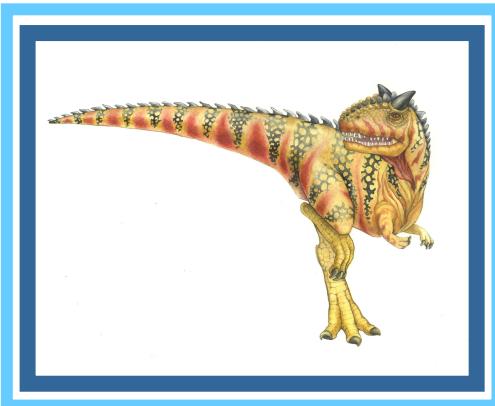


Chapter 1: Introduction

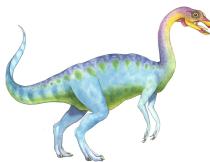




Chapter 1: Introduction

- 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
- Kernel Data Structures
- Computing Environments
- Open-Source Operating Systems





Objectives

- To describe the basic organization of computer systems
- To provide a grand tour of the major components of operating systems
- To give an overview of the many types of computing environments
- To explore several open-source operating systems





What is an Operating System?

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

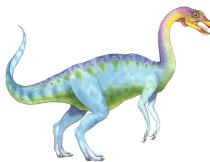




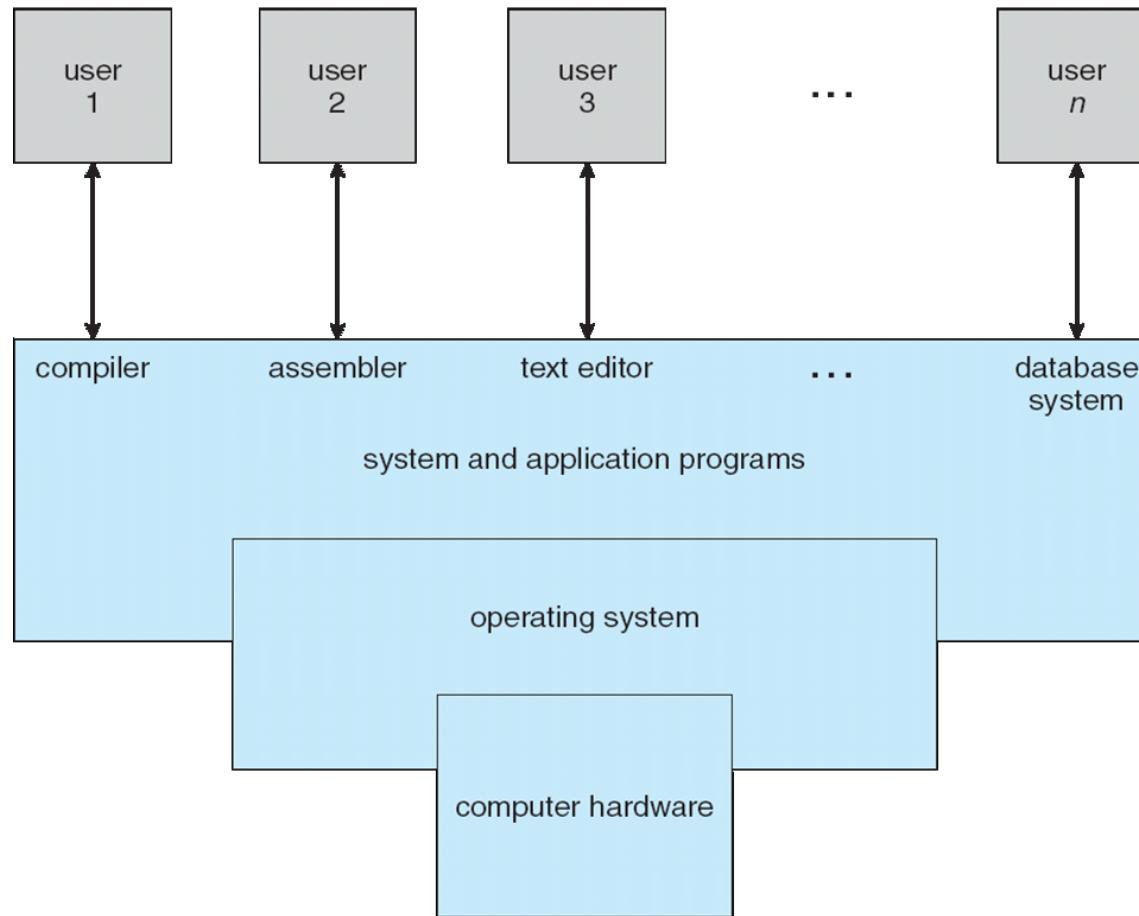
Computer System Structure

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





Four Components of a Computer System





What Operating Systems Do

- Depends on the point of view
- Users want convenience, **ease of use**
 - Don't care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicated systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles



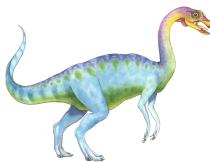


Operating System Definition

- OS is a **resource allocator**
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use

- OS is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer





Operating System Definition (Cont.)

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is good approximation
 - But varies wildly
- “The one program running at all times on the computer” is the **kernel**. Everything else is either a system program (ships with the operating system) or an application program.





Computer Startup

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

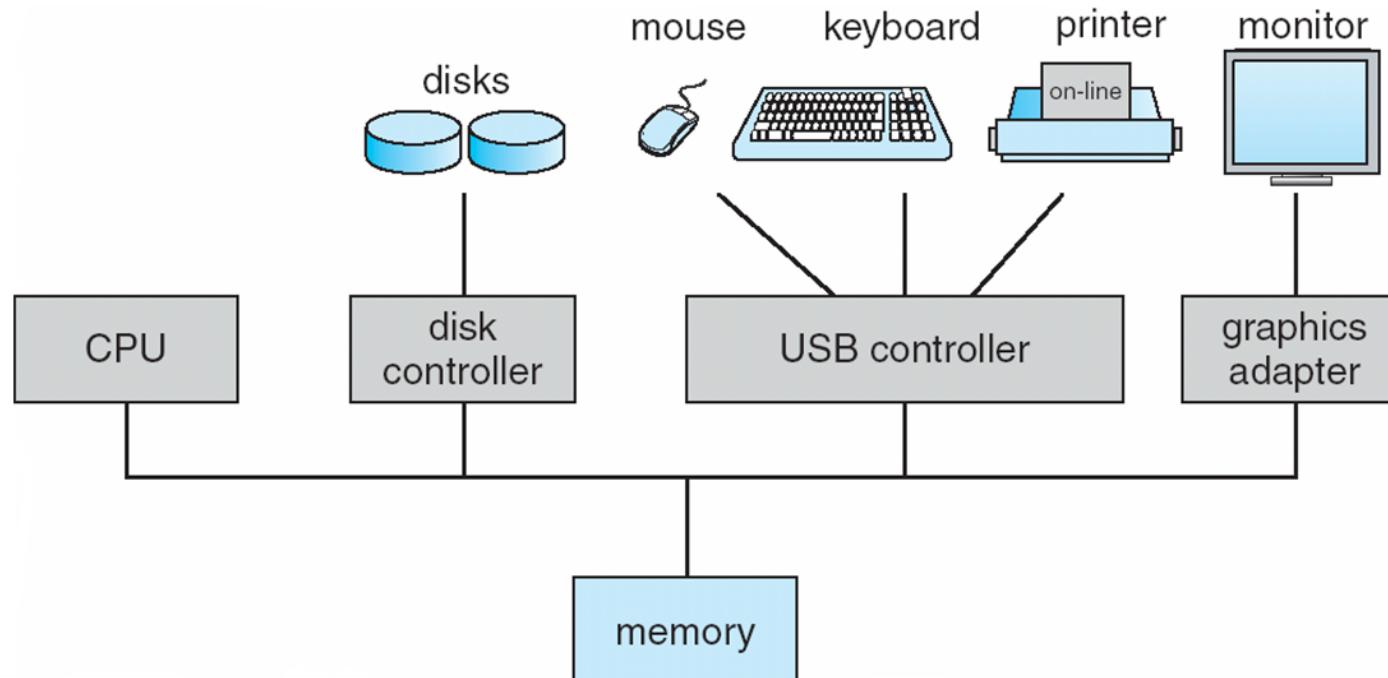




Computer System Organization

■ 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**

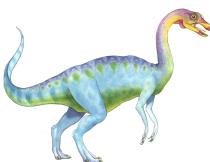




Common Functions of Interrupts

- 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
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**





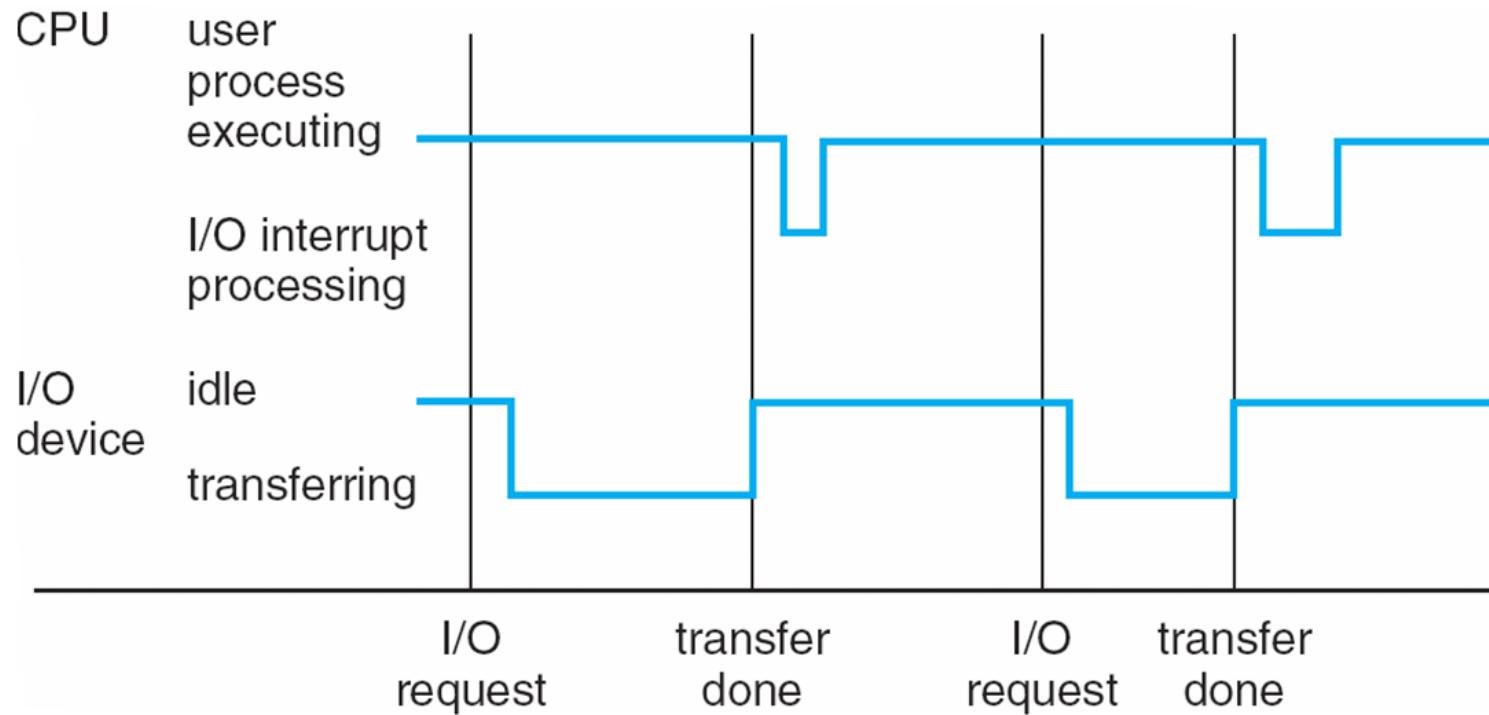
Interrupt Handling

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





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 OS 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
 - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt





Storage Definitions and Notation Review

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time. Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is 1,024² bytes; a **gigabyte**, or **GB**, is 1,024³ bytes; a **terabyte**, or **TB**, is 1,024⁴ bytes; and a **petabyte**, or **PB**, is 1,024⁵ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks





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
 - **Random access**
 - Typically **volatile**
- 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
- **Solid-state disks** – faster than magnetic disks, nonvolatile
 - Various technologies
 - Becoming more popular





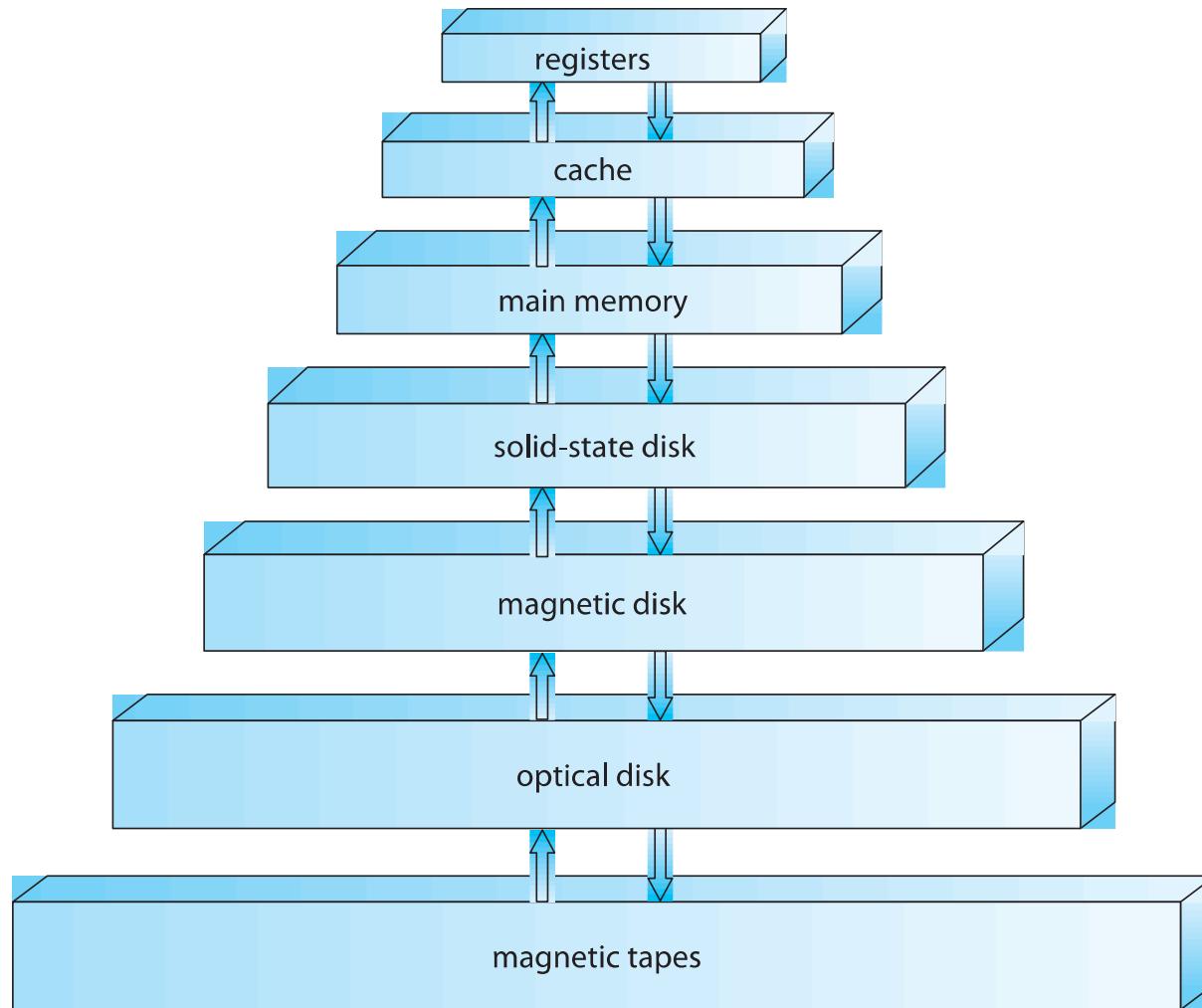
Storage Hierarchy

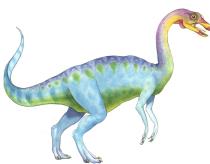
- Storage systems organized in hierarchy
 - Speed
 - Cost
 - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
 - Provides uniform interface between controller and kernel





Storage-Device Hierarchy





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





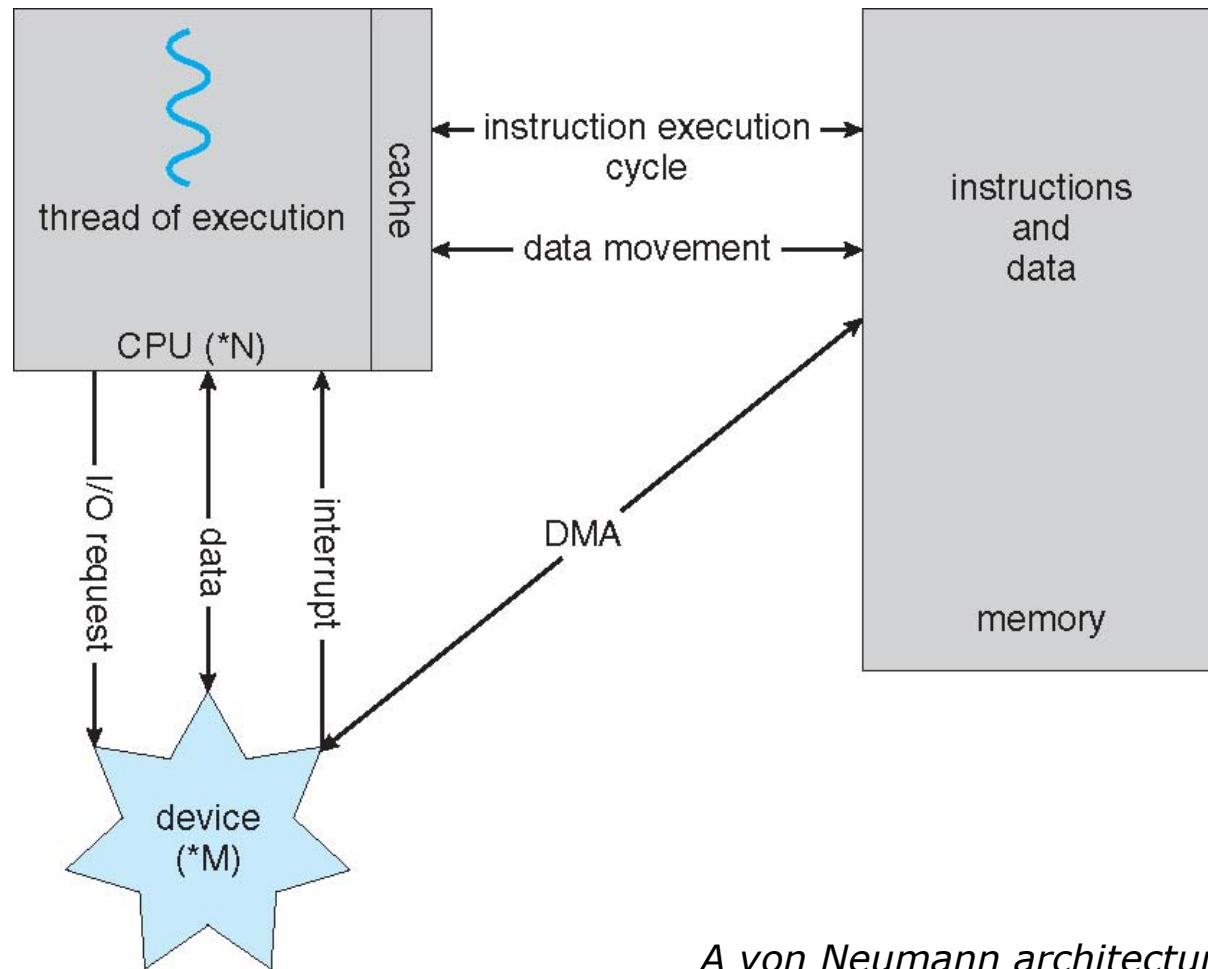
Computer-System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
 - Most systems have special-purpose processors as well
- **Multiprocessors** systems growing in use and importance
 - Also known as **parallel systems**, **tightly-coupled systems**
 - Advantages include:
 1. **Increased throughput**
 2. **Economy of scale**
 3. **Increased reliability – graceful degradation** or **fault tolerance**
 - Two types:
 1. **Asymmetric Multiprocessing**
 2. **Symmetric Multiprocessing**





How a Modern Computer Works

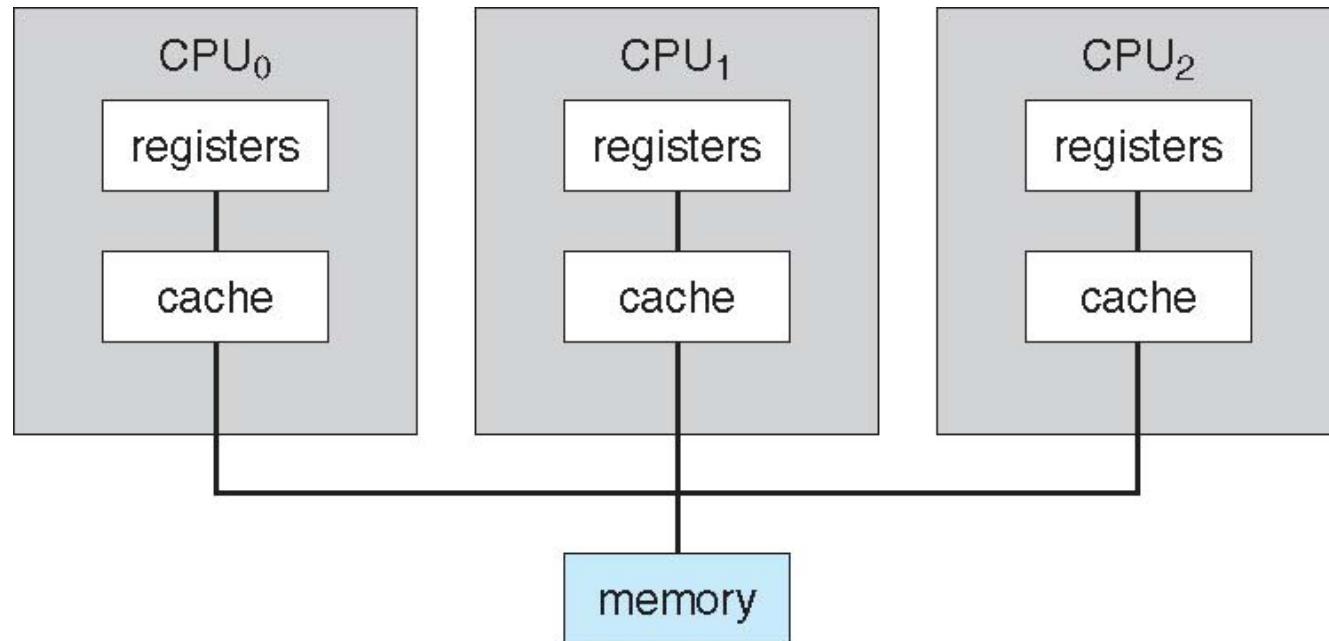


A von Neumann architecture





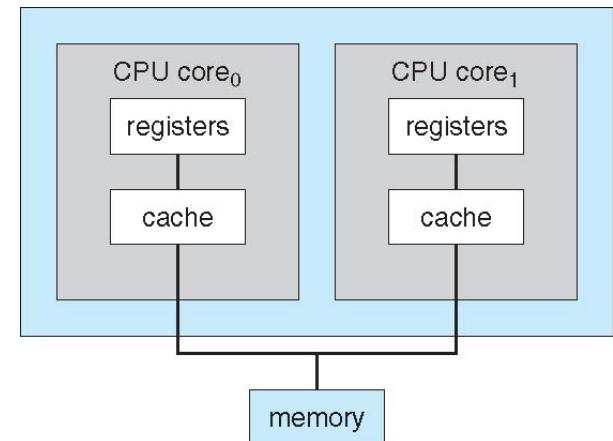
Symmetric Multiprocessing Architecture





A Dual-Core Design

- **UMA** and **NUMA**
architecture variations
- Multi-chip and **multicore**
- Systems containing all chips vs. **blade servers**
 - Chassis containing multiple separate systems

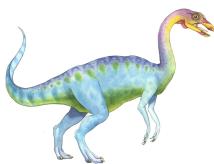




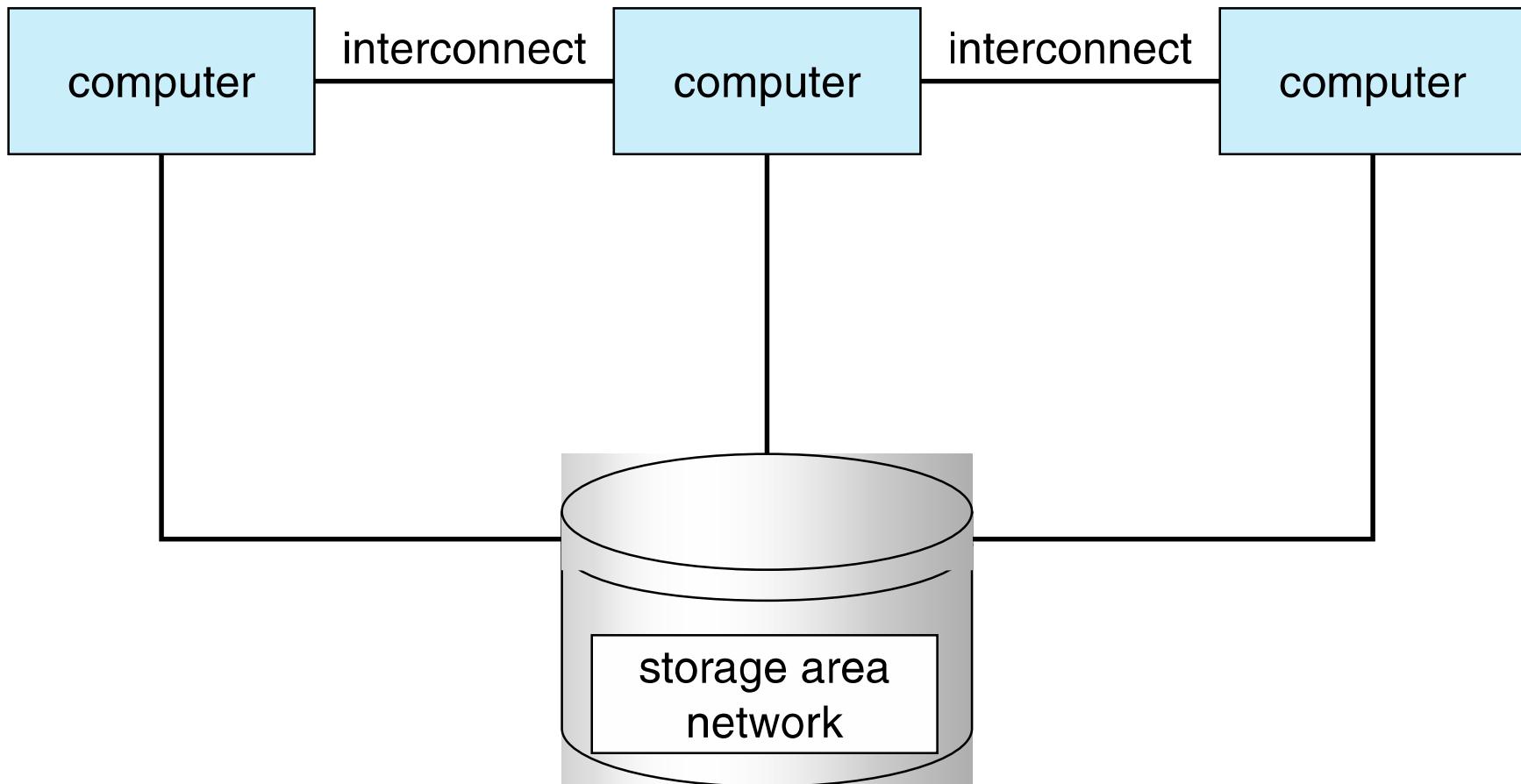
Clustered Systems

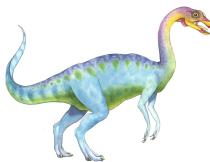
- Like multiprocessor systems, but multiple systems working together
 - Usually sharing storage via a **storage-area network (SAN)**
 - Provides a **high-availability** service which survives failures
 - ▶ **Asymmetric clustering** has one machine in hot-standby mode
 - ▶ **Symmetric clustering** has multiple nodes running applications, monitoring each other
 - Some clusters are for **high-performance computing (HPC)**
 - ▶ Applications must be written to use **parallelization**
 - Some have **distributed lock manager (DLM)** to avoid conflicting operations





Clustered Systems





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

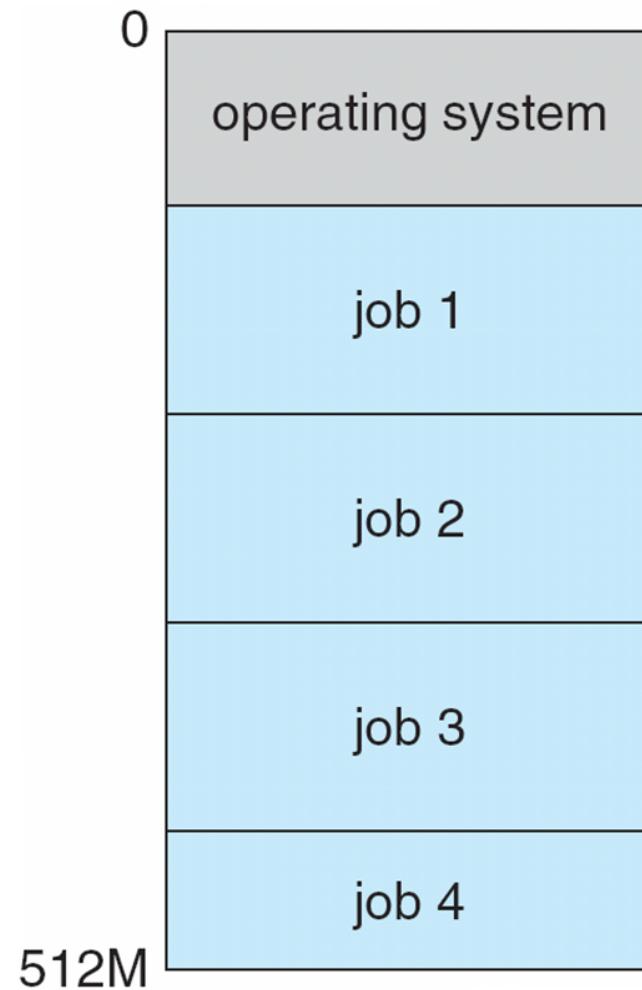
■ **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

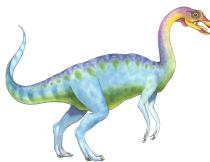
- **Response time** should be < 1 second
- Each user has at least one program executing in memory ↳ **process**
- If several jobs ready to run at the same time ↳ **CPU scheduling**
- If processes don't fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory





Memory Layout for Multiprogrammed System

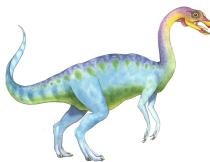




Operating-System Operations

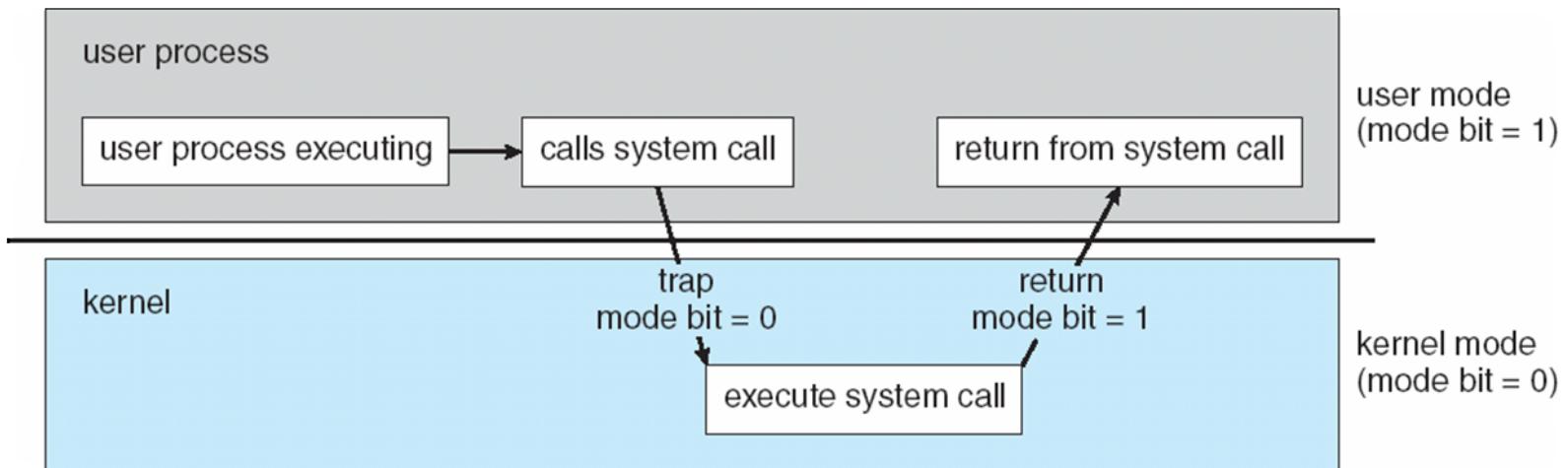
- **Interrupt driven** by hardware
- Software error or request creates **exception** or **trap**
 - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
 - **User mode** and **kernel mode**
 - **Mode bit** provided by hardware
 - ▶ Provides ability to distinguish when system is running user code or kernel code
 - ▶ Some instructions designated as **privileged**, only executable in kernel mode
 - ▶ System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
 - i.e. **virtual machine manager (VMM)** mode for guest **VMs**





Transition from User to Kernel Mode

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





Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task
 - CPU, memory, I/O, files
 - Initialization data
- Process 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 one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
 - Concurrency by multiplexing the CPUs among the processes / threads





Process Management Activities

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

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





Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
 - Optimizing CPU utilization and computer response to users
- Memory management activities
 - Keeping track of which parts of memory are currently being used and by whom
 - Deciding which processes (or parts thereof) and data to move into and out of memory
 - Allocating and deallocating memory space as needed

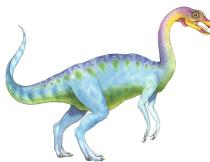




Storage Management

- 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
 - ▶ Creating and deleting files and directories
 - ▶ Primitives to manipulate files and dirs
 - ▶ Mapping files onto secondary storage
 - ▶ Backup files onto stable (non-volatile) storage media





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
 - Free-space management
 - Storage allocation
 - Disk scheduling
- Some storage need not be fast
 - Tertiary storage includes optical storage, magnetic tape
 - Still must be managed – by OS or applications
 - Varies between WORM (write-once, read-many-times) and RW (read-write)





Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

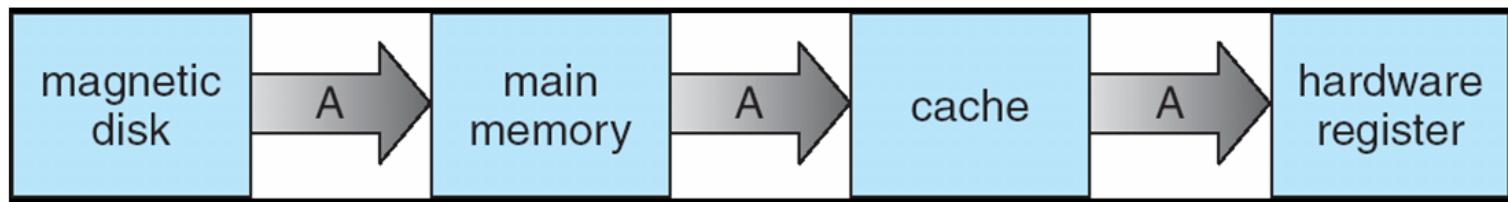
- Movement between levels of storage hierarchy can be explicit or implicit





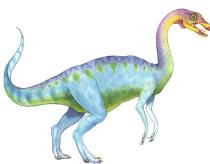
Migration of Integer A from Disk to Register

- 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
 - Various solutions covered in Chapter 17





I/O Subsystem

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





Protection and Security

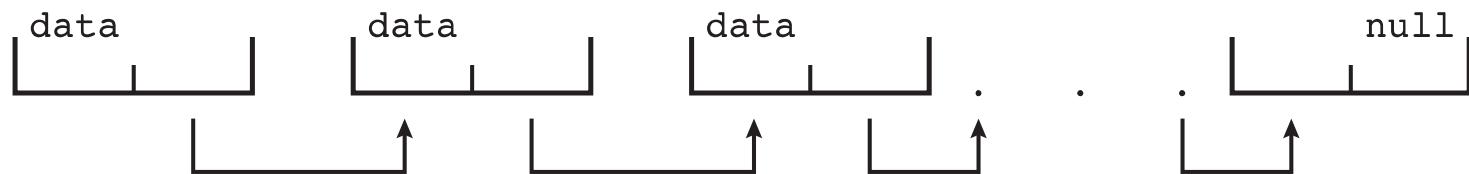
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
 - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first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



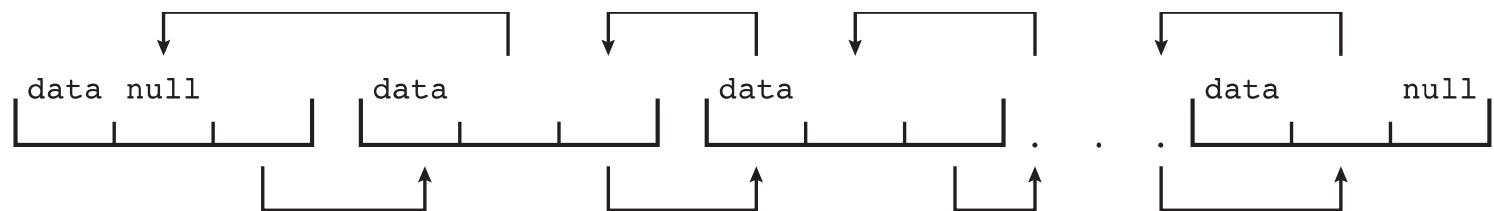


Kernel Data Structures

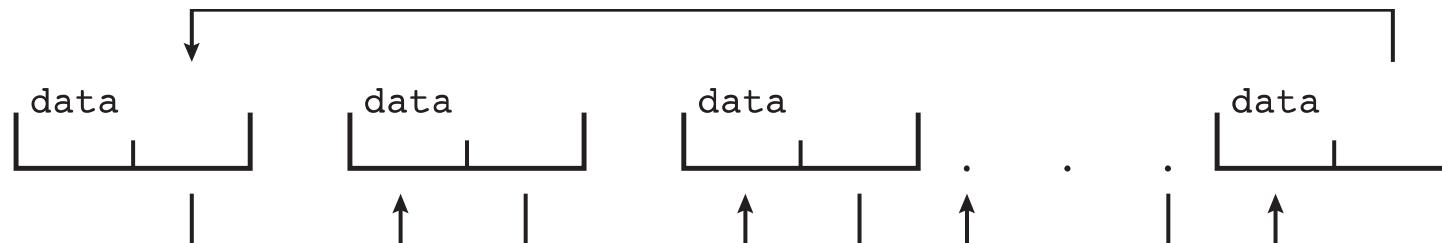
- Many similar to standard programming data structures
- ***Singly linked list***



- ***Doubly linked list***



- ***Circular linked list***



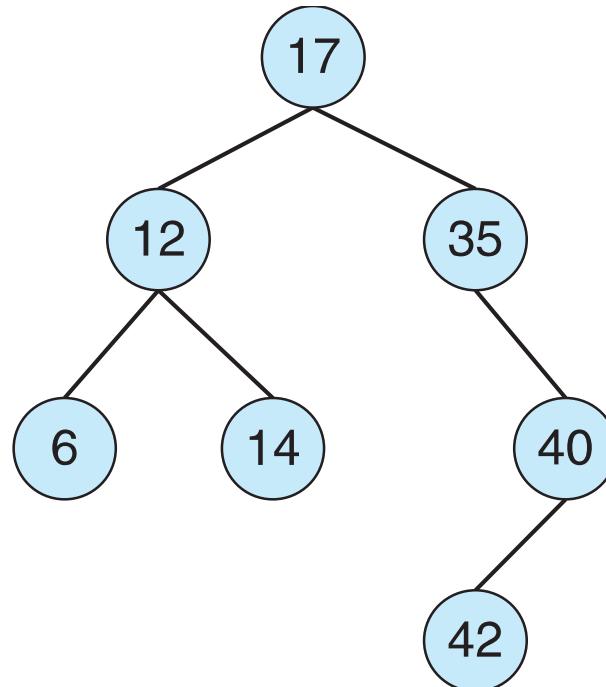


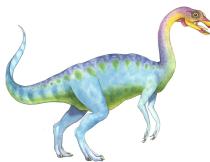
Kernel Data Structures

■ Binary search tree

left \leq right

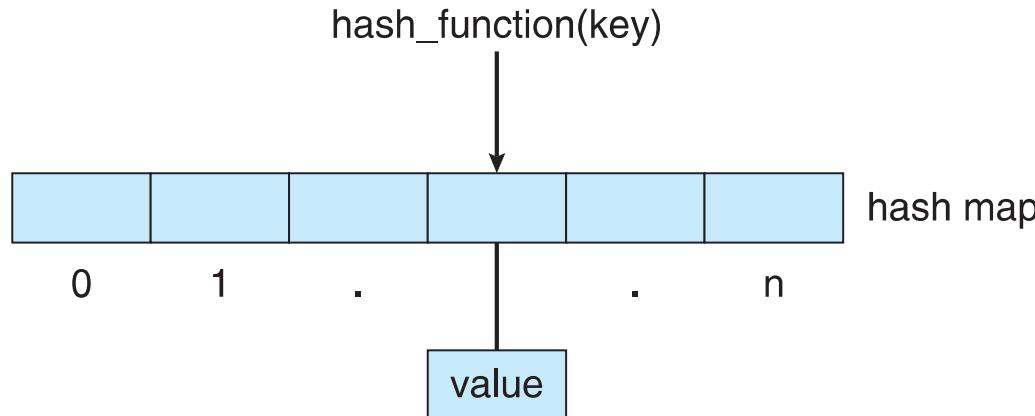
- Search performance is $O(n)$
- **Balanced binary search tree** is $O(\lg n)$





Kernel Data Structures

- Hash function can create a hash map



- Bitmap – string of n binary digits representing the status of n items
- Linux data structures defined in ***include*** files
`<linux/list.h>`, `<linux/kfifo.h>`,
`<linux/rbtree.h>`





Computing Environments - Traditional

- Stand-alone general purpose machines
- But blurred as most systems interconnect with others (i.e. the Internet)
- **Portals** provide web access to internal systems
- **Network computers (thin clients)** are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks





Computing Environments - Mobile

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





Computing Environments – Distributed

■ Distributed

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

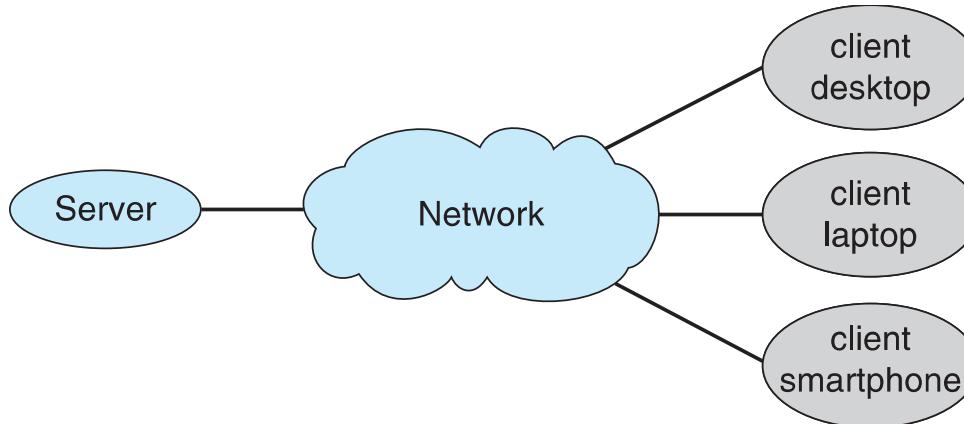




Computing Environments – Client-Server

Client-Server Computing

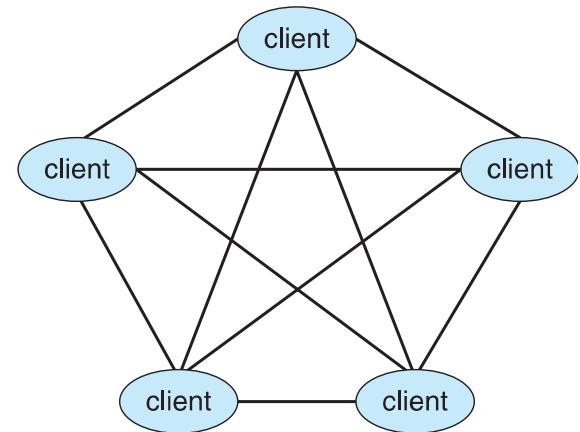
- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
 - **Compute-server system** provides an interface to client to request services (i.e., database)
 - **File-server system** provides interface for clients to store and retrieve files





Computing Environments - Peer-to-Peer

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





Computing Environments - Virtualization

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





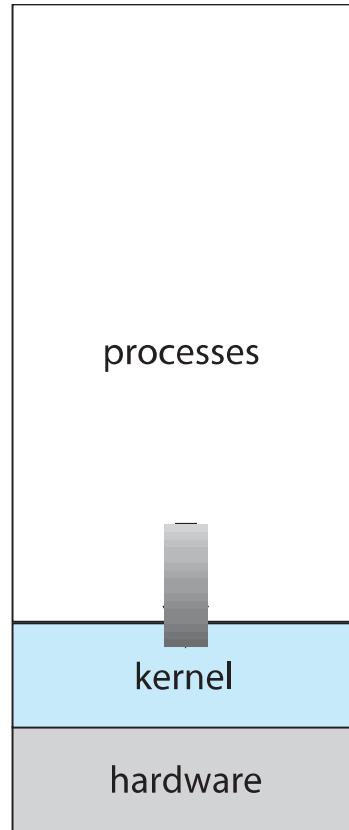
Computing Environments - Virtualization

- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
 - Apple laptop running Mac OS X host, Windows as a guest
 - Developing apps for multiple OSes without having multiple systems
 - QA testing applications without having multiple systems
 - Executing and managing compute environments within data centers
- VMM can run natively, in which case they are also the host
 - There is no general purpose host then (VMware ESX and Citrix XenServer)

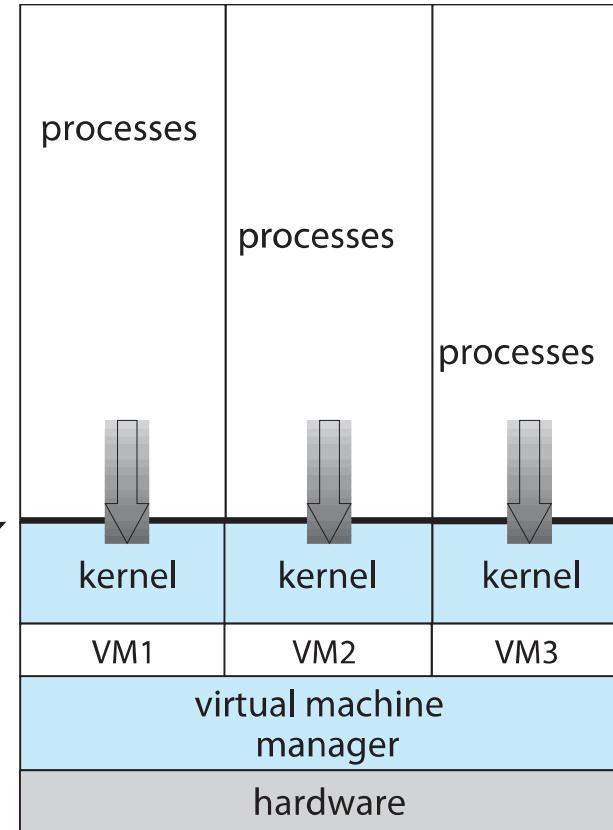




Computing Environments - Virtualization



(a)



(b)





Computing Environments – Cloud Computing

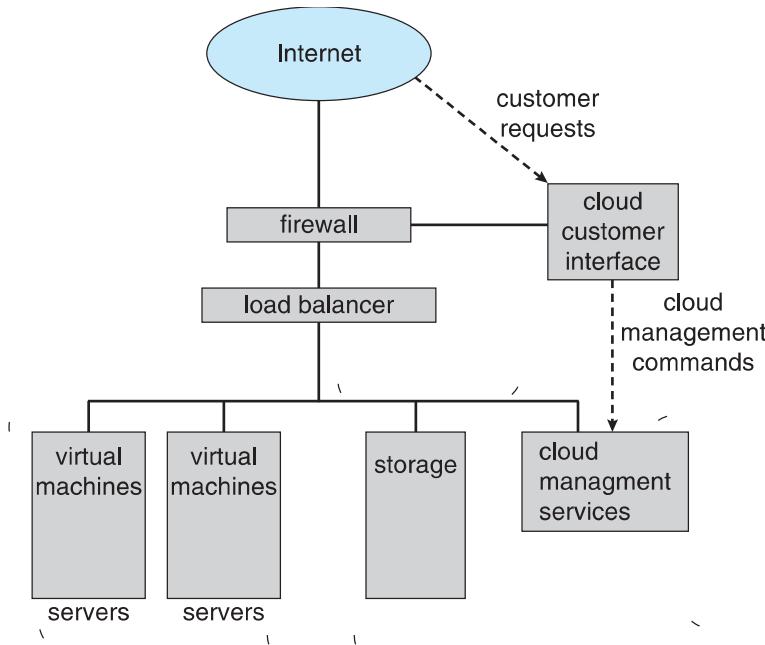
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization as based on virtualization
 - Amazon **EC2** has thousands of servers, millions of VMs, PBs of storage available across the Internet, pay based on usage
- Many types
 - **Public cloud** – available via Internet to anyone willing to pay
 - **Private cloud** – run by a company for the company's own use
 - **Hybrid cloud** – includes both public and private cloud components
 - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e. word processor)
 - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e. a database server)
 - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e. storage available for backup use)

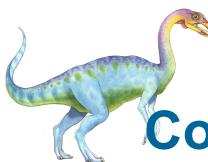




Computing Environments – Cloud Computing

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





Computing Environments – Real-Time Embedded Systems

- Real-time embedded systems most prevalent form of computers
 - Vary considerable, special purpose, limited purpose OS, **real-time OS**
 - Use expanding
- Many other special computing environments as well
 - Some have OSes, some perform tasks without an OS
- Real-time OS has well-defined fixed time constraints
 - Processing **must** be done within constraint
 - Correct operation only if constraints met



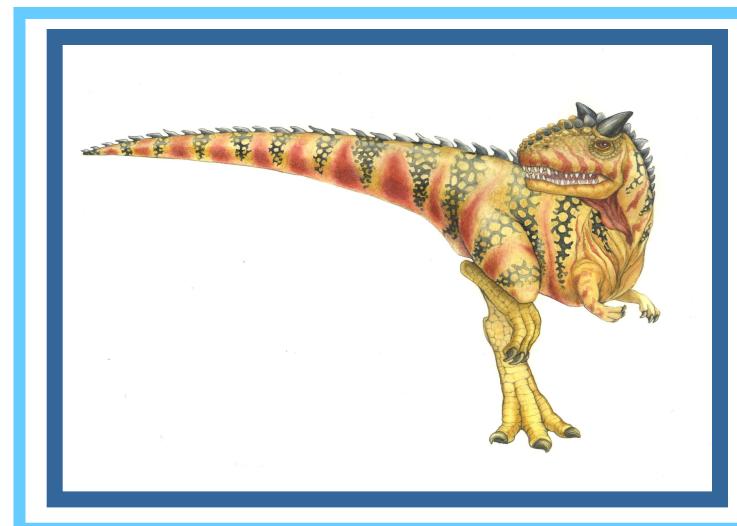


Open-Source Operating Systems

- 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** and **BSD UNIX** (including core of **Mac OS X**), and many more
- Can use VMM like VMware Player (Free on Windows), Virtualbox (open source and free on many platforms - <http://www.virtualbox.com>)
 - Use to run guest operating systems for exploration



Chapter 3: Process Concept





Chapter 3: Process Concept

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





Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems





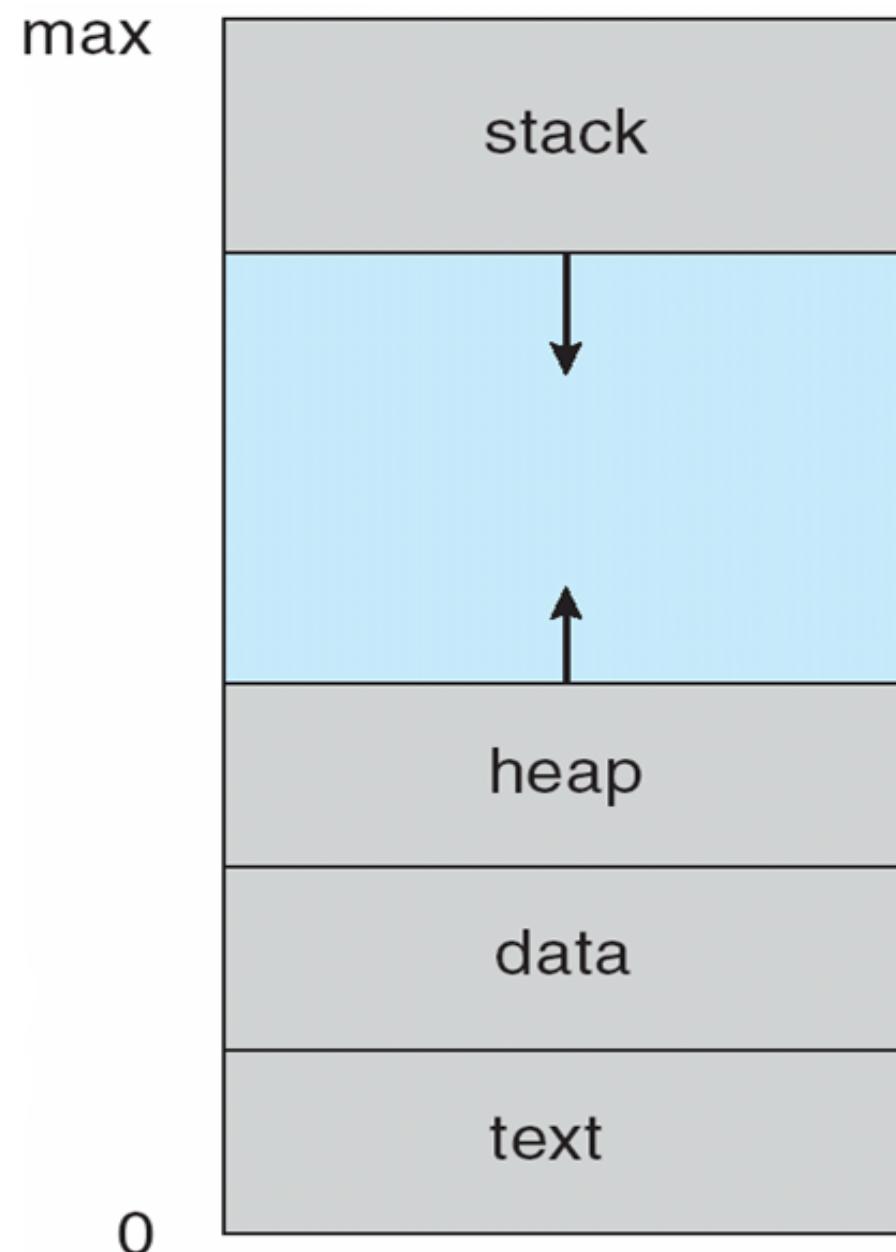
Process Concept

- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time
- Program is **passive** entity stored on disk (**executable file**), process is **active**
 - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program





Process in Memory





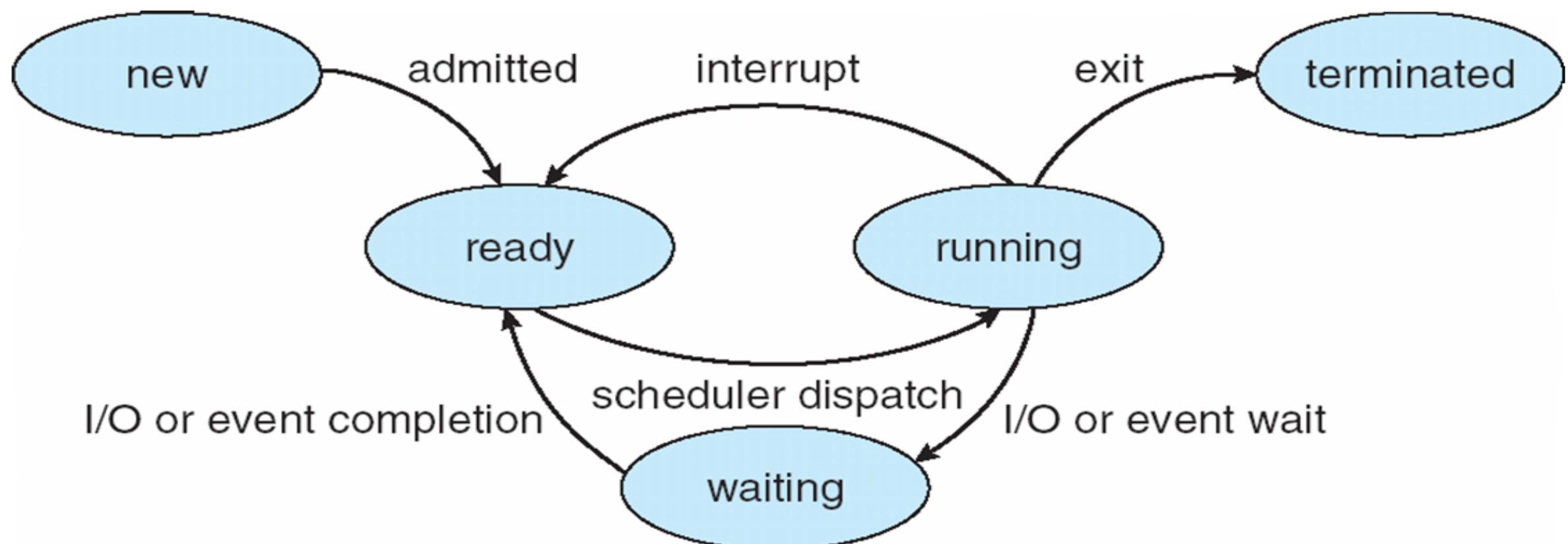
Process State

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





Diagram of Process State

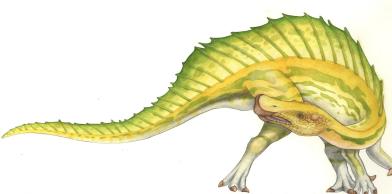
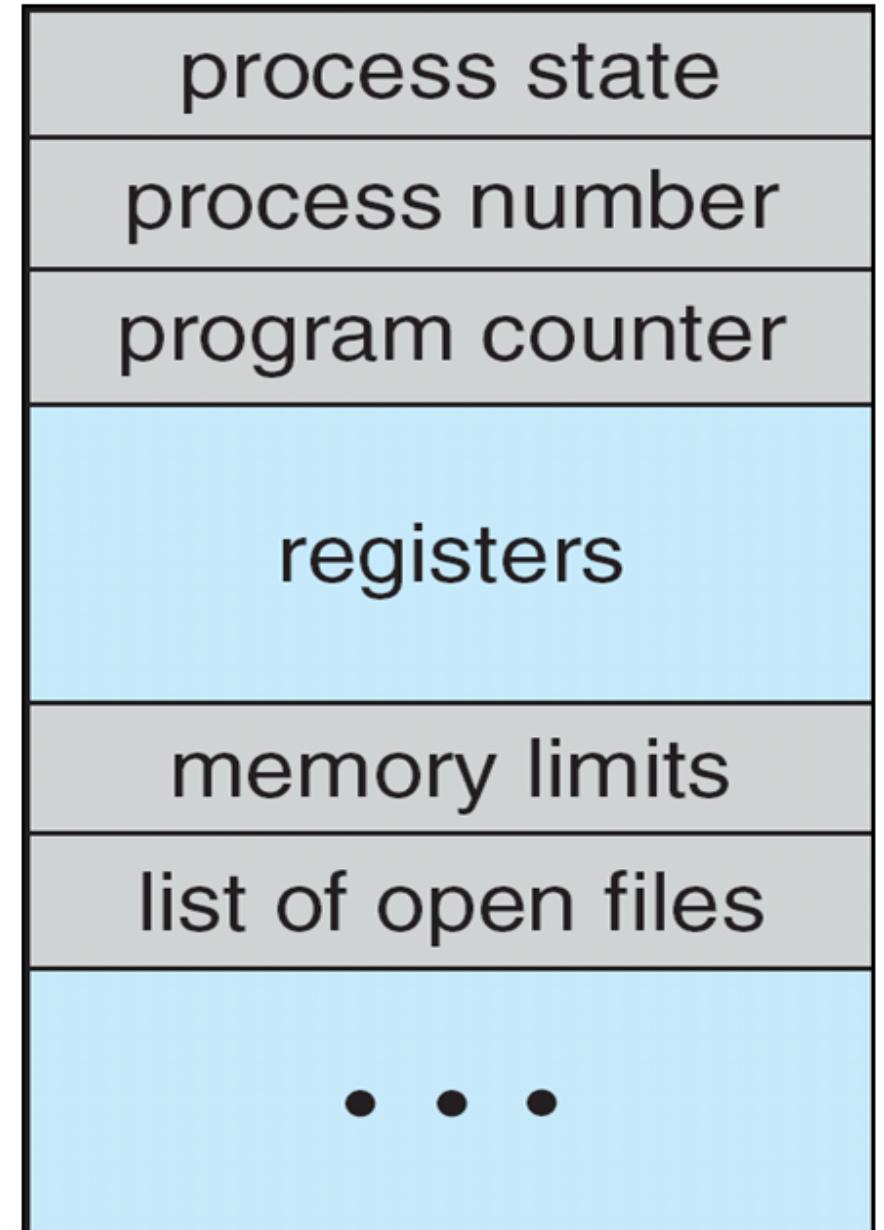




Process Control Block (PCB)

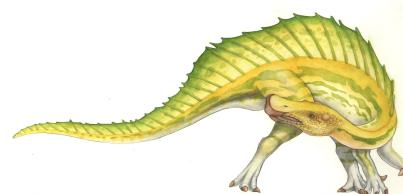
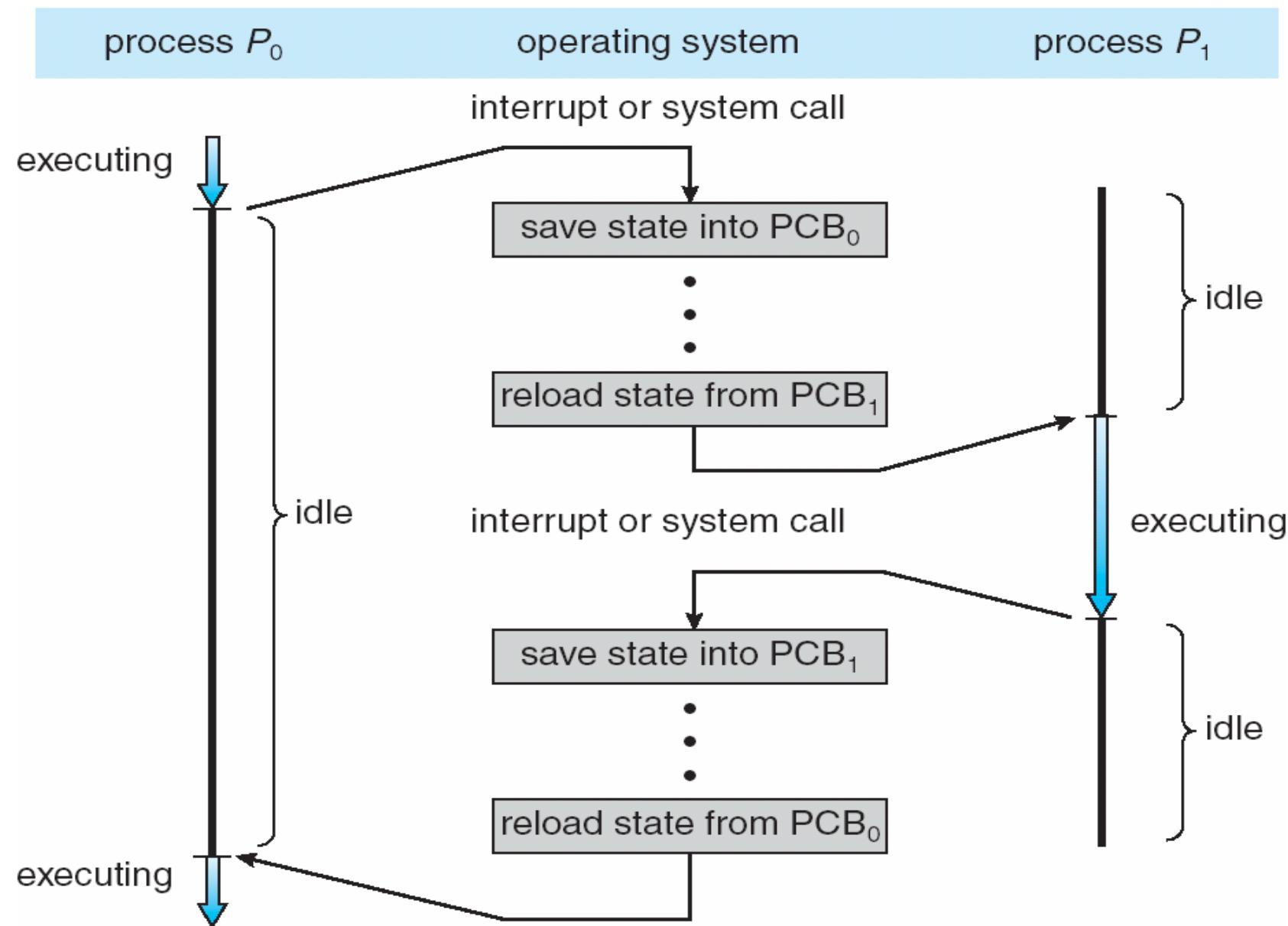
Information associated with each process
(also called **task control block**)

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





CPU Switch From Process to Process

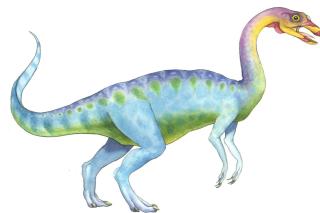




Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

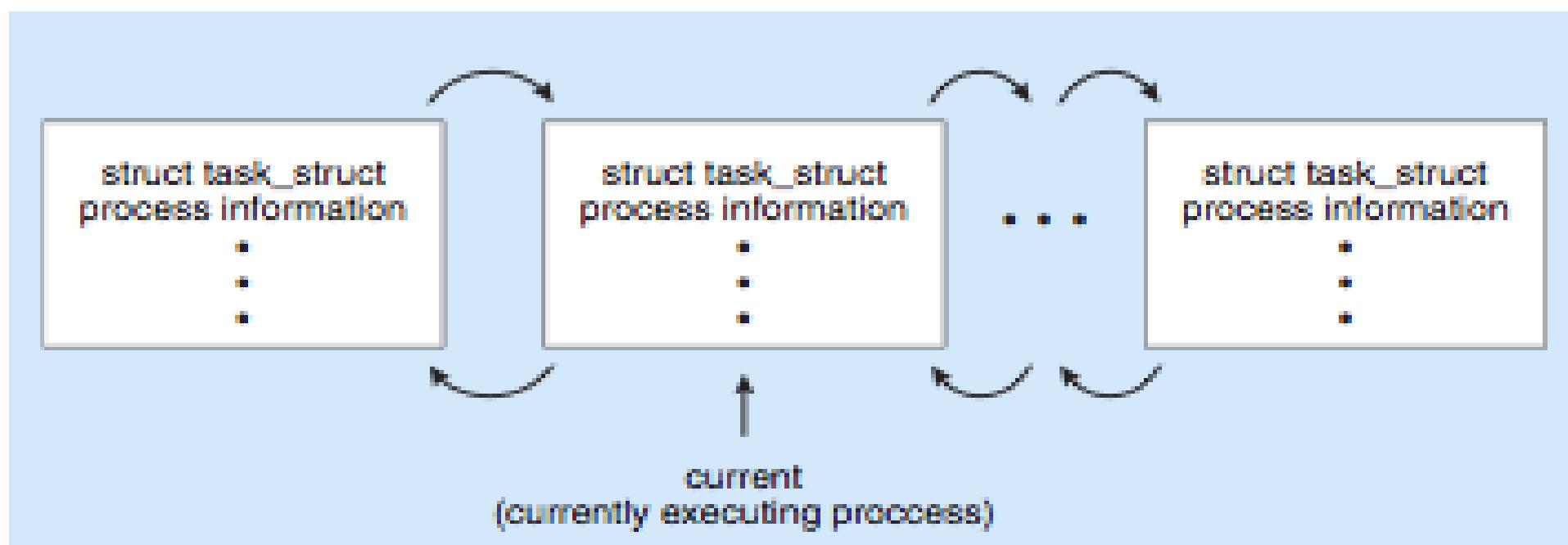




Process Representation in Linux

- Represented by the C structure `task_struct`

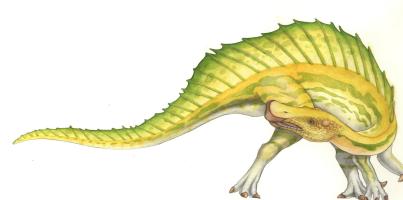
```
pid_t pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```

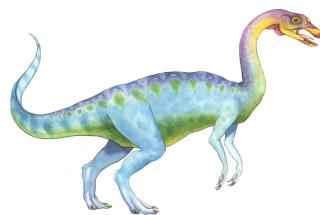




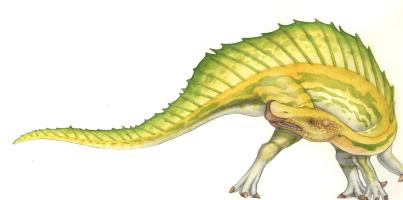
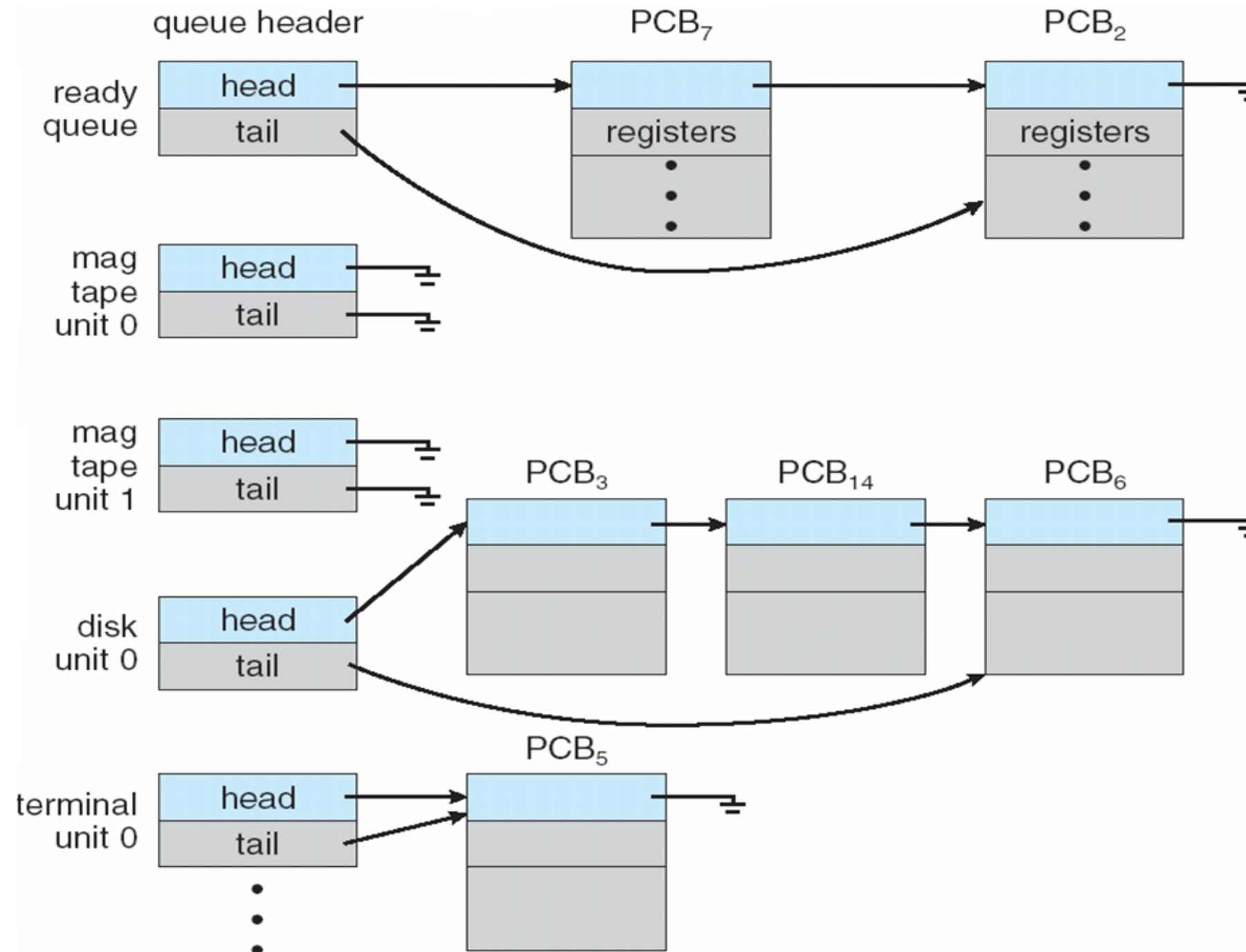
Process Scheduling

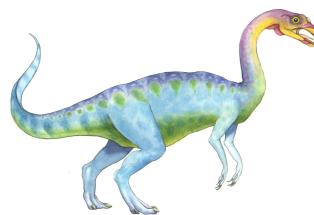
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing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





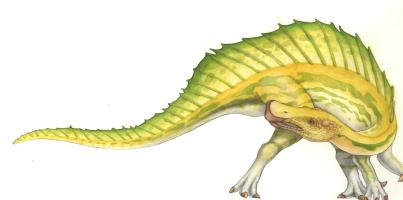
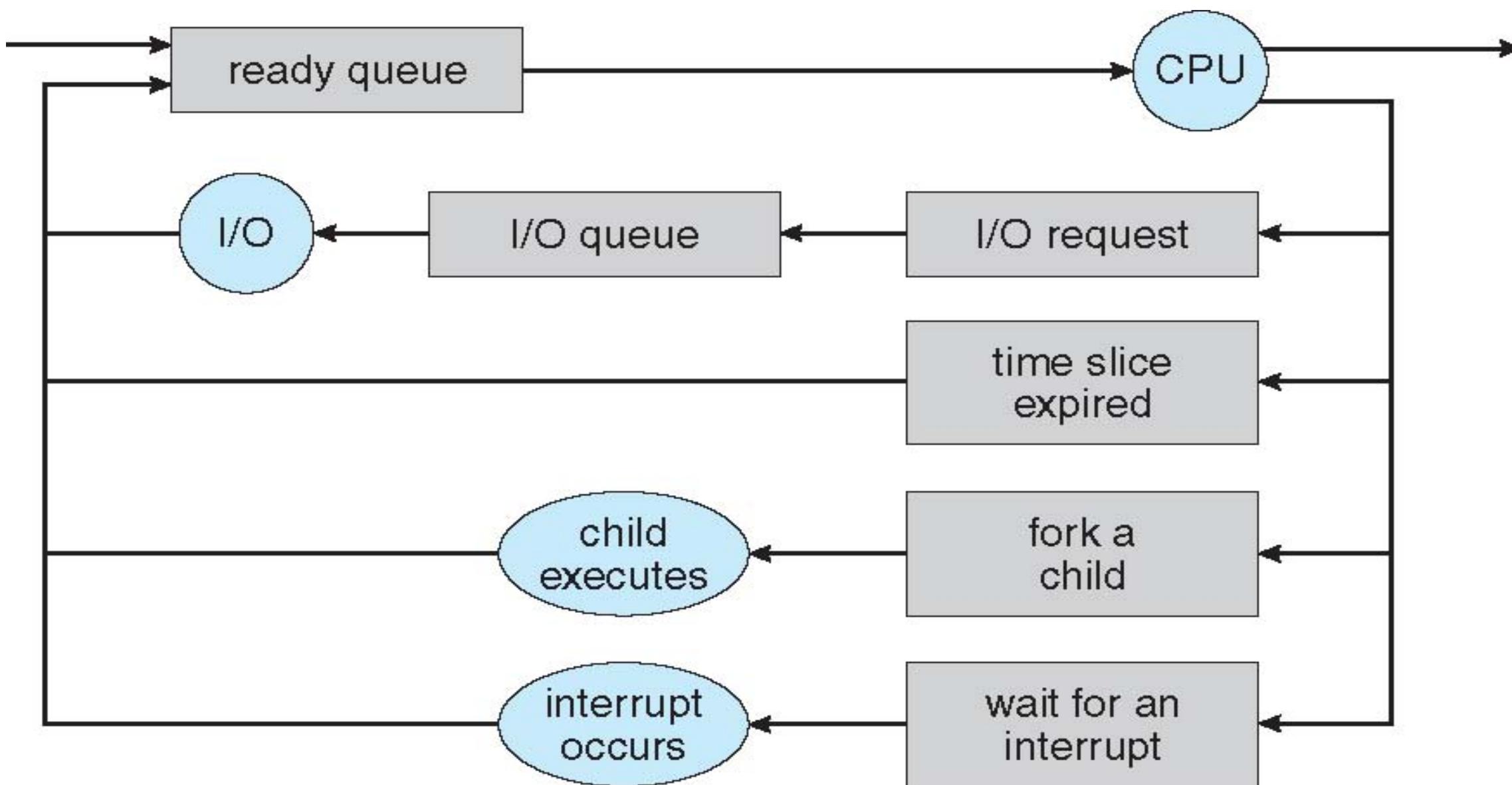
Ready Queue And Various I/O Device Queues





Representation of Process Scheduling

- Queuing diagram represents queues, resources, flows





Schedulers

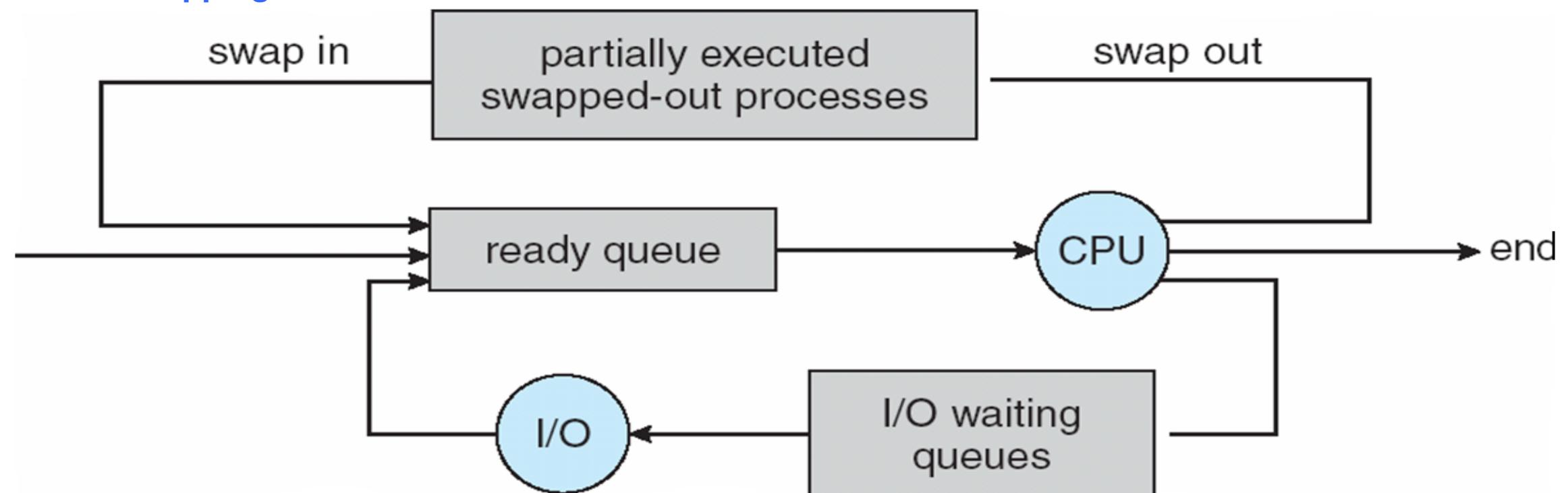
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
- Short-term scheduler is invoked very frequently (milliseconds) \Rightarrow (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) \Rightarrow (may be slow)
- The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***





Addition of Medium Term Scheduling

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





Multitasking in Mobile Systems

- Some systems / early systems allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
 - Single **foreground** process- controlled via user interface
 - Multiple **background** processes— in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
 - Background process uses a **service** to perform tasks
 - Service can keep running even if background process is suspended
 - Service has no user interface, small memory use

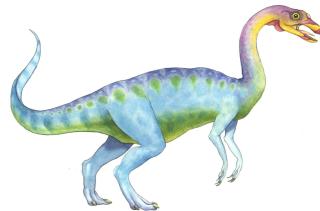




Context Switch

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





Operations on Processes

- System must provide mechanisms for process creation, termination, and so on as detailed next





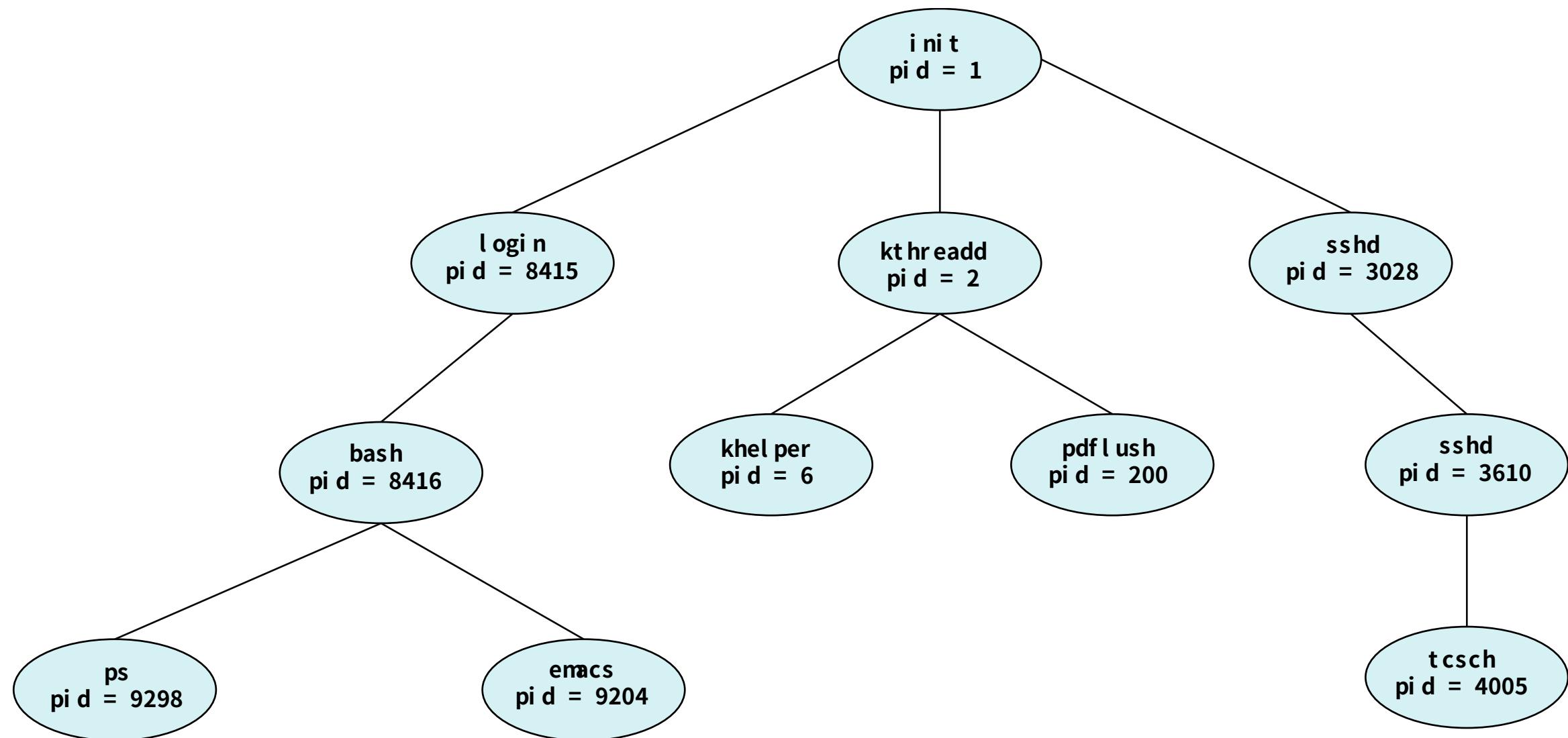
Process Creation

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





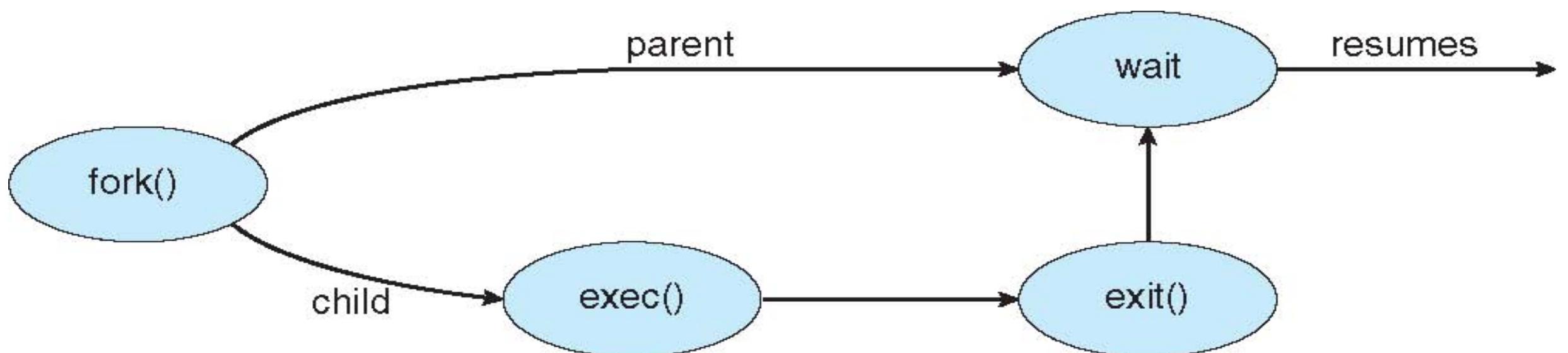
A Tree of Processes in Linux

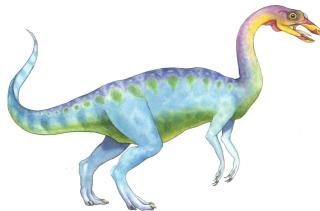




Process Creation (Cont.)

- 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





C Program Forking Separate Process

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

int main()
{
    pid_t pid;

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

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

    return 0;
}
```





Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\\\\WINDOWS\\\\system32\\\\mspaint.exe", /* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit thread handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```





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 systems do not allow child to continue if its parent terminates
 - All children terminated - **cascading termination**
- Wait for termination, returning the pid:

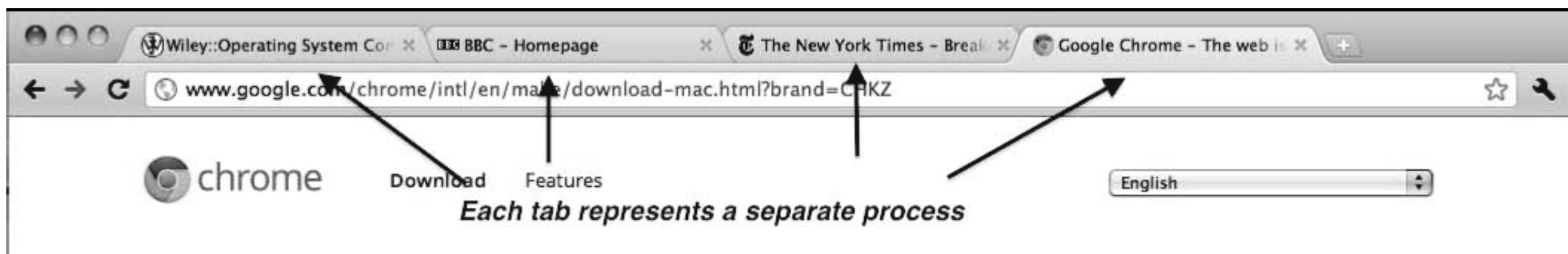
```
pid t pid; int status;  
pid = wait(&status);
```
- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**





Multiprocess Architecture – Chrome Browser

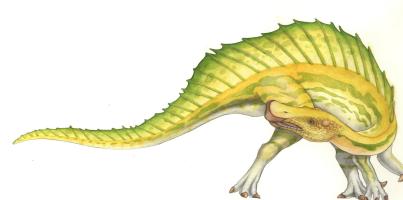
- Many web browsers ran as single process (some still do)
 - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 categories
 - **Browser** process manages user interface, disk and network I/O
 - **Renderer** process renders web pages, deals with HTML, Javascript, new one for each website opened
 - ▶ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
 - **Plug-in** process for each type of plug-in

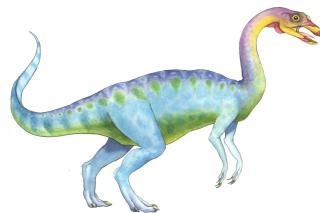




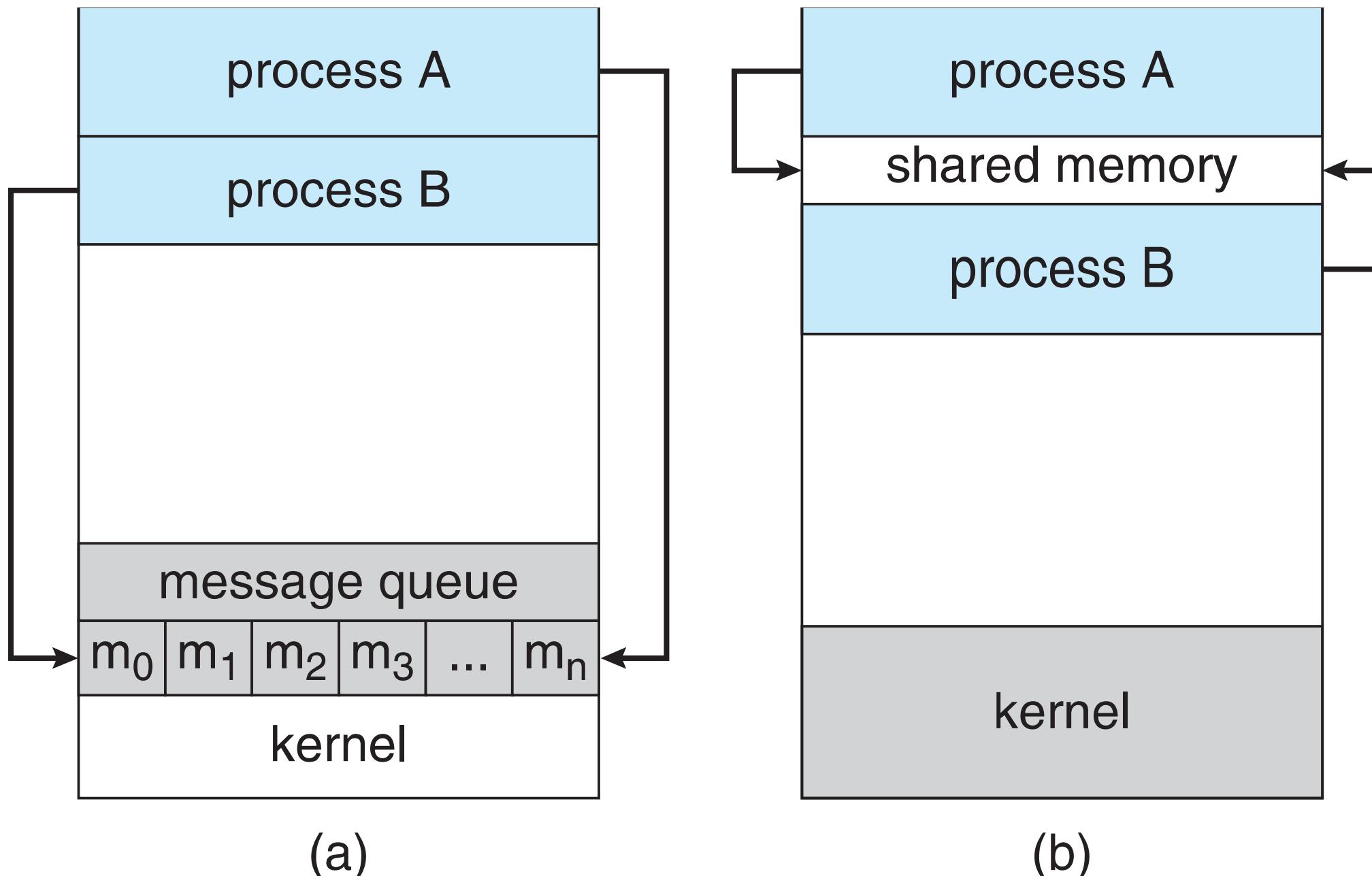
Interprocess Communication

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





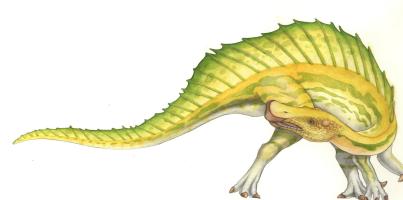
Communications Models





Cooperating Processes

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





Producer-Consumer Problem

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





Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10

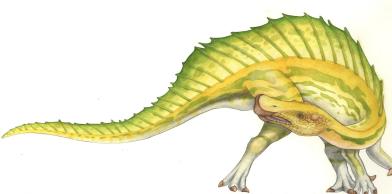
typedef struct {

    . . .

} item;

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

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





Bounded-Buffer – Producer

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





Bounded Buffer – Consumer

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





Interprocess Communication – 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., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)





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?





Direct Communication

- Processes must name each other explicitly:
 - `send(P, message)` – send a message to process P
 - `receive(Q, message)` – receive a message from process Q

- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional





Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox

- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

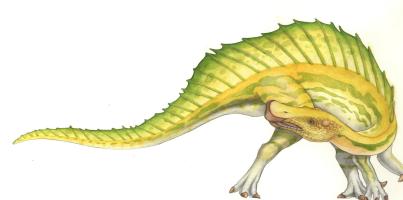




Indirect Communication

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

- Primitives are defined as:
 - send(*A, message*)** – send a message to mailbox A
 - receive(*A, message*)** – receive a message from mailbox A





Indirect Communication

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 , sends; P_2 and P_3 receive
 - Who gets the message?

- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.





Synchronization

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

}



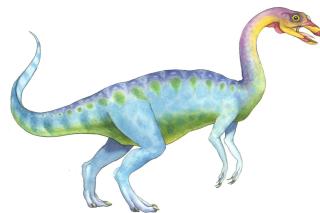


Synchronization (Cont.)

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

```
message next produced;  
  
while (true) {  
    /* produce an item in next produced */  
    send(next produced);  
  
}  
  
message next consumed;  
while (true) {  
    receive(next consumed);  
  
    /* consume the item in next consumed */  
}
```





Buffering

- Queue of messages attached to the link; implemented in one of three ways
 1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits





Examples of IPC Systems - POSIX

■ POSIX Shared Memory

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Also used to open an existing segment to share it

- Set the size of the object

```
ftruncate(shm_fd, 4096);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```





IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

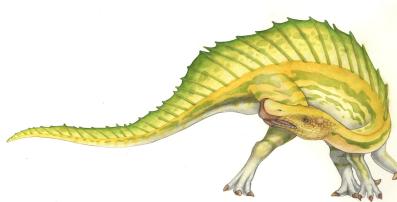
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```





IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```





Examples of IPC Systems - Mach

- Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
`msg_send()`, `msg_receive()`, `msg_rpc()`
 - Mailboxes needed for communication, created via
`port_allocate()`
 - Send and receive are flexible, for example four options if mailbox full:
 - ▶ Wait indefinitely
 - ▶ Wait at most n milliseconds
 - ▶ Return immediately
 - ▶ Temporarily cache a message

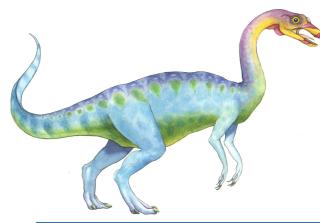




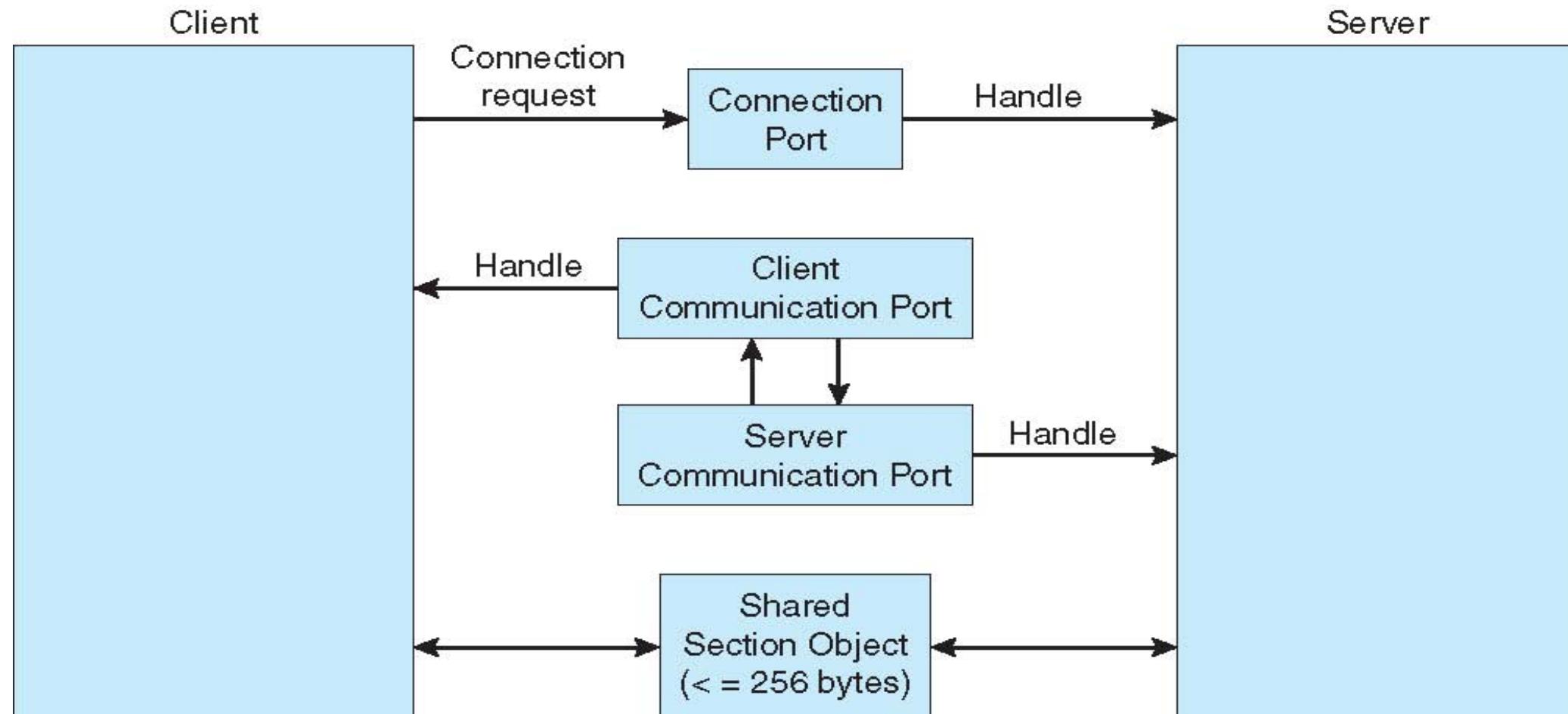
Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
 - Only works between processes on the same system
 - Uses ports (like mailboxes) to establish and maintain communication channels
 - Communication works as follows:
 - ▶ The client opens a handle to the subsystem's **connection port** object.
 - ▶ The client sends a connection request.
 - ▶ The server creates two private **communication ports** and returns the handle to one of them to the client.
 - ▶ The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.





Local Procedure Calls in Windows XP





Communications in Client-Server Systems

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





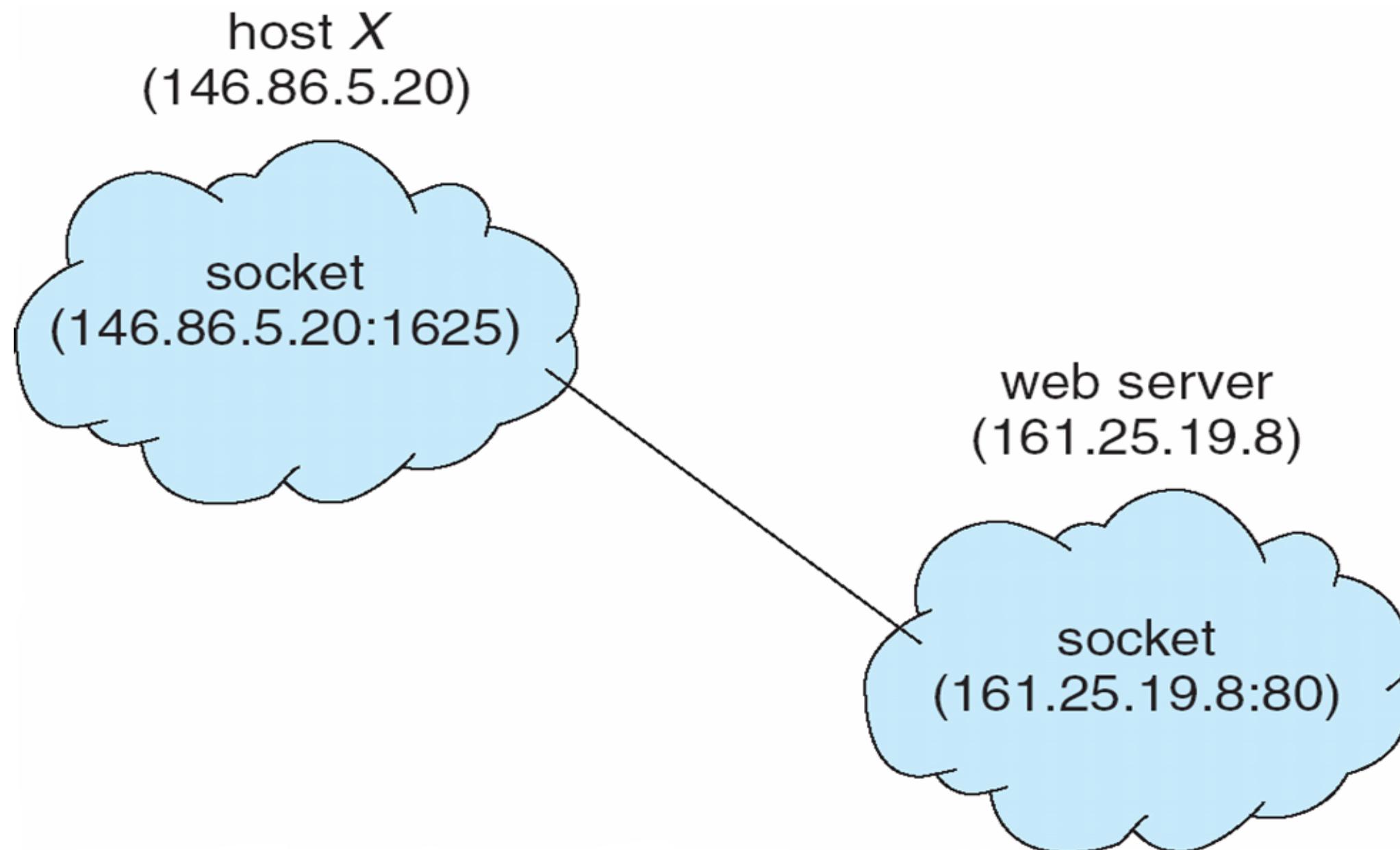
Sockets

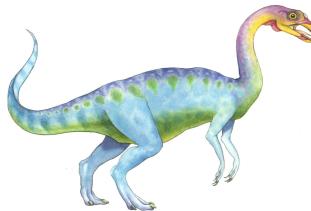
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running





Socket Communication





Sockets in Java

- Three types of sockets
 - **Connection-oriented (TCP)**
 - **Connectionless (UDP)**
 - **MulticastSocket** class— data can be sent to multiple recipients
- Consider this “Date” server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

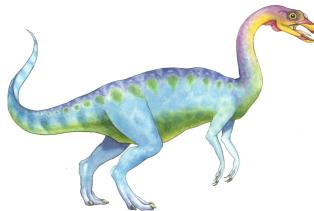
            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

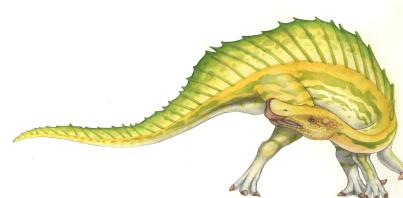
                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```





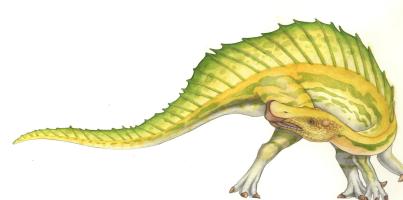
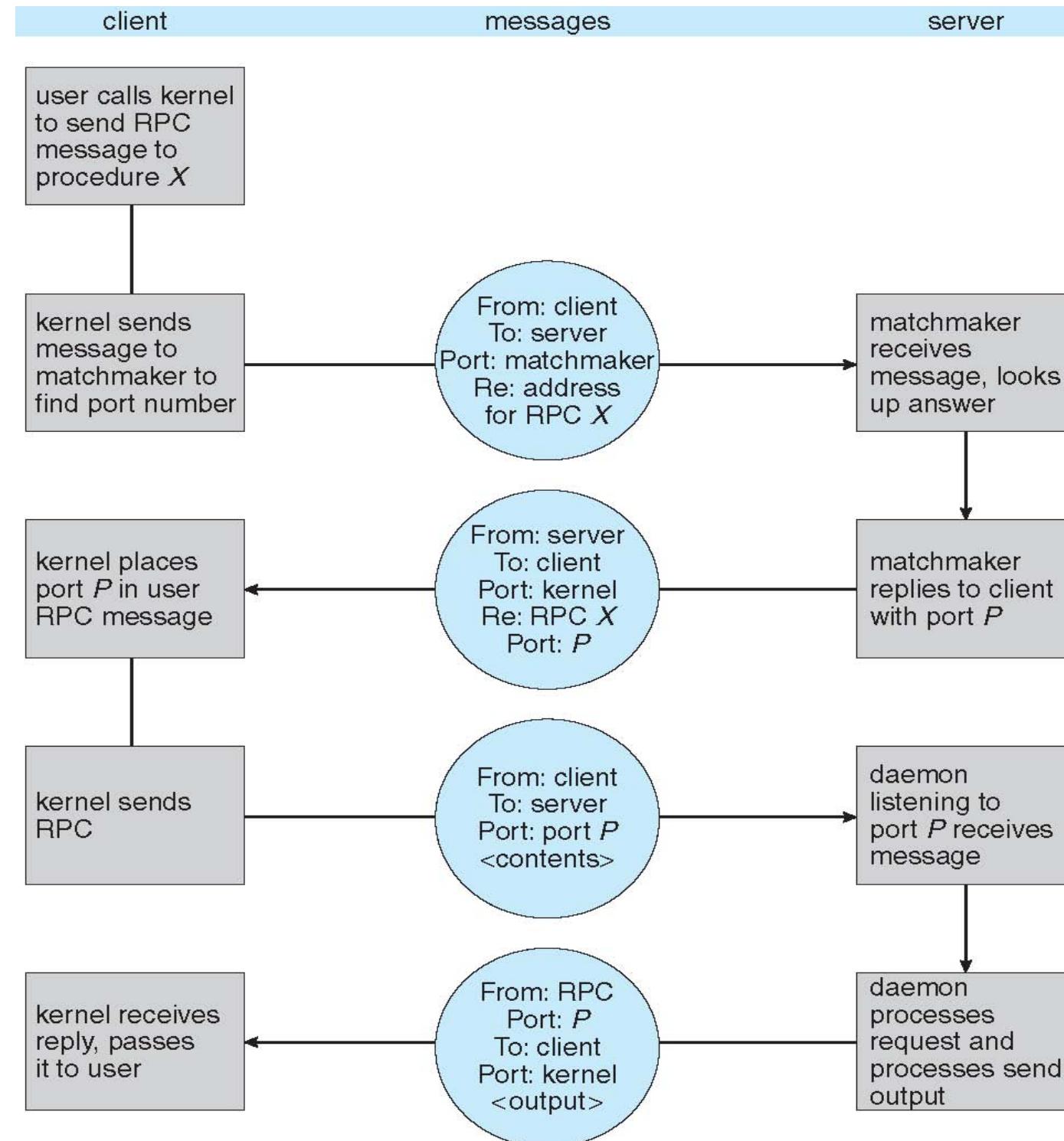
Remote Procedure Calls

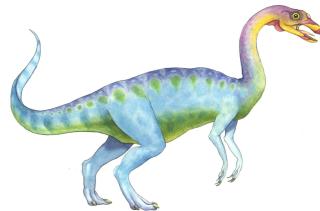
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
 - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**
- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
 - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
 - Messages can be delivered **exactly once** rather than **at most once**
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server





Execution of RPC





Pipes

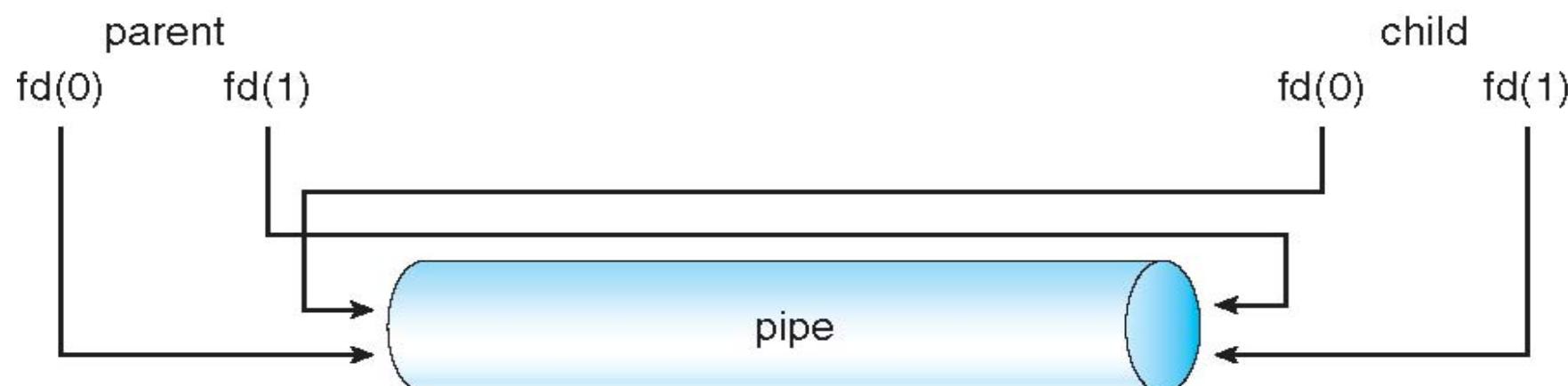
- Acts as a conduit allowing two processes to communicate
- Issues
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e. **parent-child**) between the communicating processes?
 - Can the pipes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





Ordinary Pipes

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



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook



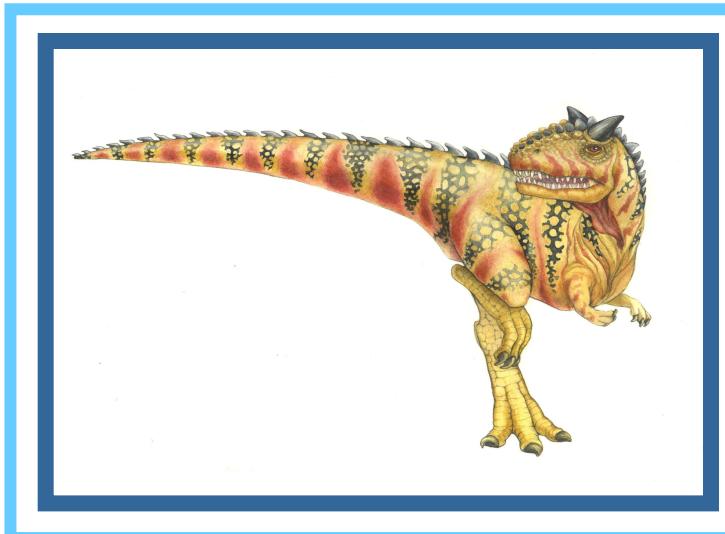


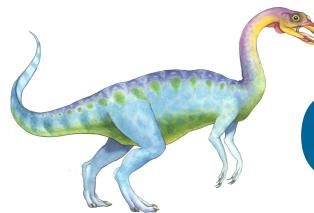
Named Pipes

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



Chapter 4: Multithreaded Programming





Chapter 4: Multithreaded Programming

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples





Objectives

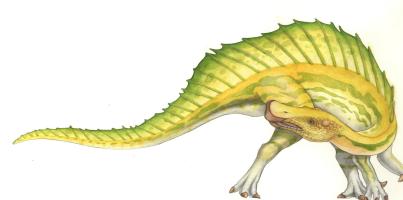
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

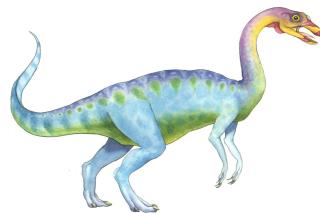




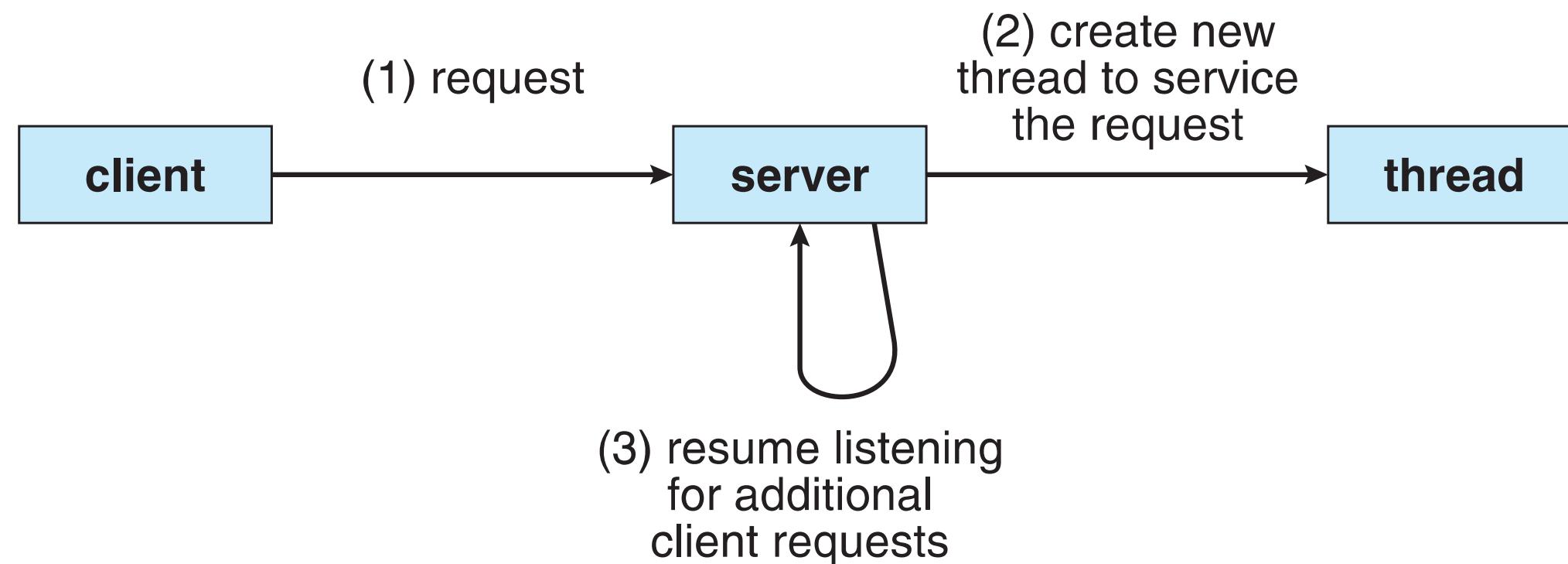
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded





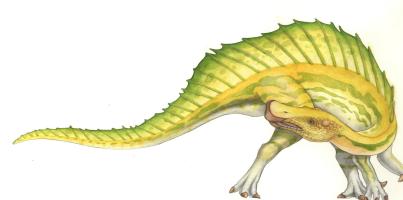
Multithreaded Server Architecture





Benefits

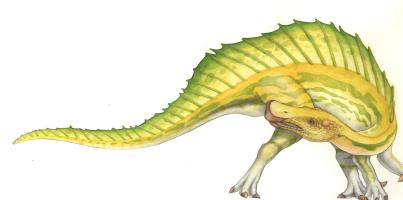
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures





Multicore Programming

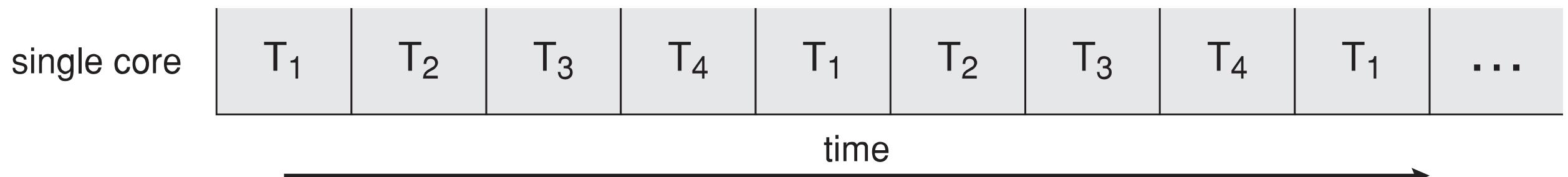
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency
- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
 - CPUs have cores as well as **hardware threads**
 - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core



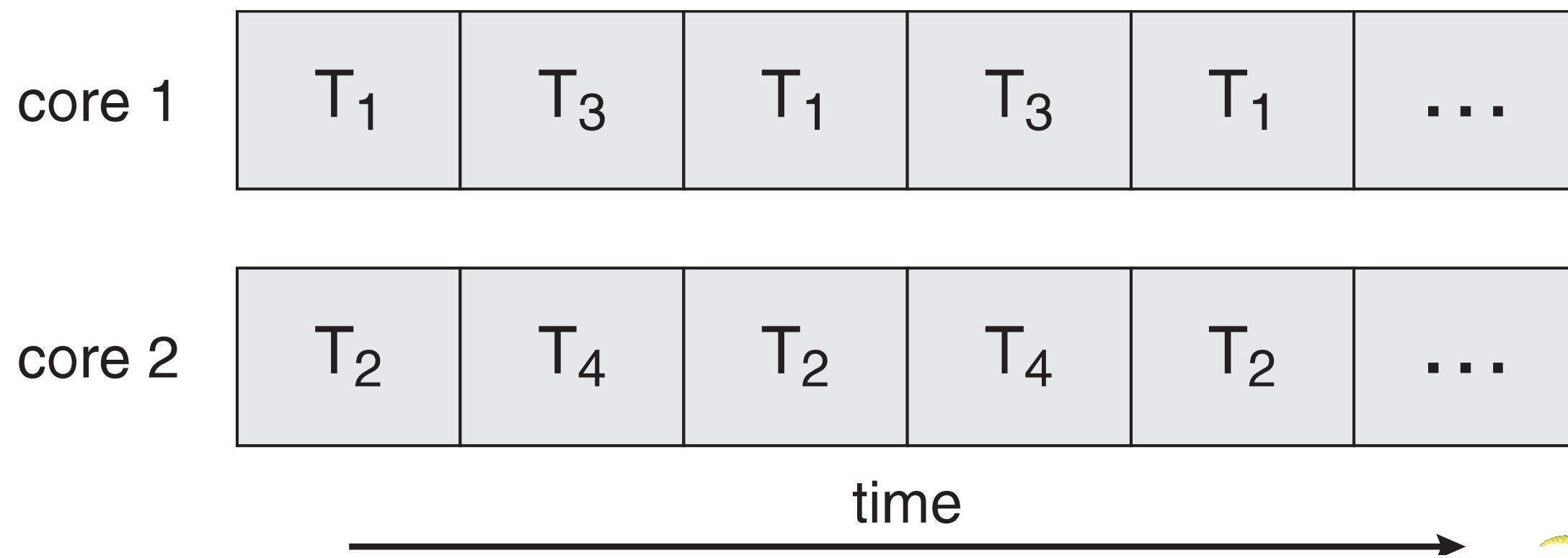


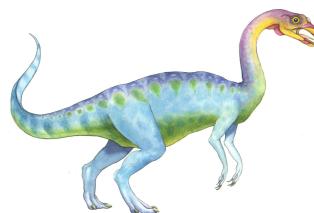
Concurrency vs. Parallelism

- Concurrent execution on single-core system:

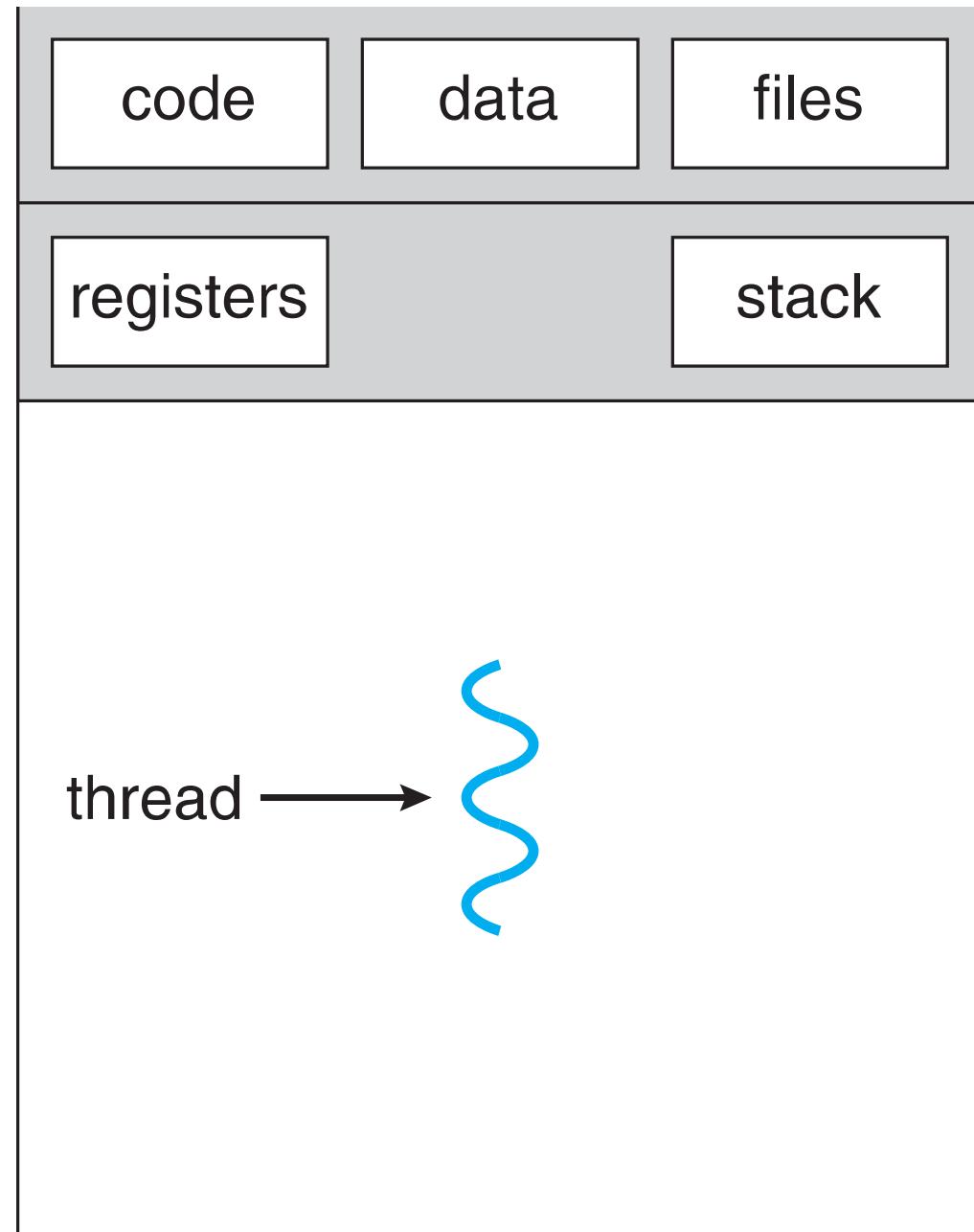


- Parallelism on a multi-core system:

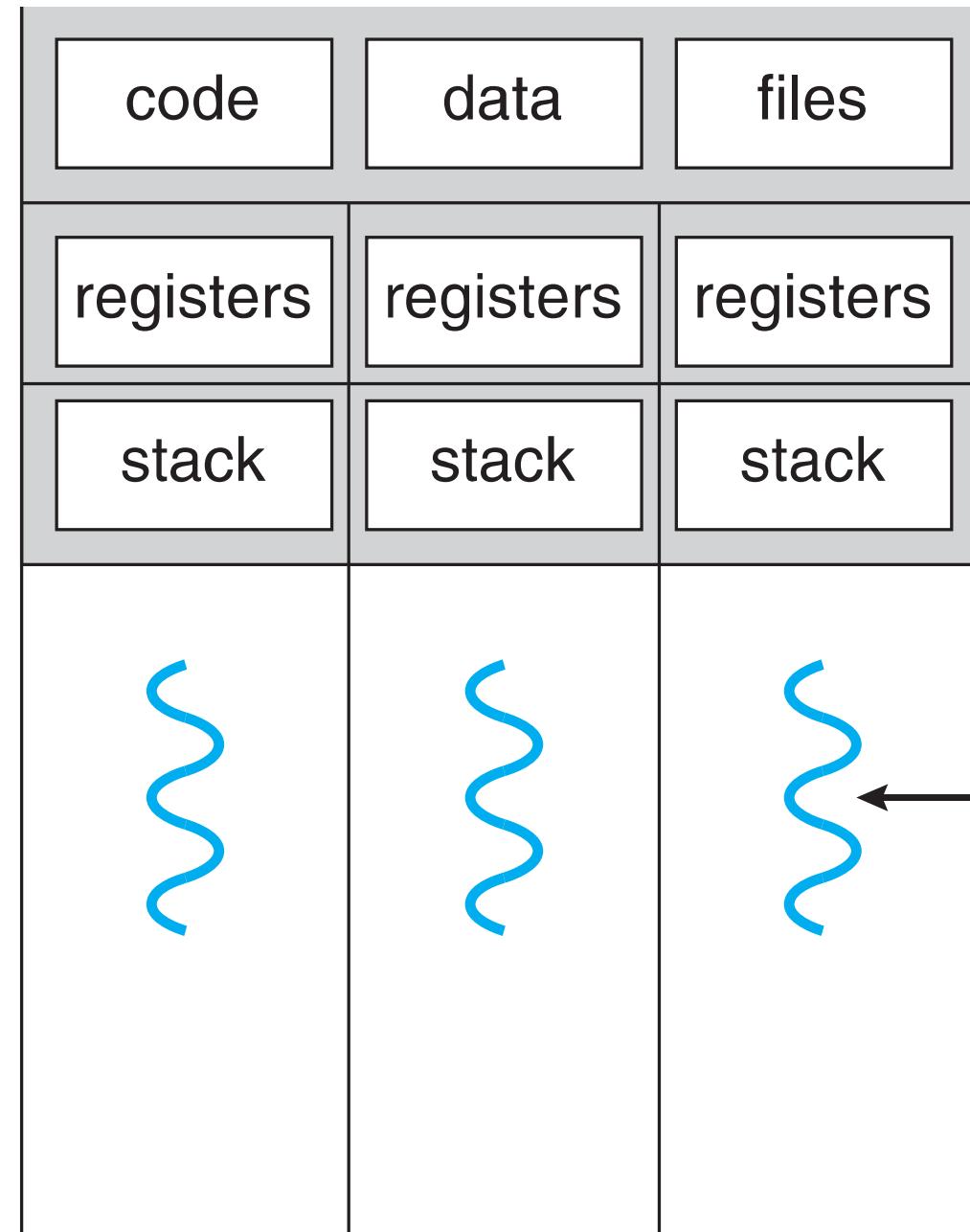




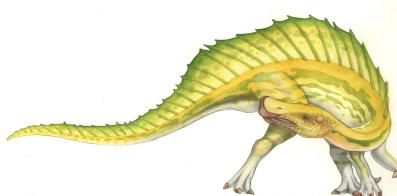
Single and Multithreaded Processes



single-threaded process



multithreaded process





Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

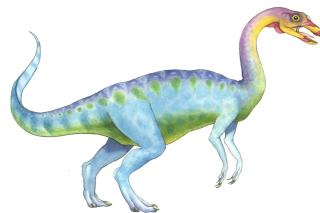
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- I.e. if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches $1 / S$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?





User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Win32 threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

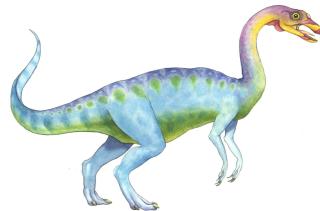




Multithreading Models

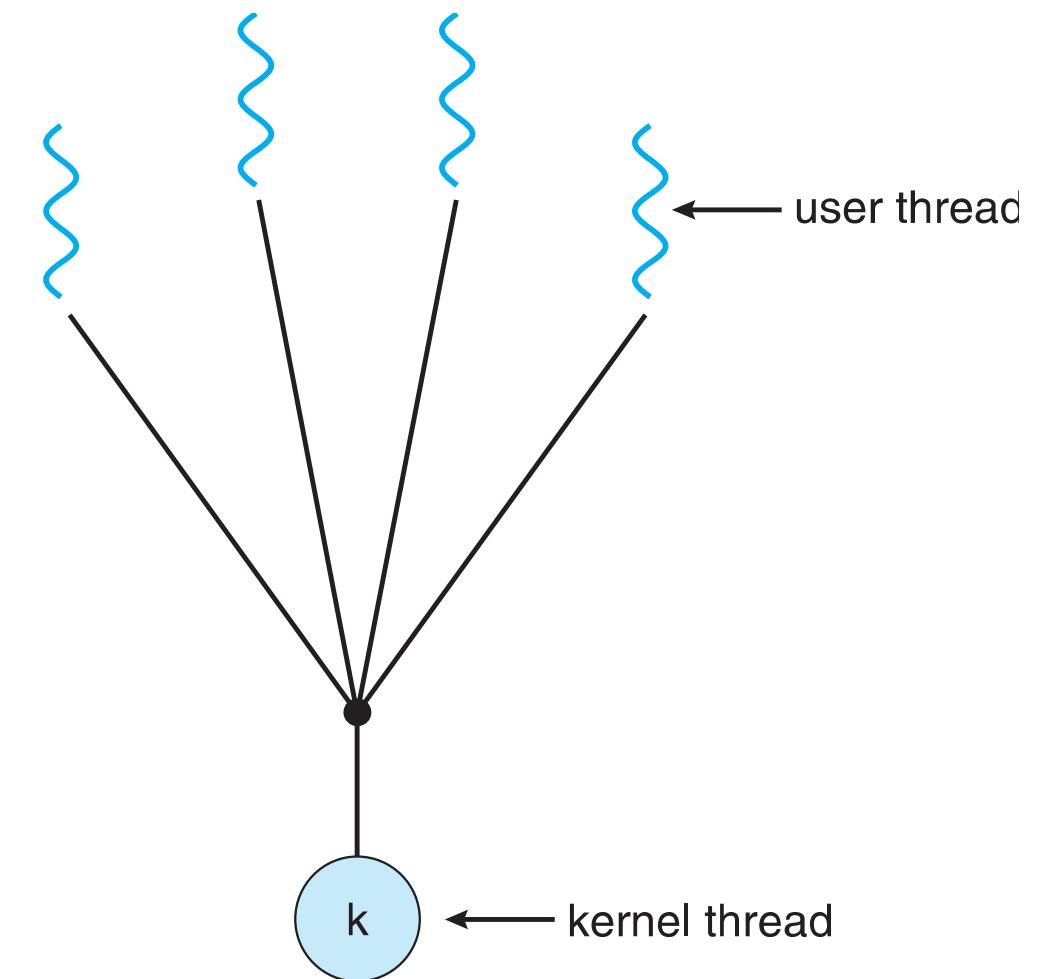
- Many-to-One
- One-to-One
- Many-to-Many

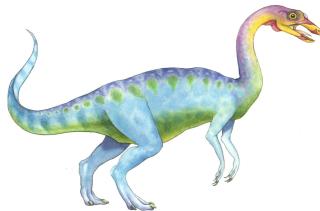




Many-to-One

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

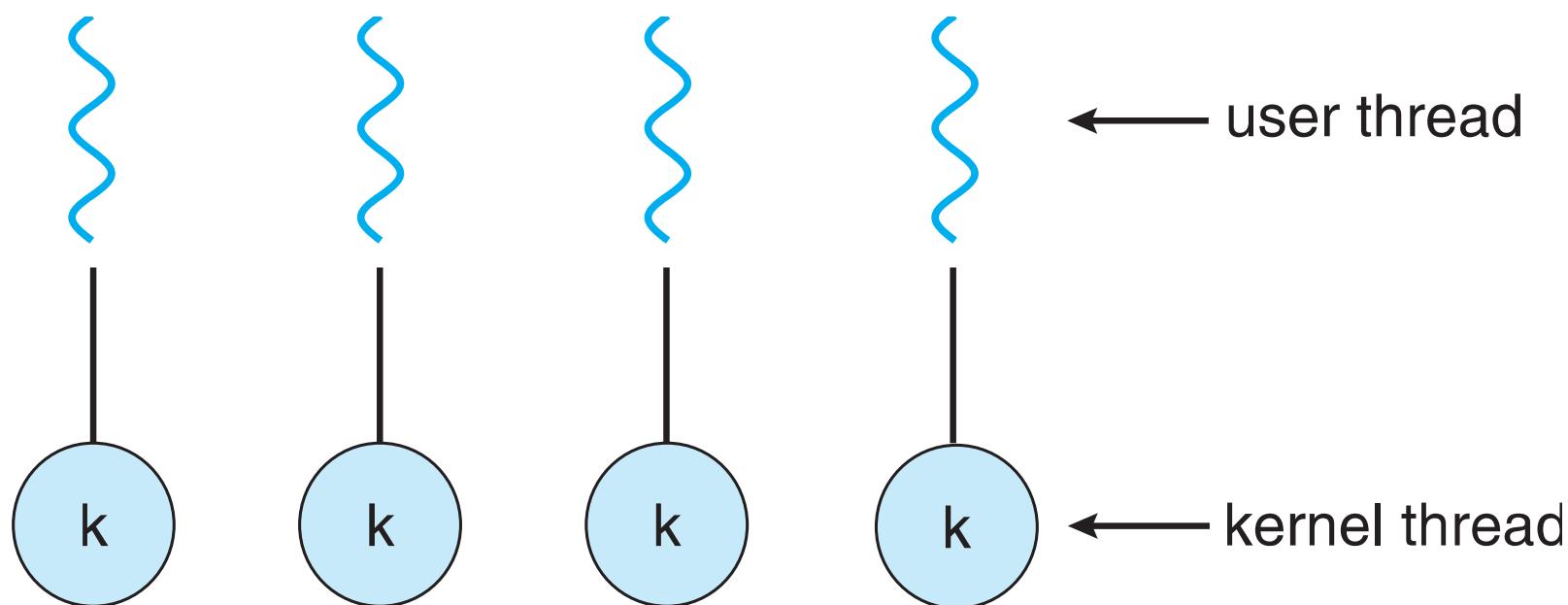


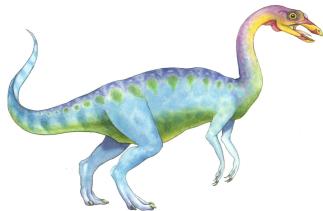


One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

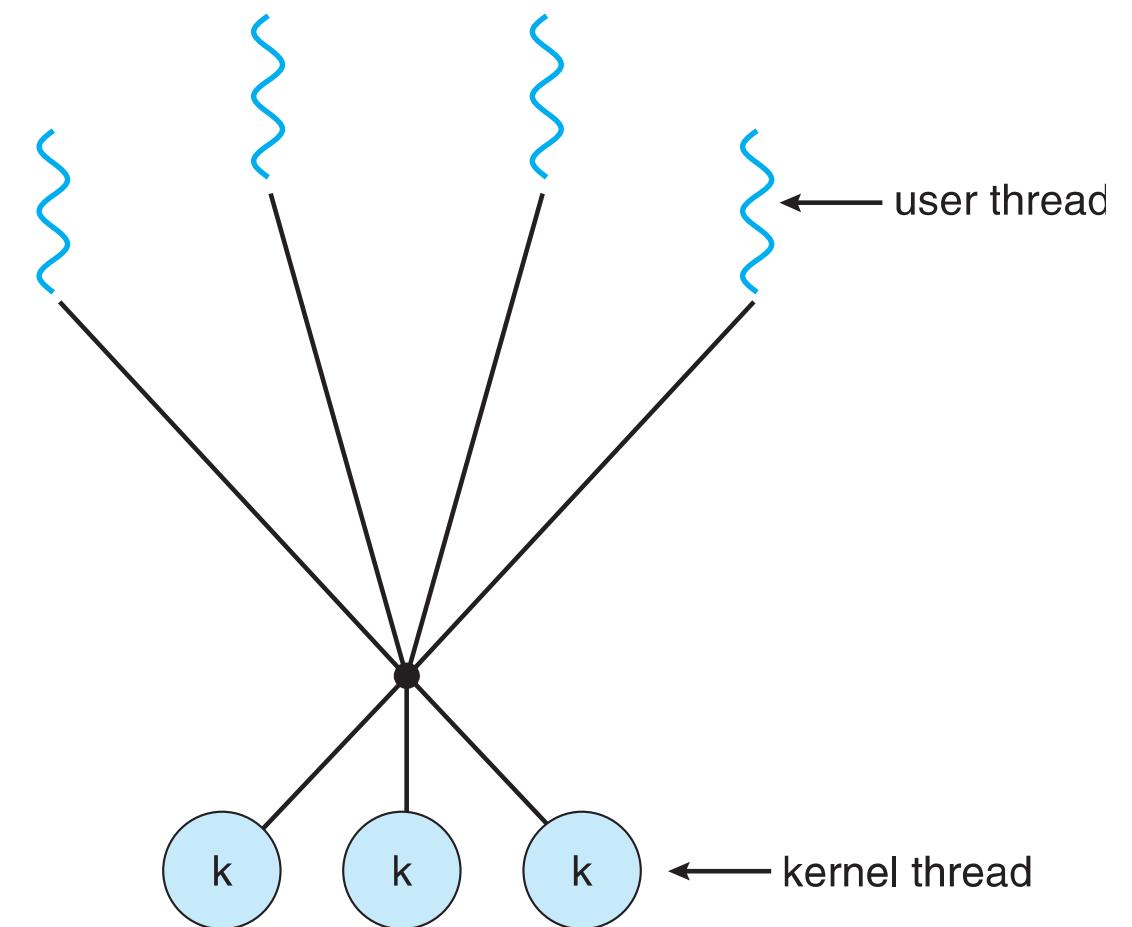
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

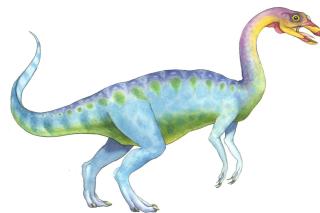




Many-to-Many Model

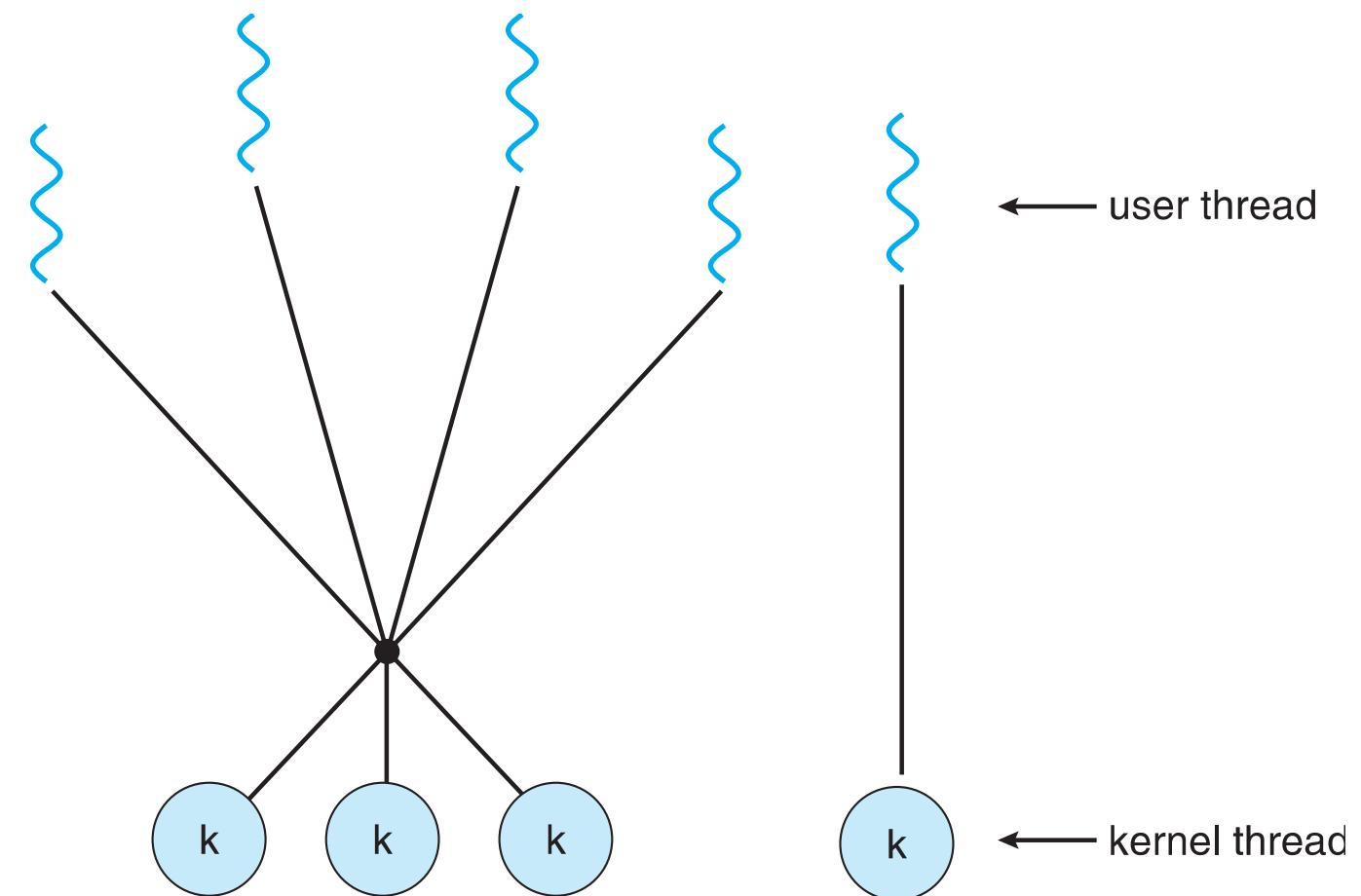
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package





Two-level Model

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





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
- ***Specification*, not *implementation***
- 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)





Pthreads Example

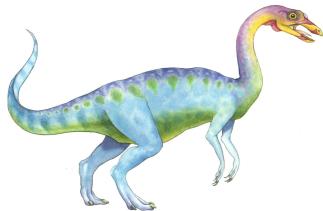
```
#include <pthread.h>
#include <stdio.h>

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

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;
    }
}
```





Pthreads Example (Cont.)

```
/* 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.





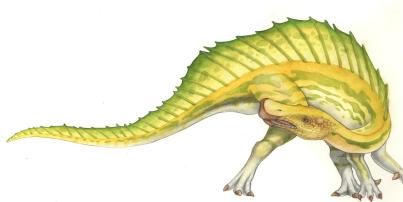
Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.





Win32 API Multithreaded C Program

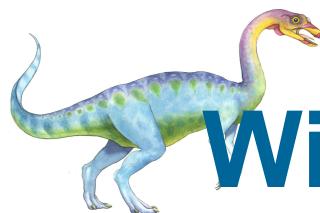
```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* 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;

    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;
    }
}
```





Win32 API Multithreaded C Program (Cont.)

```
/* 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);
}

}
```





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

```
public interface Runnable
{
    public abstract void run();
}
```





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);
    }
}
```





Java Multithreaded Program (Cont.)

```
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 {
                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>"); }
    }
}
```





Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
 - Thread Pools
 - OpenMP
 - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package



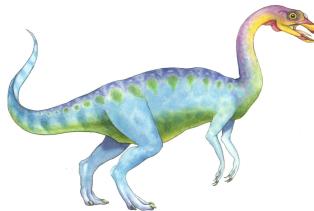


Thread Pools

- 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
 - Separating task to be performed from mechanics of creating task allows different strategies for running task
 - ▶ i.e. Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```





OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

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

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```





Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{}” - ^{ printf("I am a block"); }
- Blocks placed in dispatch queue
 - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
 - serial – blocks removed in FIFO order, queue is per process, called **main queue**
 - ▶ Programmers can create additional serial queues within program
 - concurrent – removed in FIFO order but several may be removed at a time
 - ▶ Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue  
(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

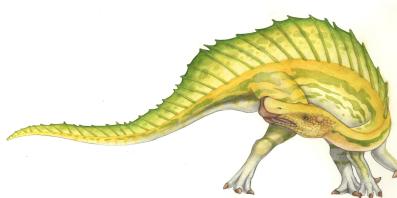
```
dispatch_async(queue, ^{ printf("I am a block."); });
```





Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of fork

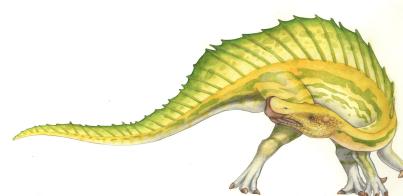
- **Exec()** usually works as normal – replace the running process including all threads





Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- 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 by one of two signal handlers:
 1. default
 2. user-defined
- Every signal has default handler that kernel runs when handling signal
 - User-defined signal handler can override default
 - For single-threaded, signal delivered to process
- Where should a signal be delivered for multi-threaded?
 - 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





Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
.  
.  
  
/* cancel the thread */  
pthread_cancel(tid);
```





Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
 - Cancellation only occurs when thread reaches **cancellation point**
 - I.e. `pthread_cancel()`
 - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals





Thread-Local Storage

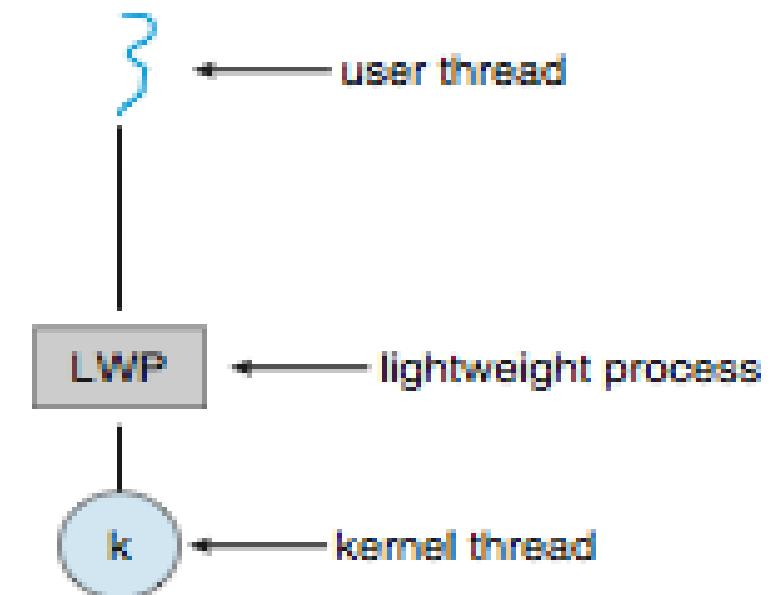
- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to `static` data
 - TLS is unique to each thread





Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP attached to kernel thread
 - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





Operating System Examples

- Windows XP Threads
- Linux Thread

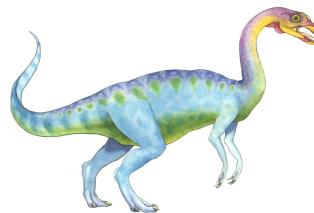




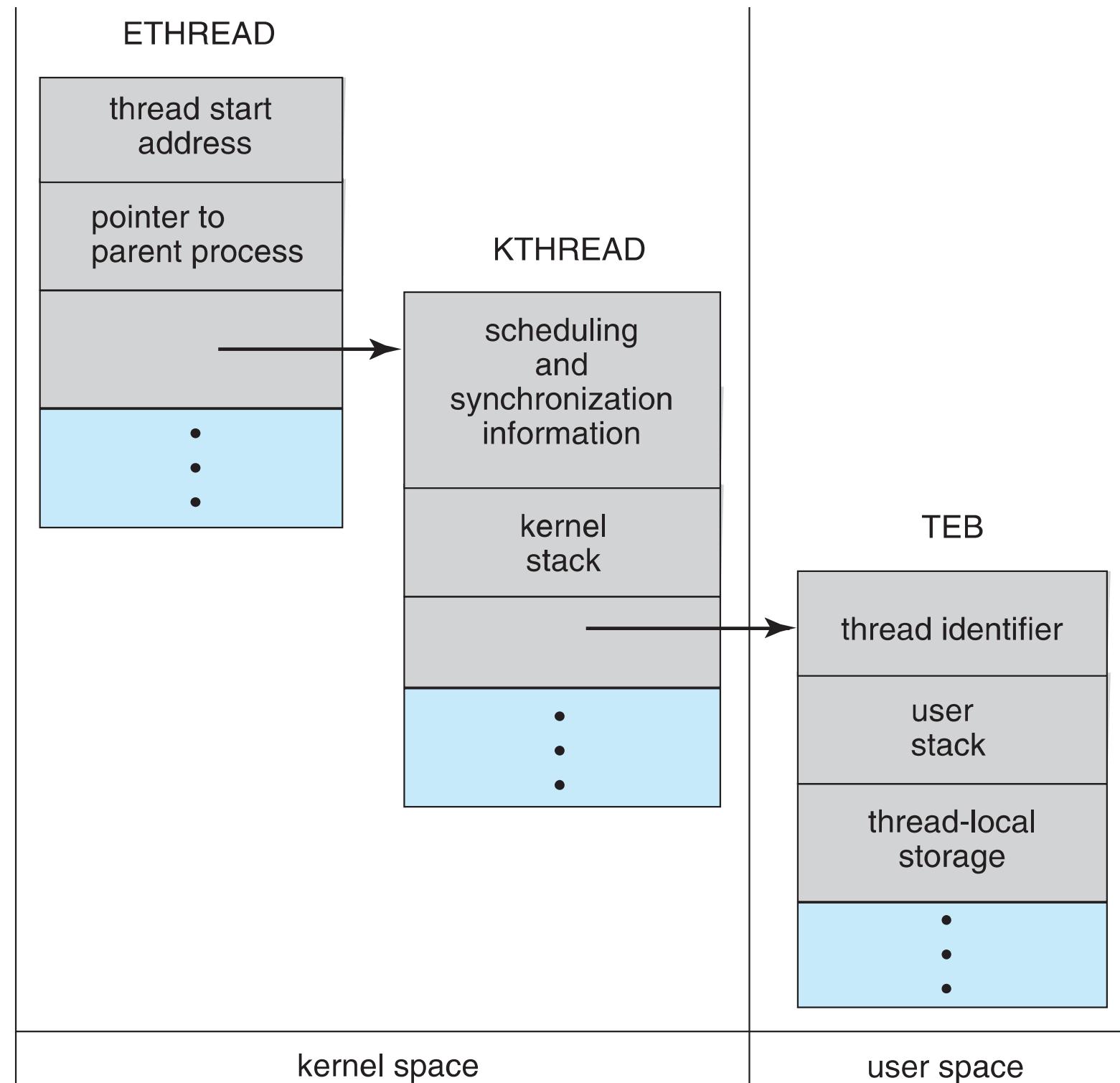
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks for when thread runs in user mode or kernel mode
 - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread
- The primary data structures of a thread include:
 - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
 - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
 - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space





Windows XP Threads Data Structures





Linux Threads

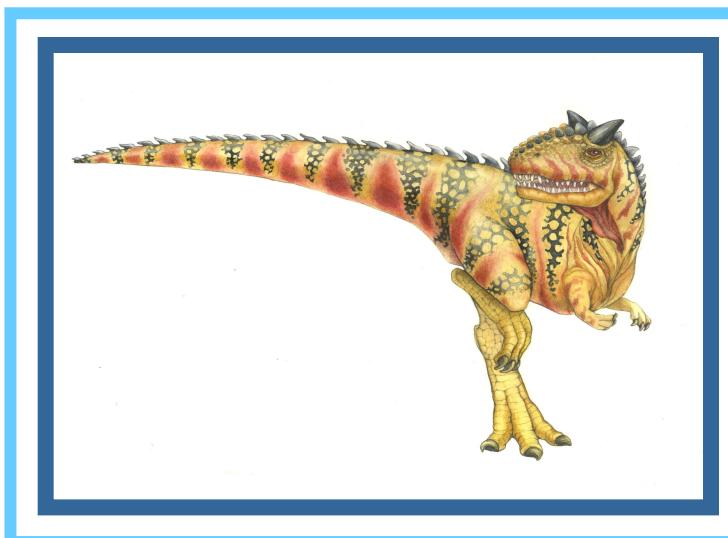
- Linux refers to them as **tasks** rather than **threads**
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
 - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **struct task_struct** points to process data structures (shared or unique)



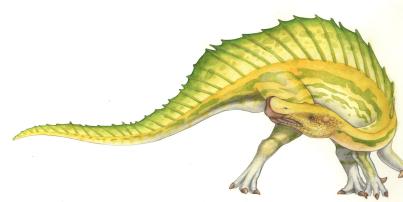
Chapter 5: Process Scheduling





Chapter 5: Process Scheduling

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





Objectives

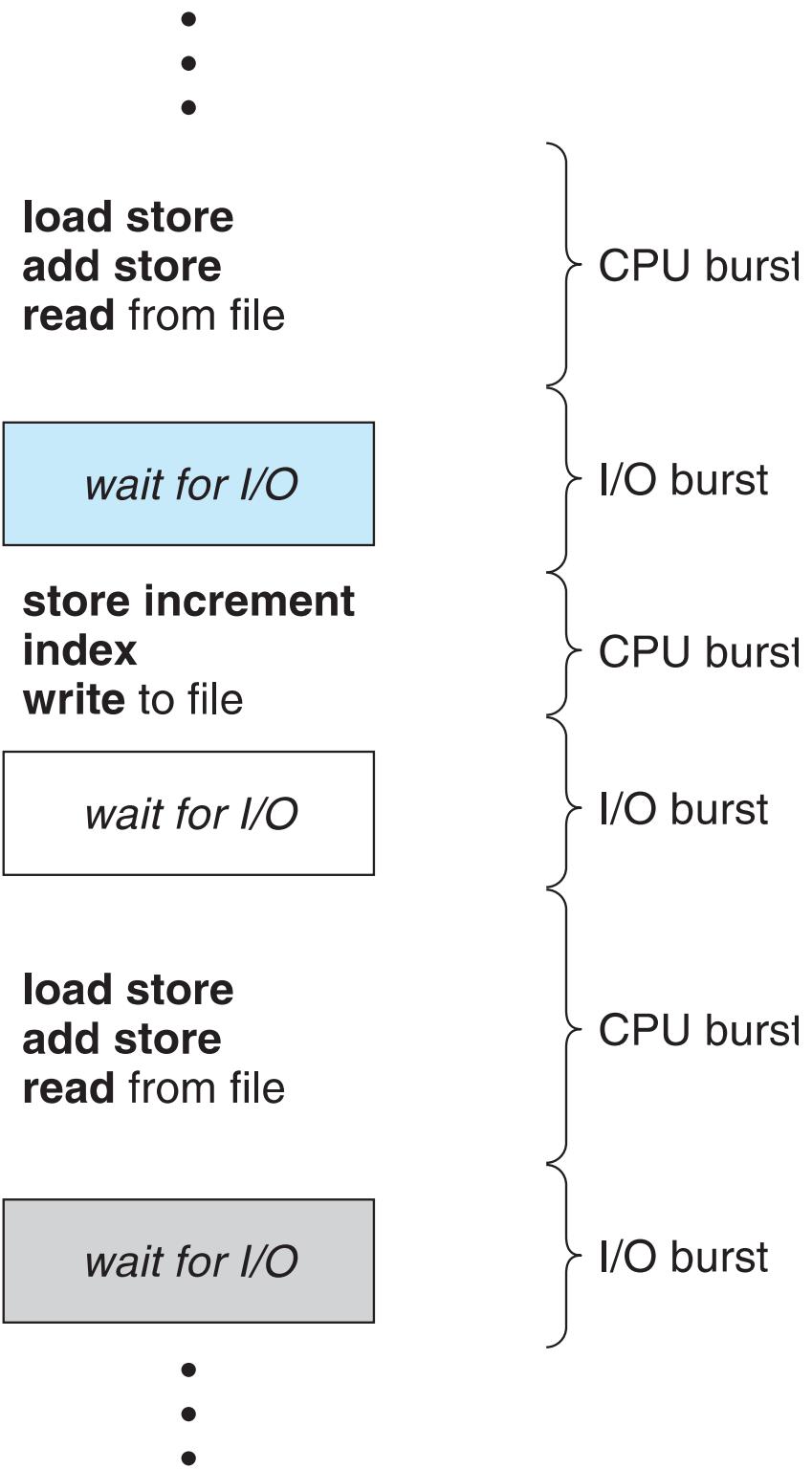
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

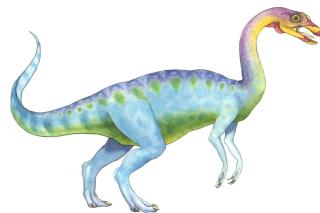




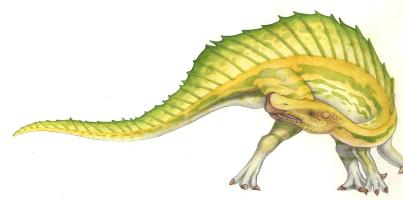
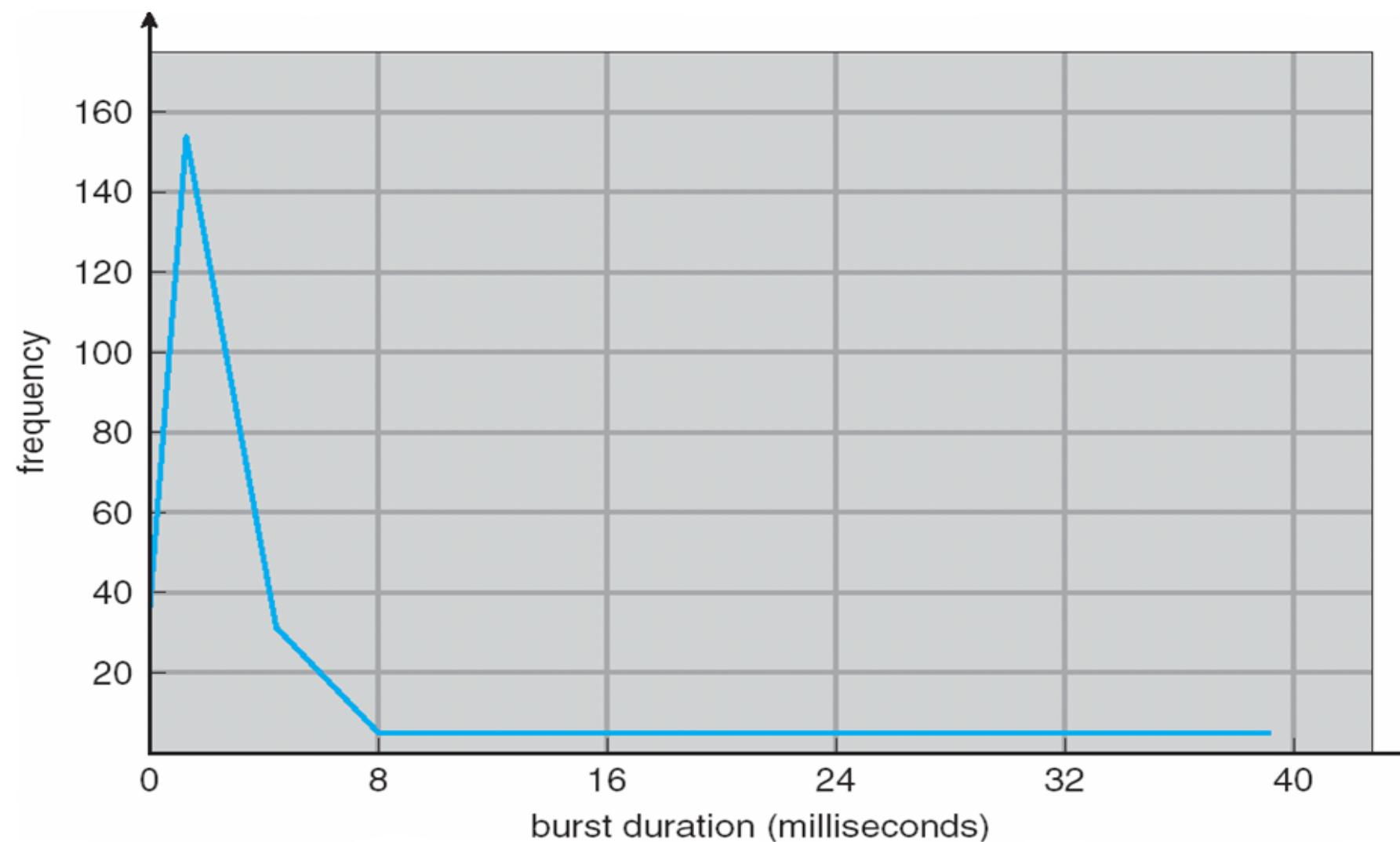
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern





Histogram of CPU-burst Times





CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
 - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
 - Consider access to shared data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities





Dispatcher

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





Scheduling Criteria

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





Scheduling Algorithm Optimization Criteria

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





First-Come, First-Served (FCFS) Scheduling

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

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



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



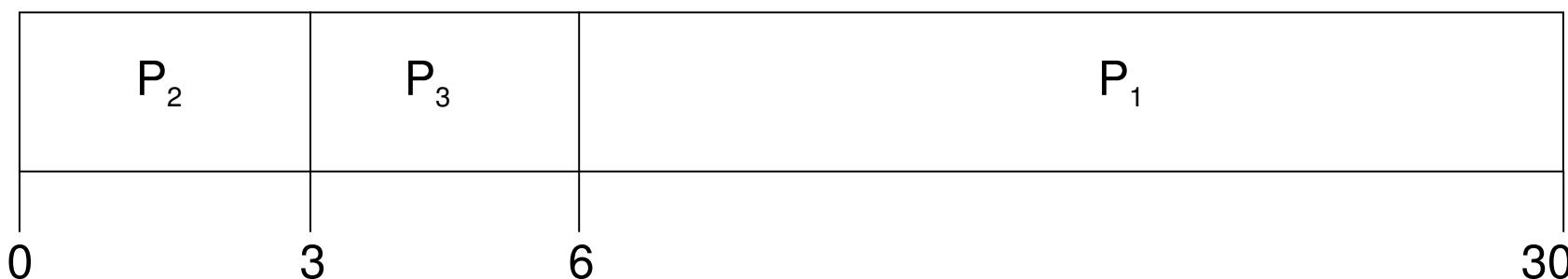


FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes





Shortest-Job-First (SJF) Scheduling

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

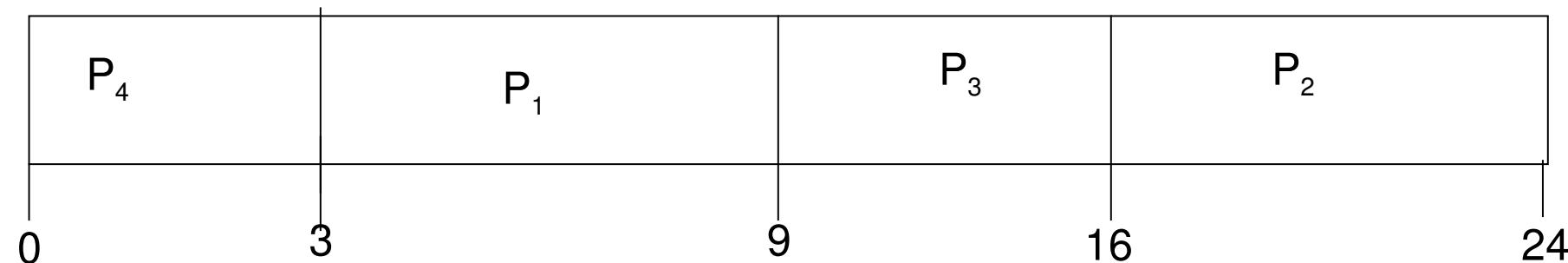




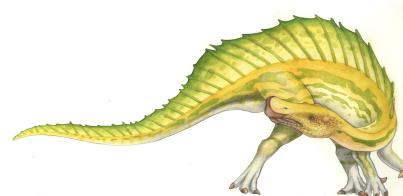
Example of SJF

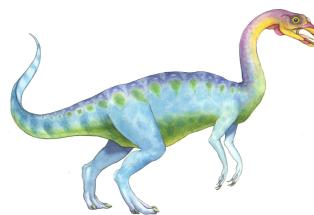
<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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





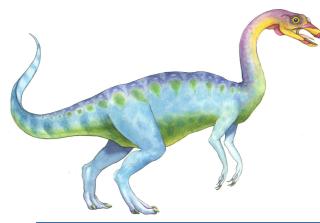
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

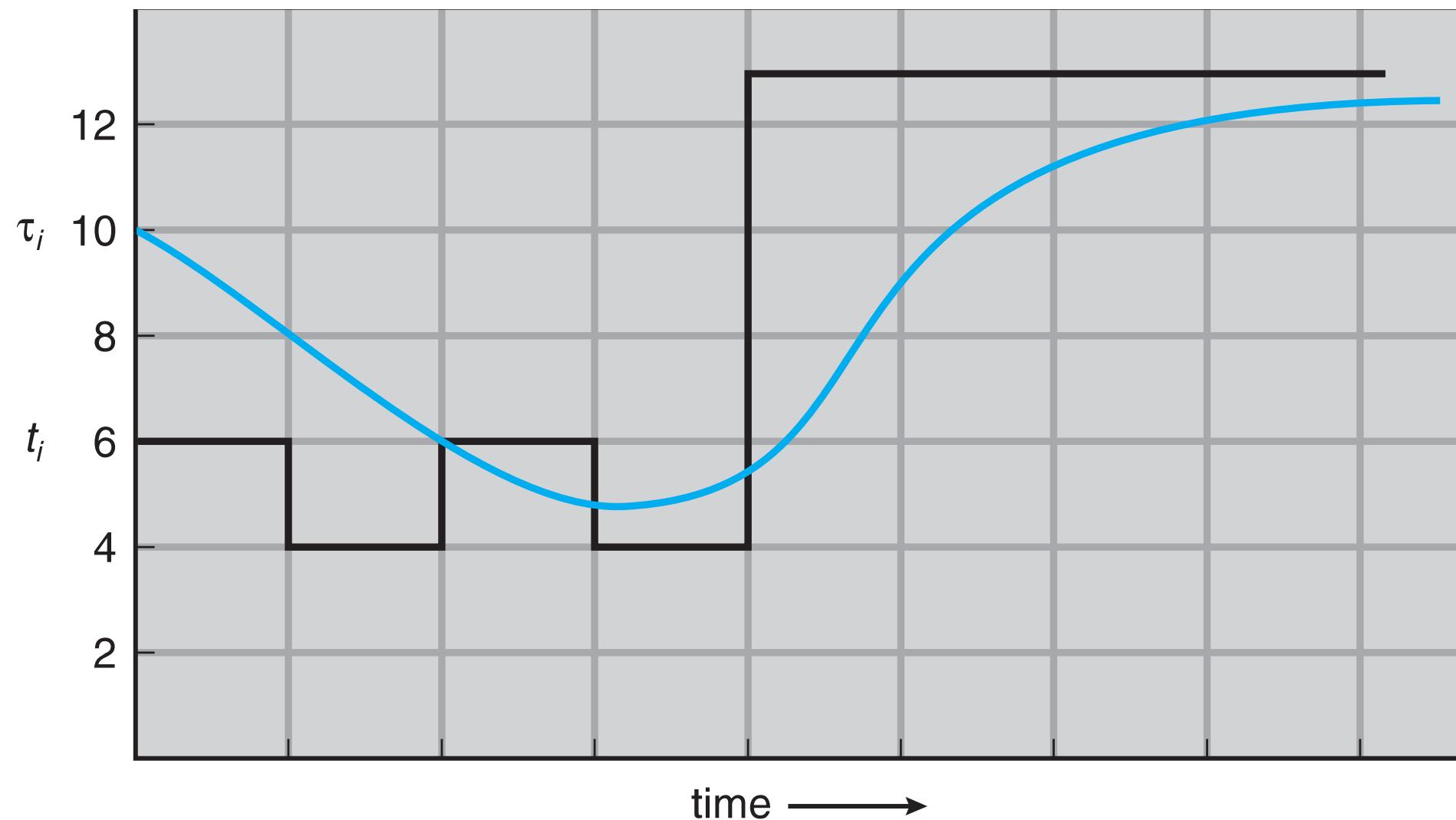
1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

- Commonly, α set to $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**





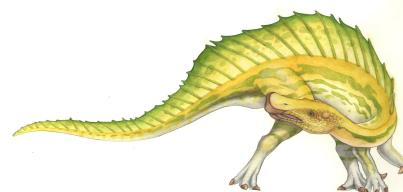
Prediction of the Length of the Next CPU Burst



CPU burst (t_i)

6 4 6 6 4 13 13 13 ...

"guess" (τ_i) 10 8 6 5 5 9 11 12 ...





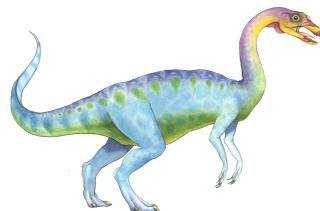
Examples of Exponential Averaging

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

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_n - 1 + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

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



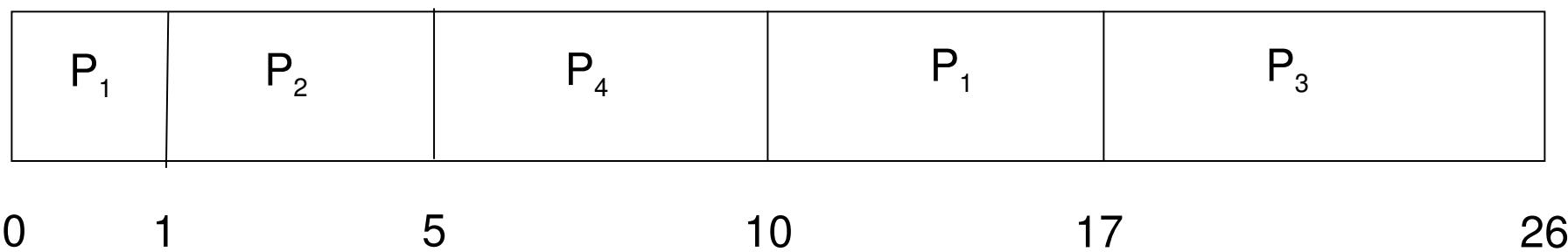


Example of Shortest-remaining-time-first

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

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart*



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





Priority Scheduling

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

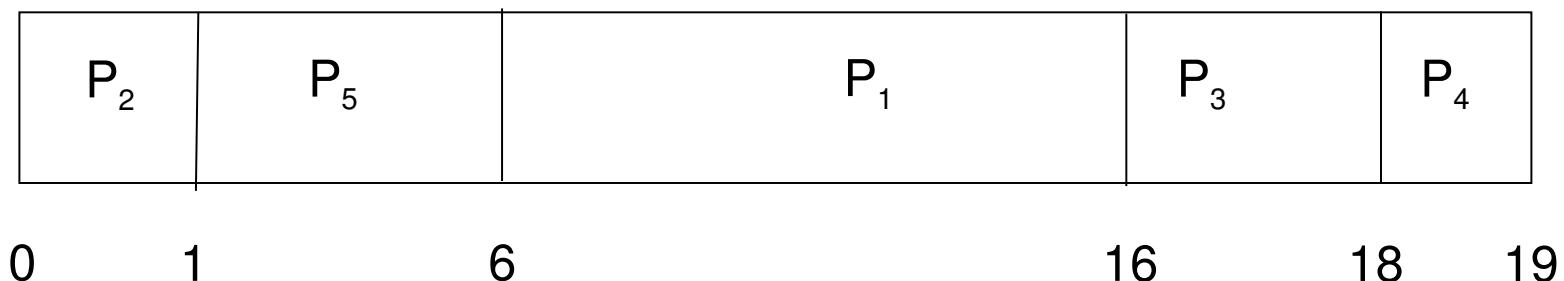




Example of Priority Scheduling

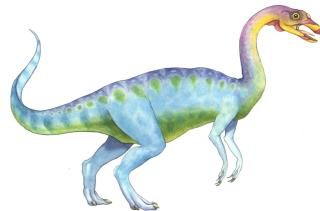
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec





Round Robin (RR)

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

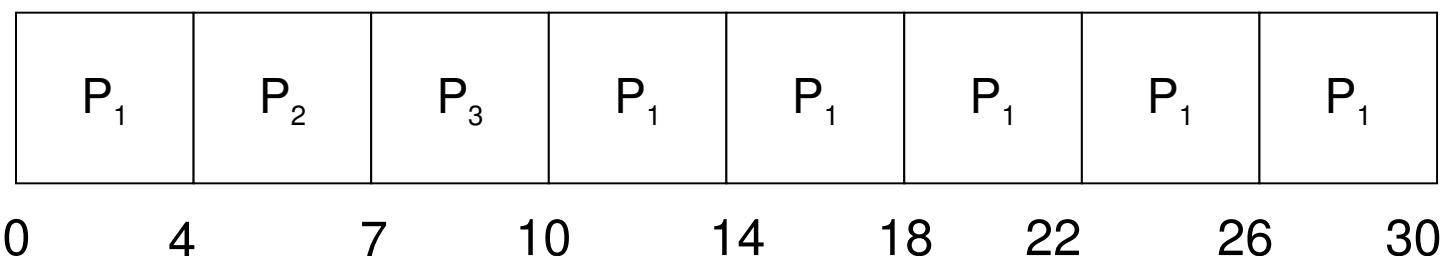




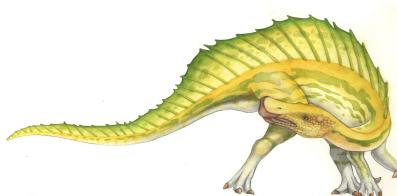
Example of RR with Time Quantum = 4

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

- The Gantt chart is:

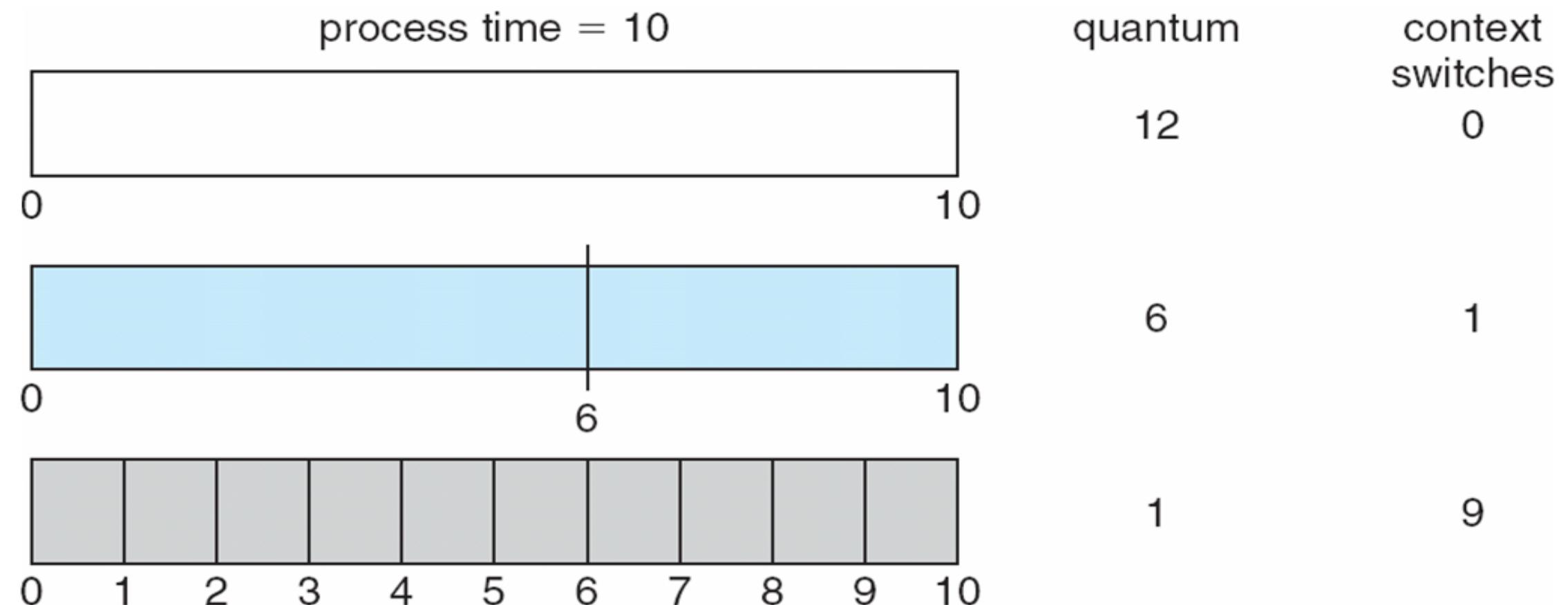


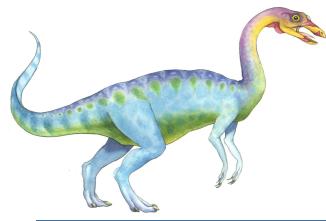
- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec



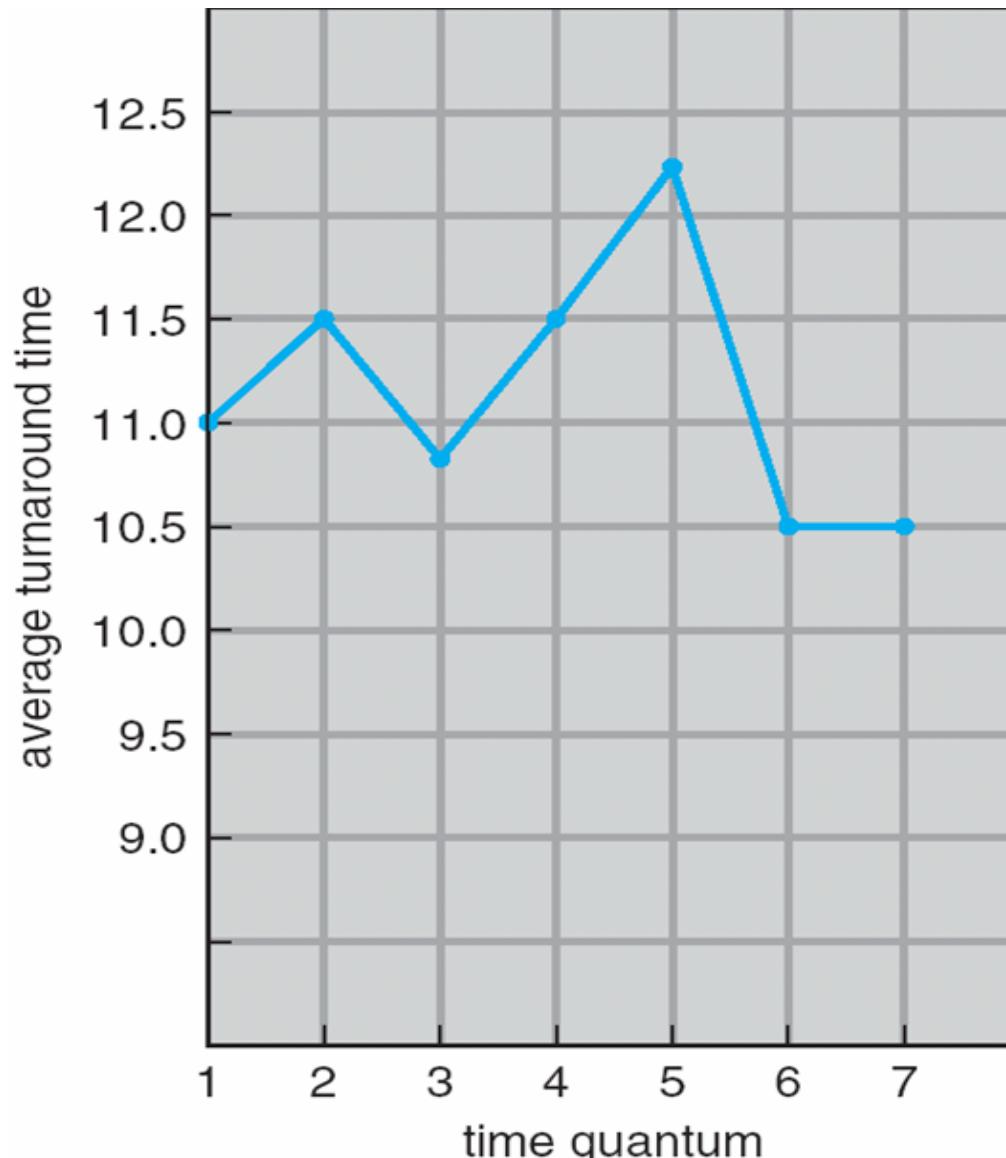


Time Quantum and Context Switch Time





Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

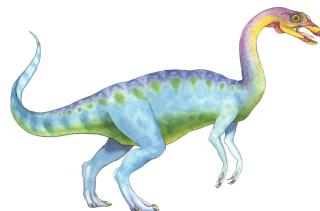




Multilevel Queue

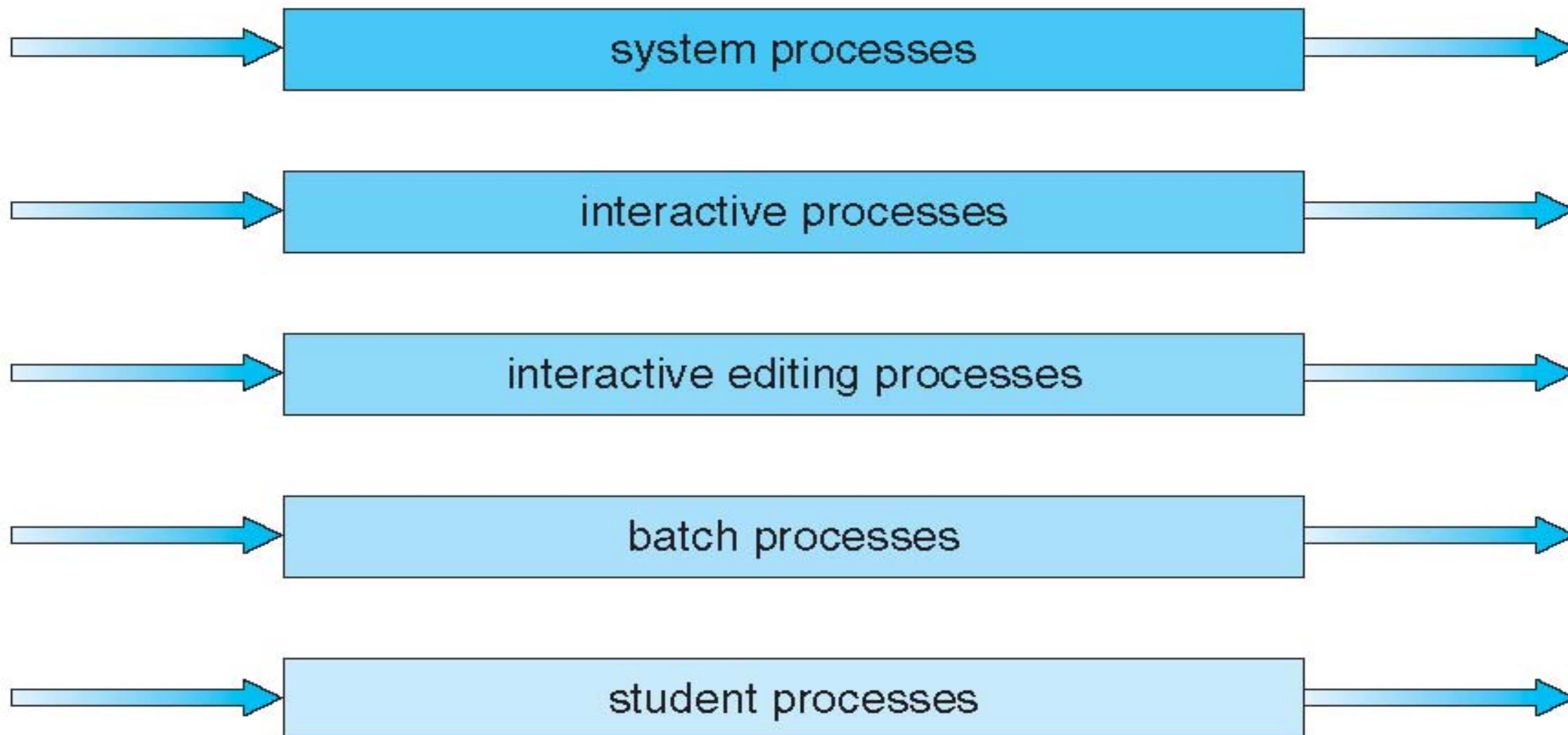
- Ready queue is partitioned into separate queues, eg:
 - **foreground** (interactive)
 - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues:
 - Fixed priority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



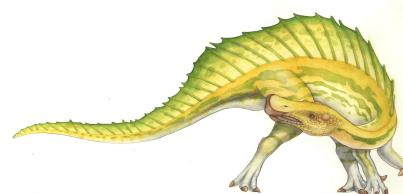


Multilevel Queue Scheduling

highest priority



lowest priority

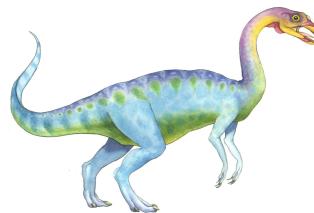




Multilevel Feedback Queue

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





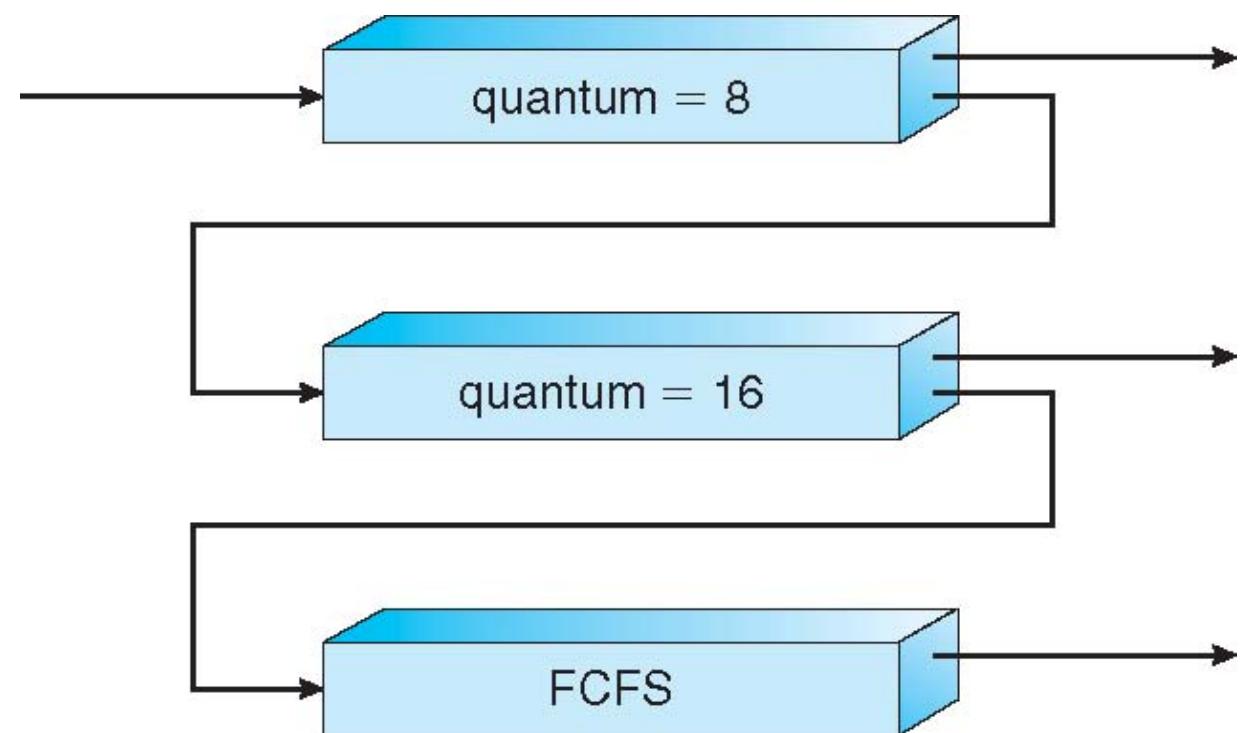
Example of Multilevel Feedback Queue

- Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

- Scheduling

- A new job enters queue Q_0 which is served FCFS
 - ▶ When it gains CPU, job receives 8 milliseconds
 - ▶ If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
 - ▶ If it still does not complete, it is preempted and moved to queue Q_2





Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
 - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system





Pthread Scheduling

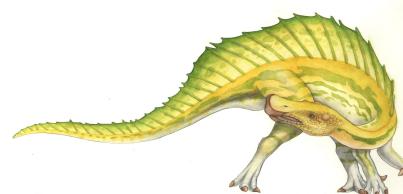
- API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM





Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```





Pthread Scheduling API

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)

    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)

    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */

void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);

}
```

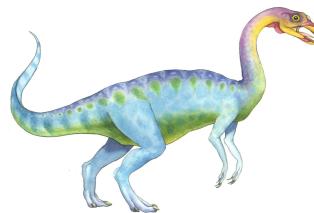




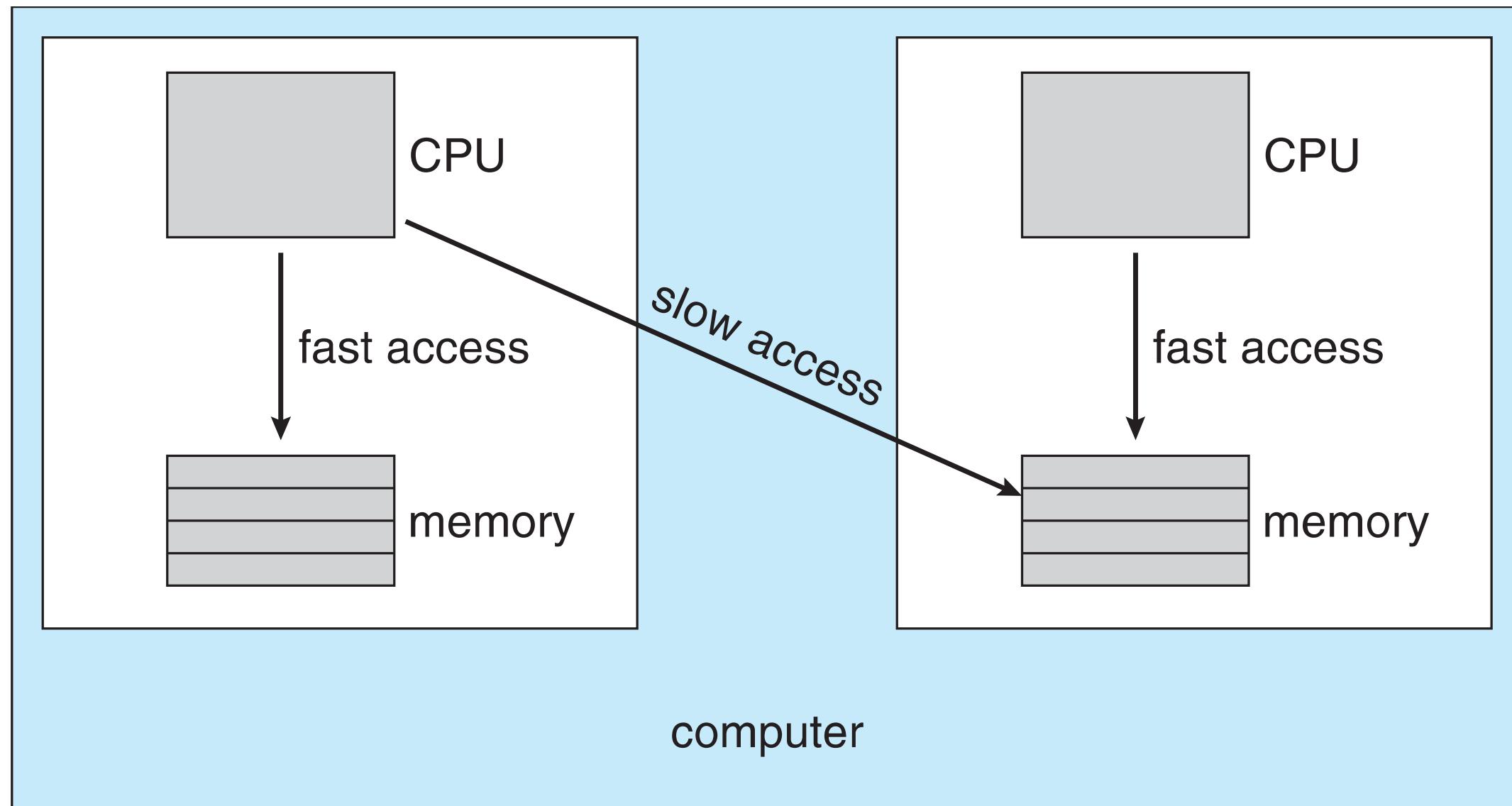
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
 - **soft affinity**
 - **hard affinity**
 - Variations including **processor sets**



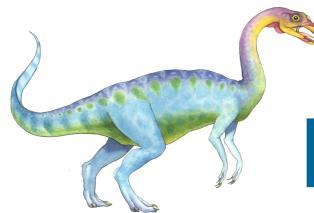


NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity





Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





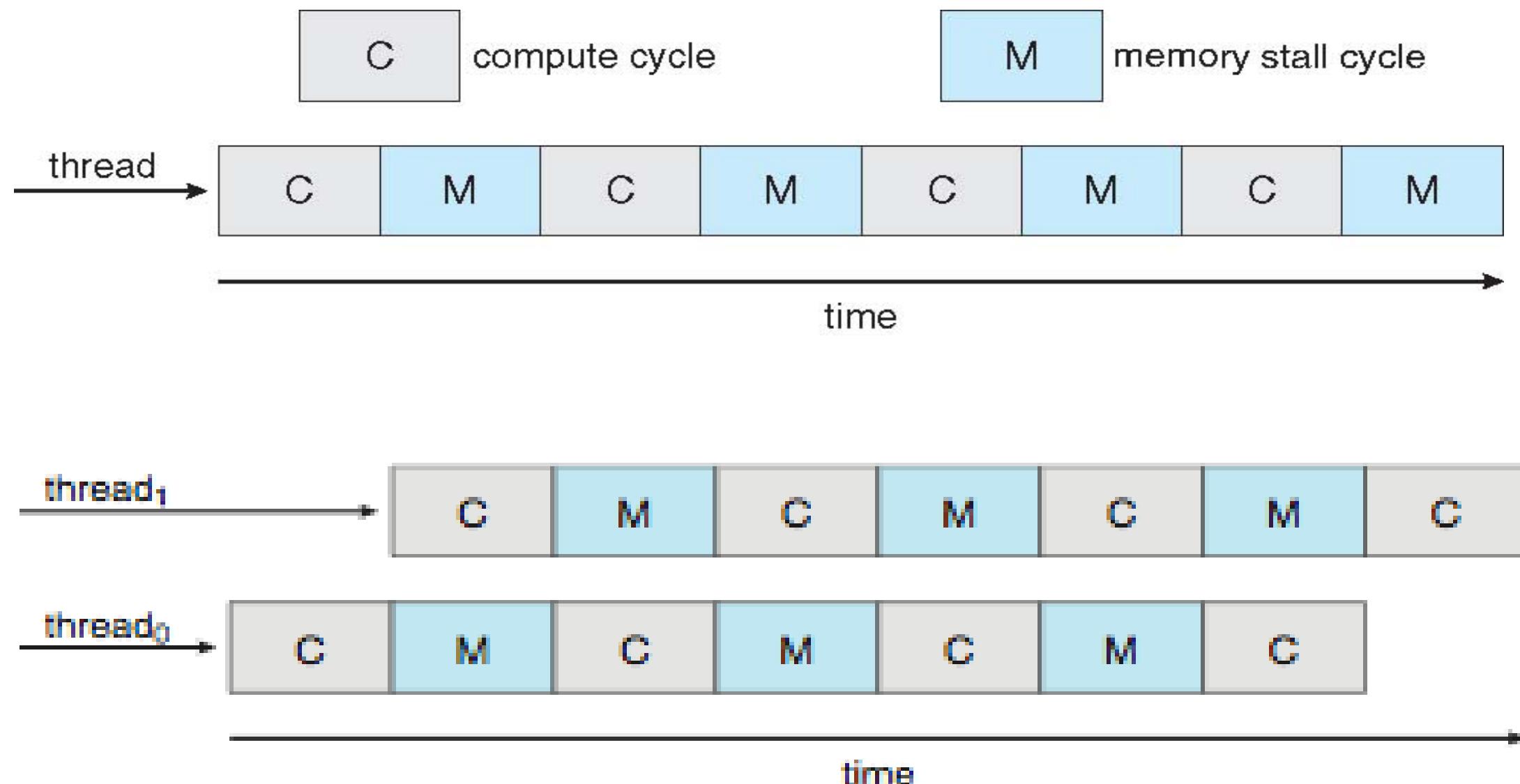
Multicore Processors

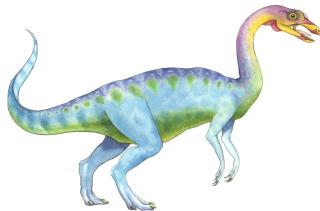
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens





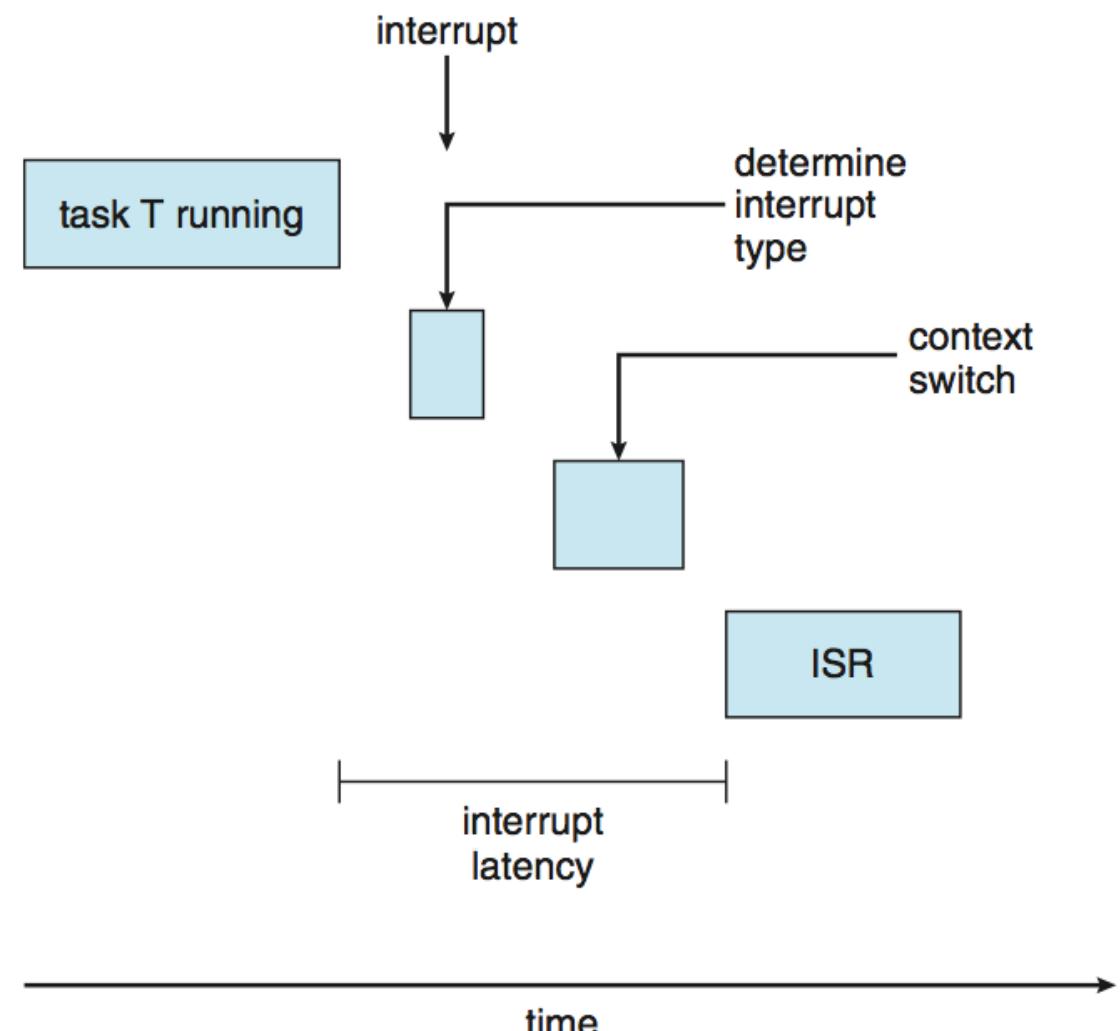
Multithreaded Multicore System





Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
 1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
 2. Dispatch latency – time for scheduler to take current process off CPU and switch to another

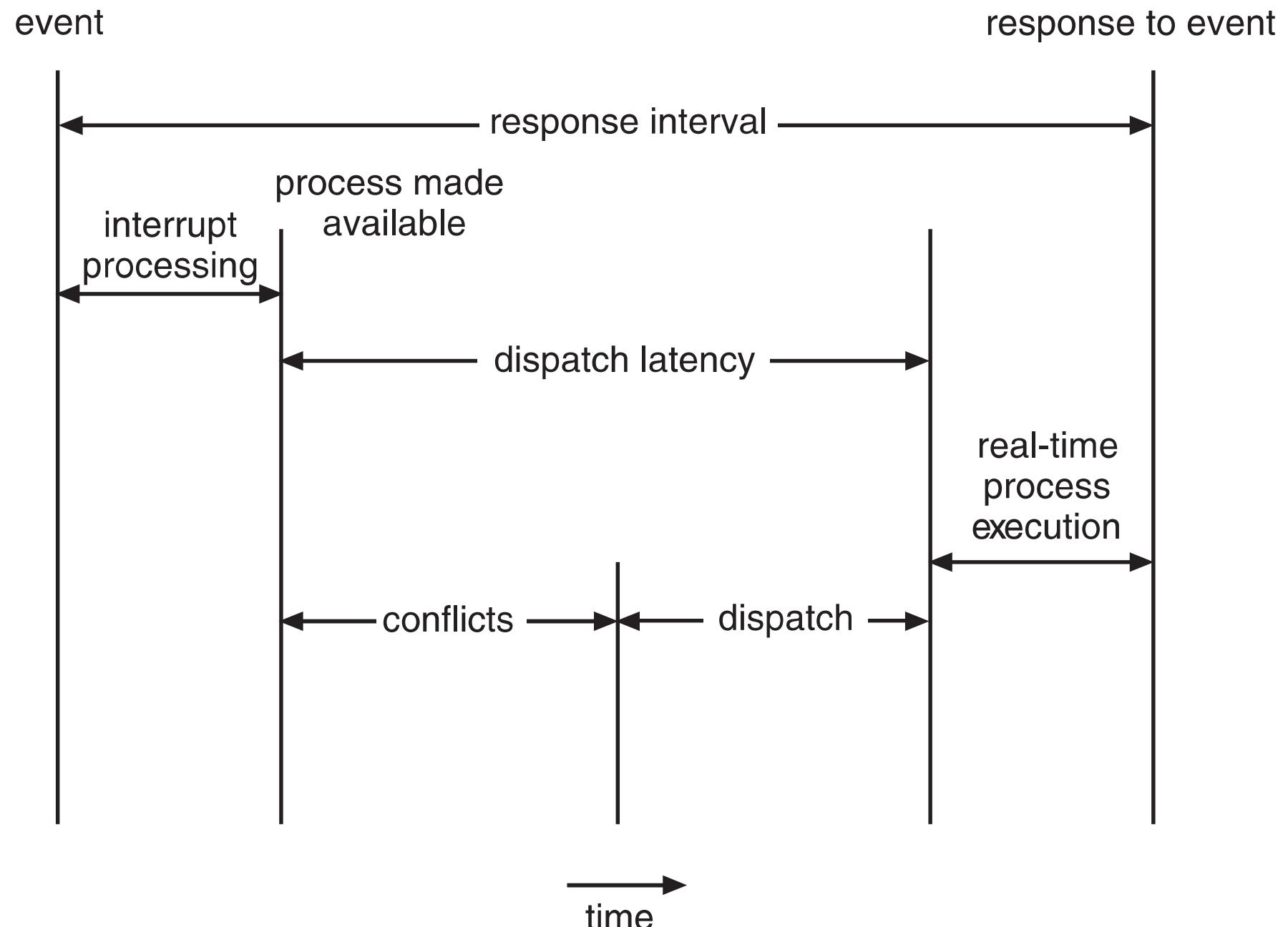




Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:

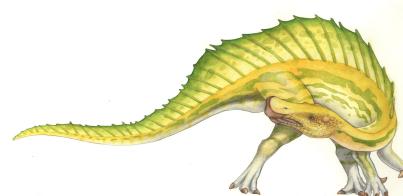
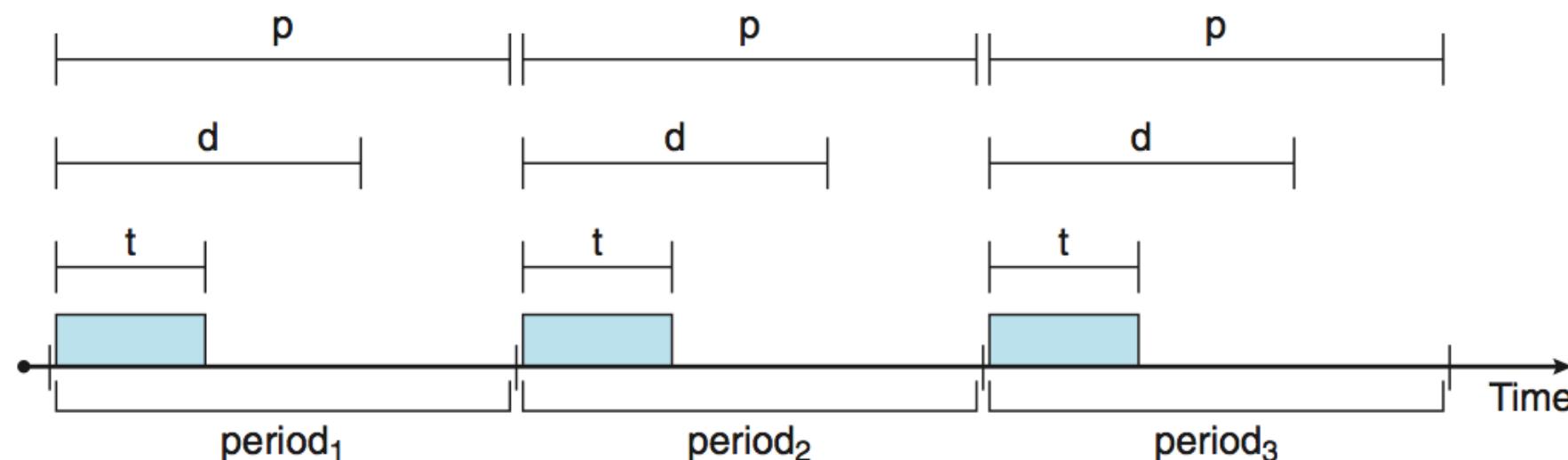
1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes





Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
 - Has processing time t , deadline d , period p
 - $0 \leq t \leq d \leq p$
 - **Rate** of periodic task is $1/p$

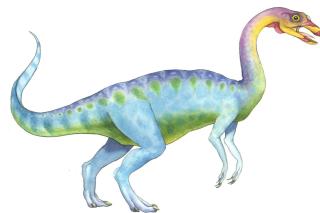




Virtualization and Scheduling

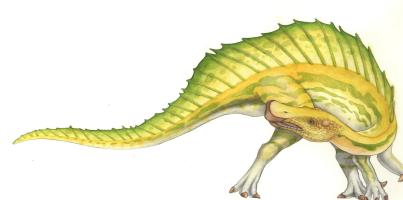
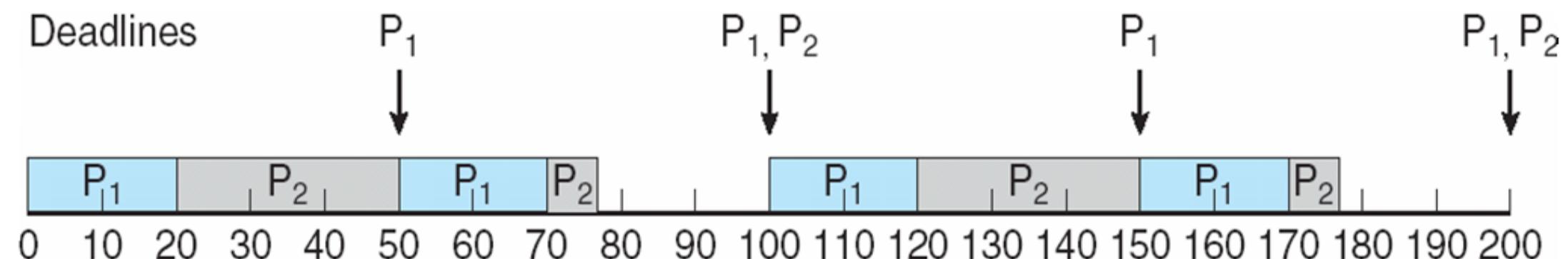
- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
 - Not knowing it doesn't own the CPUs
 - Can result in poor response time
 - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

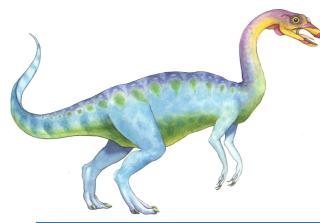




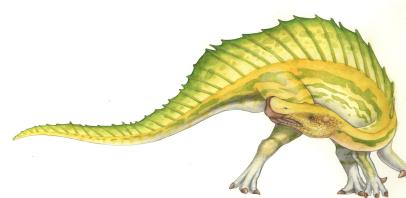
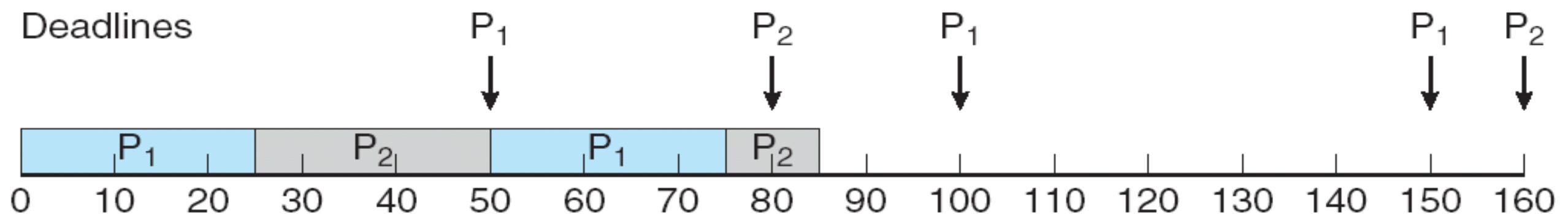
Rate Montonic Scheduling

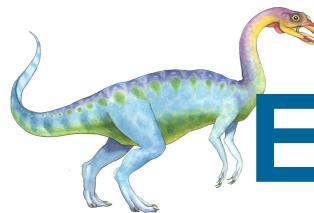
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2 .





Missed Deadlines with Rate Monotonic Scheduling



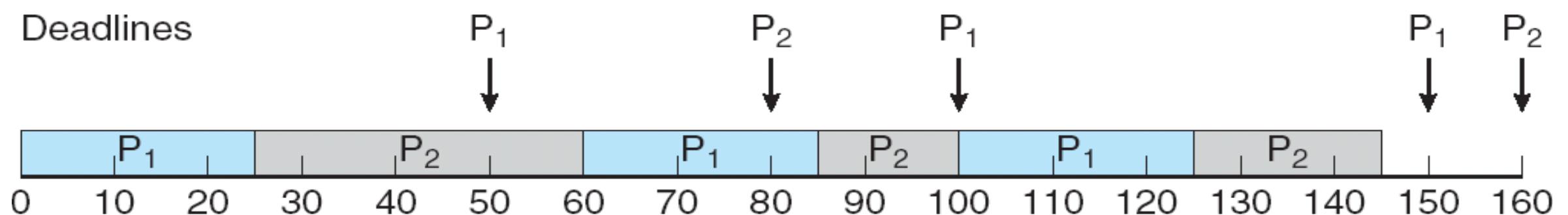


Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;

the later the deadline, the lower the priority





Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive N / T of the total processor time





POSIX Real-Time Scheduling

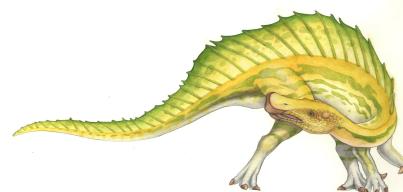
- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
 1. `SCHED_FIFO` - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
 2. `SCHED_RR` - similar to `SCHED_FIFO` except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
 1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
 2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`





POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED FIFO\n");
    }
}
```





POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```





Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling





Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
 - Preemptive, priority based
 - Two priority ranges: time-sharing and real-time
 - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
 - Map into global priority with numerically lower values indicating higher priority
 - Higher priority gets larger q
 - Task run-able as long as time left in time slice (**active**)
 - If no time left (**expired**), not run-able until all other tasks use their slices
 - All run-able tasks tracked in per-CPU **runqueue** data structure
 - ▶ Two priority arrays (active, expired)
 - ▶ Tasks indexed by priority
 - ▶ When no more active, arrays are exchanged
 - Worked well, but poor response times for interactive processes

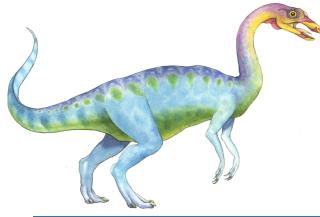




Linux Scheduling in Version 2.6.23 +

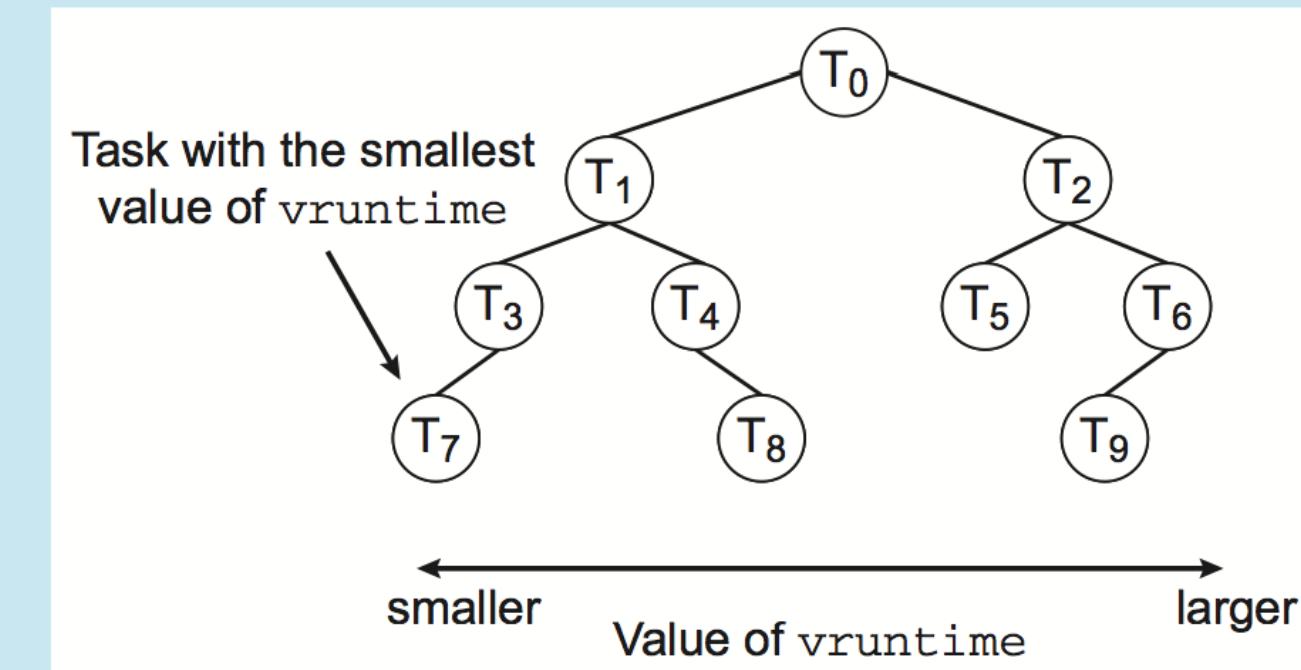
- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
 - Each has specific priority
 - Scheduler picks highest priority task in highest scheduling class
 - Rather than quantum based on fixed time allotments, based on proportion of CPU time
 - 2 scheduling classes included, others can be added
 1. default
 2. real-time
- Quantum calculated based on **nice value** from -20 to +19
 - Lower value is higher priority
 - Calculates **target latency** – interval of time during which task should run at least once
 - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
 - Associated with decay factor based on priority of task – lower priority is higher decay rate
 - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time



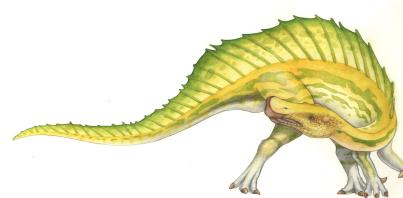


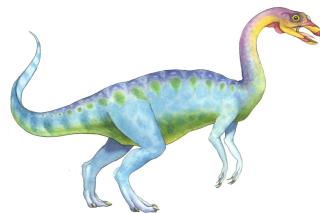
CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



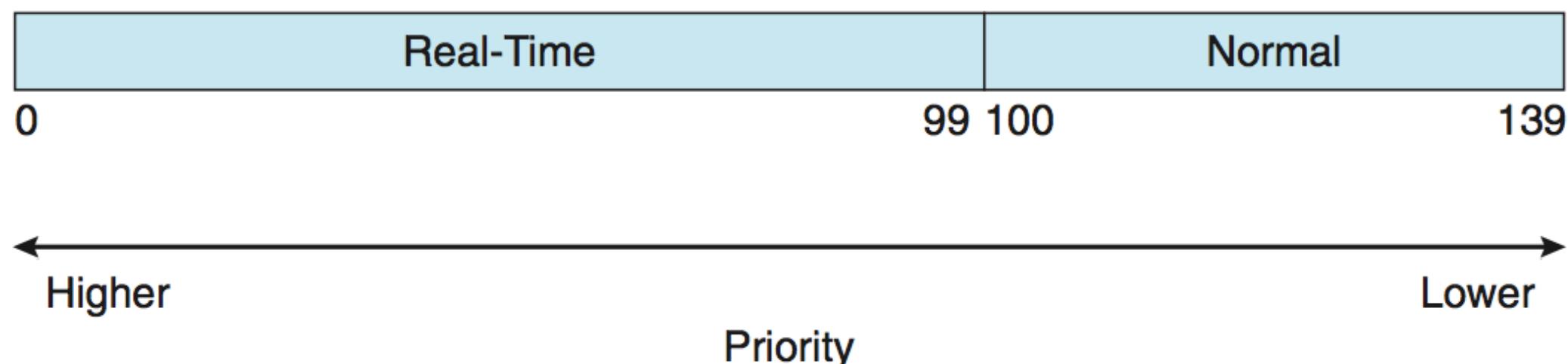
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.





Linux Scheduling (Cont.)

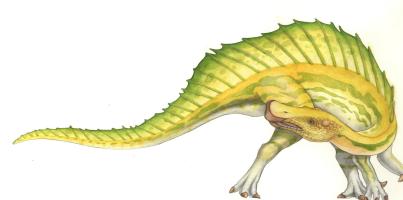
- Real-time scheduling according to POSIX.1b
 - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139





Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

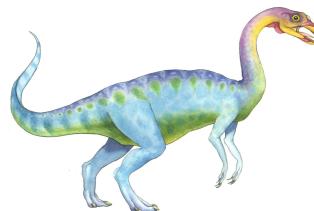




Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
 - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
 - All are variable except REALTIME
- A thread within a given priority class has a relative priority
 - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
 - Applications create and manage threads independent of kernel
 - For large number of threads, much more efficient
 - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

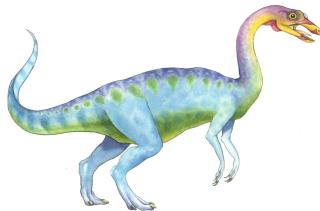




Windows Priorities

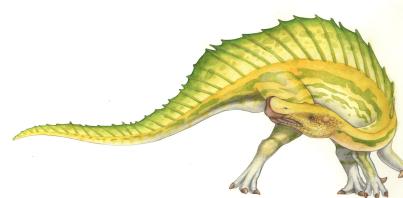
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1





Solaris

- Priority-based scheduling
- Six classes available
 - Time sharing (default) (TS)
 - Interactive (IA)
 - Real time (RT)
 - System (SYS)
 - Fair Share (FSS)
 - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
 - Loadable table configurable by sysadmin

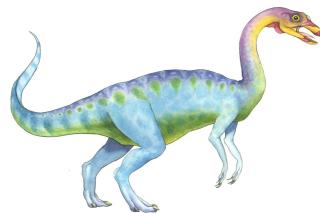




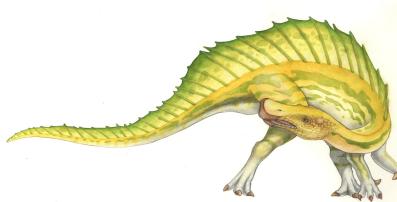
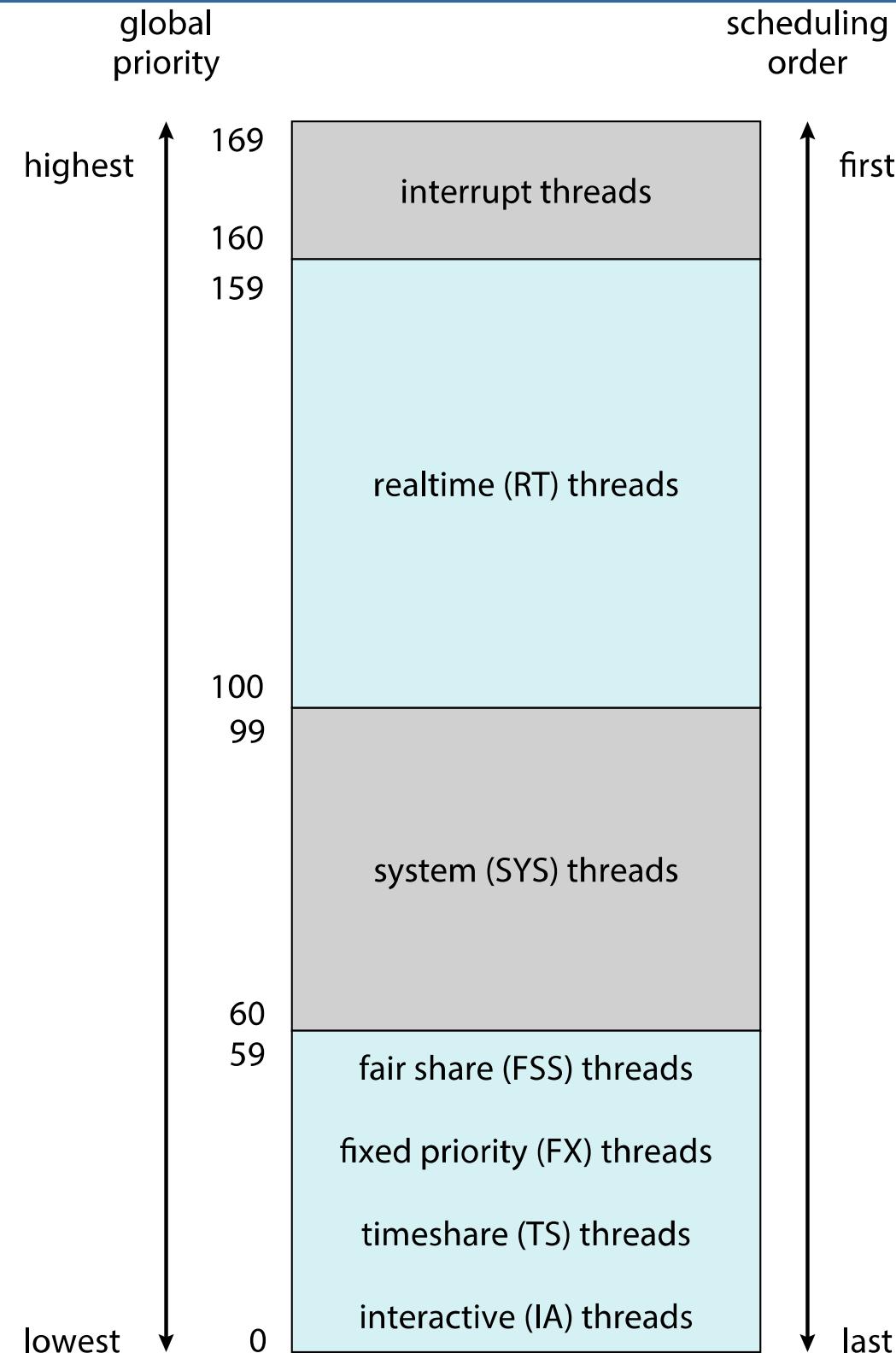
Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59





Solaris Scheduling





Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
 - Thread with highest priority runs next
 - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
 - Multiple threads at same priority selected via RR





Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
 - Type of **analytic evaluation**
 - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



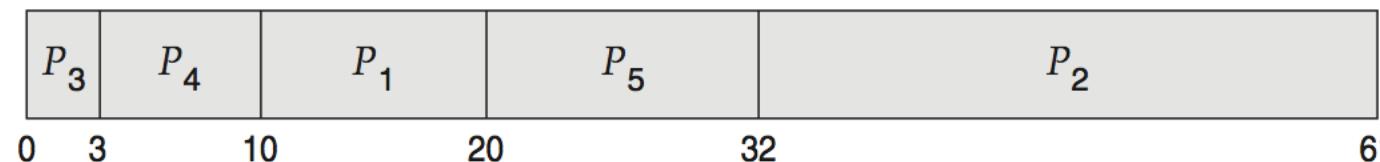


Deterministic Evaluation

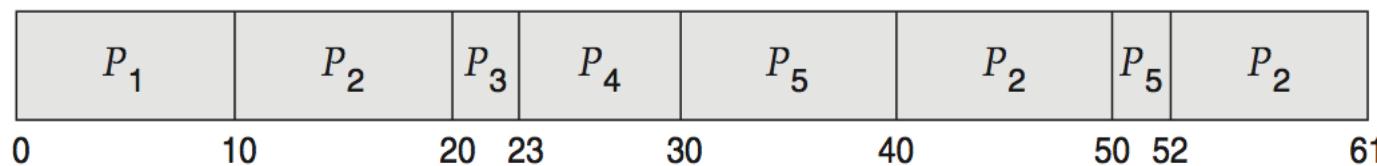
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
 - FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:





Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
 - Commonly exponential, and described by mean
 - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
 - Knowing arrival rates and service rates
 - Computes utilization, average queue length, average wait time, etc

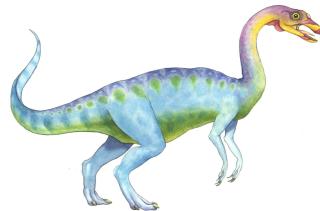




Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus
$$n = \lambda \times W$$
 - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

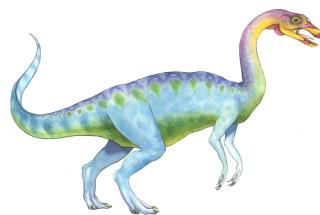




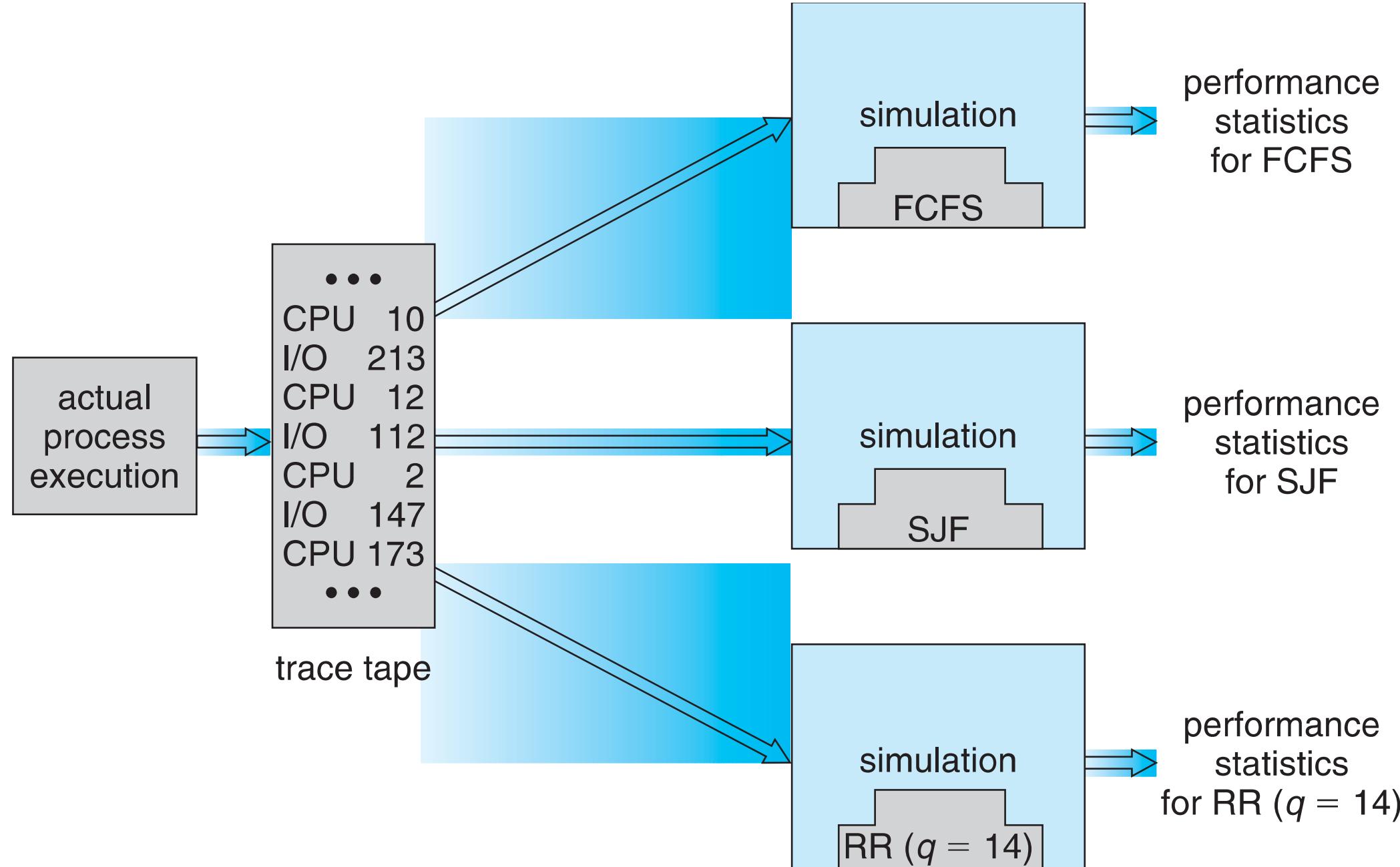
Simulations

- Queueing models limited
- **Simulations** more accurate
 - Programmed model of computer system
 - Clock is a variable
 - Gather statistics indicating algorithm performance
 - Data to drive simulation gathered via
 - ▶ Random number generator according to probabilities
 - ▶ Distributions defined mathematically or empirically
 - ▶ Trace tapes record sequences of real events in real systems





Evaluation of CPU Schedulers by Simulation



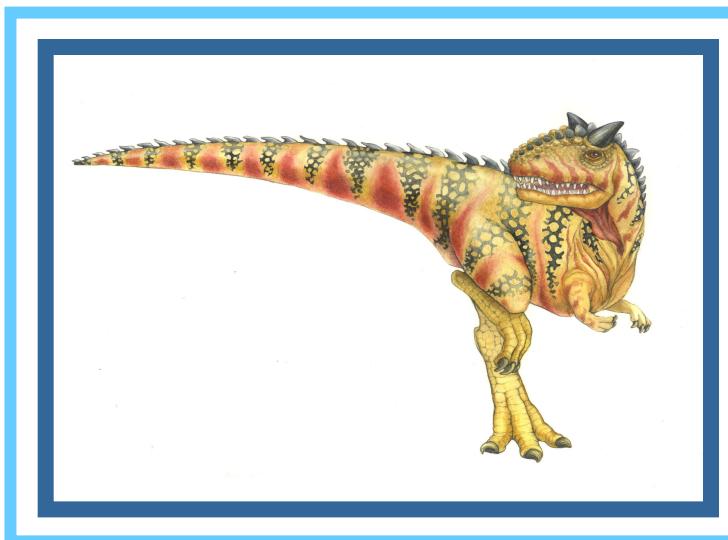


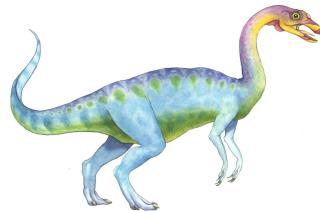
Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
 - High cost, high risk
 - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary



Chapter 6: Process Scheduling





Chapter 6: Process Scheduling

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

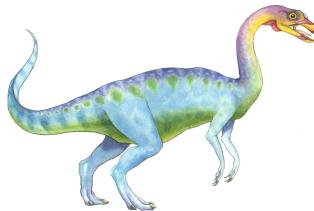




Objectives

- 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 examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems





Background

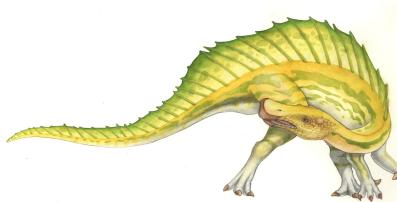
- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the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





Producer

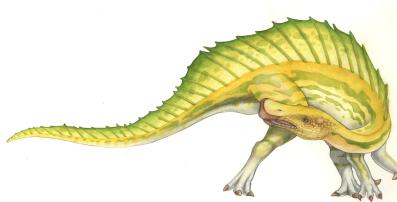
```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER SIZE) ;  
        /* do nothing */  
    buffer[in] = next produced;  
    in = (in + 1) % BUFFER SIZE;  
    counter++;  
}
```

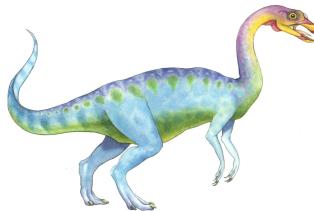




Consumer

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





Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

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

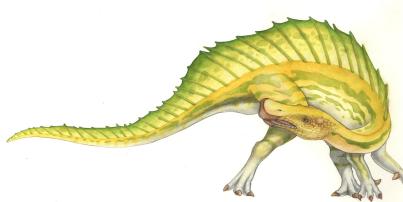
S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}





Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**





Critical Section

- General structure of process p_i is

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





Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
 2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
 3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes
- Two approaches depending on if kernel is preemptive or non-preemptive
 - **Preemptive** – allows preemption of process when running in kernel mode
 - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - ▶ Essentially free of race conditions in kernel mode





Peterson's Solution

- Good algorithmic description of solving the problem
- 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!



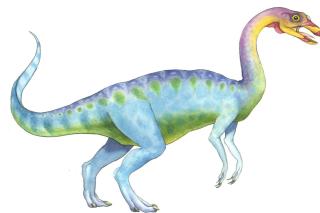


Algorithm for Process P_i

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

- Provable that
 - 1. Mutual exclusion is preserved
 - 2. Progress requirement is satisfied
 - 3. Bounded-waiting requirement is met





Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic** = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words





Solution to Critical-section Problem Using Locks

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





test_and_set Instruction

- Definition:

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





Solution using test_and_set()

- Shared boolean variable lock, initialized to FALSE
- Solution:

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



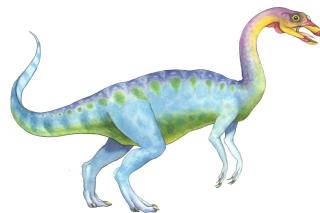


compare_and_swap Instruction

- Definition:

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



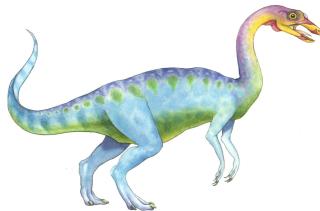


Solution using compare_and_swap

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

```
do {  
    while (compare and swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```





Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    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);
```





Mutex Locks

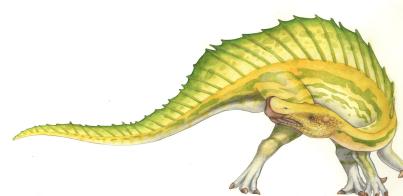
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect critical regions with it by first `acquire()` a lock then `release()` it
 - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

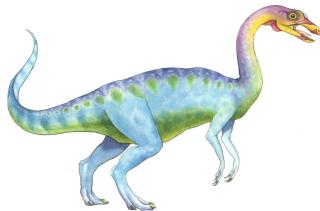




acquire() and release()

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

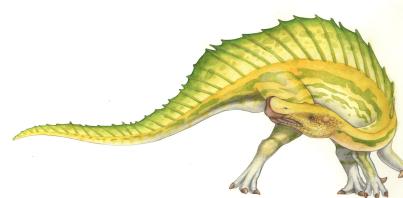




Semaphore

- 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)  
        ; // busy wait  
    S--;  
}  
  
signal (S) {  
    S++;  
}
```





Semaphore Usage

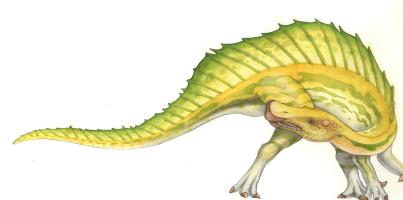
- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Then a **mutex lock**
- Can implement a counting semaphore S as a binary semaphore
- Can solve various synchronization problems
- Consider P_1 , and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

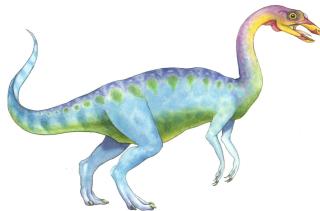
P1:

```
S1;  
signal(synch);
```

P2:

```
wait(synch);  
S2;
```

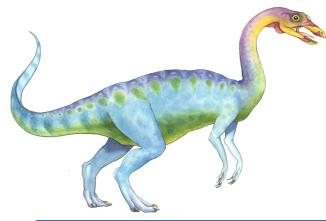




Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal 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



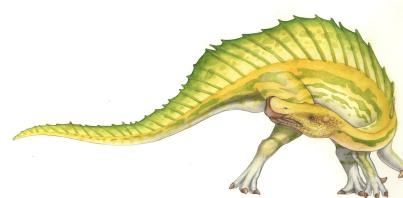


Semaphore Implementation with no Busy waiting (Cont.)

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

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

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





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

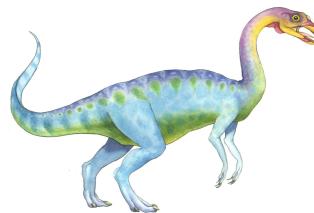
```
wait(S) ;  
wait(Q) ;  
.  
signal(S) ;  
signal(Q) ;
```

P_1

```
wait(Q) ;  
wait(S) ;  
.  
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
 - Solved via **priority-inheritance protocol**





Classical Problems of Synchronization

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





Bounded-Buffer Problem

- 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

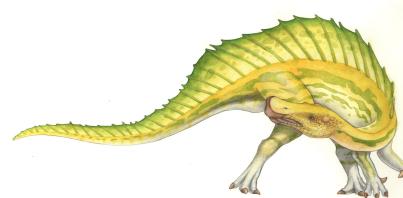




Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced 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 next_consumed */  
  
    ...  
  
    signal(mutex);  
    signal(empty);  
  
    ...  
    /* consume the item in next consumed */  
  
    ...  
} while (true);
```





Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do *not* perform any updates
 - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
 - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
 - Data set
 - Semaphore `rw_mutex` initialized to 1
 - Semaphore `mutex` initialized to 1
 - Integer `read_count` initialized to 0





Readers-Writers Problem (Cont.)

- The structure of a writer process

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

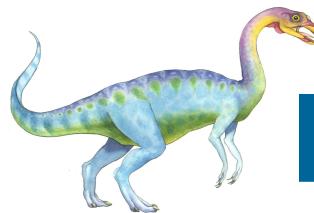




Readers-Writers Problem (Cont.)

- The structure of a reader process
- do {
 wait(mutex);
 read count++;
 if (read count == 1)
 ■ wait(rw mutex); signal(mutex);
 ■ ...
 /* reading is performed */
 ■ ... wait(mutex);
 read count--;
 if (read count == 0)
 ■ signal(rw mutex); signal(mutex);
 ■ } while (true);





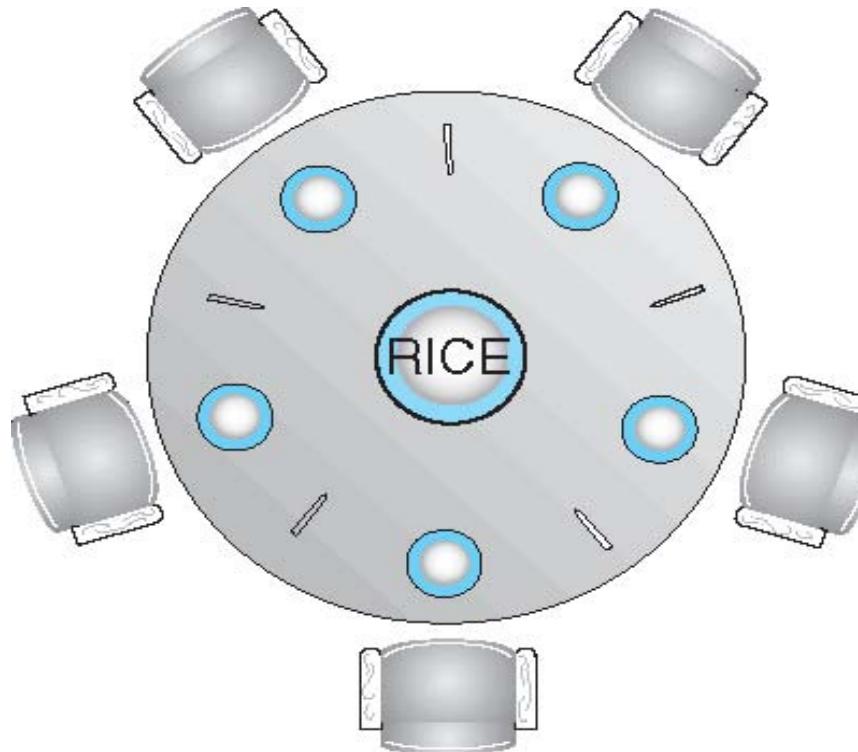
Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks





Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - Need both to eat, then release both when done
- In the case of 5 philosophers
 - Shared data
 - ▶ Bowl of rice (data set)
 - ▶ Semaphore **chopstick [5]** initialized to 1



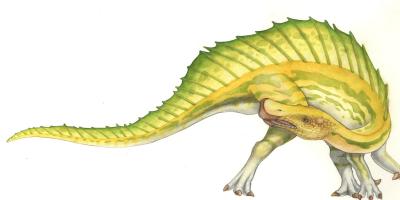


Dining-Philosophers Problem Algorithm

- 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 this algorithm?



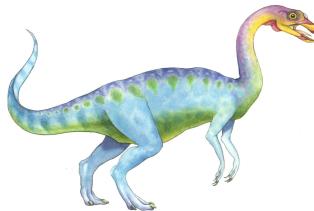


Problems with Semaphores

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

- Deadlock and starvation





Monitors

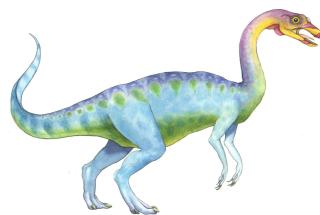
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

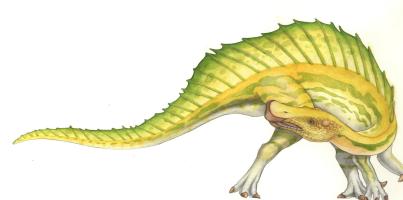
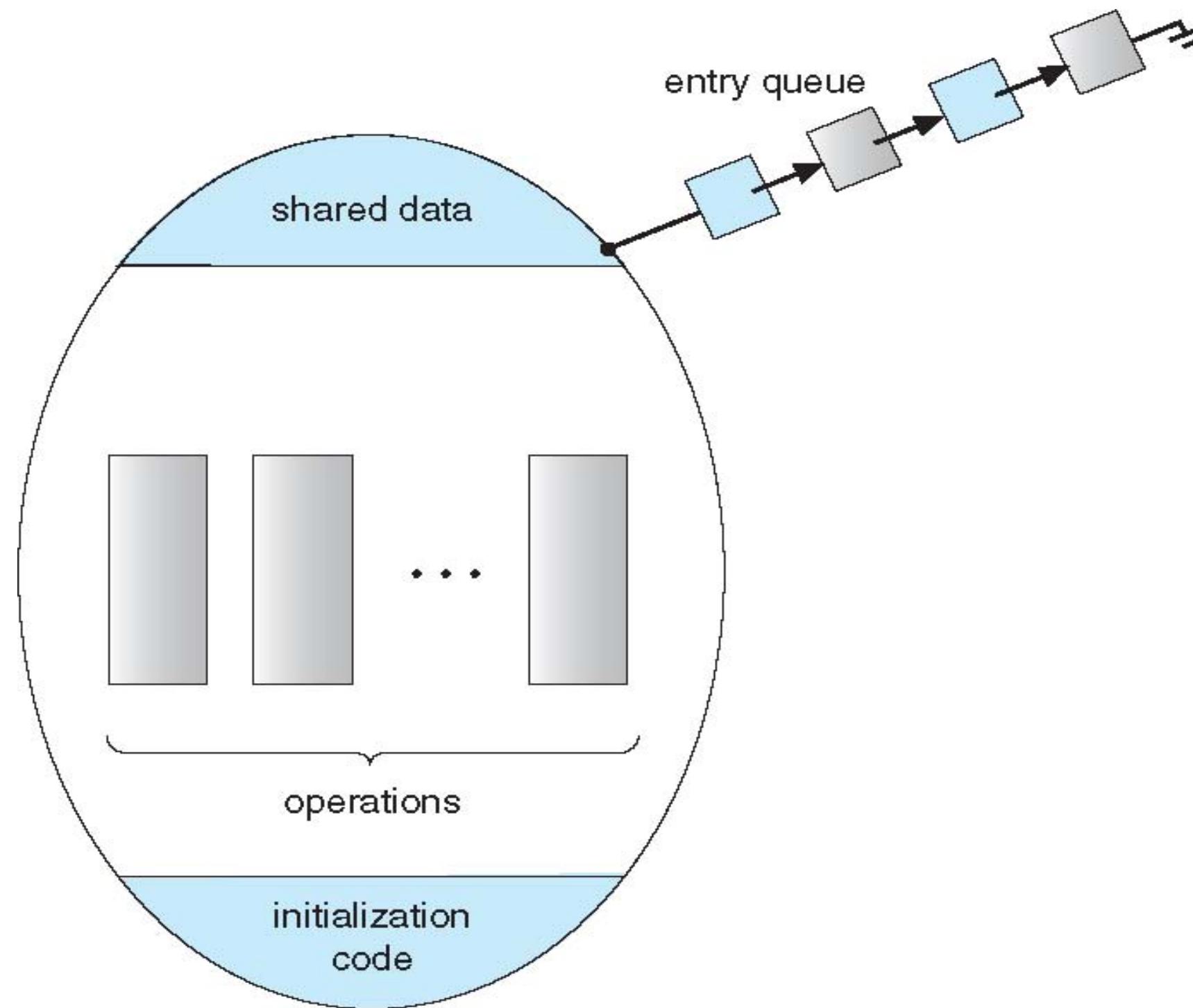
    procedure Pn (...) {.....}

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





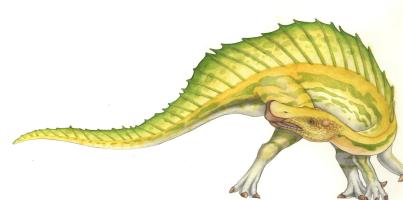
Schematic view of a Monitor





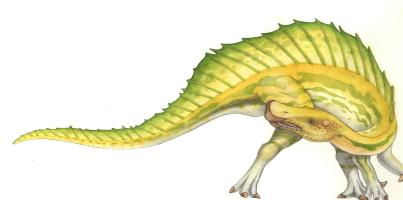
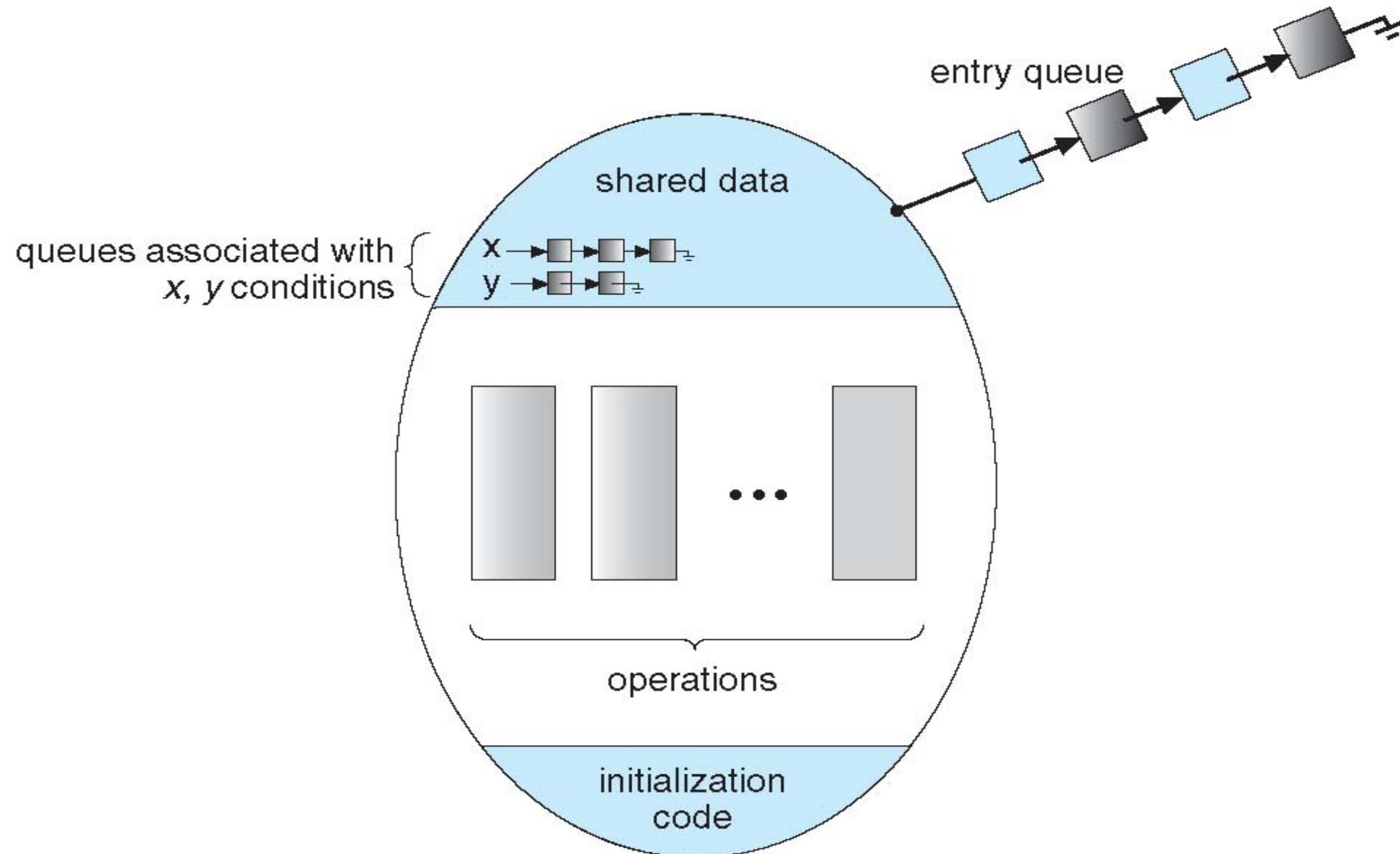
Condition Variables

- `condition x, y;`
- Two operations on a condition variable:
 - `x.wait ()` – a process that invokes the operation is suspended until `x.signal ()`
 - `x.signal ()` – resumes one of processes (if any) that invoked `x.wait ()`
 - ▶ If no `x.wait ()` on the variable, then it has no effect on the variable





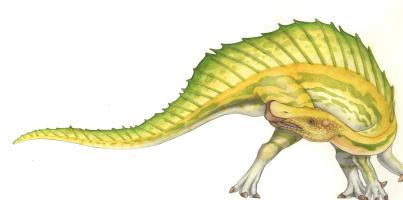
Monitor with Condition Variables





Condition Variables Choices

- If process P invokes `x.signal ()`, with Q in `x.wait ()` state, what should happen next?
 - If Q is resumed, then P must wait
- Options include
 - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
 - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
 - Both have pros and cons – language implementer can decide
 - Monitors implemented in Concurrent Pascal compromise
 - ▶ P executing signal immediately leaves the monitor, Q is resumed
 - Implemented in other languages including Mesa, C#, Java



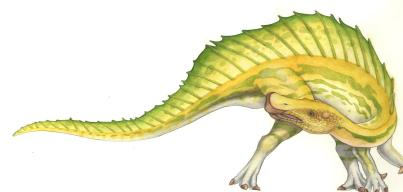


Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    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:

`DiningPhilosophers.pickup (i);`

`EAT`

`DiningPhilosophers.putdown (i);`

- No deadlock, but starvation is possible





Monitor Implementation Using Semaphores

- Variables

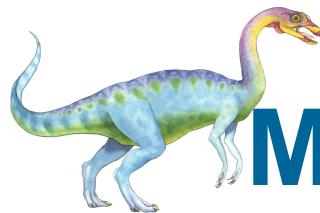
```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- 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





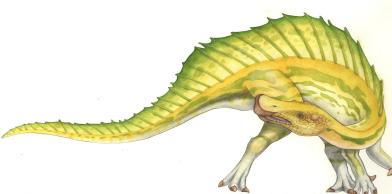
Monitor Implementation – Condition Variables

- For each condition variable **x**, we have:

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

- The operation **x.wait** can be implemented as:

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





Monitor Implementation (Cont.)

- The operation `x.signal` can be implemented as:

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





Resuming Processes within a Monitor

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





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;
    }
}
```

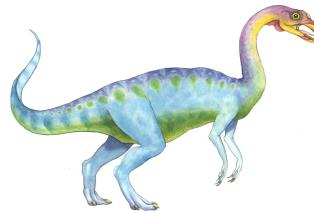




Synchronization Examples

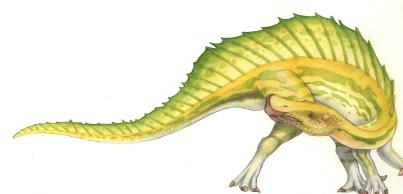
- Solaris
- Windows XP
- Linux
- Pthreads





Solaris Synchronization

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





Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
 - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
 - **Events**
 - ▶ An event acts much like a condition variable
 - Timers notify one or more thread when time expired
 - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)





Linux Synchronization

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





Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
 - mutex locks
 - condition variables
- Non-portable extensions include:
 - read-write locks
 - spinlocks





Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions





System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- **Transaction** - collection of instructions or operations that performs single logical function
 - Here we are concerned with changes to stable storage – disk
 - Transaction is series of **read** and **write** operations
 - Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation
 - Aborted transaction must be **rolled back** to undo any changes it performed





Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
 - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
 - Example: disk and tape
- Stable storage – Information never lost
 - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage





Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is [write-ahead logging](#)
 - Log on stable storage, each log record describes single transaction write operation, including
 - ▶ Transaction name
 - ▶ Data item name
 - ▶ Old value
 - ▶ New value
 - $\langle T_i \text{ starts} \rangle$ written to log when transaction T_i starts
 - $\langle T_i \text{ commits} \rangle$ written when T_i commits
- Log entry must reach stable storage before operation on data occurs





Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
 - $\text{Undo}(T_i)$ restores value of all data updated by T_i
 - $\text{Redo}(T_i)$ sets values of all data in transaction T_i to new values
- $\text{Undo}(T_i)$ and $\text{redo}(T_i)$ must be **idempotent**
 - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
 - If log contains $\langle T_i \text{ starts} \rangle$ without $\langle T_i \text{ commits} \rangle$, $\text{undo}(T_i)$
 - If log contains $\langle T_i \text{ starts} \rangle$ and $\langle T_i \text{ commits} \rangle$, $\text{redo}(T_i)$

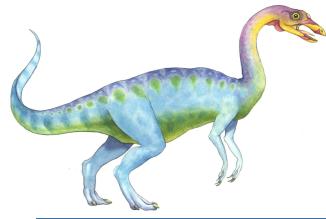




Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
 1. Output all log records currently in volatile storage to stable storage
 2. Output all modified data from volatile to stable storage
 3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes T_i , such that T_i started executing before the most recent checkpoint, and all transactions after T_i . All other transactions already on stable storage





Concurrent Transactions

- Must be equivalent to serial execution – [serializability](#)
- Could perform all transactions in critical section
 - Inefficient, too restrictive
- [Concurrency-control algorithms](#) provide serializability

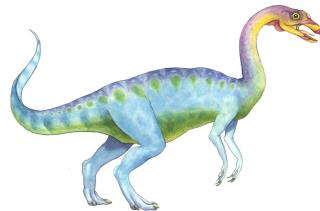




Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0 , T_1 atomically
- Execution sequence called **schedule**
- Atomically executed transaction order called **serial schedule**
- For N transactions, there are $N!$ valid serial schedules

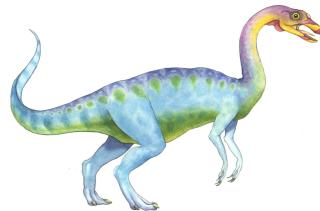




Schedule 1: T_0 then T_1

	T_0	T_1
read(A)		
write(A)		
read(B)		
write(B)		
		read(A)
		write(A)
		read(B)
		write(B)

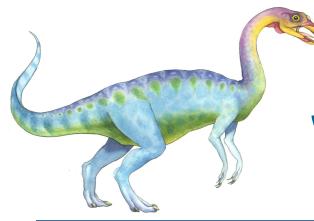




Nonserial Schedule

- Nonserial schedule allows overlapped execute
 - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
 - Conflict if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
 - Then S' with swapped order O_j, O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
 - S is conflict serializable





Schedule 2: Concurrent Serializable Schedule

	T_0	T_1
	read(A) write(A)	
	read(B) write(B)	read(A) write(A)
		read(B) write(B)





Locking Protocol

- Ensure serializability by associating lock with each data item
 - Follow locking protocol for access control
- Locks
 - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
 - Exclusive – T_i has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
 - Similar to readers-writers algorithm





Two-phase Locking Protocol

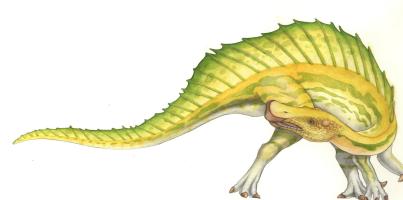
- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
 - Growing – obtaining locks
 - Shrinking – releasing locks
- Does not prevent deadlock

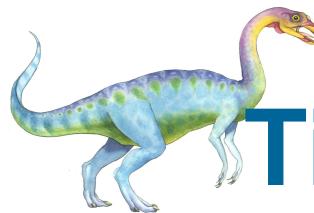




Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
 - $TS(T_i) < TS(T_j)$ if T_i entered system before T_j
 - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
 - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j





Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
 - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
 - R-timestamp(Q) – largest timestamp of successful read(Q)
 - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting **read** and **write** executed in timestamp order
- Suppose T_i executes **read(Q)**
 - If $TS(T_i) < W\text{-timestamp}(Q)$, T_i needs to read value of Q that was already overwritten
 - ▶ **read** operation rejected and T_i rolled back
 - If $TS(T_i) \geq W\text{-timestamp}(Q)$
 - ▶ **read** executed, R-timestamp(Q) set to $\max(R\text{-timestamp}(Q), TS(T_i))$

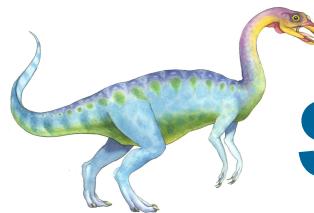




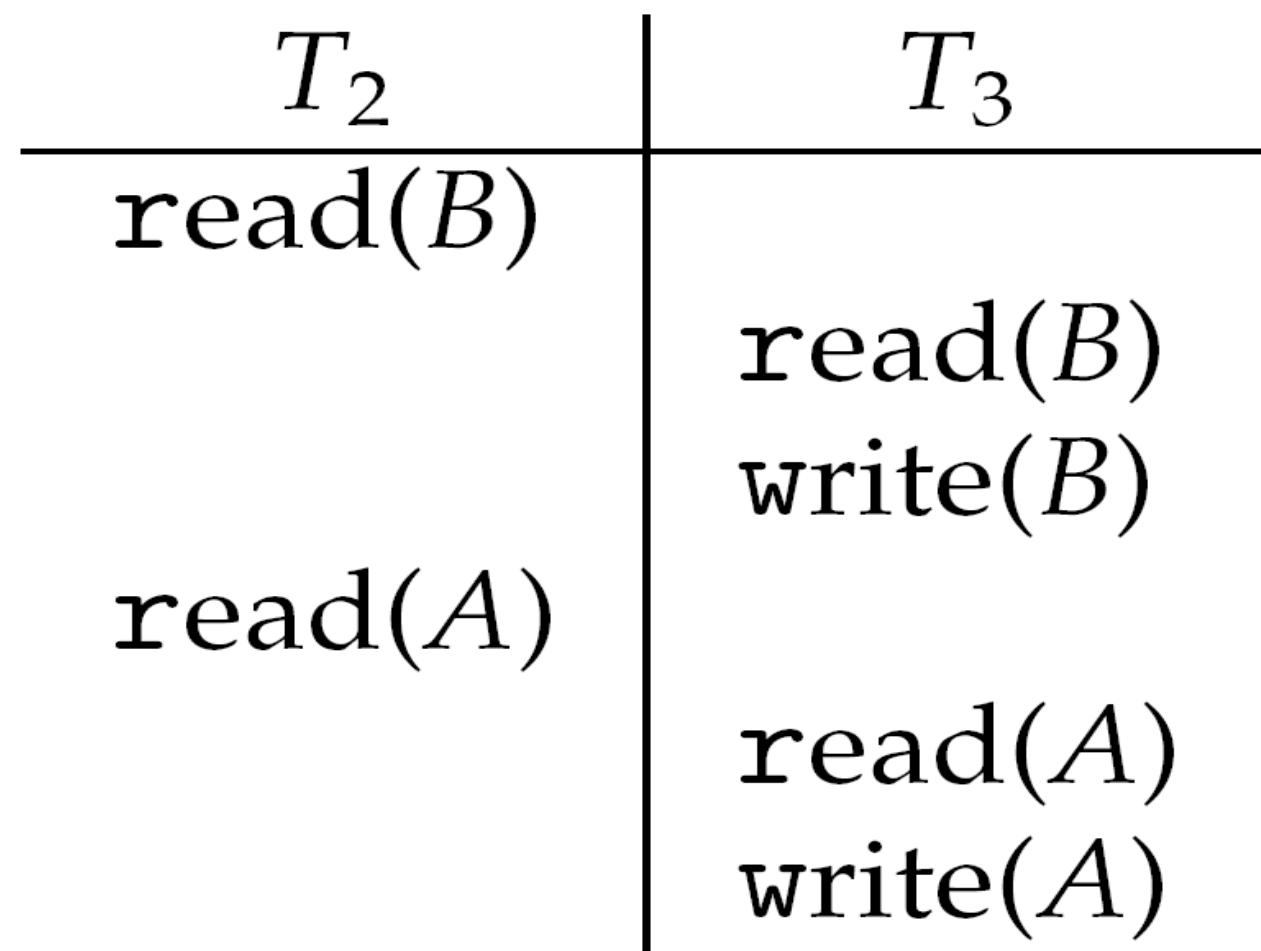
Timestamp-ordering Protocol

- Suppose T_i executes `write(Q)`
 - If $TS(T_i) < R\text{-timestamp}(Q)$, value Q produced by T_i was needed previously and T_i assumed it would never be produced
 - ▶ `Write` operation rejected, T_i rolled back
 - If $TS(T_i) < W\text{-timestamp}(Q)$, T_i attempting to write obsolete value of Q
 - ▶ `Write` operation rejected and T_i rolled back
 - Otherwise, `write` executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

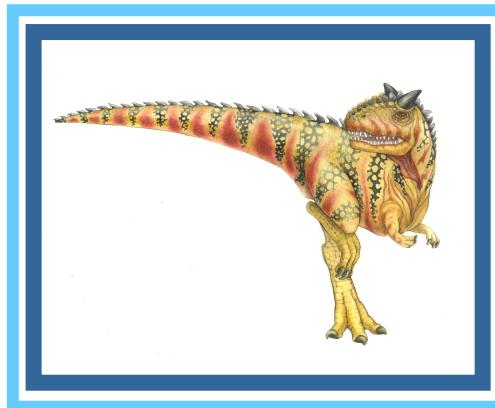


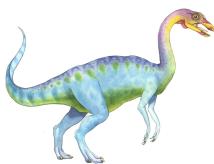


Schedule Possible Under Timestamp Protocol



Chapter 7: Deadlocks

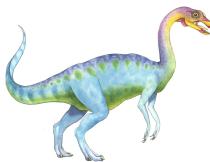




Chapter 7: Deadlocks

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



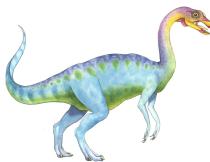


Chapter Objectives

- 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

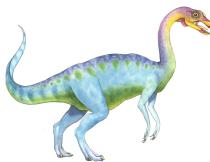




System Model

- System consists of resources
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**





Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2, \dots, 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 .





Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc
- See example box in text page 318 for mutex deadlock





Resource-Allocation Graph

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

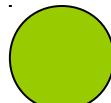
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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





Resource-Allocation Graph (Cont.)

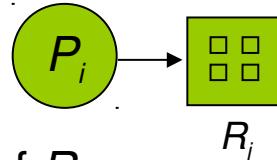
- Process



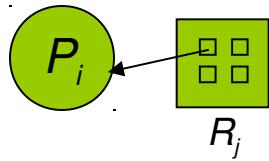
- Resource Type with 4 instances



- P_i requests instance of R_j

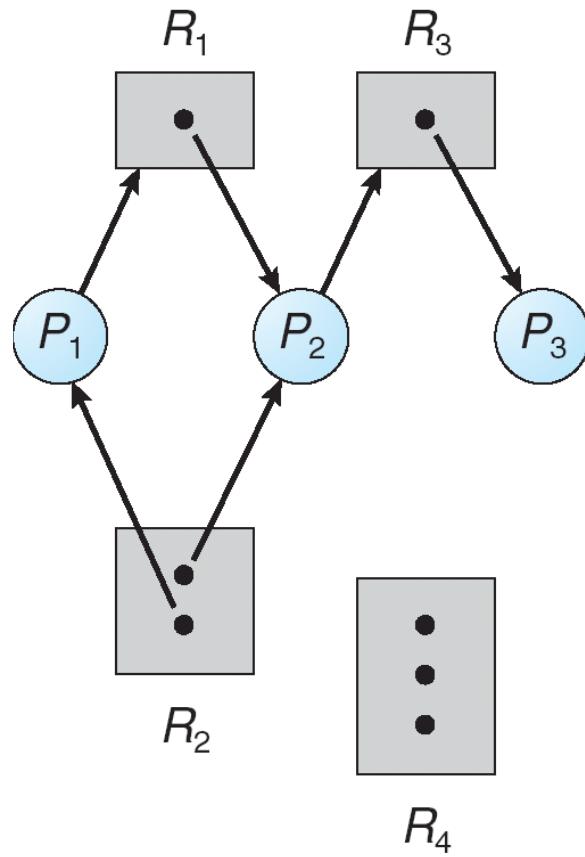


- P_i is holding an instance of R_j



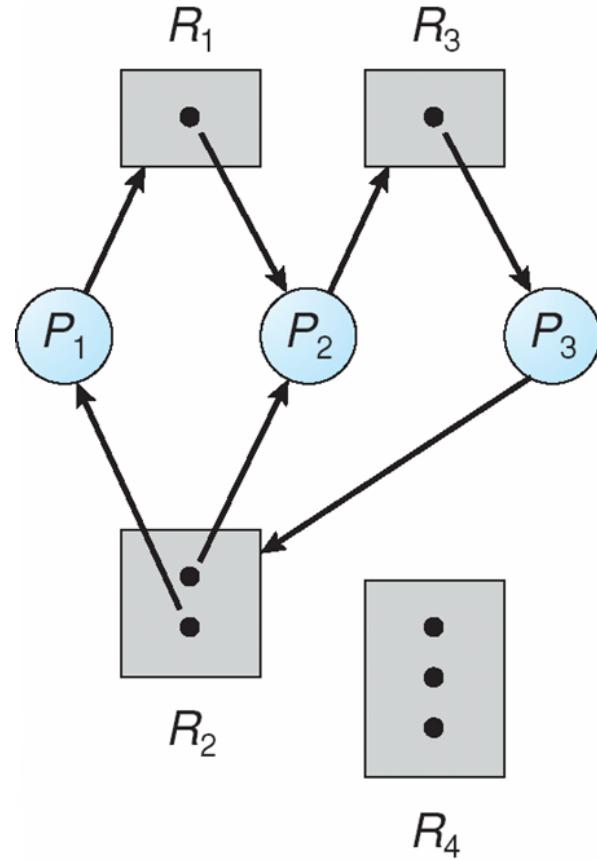


Example of a Resource Allocation Graph



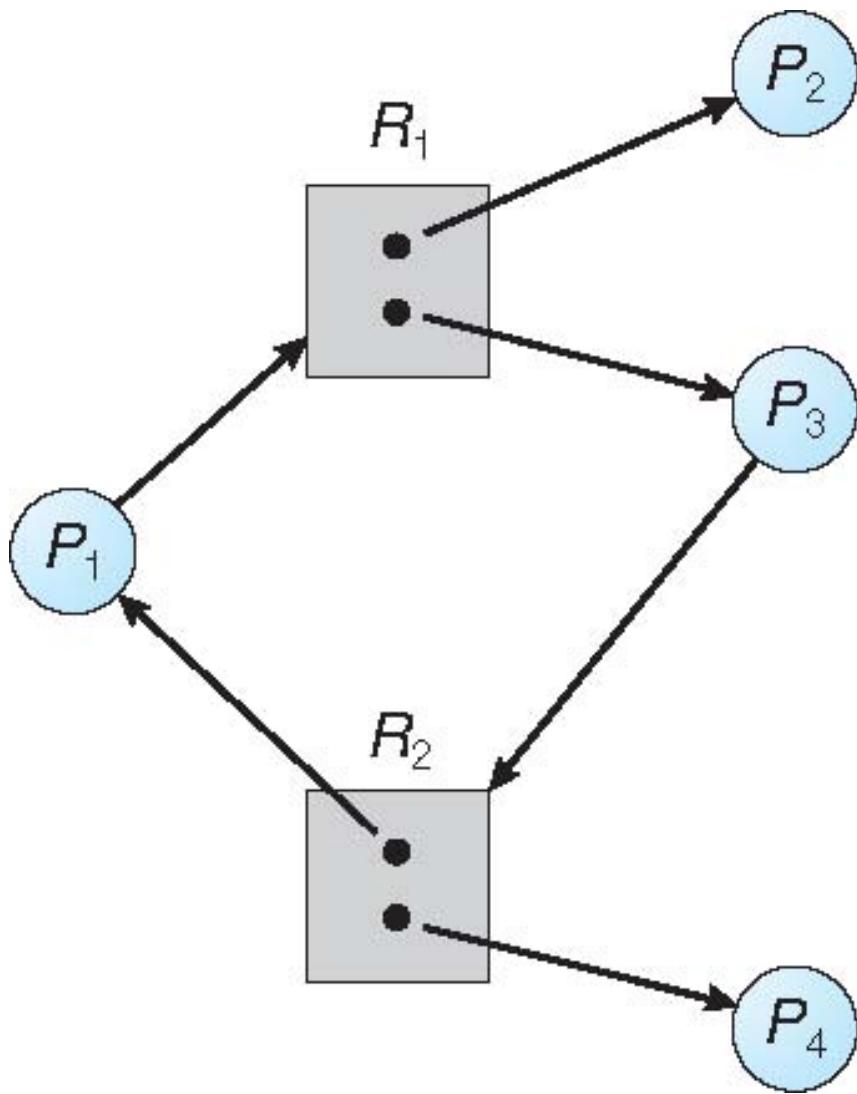


Resource Allocation Graph With A Deadlock





Graph With A Cycle But No Deadlock





Basic Facts

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





Methods for Handling Deadlocks

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



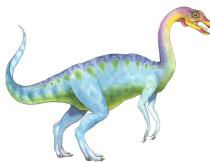


Deadlock Prevention

Restrain the ways request can be made

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





Deadlock Prevention (Cont.)

■ No Preemption –

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

■ Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



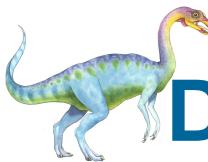


Deadlock Example

```
/* thread one runs in this function */
void *do work one(void *param)
{
    pthread mutex lock(&first mutex);
    pthread mutex lock(&second mutex);
    /** * Do some work */
    pthread mutex unlock(&second mutex);
    pthread mutex unlock(&first mutex);
    pthread exit(0);
}

/* thread two runs in this function */
void *do work two(void *param)
{
    pthread mutex lock(&second mutex);
    pthread mutex lock(&first mutex);
    /** * Do some work */
    pthread mutex unlock(&first mutex);
    pthread mutex unlock(&second mutex);
    pthread exit(0);
}
```

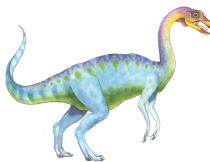




Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get lock(from);
    lock2 = get lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```





Deadlock Avoidance

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

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

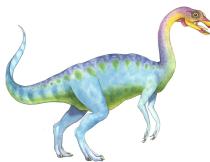




Safe State

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





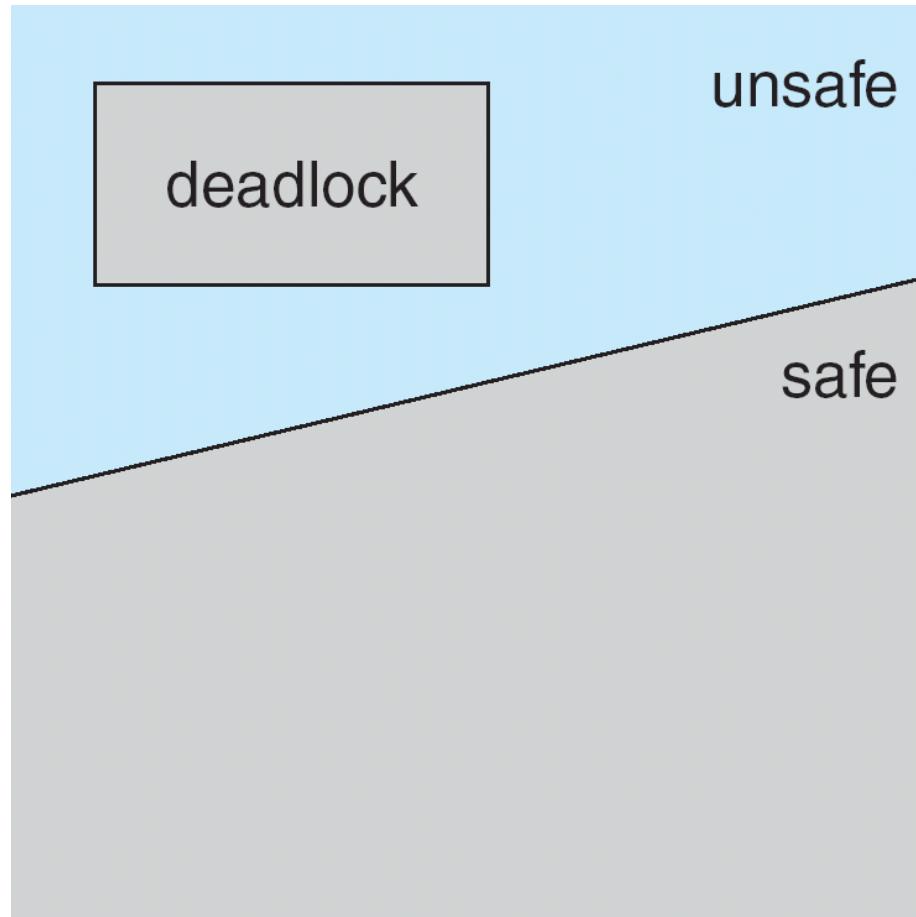
Basic Facts

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





Safe, Unsafe, Deadlock State





Avoidance algorithms

- Single instance of a resource type
 - Use a resource-allocation graph

- Multiple instances of a resource type
 - Use the banker's algorithm

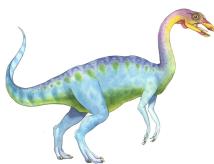




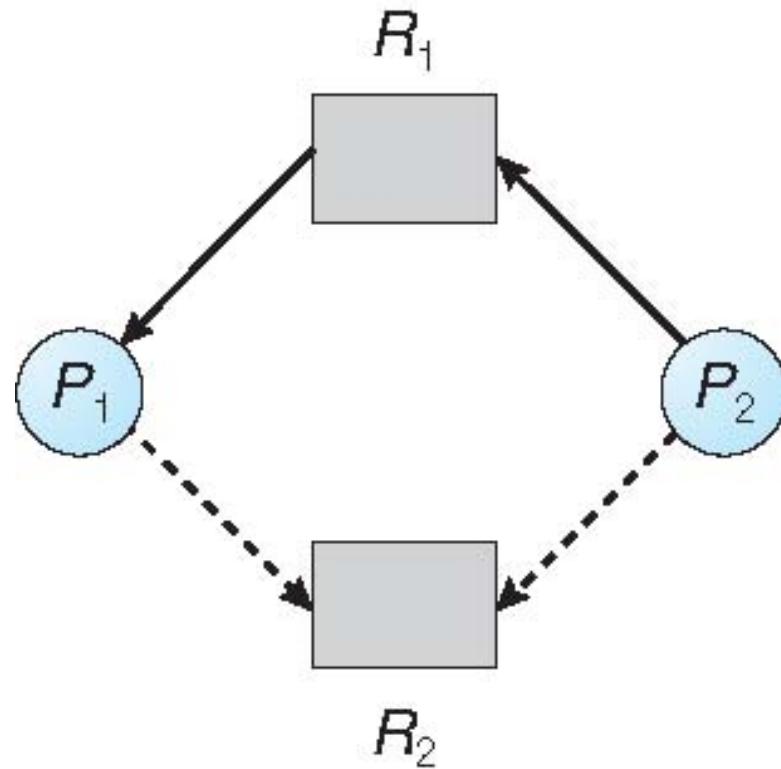
Resource-Allocation Graph Scheme

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



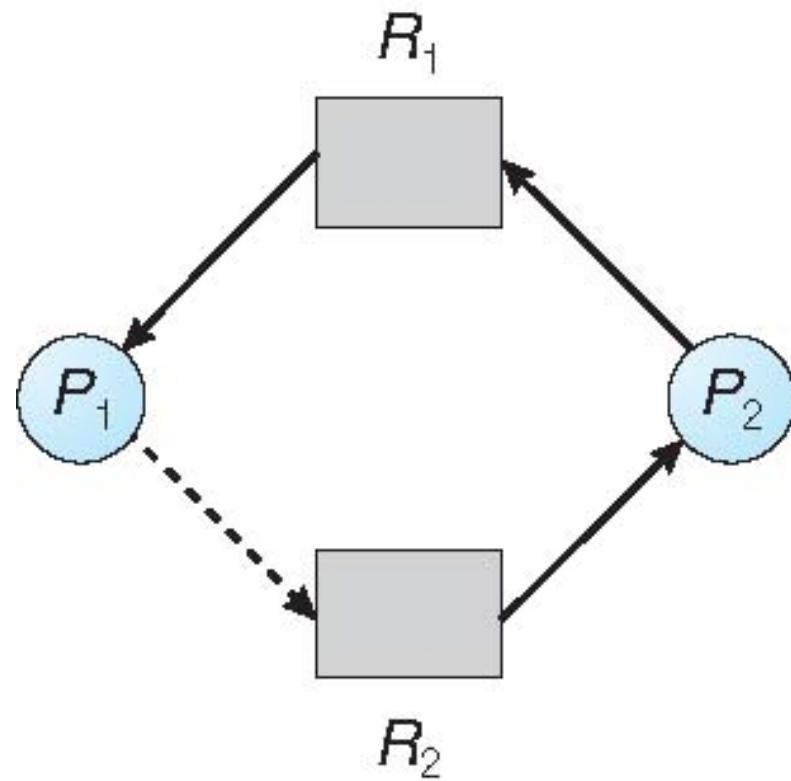


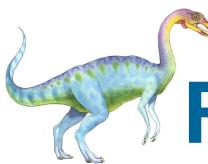
Resource-Allocation Graph





Unsafe State In Resource-Allocation Graph

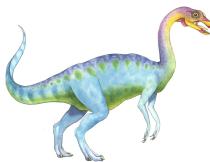




Resource-Allocation Graph Algorithm

- Suppose 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

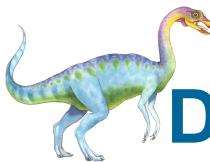




Banker's Algorithm

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

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





Safety Algorithm

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

Work = Available

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

2. Find an i such that both:
 - (a) **Finish [i] = false**
 - (b) **Need_i ≤ 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 $\text{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\text{Request}_i \leq \text{Available}$, 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:

$$\text{Available} = \text{Available} - \text{Request};$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

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





Example of Banker's Algorithm

■ 5 processes P_0 through P_4 ;

3 resource types:

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

Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
P_1	2	0	0	3	2	2
P_2	3	0	2	9	0	2
P_3	2	1	1	2	2	2
P_4	0	0	2	4	3	3





Example (Cont.)

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

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

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria





Example: P_1 Request (1,0,2)

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

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

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





Deadlock Detection

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





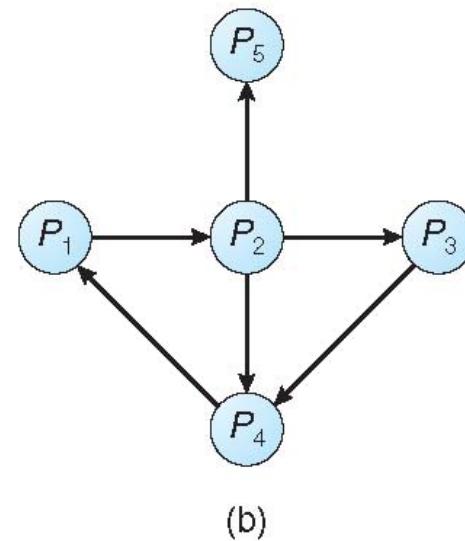
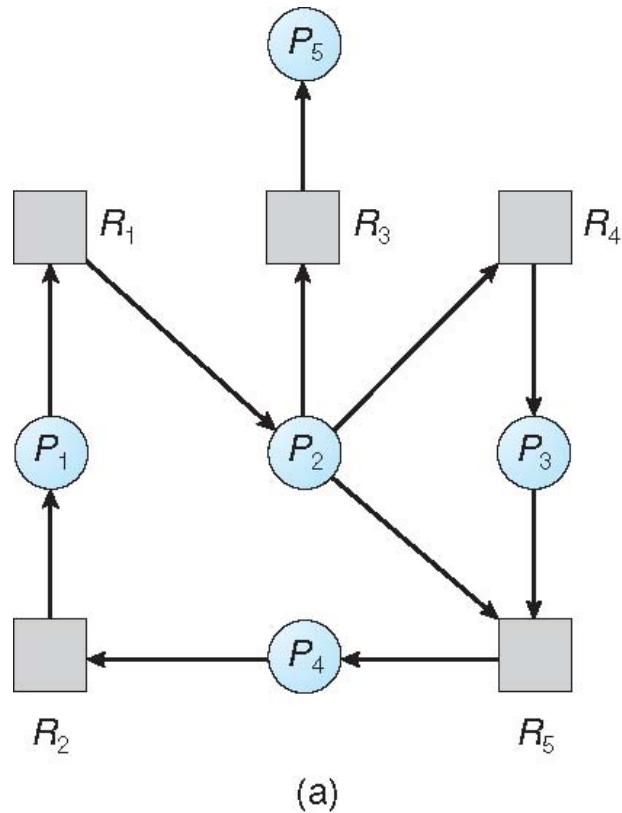
Single Instance of Each Resource Type

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





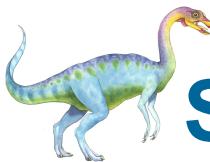
Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph





Several Instances of a Resource Type

- **Available:** A vector of length m indicates the number of available resources of each type
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i][j] = k$, 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 $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then
Finish[i] = false; otherwise, **Finish[i] = true**

2. Find an index i such that both:

- (a) **Finish[i] == false**
- (b) **Request_i ≤ Work**

If no such i exists, go to step 4





Detection Algorithm (Cont.)

3. $\text{Work} = \text{Work} + \text{Allocation}_i$,
 $\text{Finish}[i] = \text{true}$
go to step 2
4. If $\text{Finish}[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $\text{Finish}[i] == \text{false}$, then P_i is deadlocked

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





Example of Detection Algorithm

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

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all i





Example (Cont.)

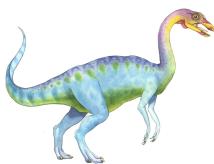
- P_2 requests an additional instance of type **C**

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?
 1. Priority of the process
 2. How long process has computed, and how much longer to completion
 3. Resources the process has used
 4. Resources process needs to complete
 5. How many processes will need to be terminated
 6. Is process interactive or batch?



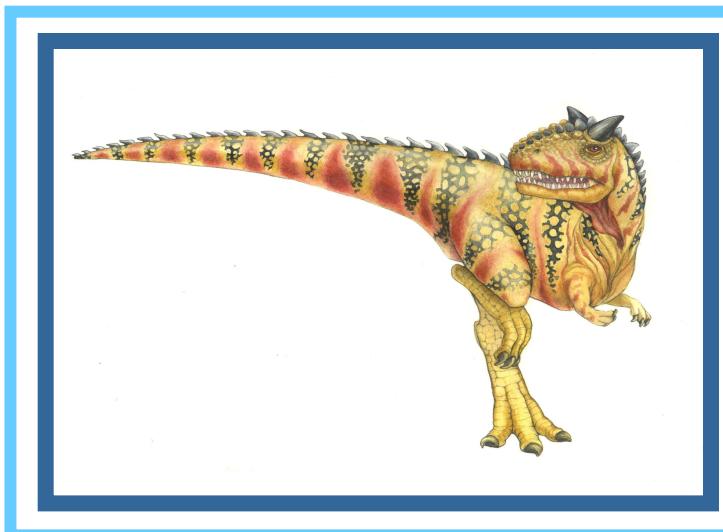


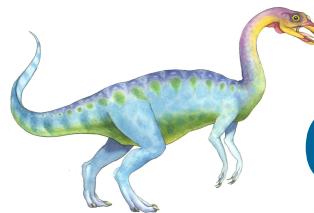
Recovery from Deadlock: Resource Preemption

- **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



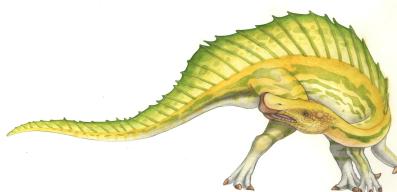
Chapter 8: Memory-Management Strategies





Chapter 8: Memory Management Strategies

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture





Objectives

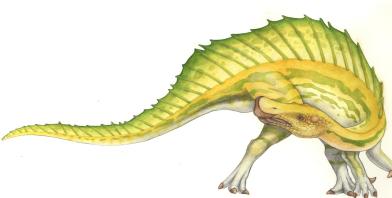
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging





Background

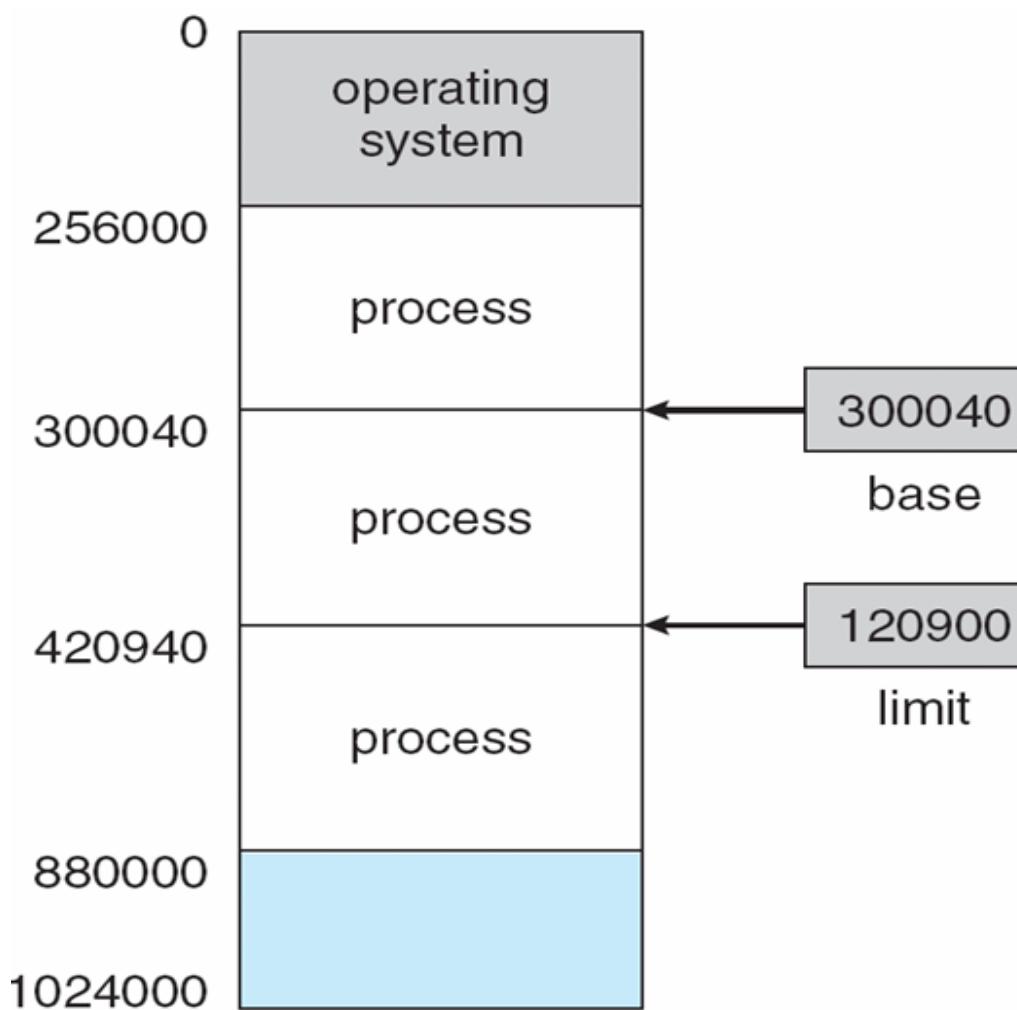
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or 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

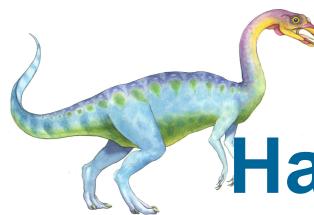




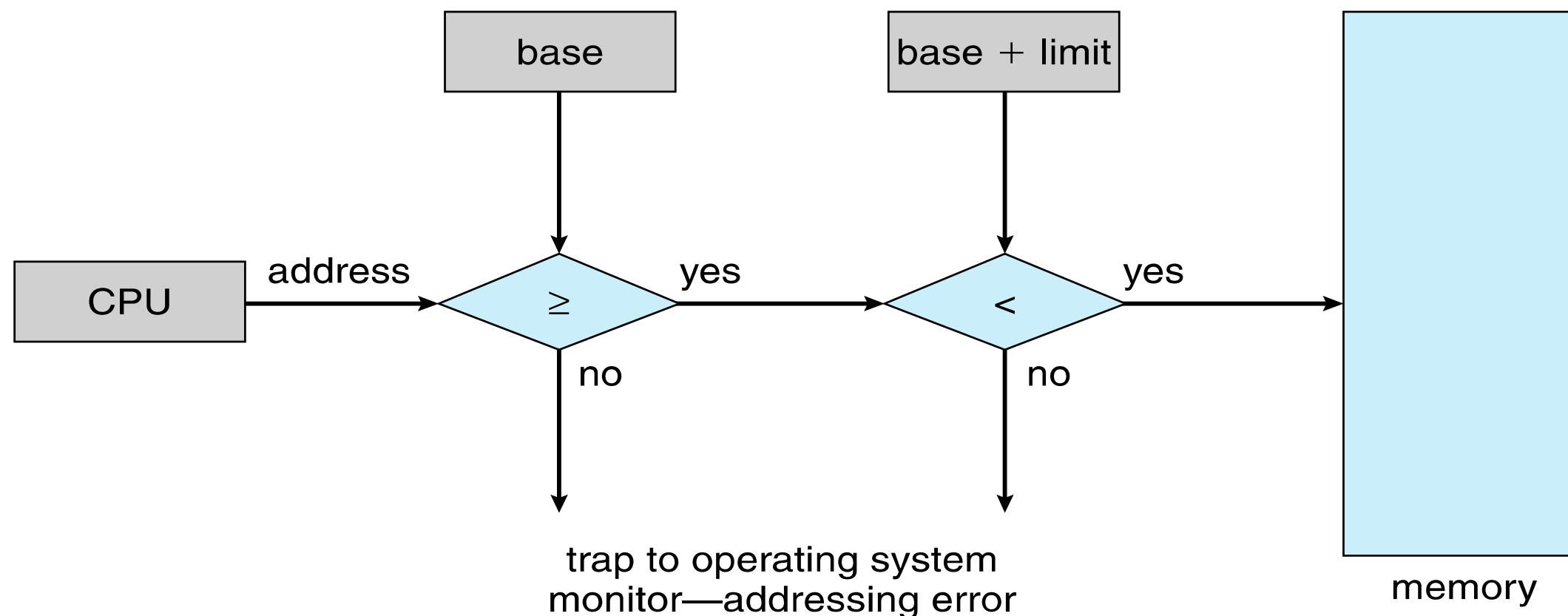
Base and Limit Registers

- 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





Hardware Address Protection with Base and Limit Registers





Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- 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





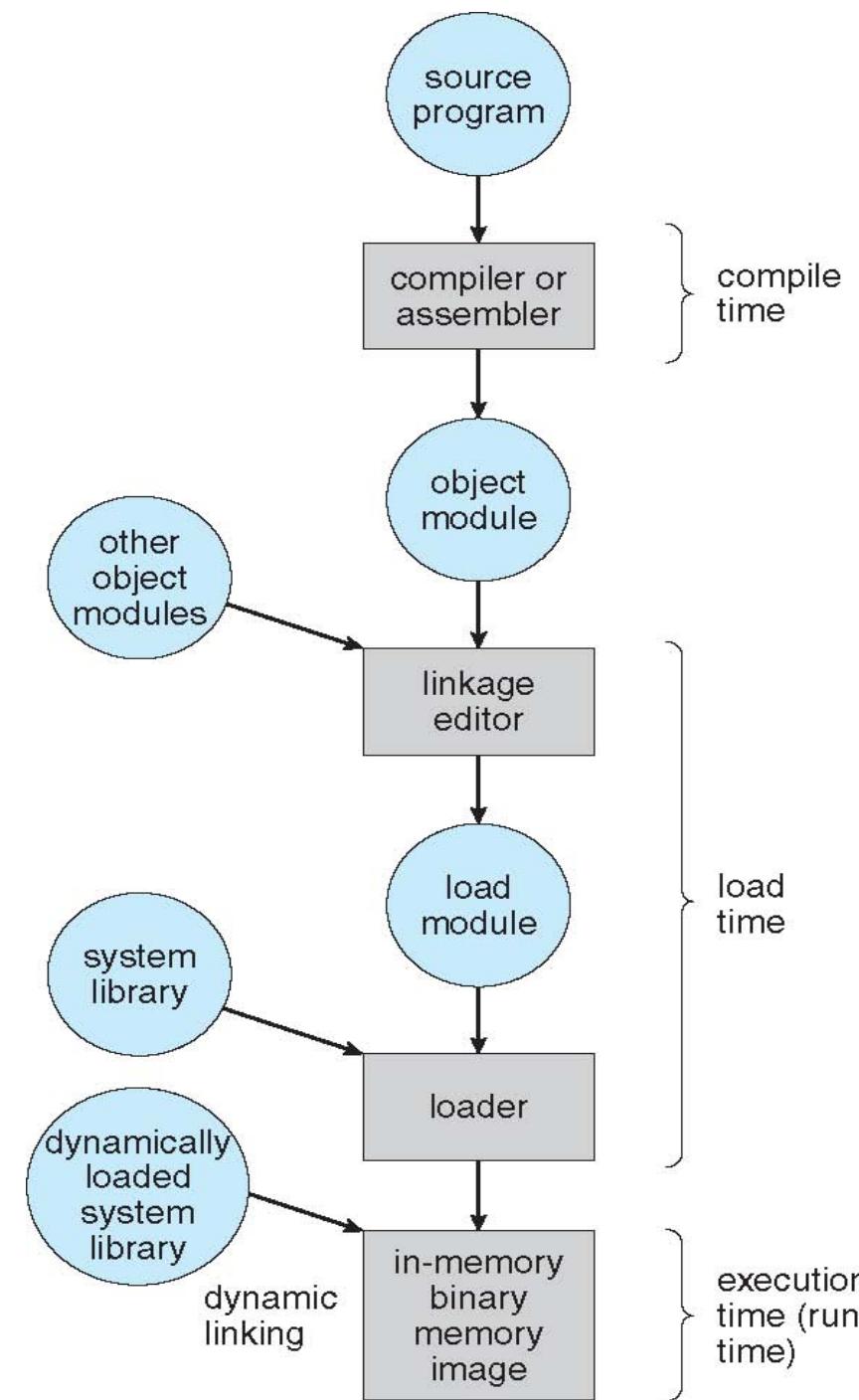
Binding of Instructions and Data to Memory

- 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
 - **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

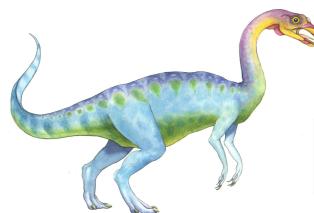




Memory-Management Unit (MMU)

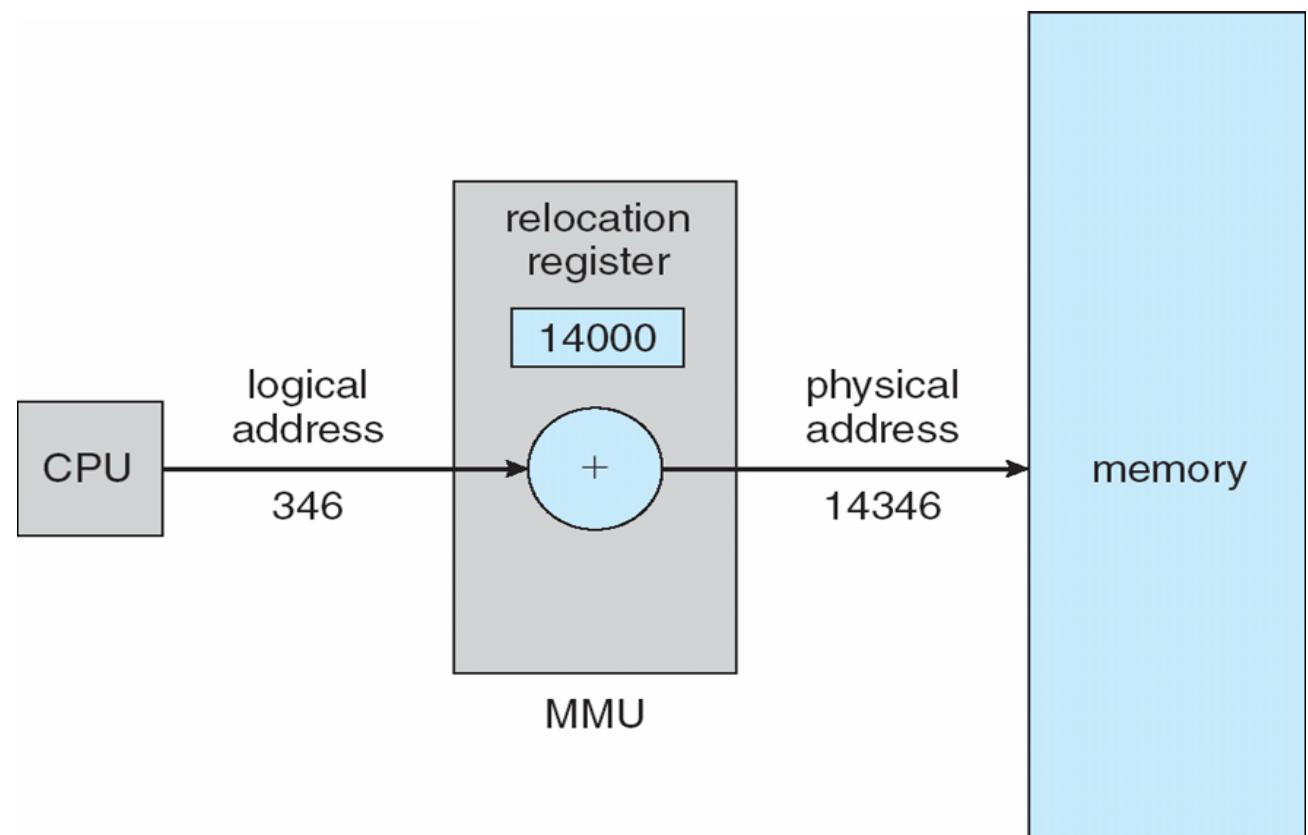
- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - Base 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
- 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

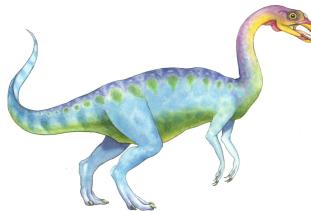




Dynamic Linking

- **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





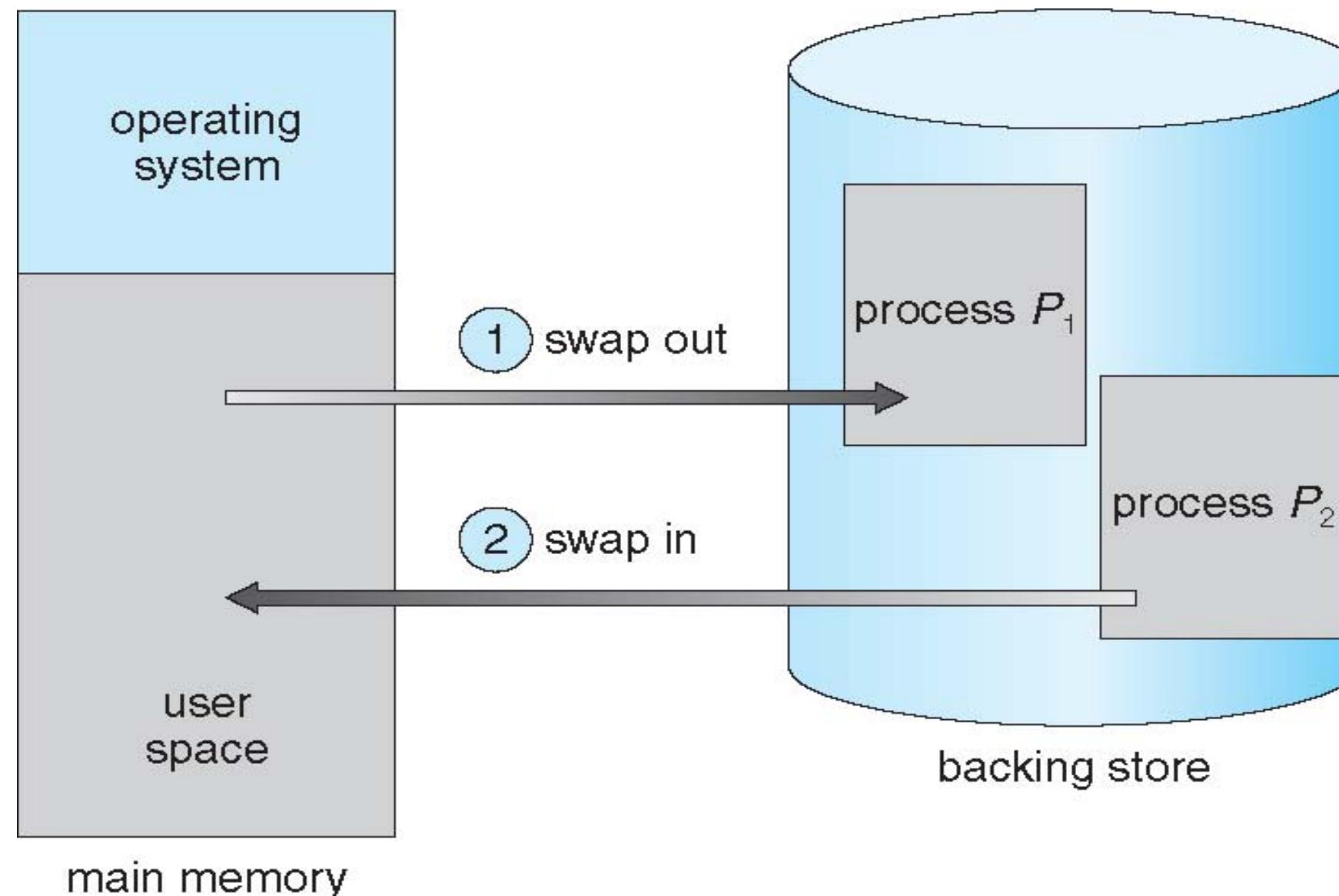
Swapping

- 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

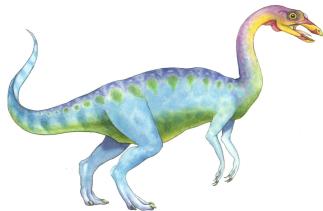




Context Switch Time including Swapping

- 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
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - Swap out time of 2000 ms
 - Plus swap in of same sized process
 - Total context switch swapping component time of 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
 - But modified version common
 - ▶ Swap only when free memory extremely low





Swapping on Mobile Systems

- Not typically supported
 - Flash memory based
 - ▶ Small amount of space
 - ▶ Limited number of write cycles
 - ▶ Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
 - iOS **asks** apps to voluntarily relinquish allocated memory
 - ▶ Read-only data thrown out and reloaded from flash if needed
 - ▶ Failure to free can result in termination
 - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
 - Both OSes support paging as discussed below





Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method

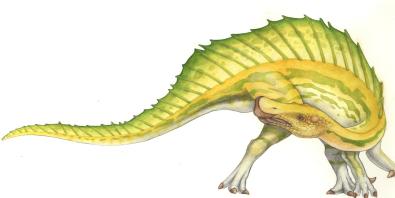
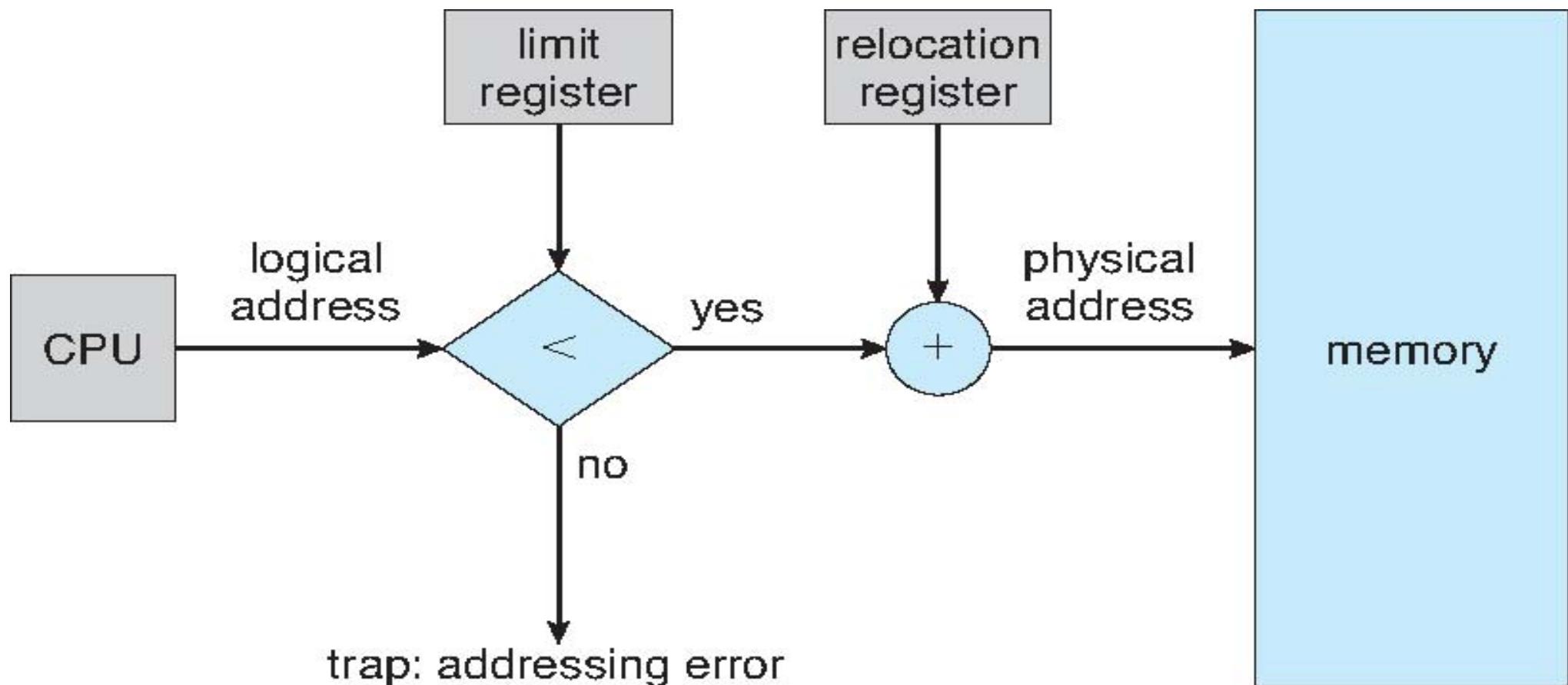
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory

- 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

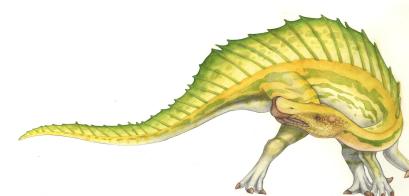
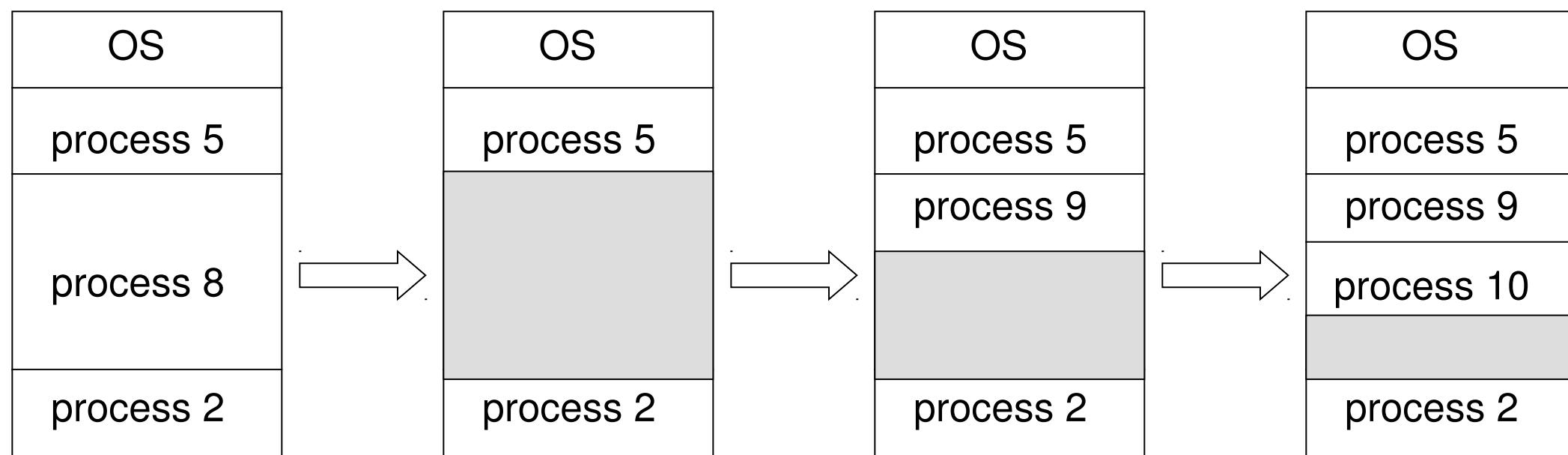


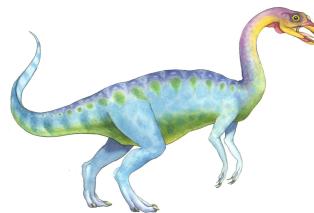


Contiguous Allocation (Cont.)

■ 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





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
 - I/O problem
 - ▶ Latch job in memory while it is involved in I/O
 - ▶ Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems





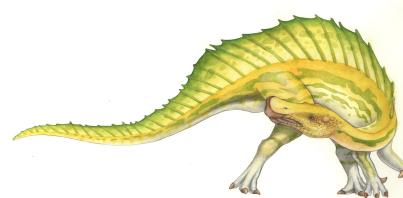
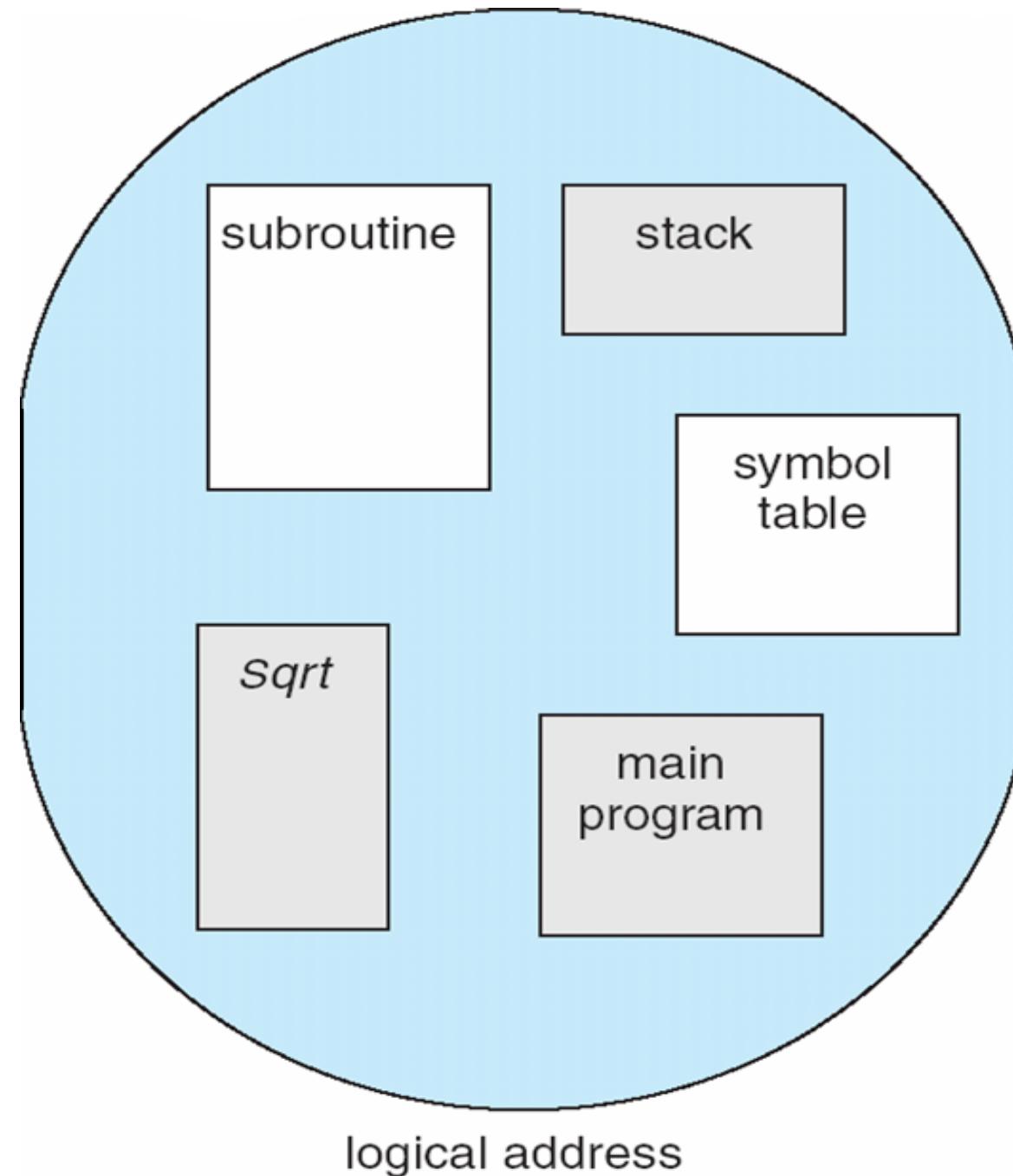
Segmentation

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



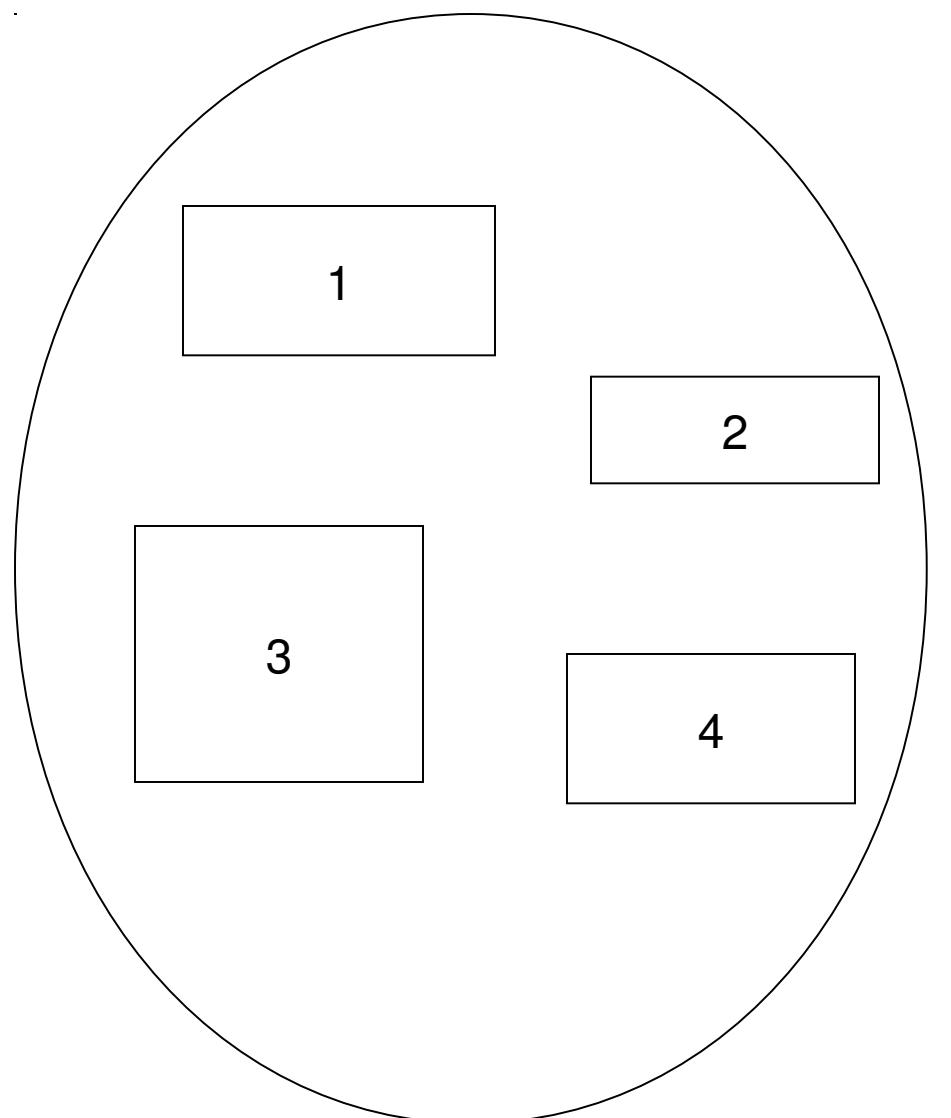


User's View of a Program

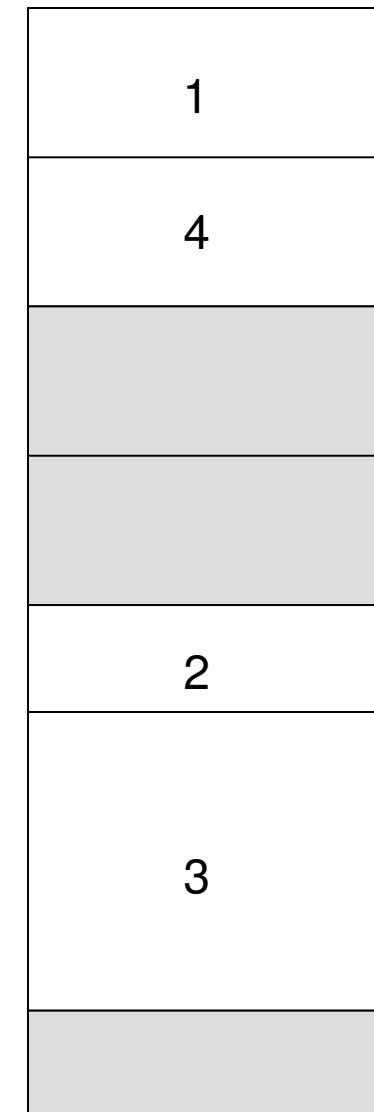




Logical View of Segmentation



user space



physical memory space





Segmentation Architecture

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





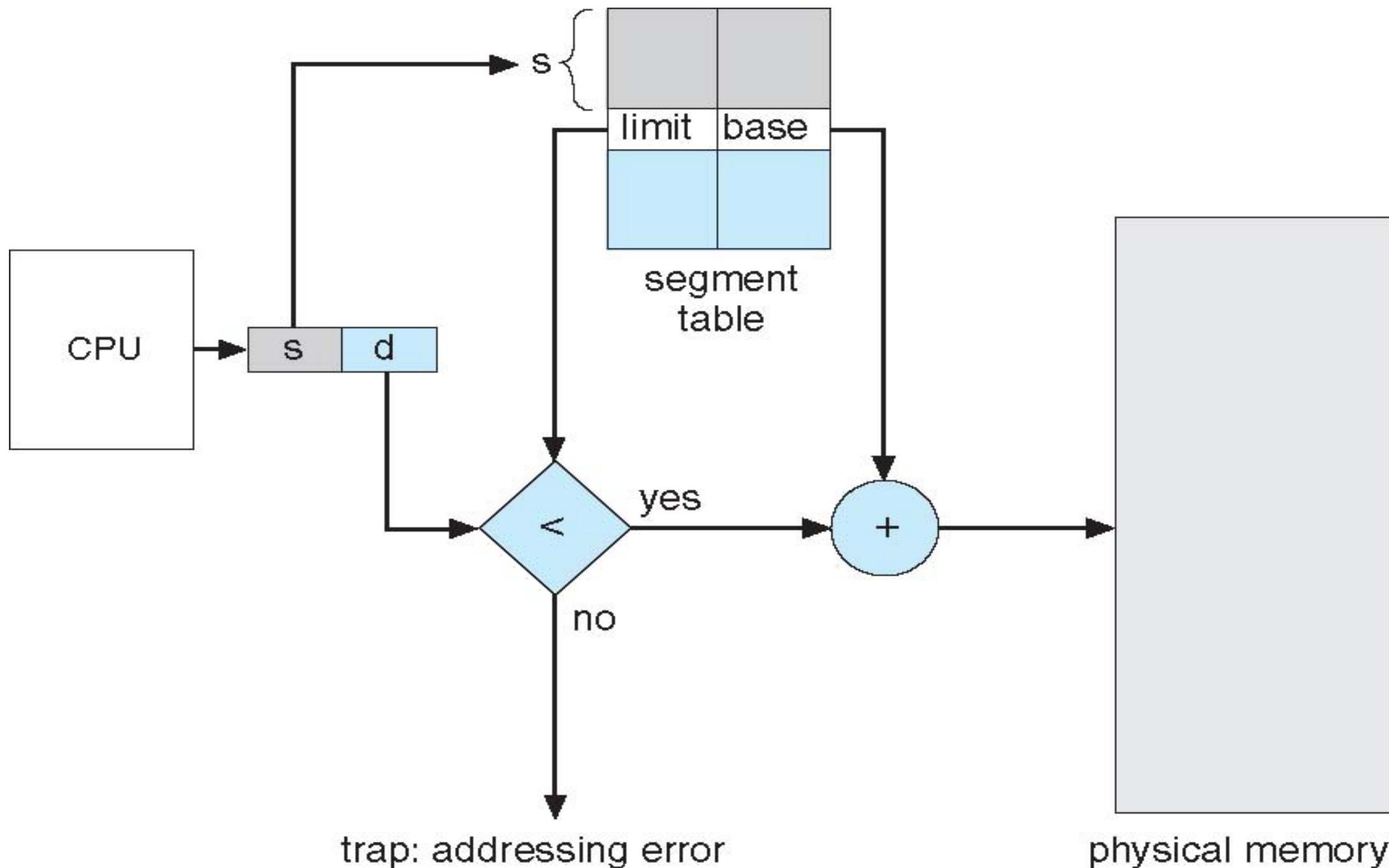
Segmentation Architecture (Cont.)

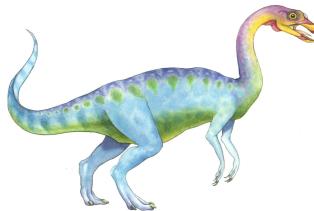
- Protection
 - With each entry in segment table associate:
 - ▶ validation bit = 0 \Rightarrow illegal segment
 - ▶ read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation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





Segmentation Hardware

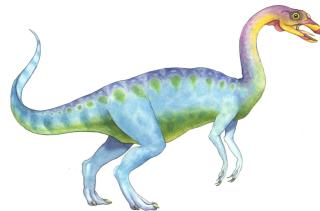




Paging

- 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
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

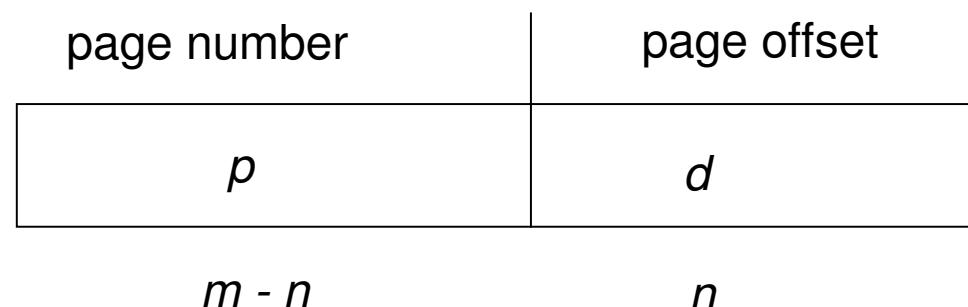




Address Translation Scheme

- 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

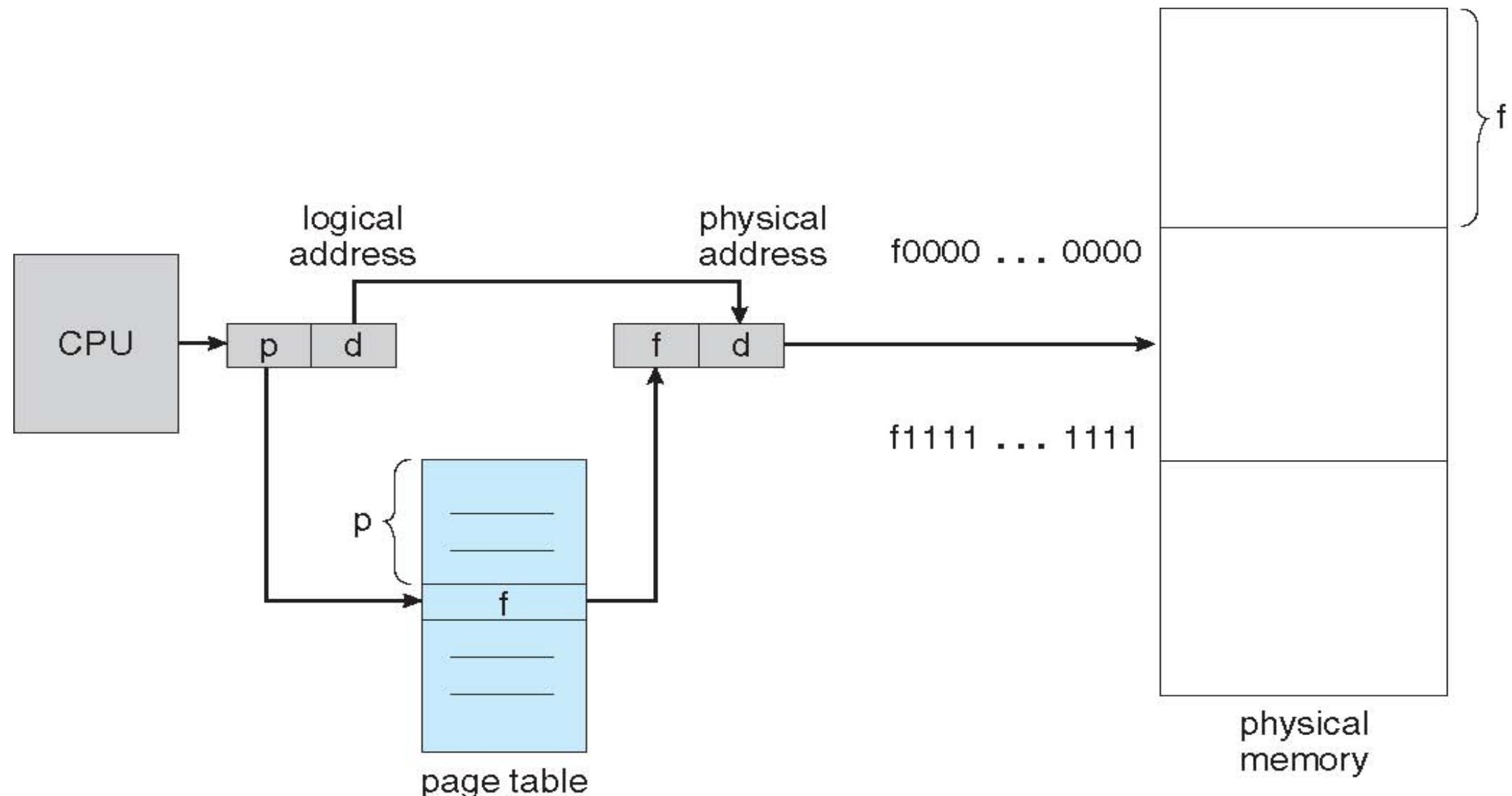


- For given logical address space 2^m and page size 2^n



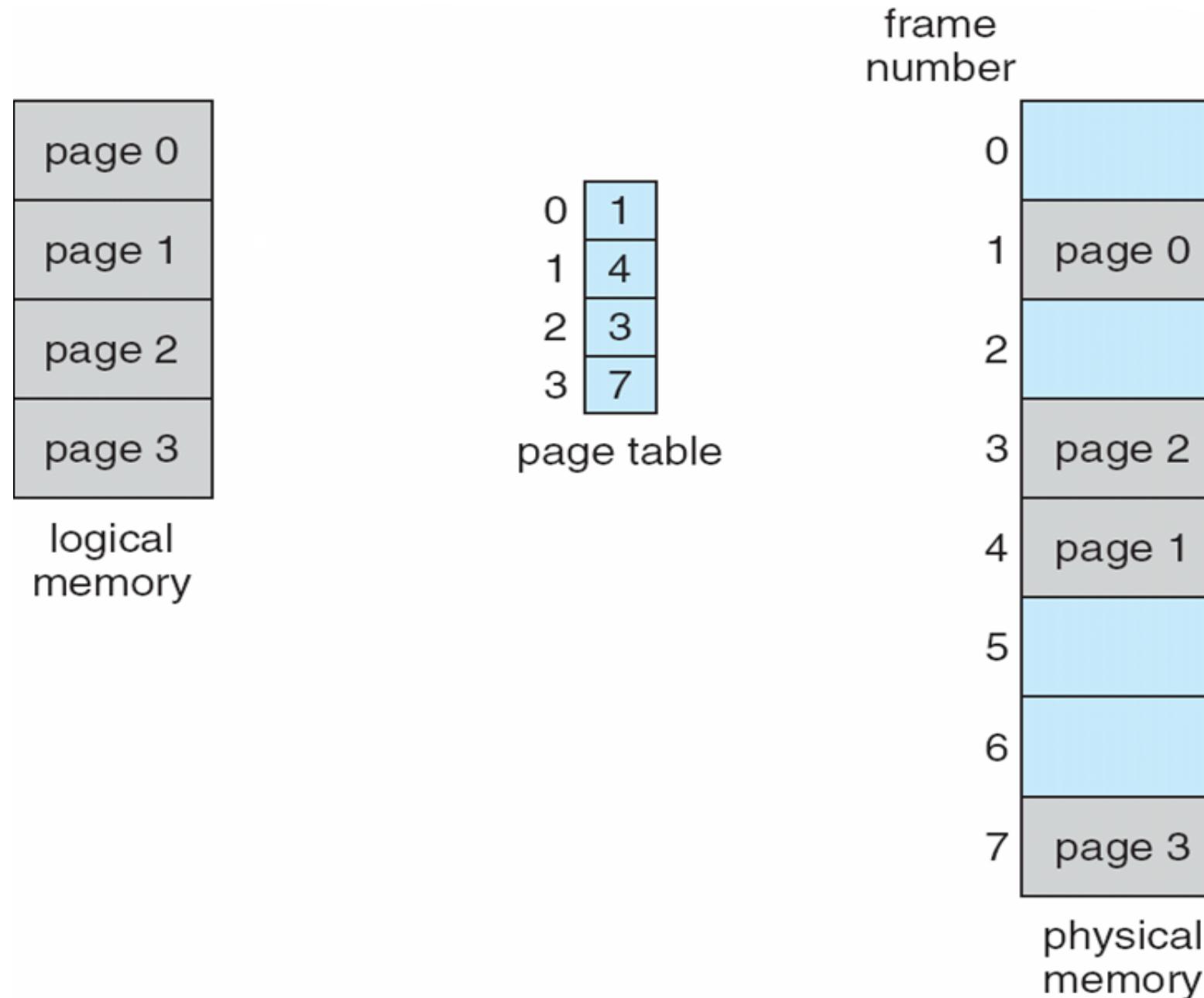


Paging Hardware





Paging Model of Logical and Physical Memory





Paging Example

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

logical memory

0	5
1	6
2	1
3	2

page table

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

physical memory

$n=2$ and $m=4$ 32-byte memory and 4-byte pages





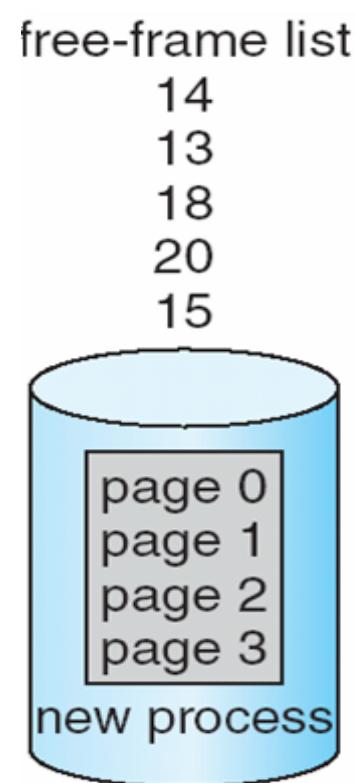
Paging (Cont.)

- 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
- By implementation process can only access its own memory



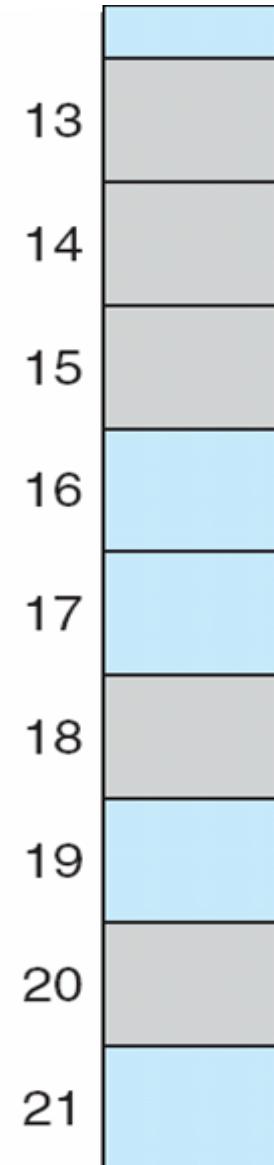


Free Frames



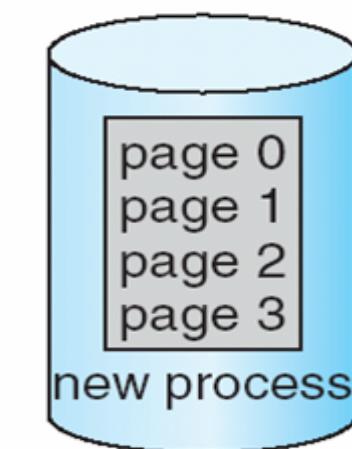
(a)

Before allocation



free-frame list

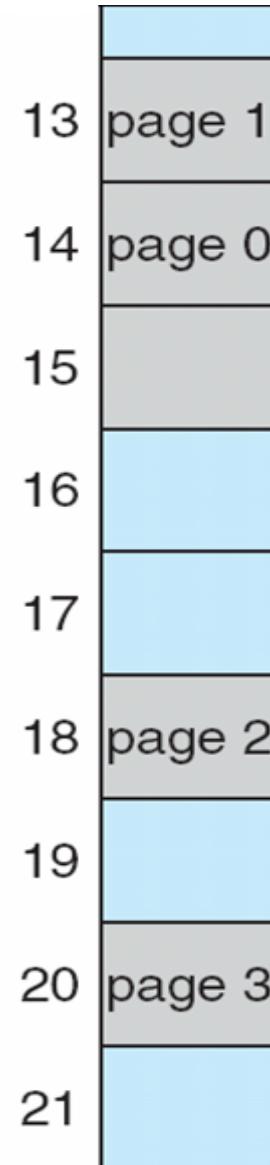
- 15



new-process page table

(b)

After allocation





Implementation of Page Table

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





Associative Memory

- Associative memory – parallel search

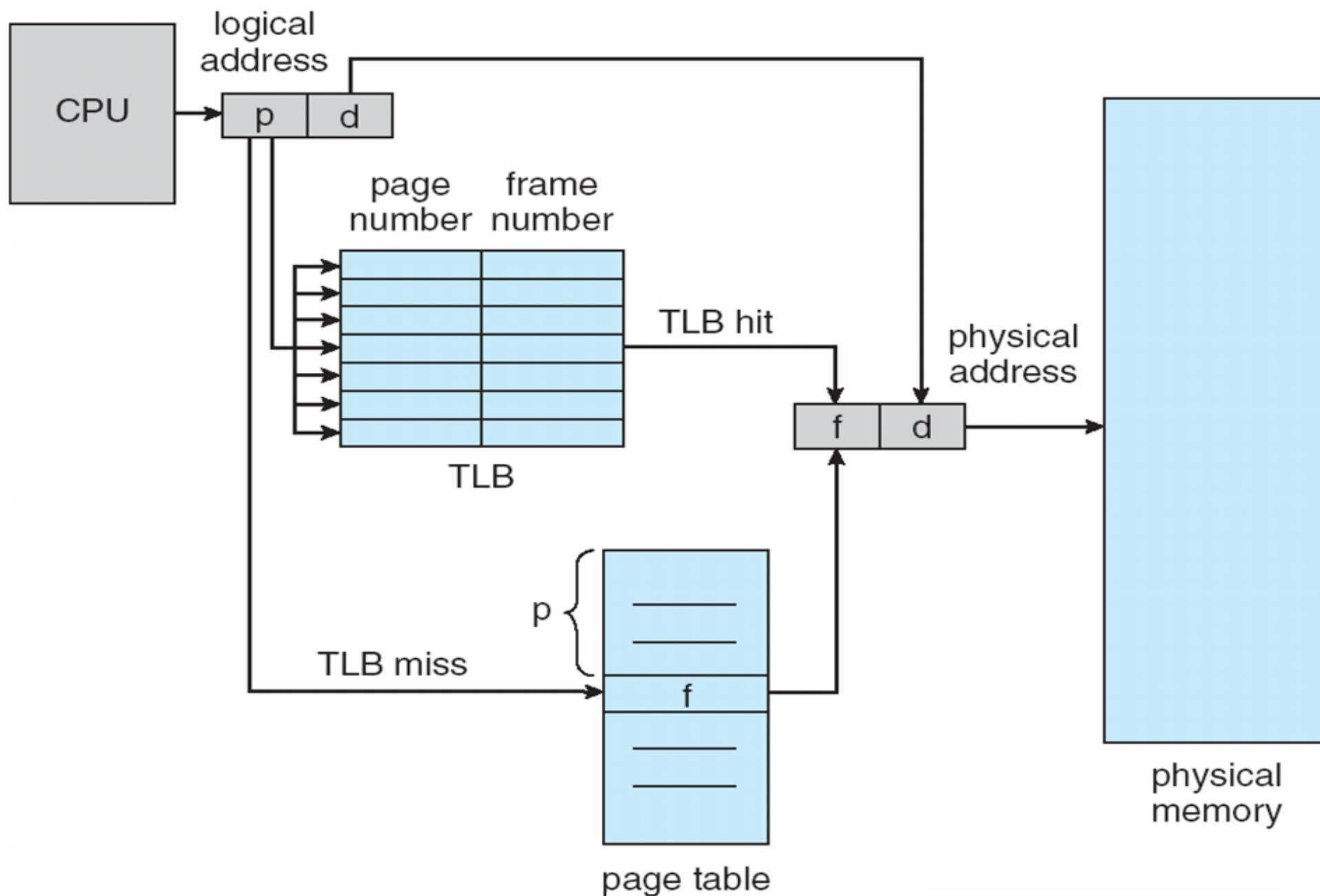
Page #	Frame #

- 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; ratio related to number of associative registers
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
- **Effective Access Time (EAT)**
$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
 - $\text{EAT} = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns}$
- Consider more realistic hit ratio $\rightarrow \alpha = 99\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
 - $\text{EAT} = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns}$





Memory Protection

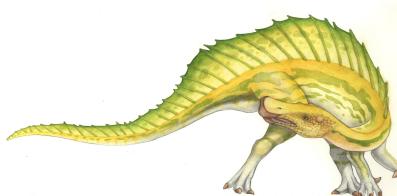
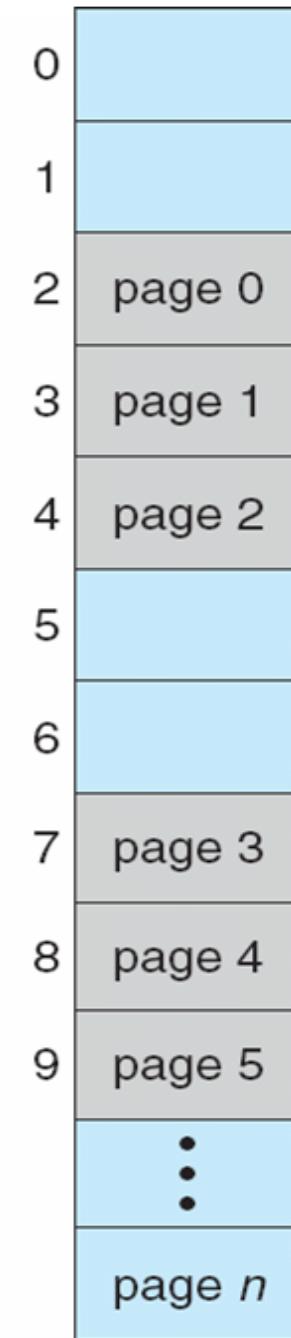
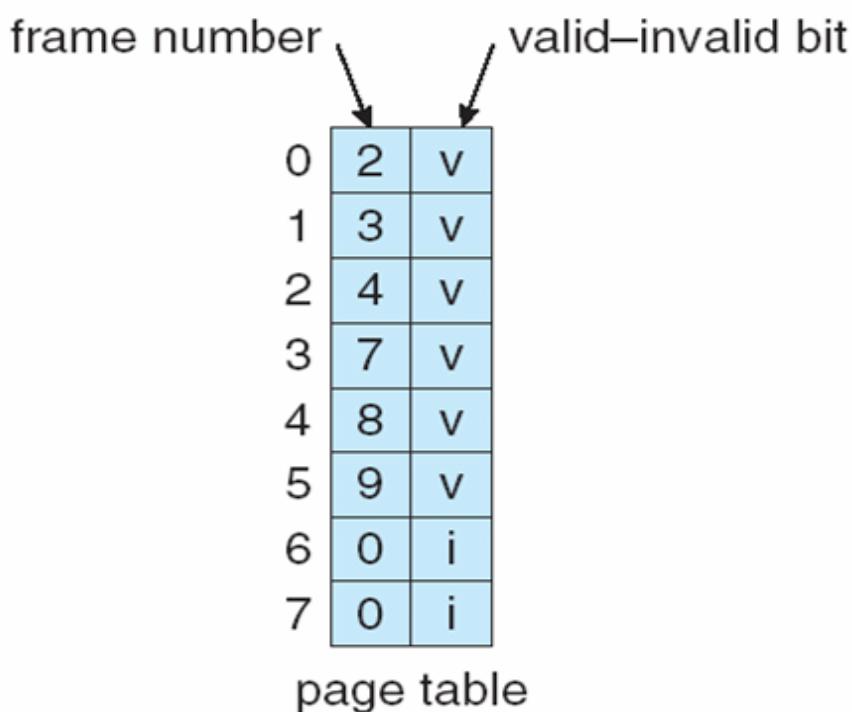
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated 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

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





Shared Pages

■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- 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

ed 1
ed 2
ed 3
data 1

process P_1

3
4
6
1

page table
for P_1

ed 1
ed 2
ed 3
data 3

process P_3

3
4
6
2

page table
for P_3

ed 1
ed 2
ed 3
data 2

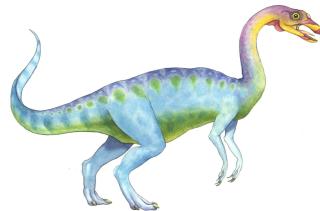
process P_2

3
4
6
7

page table
for P_2

0
1
2
3
4
5
6
7
8
9
10
11





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





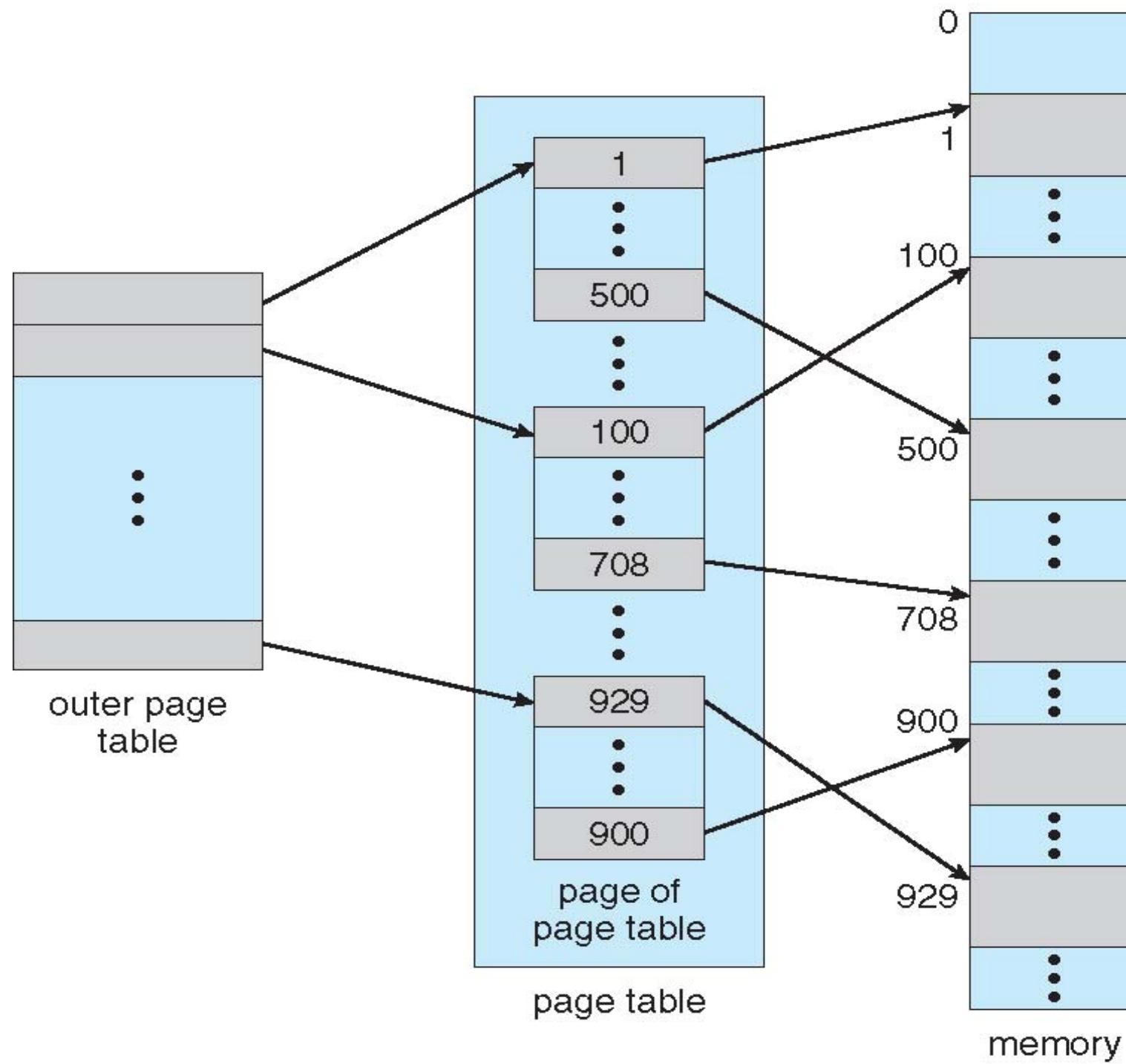
Hierarchical Page Tables

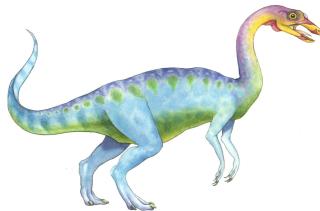
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table





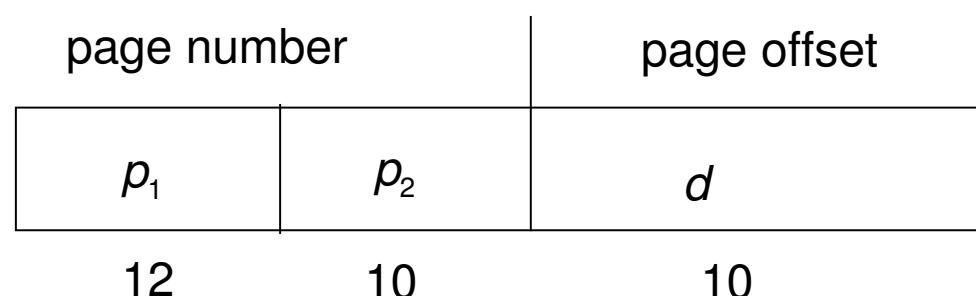
Two-Level Page-Table Scheme





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:

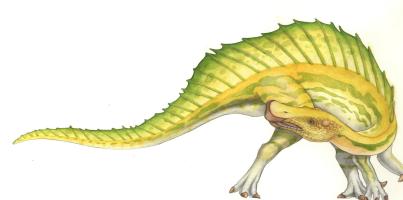
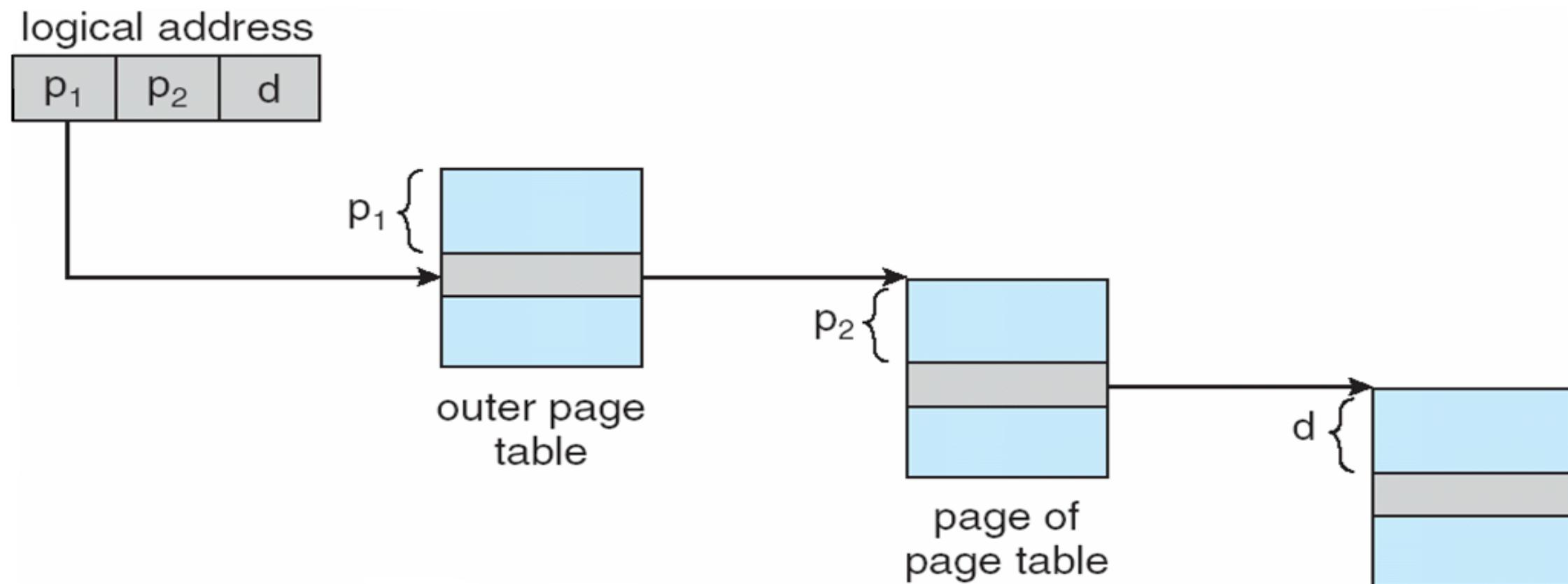


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**





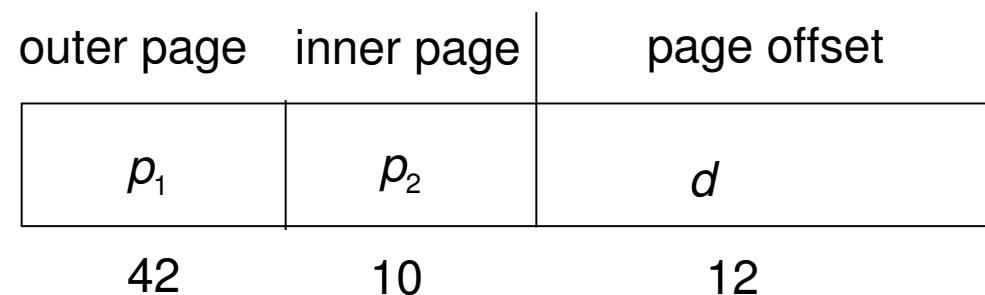
Address-Translation Scheme





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

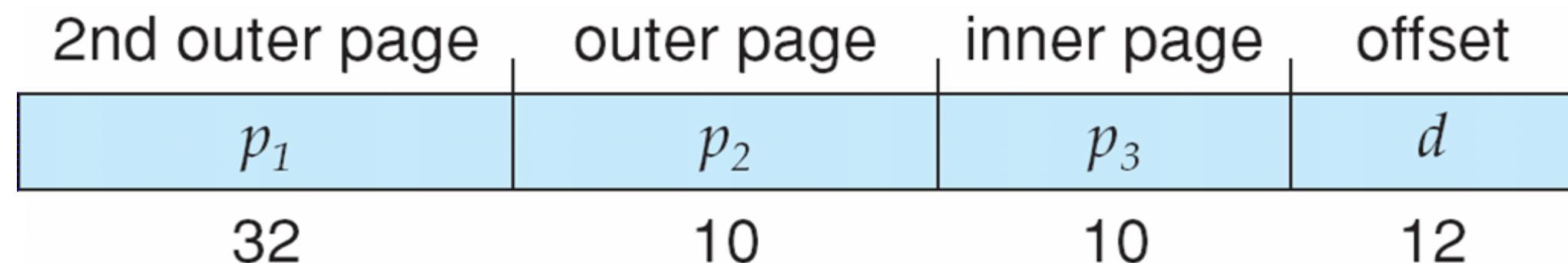
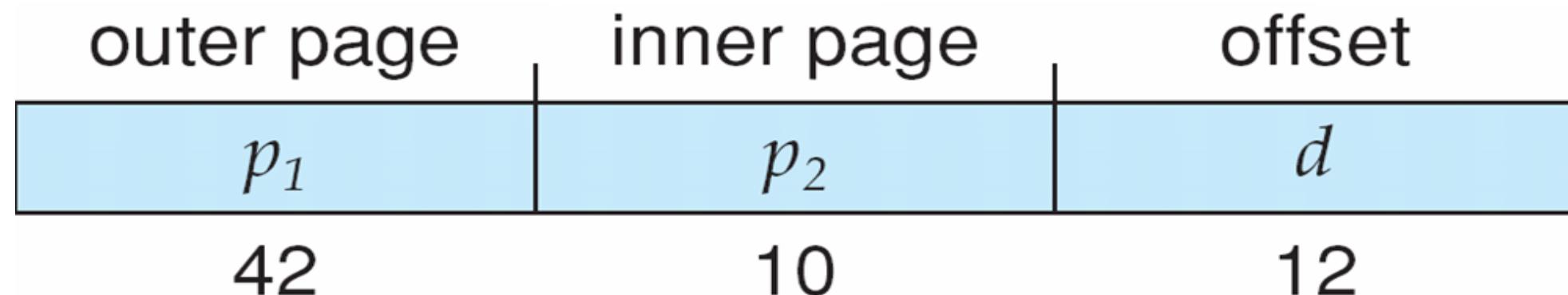


- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location





Three-level Paging Scheme





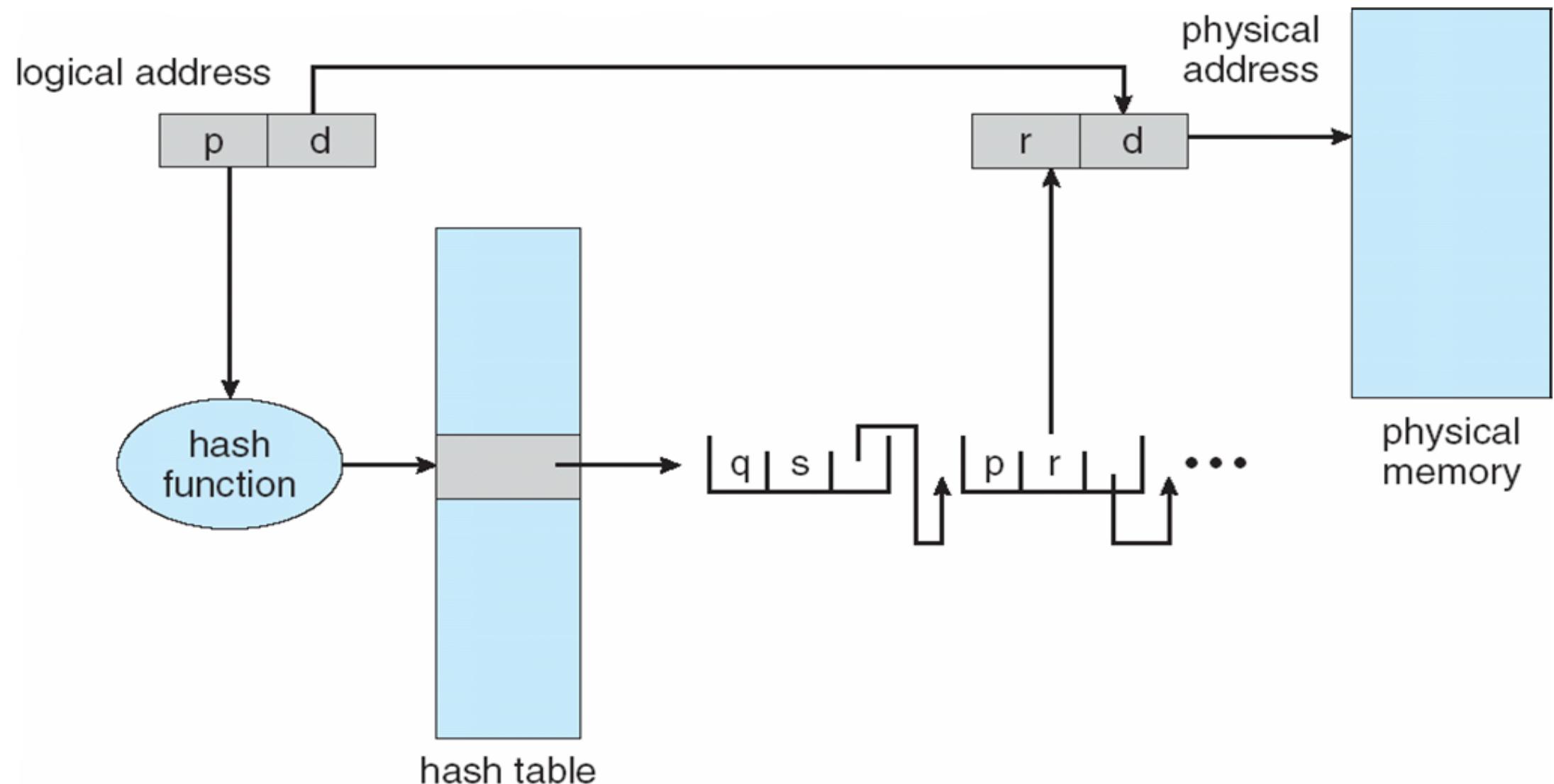
Hashed Page Tables

- 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)





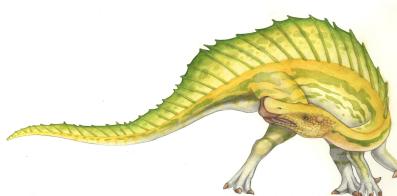
Hashed Page Table





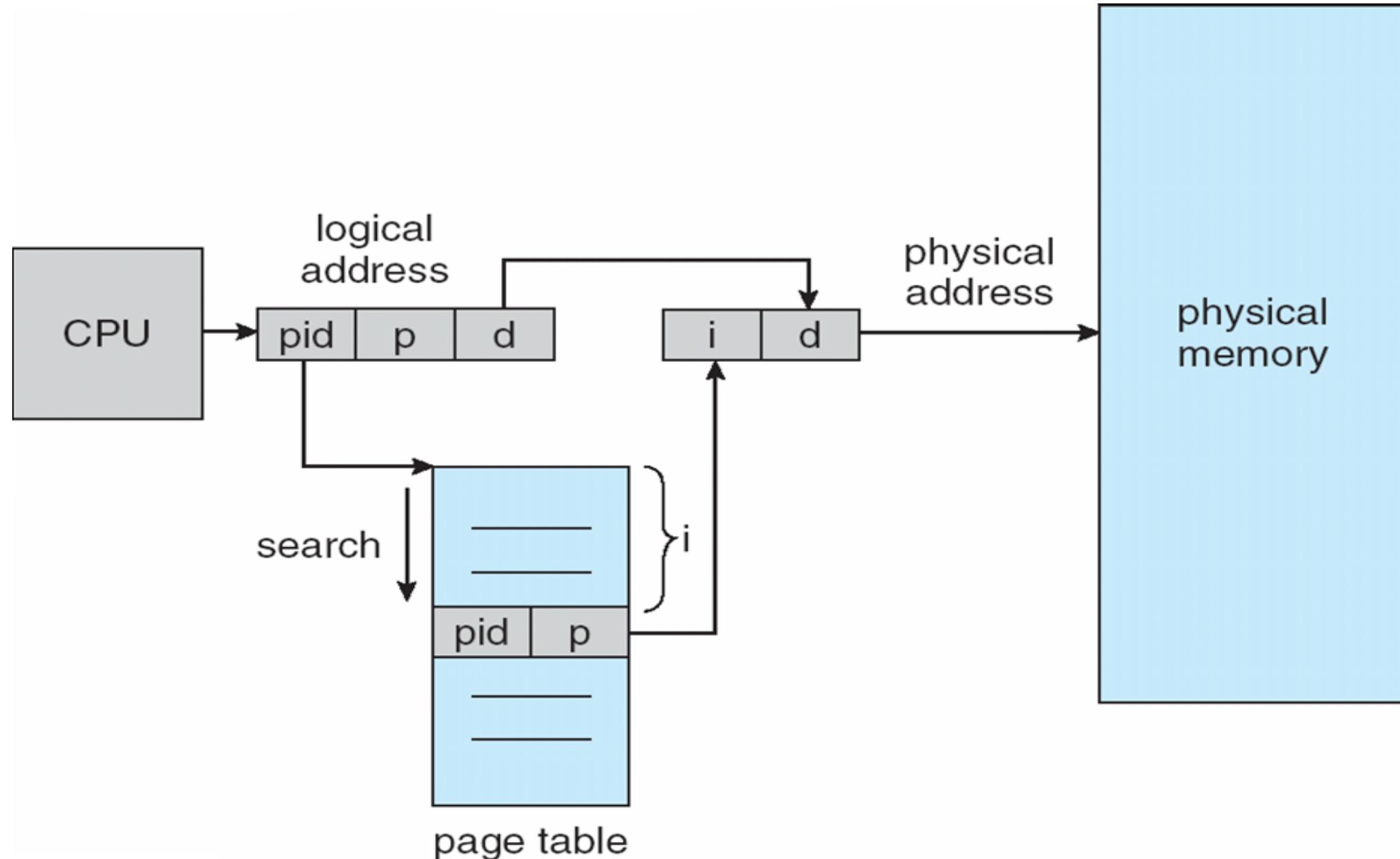
Inverted Page Table

- 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





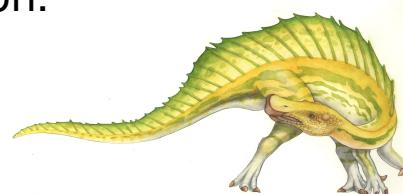
Inverted Page Table Architecture





Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
 - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
 - One kernel and one for all user processes
 - Each maps memory addresses from virtual to physical memory
 - Each entry represents a contiguous area of mapped virtual memory,
 - ▶ More efficient than having a separate hash-table entry for each page
 - Each entry has base address and span (indicating the number of pages the entry represents)
- TLB holds translation table entries (TTEs) for fast hardware lookups
 - A cache of TTEs reside in a translation storage buffer (TSB)
 - ▶ Includes an entry per recently accessed page
- Virtual address reference causes TLB search
 - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
 - ▶ If match found, the CPU copies the TSB entry into the TLB and translation completes
 - ▶ If no match found, kernel interrupted to search the hash table
 - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.





Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here



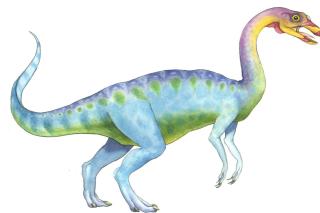


Example: The Intel IA-32 Architecture

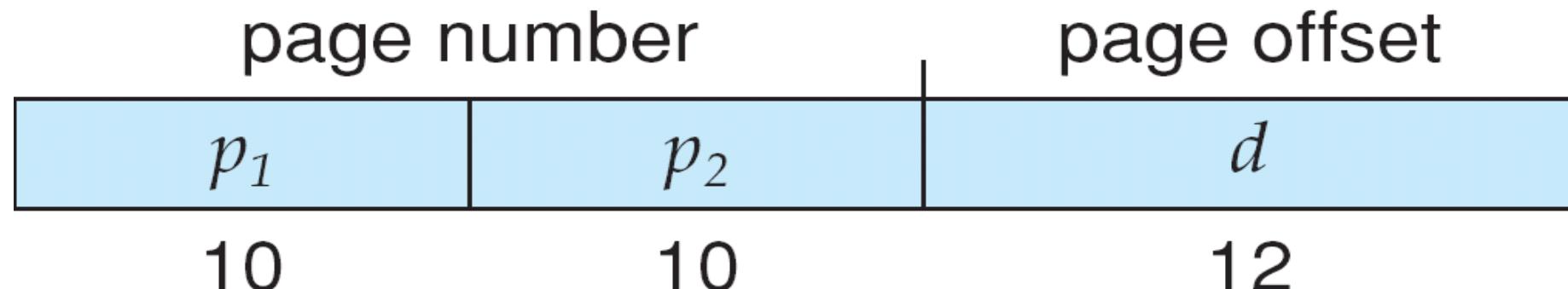
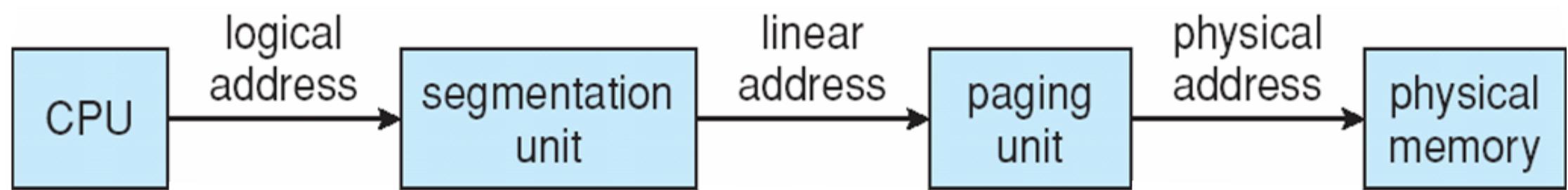
- Supports both segmentation and segmentation with paging
 - Each segment can be 4 GB
 - Up to 16 K segments per process
 - Divided into two partitions
 - ▶ First partition of up to 8 K segments are private to process (kept in **local descriptor table (LDT)**)
 - ▶ Second partition of up to 8K segments shared among all processes (kept in **global descriptor table (GDT)**)
- CPU generates logical address
 - Selector given to segmentation unit
 - ▶ Which produces linear addresses
- | | | |
|----------|----------|----------|
| <i>s</i> | <i>g</i> | <i>p</i> |
| 13 | 1 | 2 |

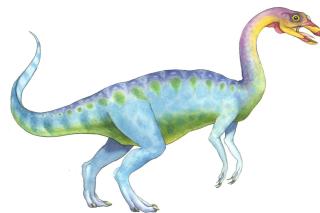
 : in main memory
 - ▶ Paging units form equivalent of MMU
 - ▶ Pages sizes can be 4 KB or 4 MB



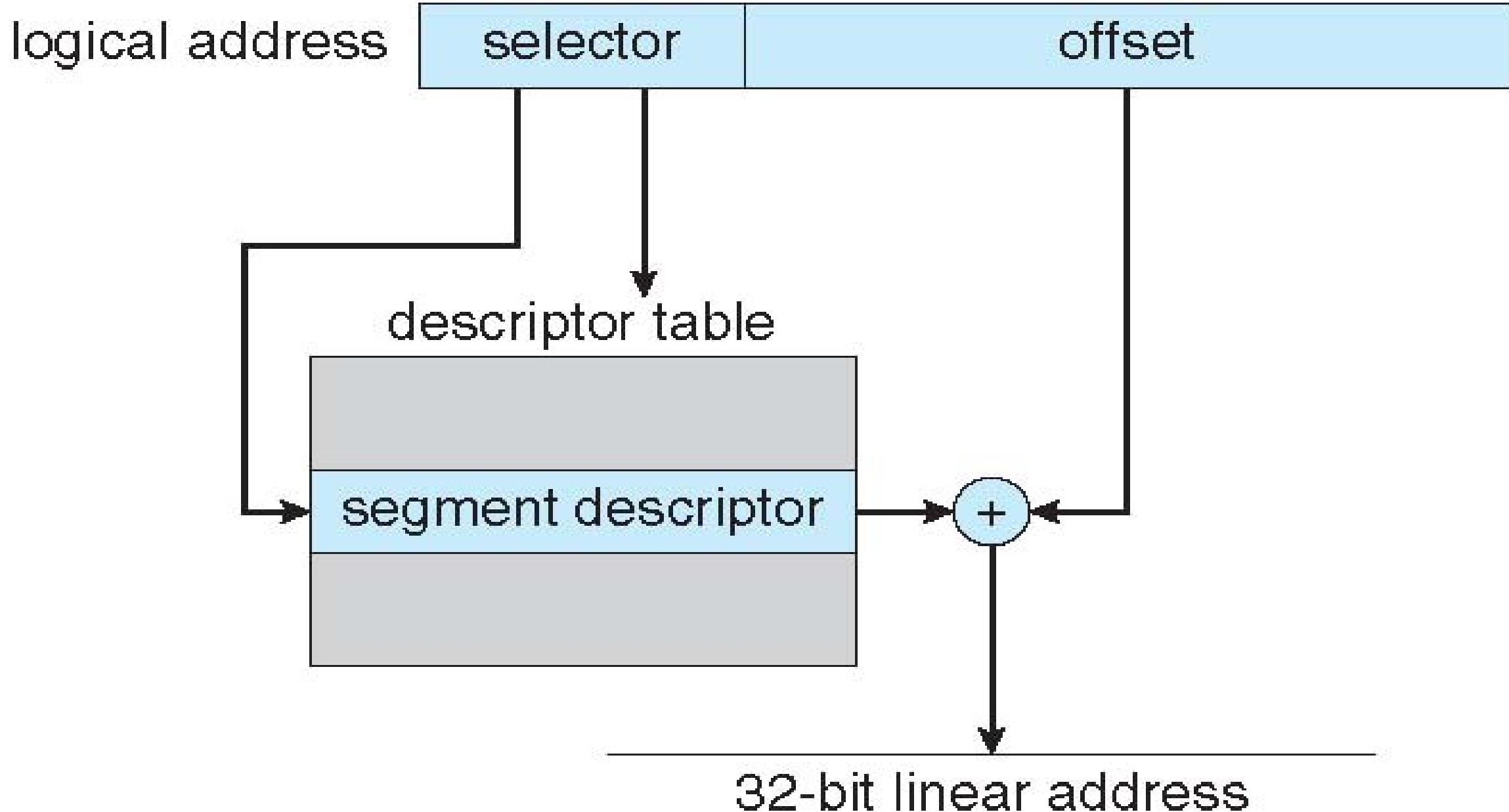


Logical to Physical Address Translation in IA-32



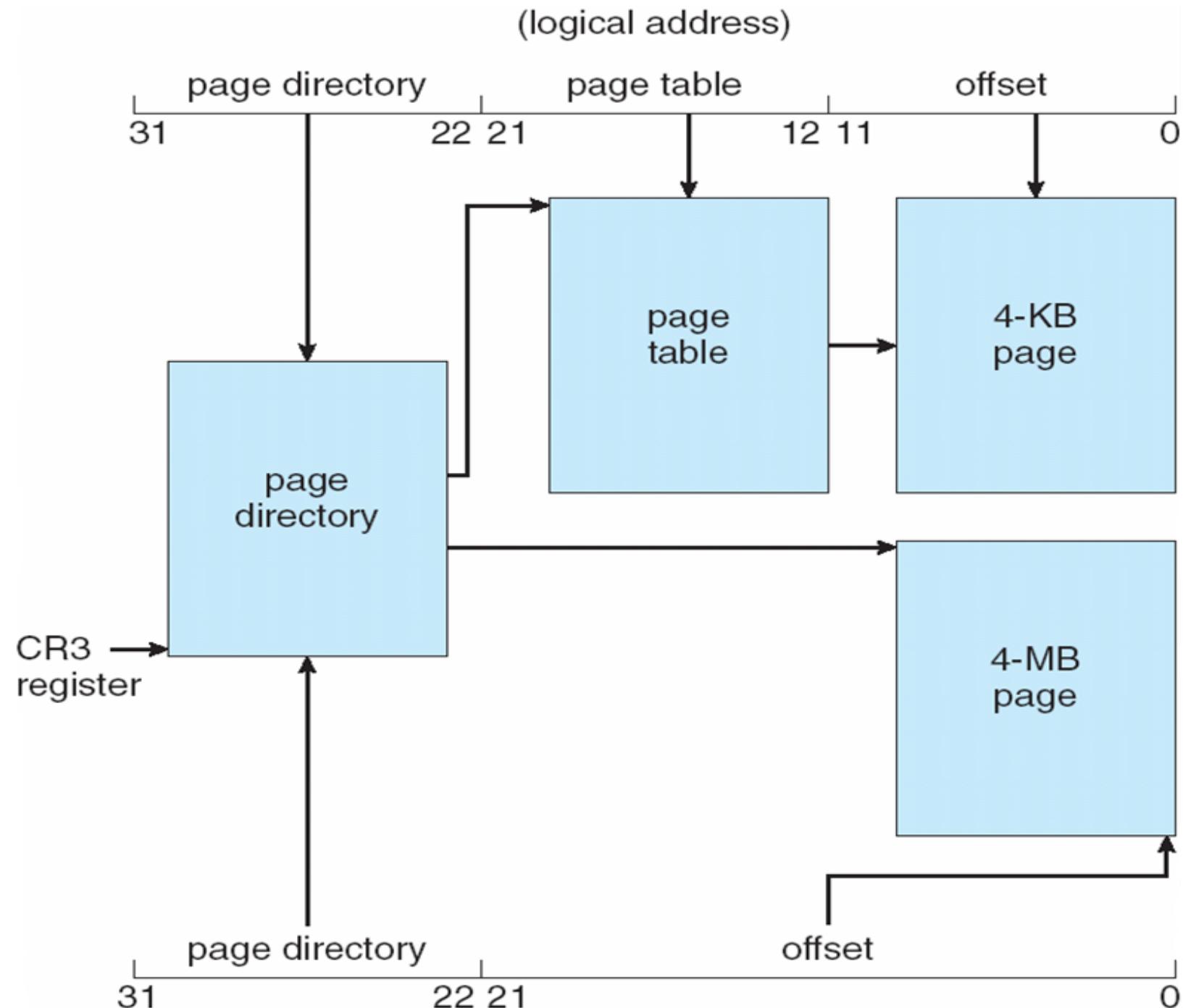


Intel IA-32 Segmentation





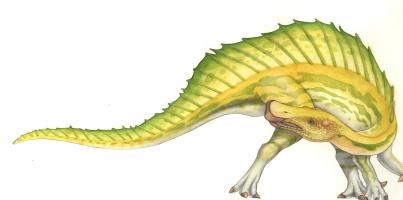
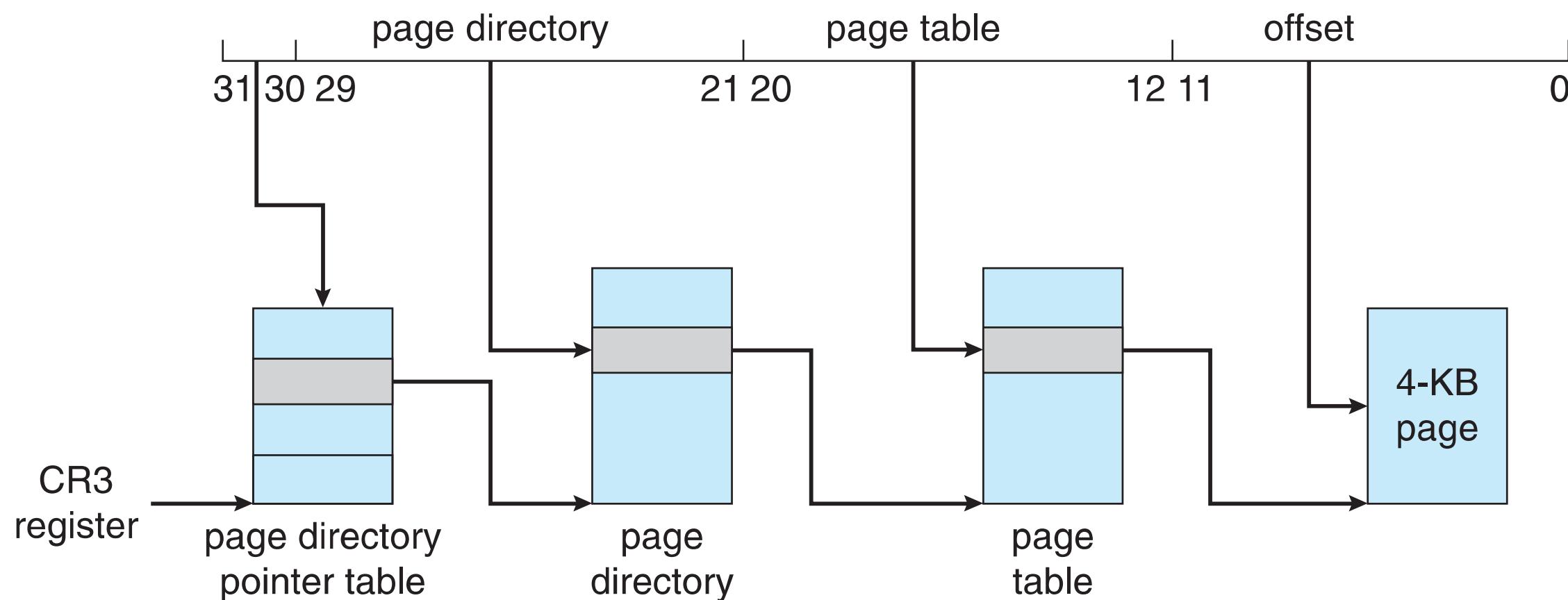
Intel IA-32 Paging Architecture

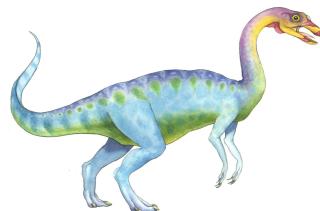




Intel IA-32 Page Address Extensions

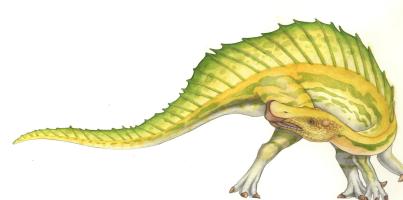
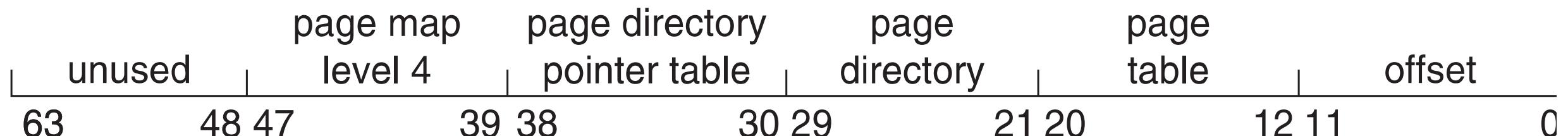
- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space
 - Paging went to a 3-level scheme
 - Top two bits refer to a **page directory pointer table**
 - Page-directory and page-table entries moved to 64-bits in size
 - Net effect is increasing address space to 36 bits – 64GB of physical memory





Intel x86-64

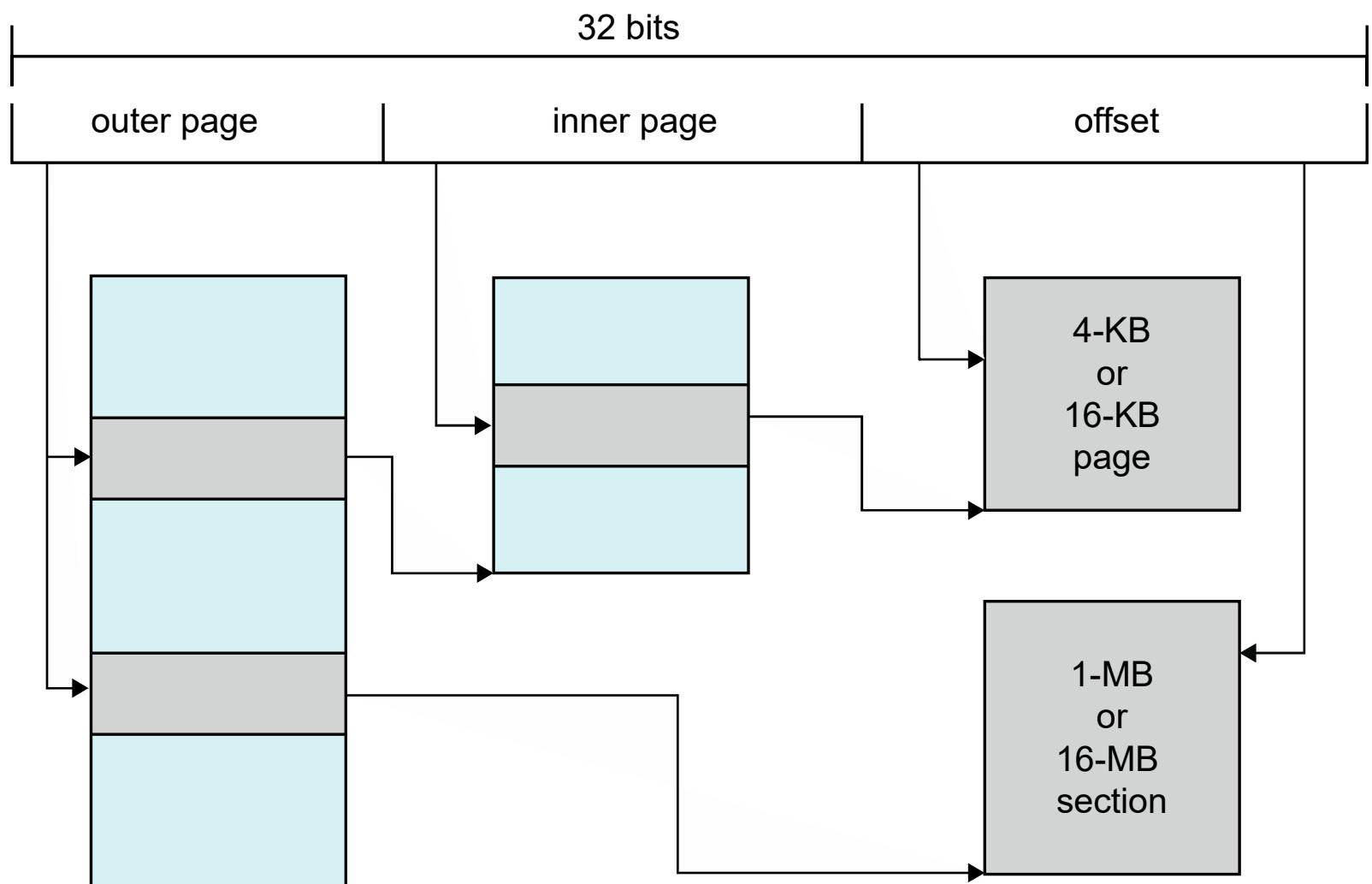
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
 - Page sizes of 4 KB, 2 MB, 1 GB
 - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



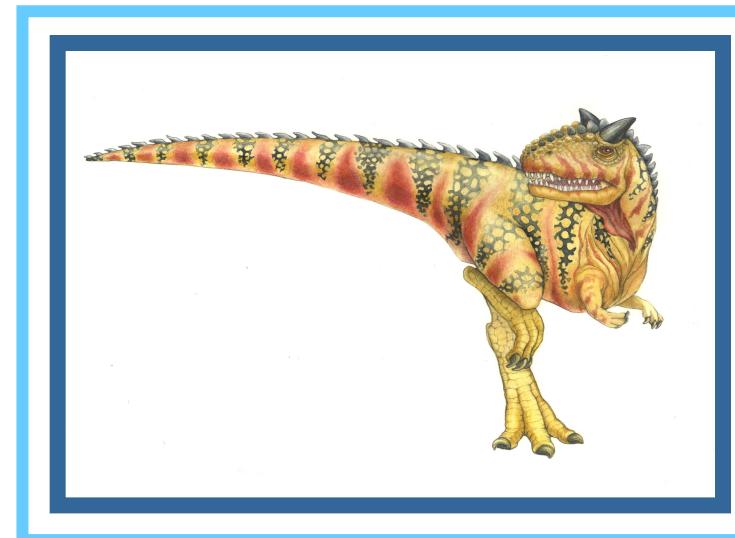


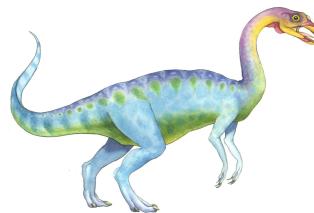
Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
 - Outer level has two micro TLBs (one data, one instruction)
 - Inner is single main TLB
 - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



Chapter 9: Virtual-Memory Management

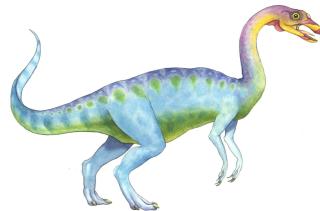




Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples





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





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
 - Program and programs could be larger than physical memory

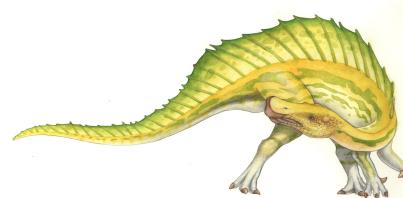




Background

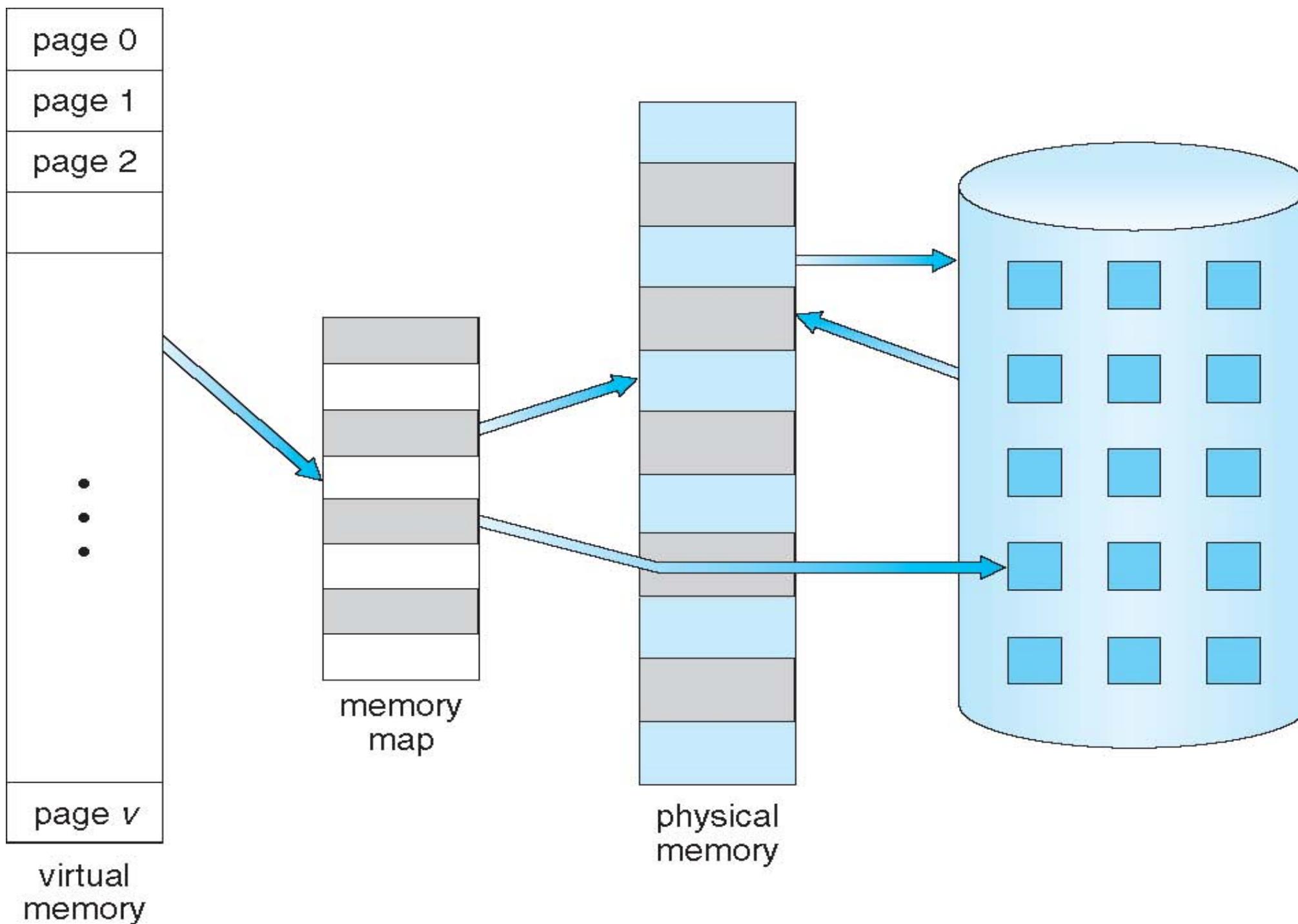
- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more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

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



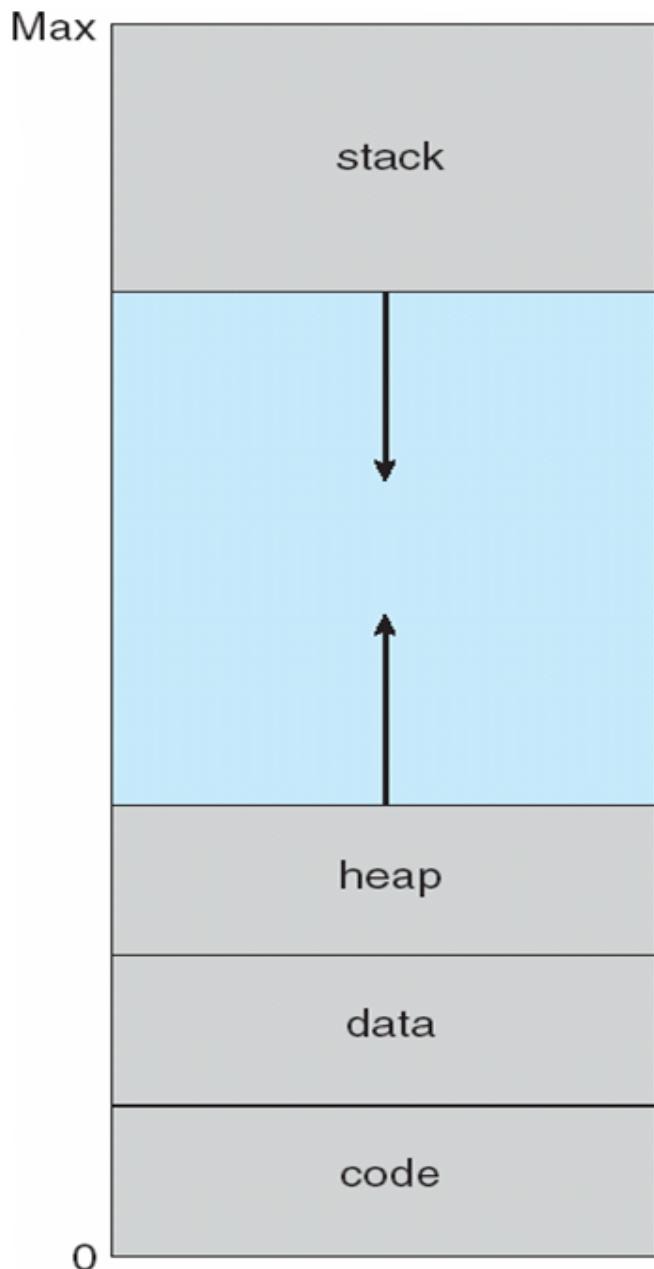


Virtual Memory That is Larger Than Physical Memory





Virtual-address Space

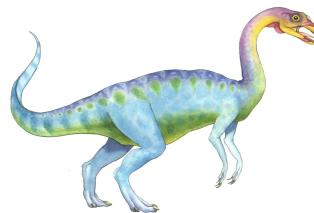




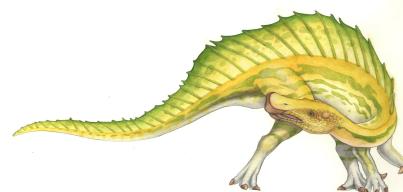
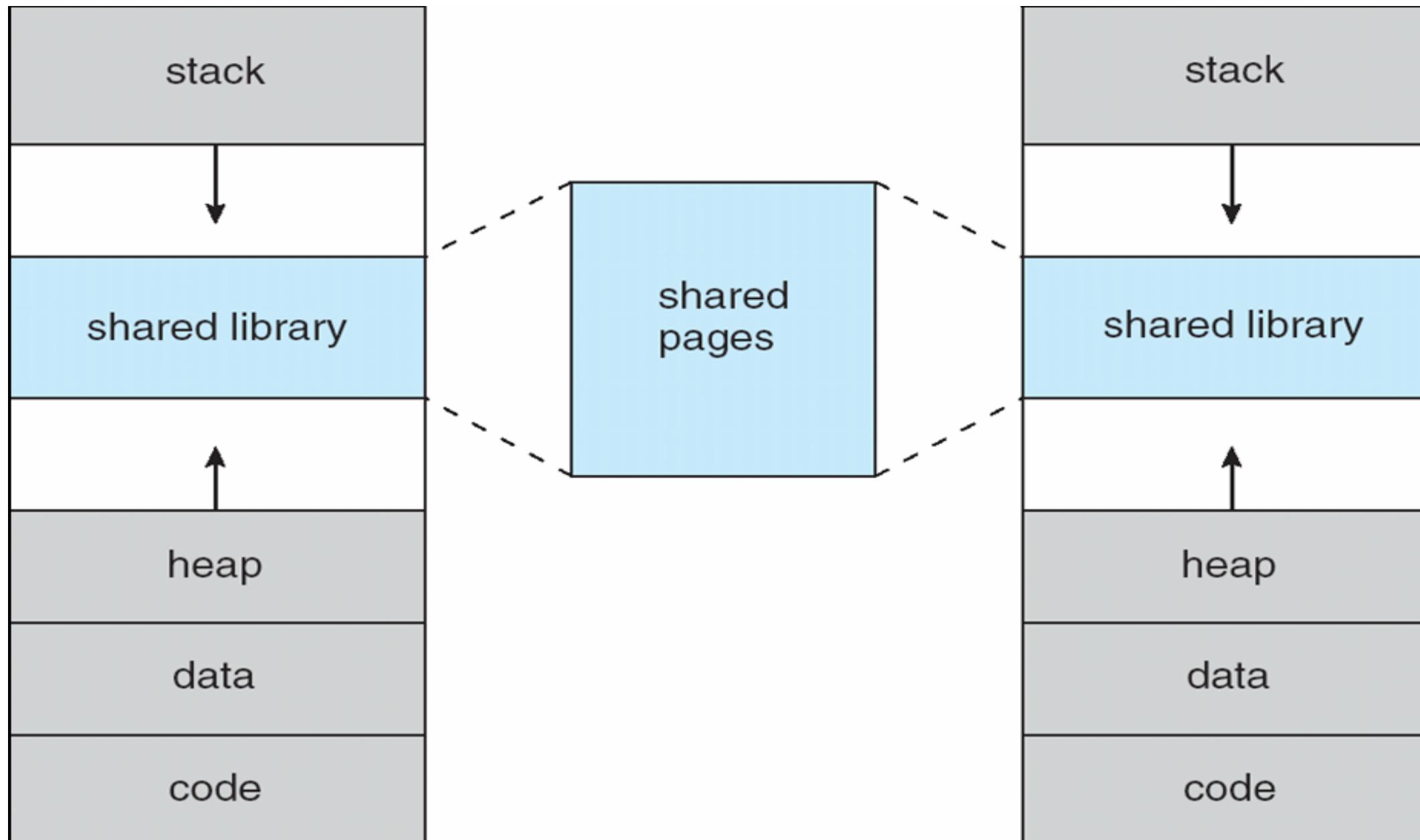
Virtual Address Space

- 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





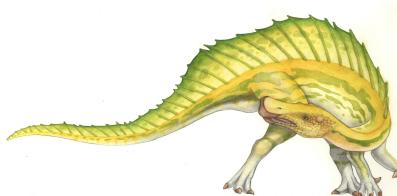
Shared Library Using Virtual Memory

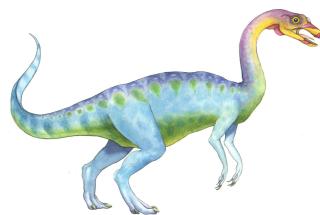




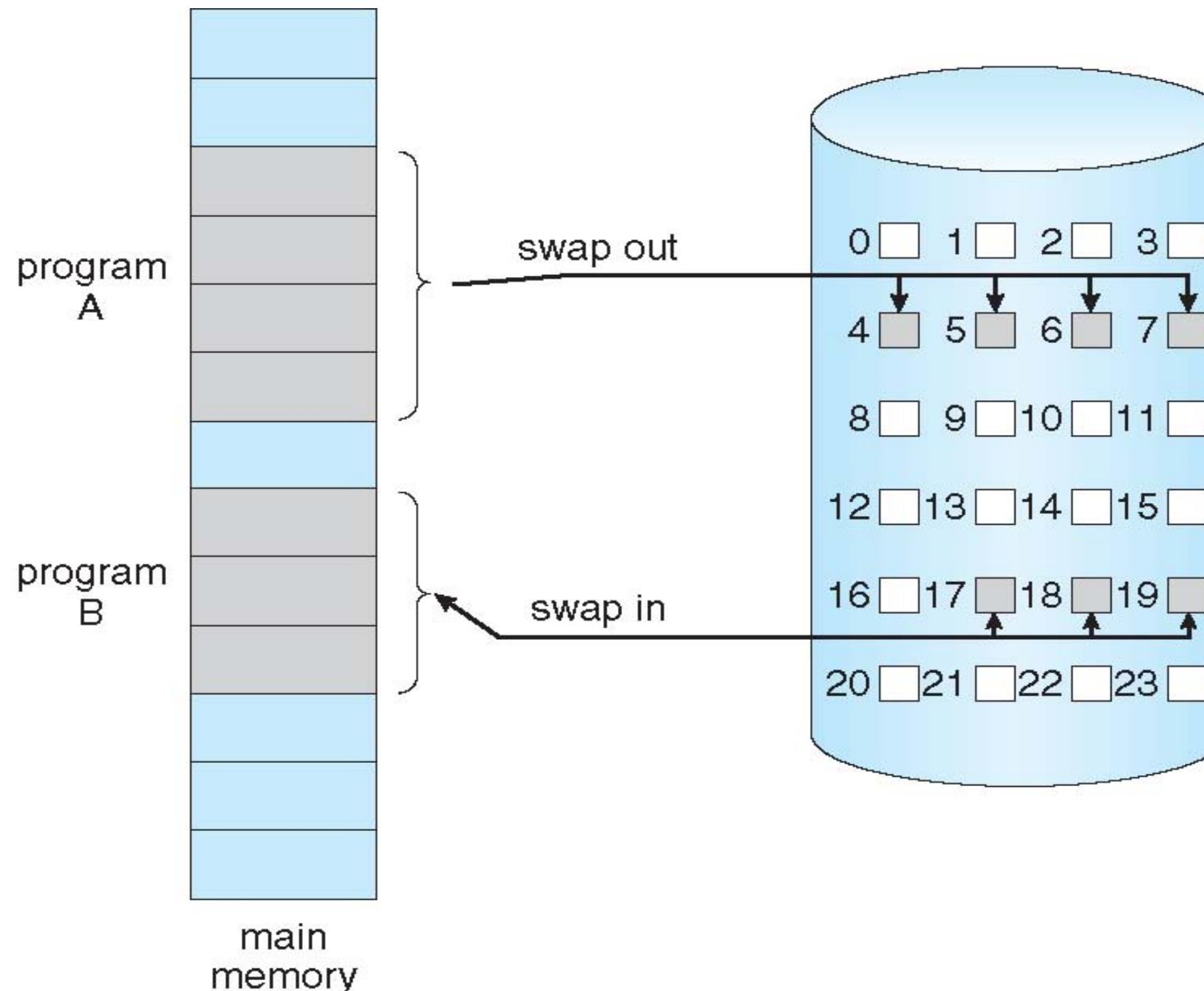
Demand Paging

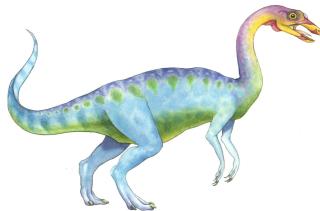
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Page is needed ⇒ reference to it
 - invalid reference ⇒ abort
 - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**





Transfer of a Paged Memory to Contiguous Disk Space





Valid-Invalid Bit

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

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

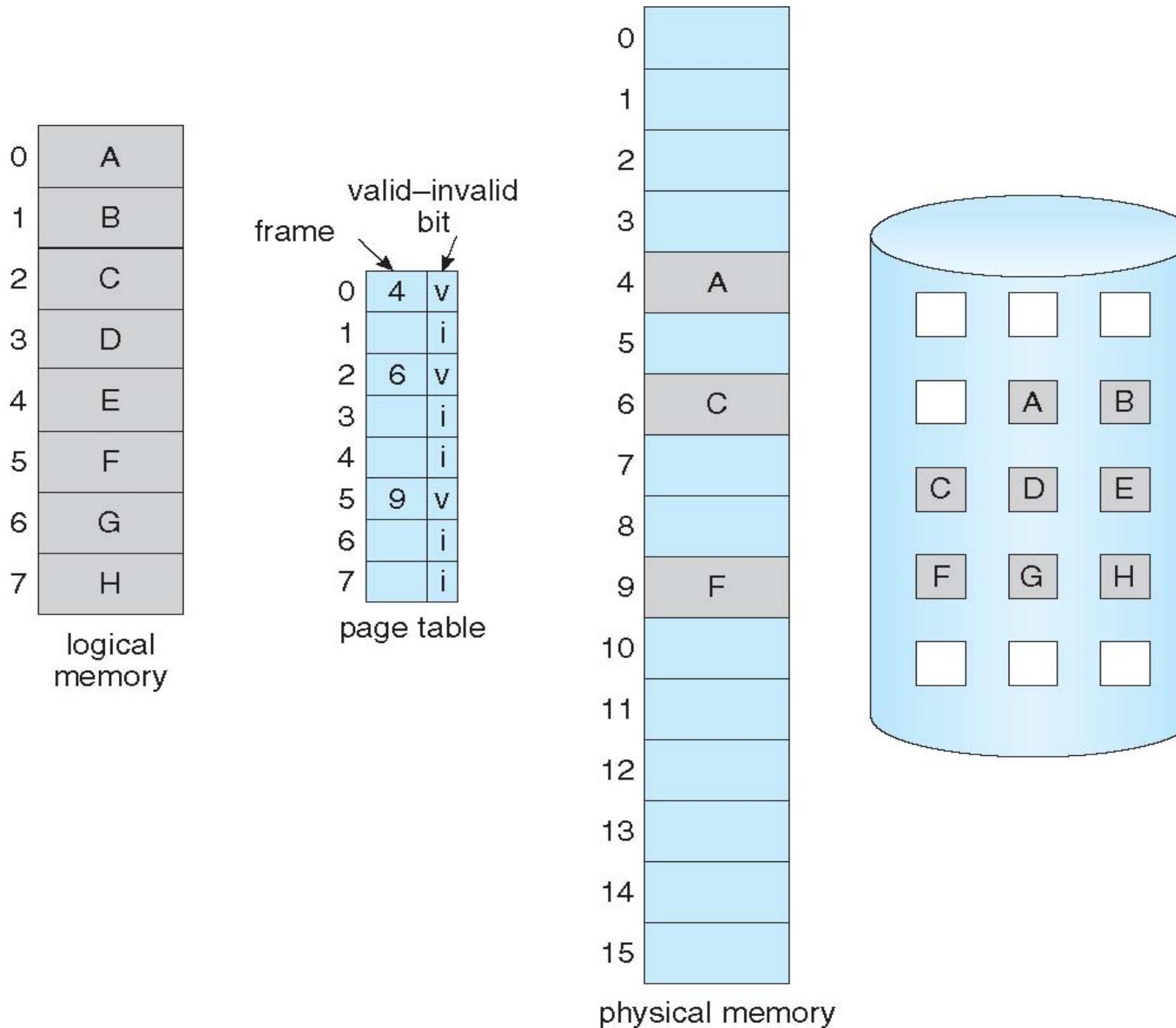
page table

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





Page Table When Some Pages Are Not in Main Memory





Page Fault

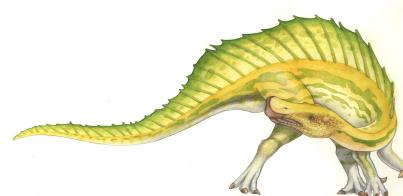
- If there is a reference to a page, first reference to that page will trap to operating system:
page fault
 1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
 2. Get empty frame
 3. Swap page into frame via scheduled disk operation
 4.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





Aspects of Demand Paging

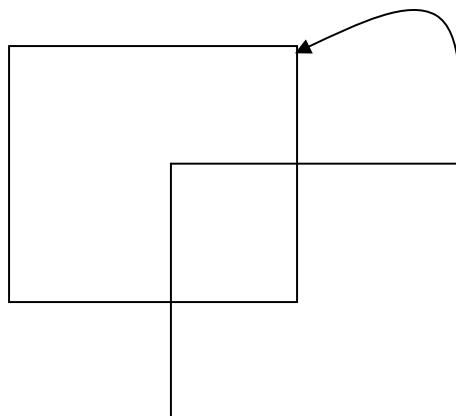
- Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with **swap space**)
 - Instruction restart





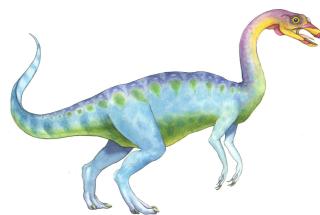
Instruction Restart

- Consider an instruction that could access several different locations
 - block move

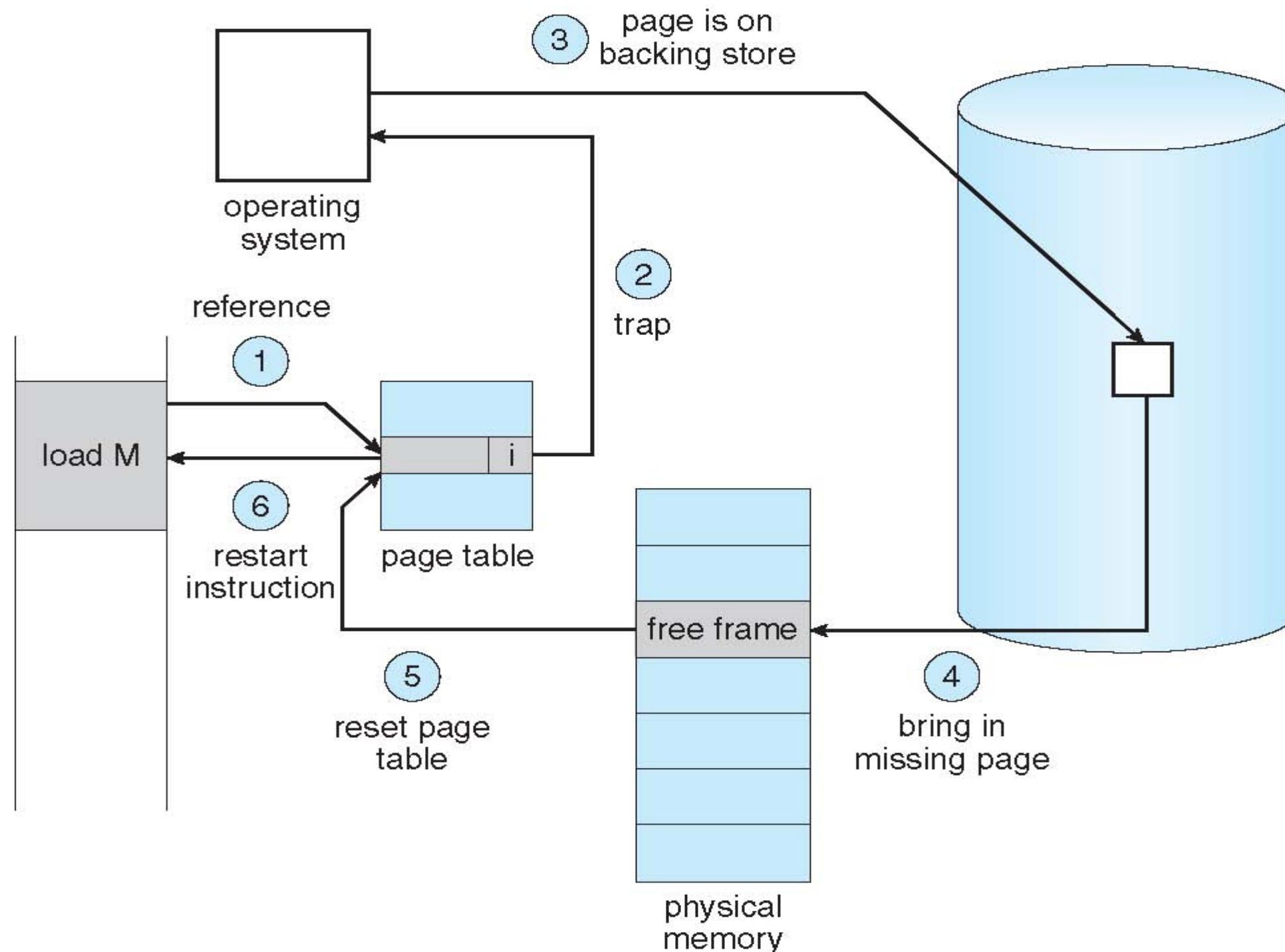


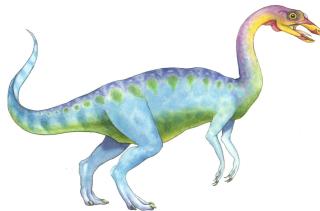
- auto increment/decrement location
- Restart the whole operation?
 - ▶ What if source and destination overlap?





Steps in Handling a Page Fault

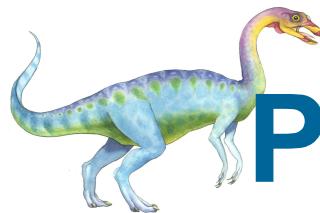




Performance of Demand Paging

- Stages in Demand Paging
 - 1. Trap to the operating system
 - 2. Save the user registers and process state
 - 3. Determine that the interrupt was a page fault
 - 4. Check that the page reference was legal and determine the location of the page on the disk
 - 5. Issue a read from the disk to a free frame:
 - 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.)

- 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)

$$\begin{aligned} EAT = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead} \\ &) \end{aligned}$$





Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p \text{ (8 milliseconds)} \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then
 $\text{EAT} = 8.2 \text{ microseconds.}$
This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
 - $$\begin{aligned} 220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p \end{aligned}$$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses





Demand Paging Optimizations

- 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





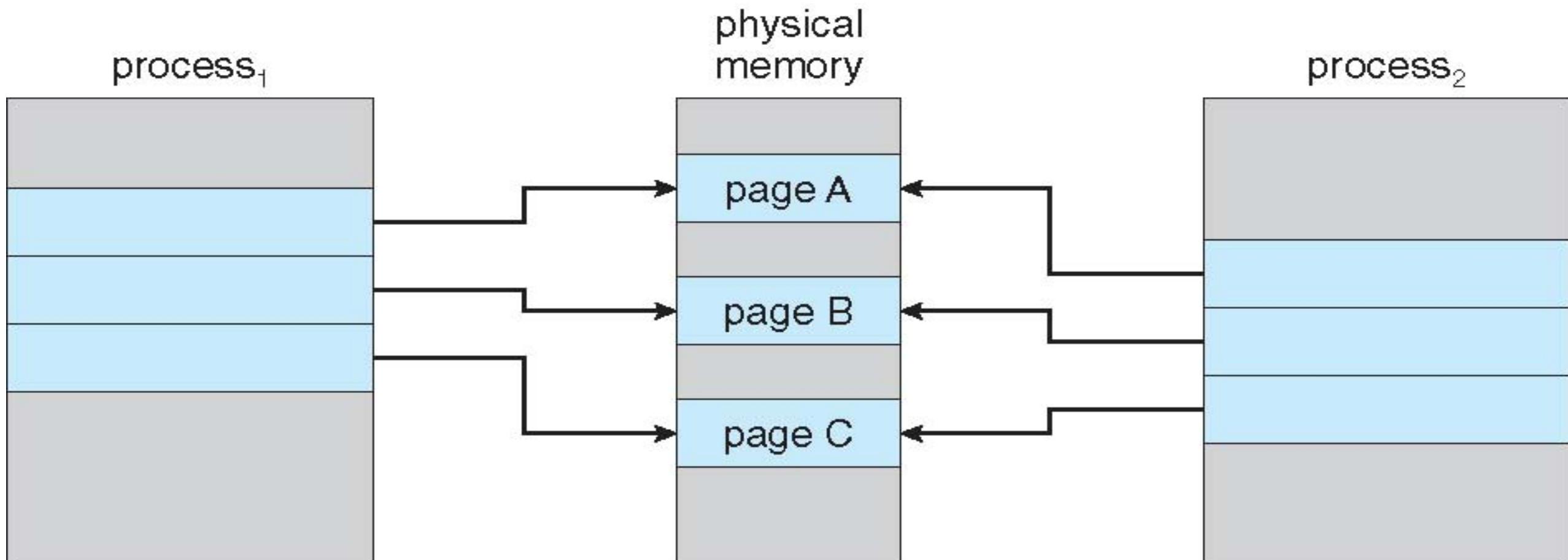
Copy-on-Write

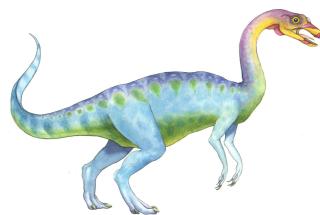
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
 - 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



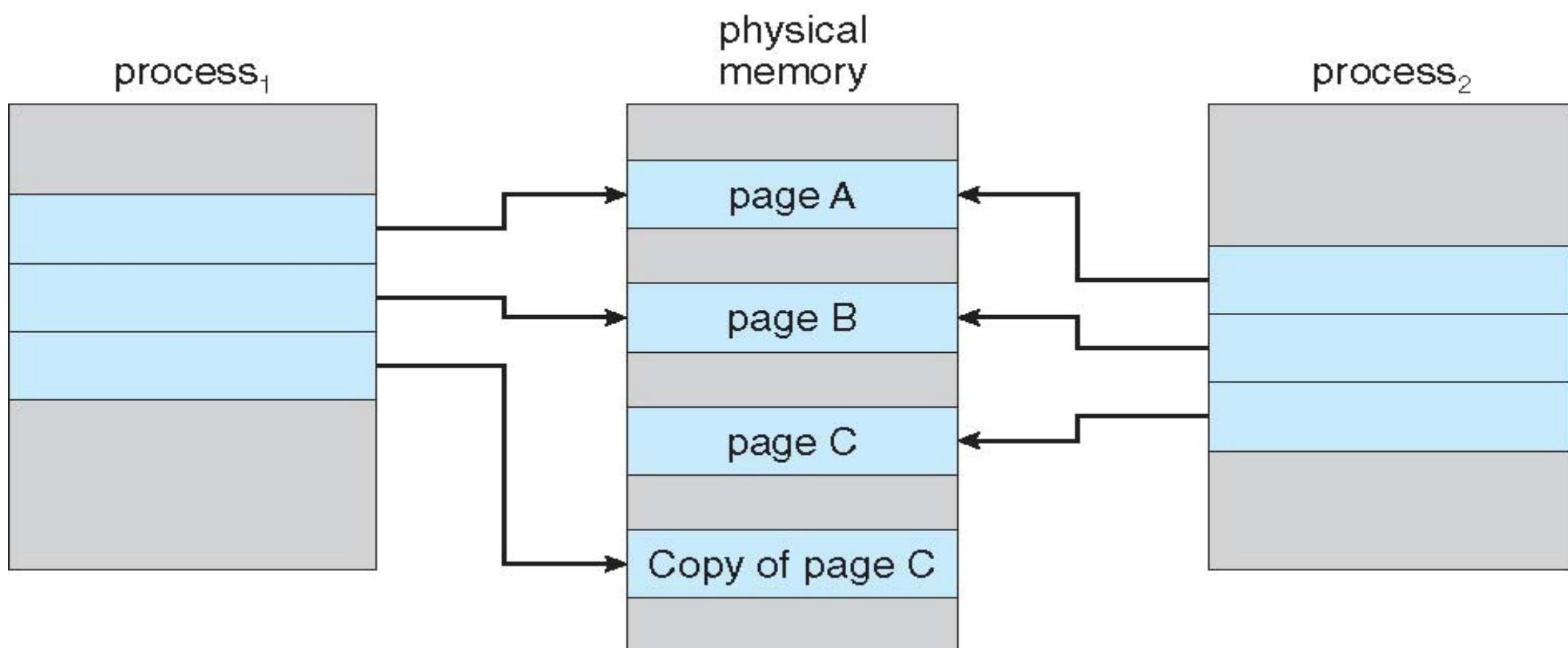


Before Process 1 Modifies Page C





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





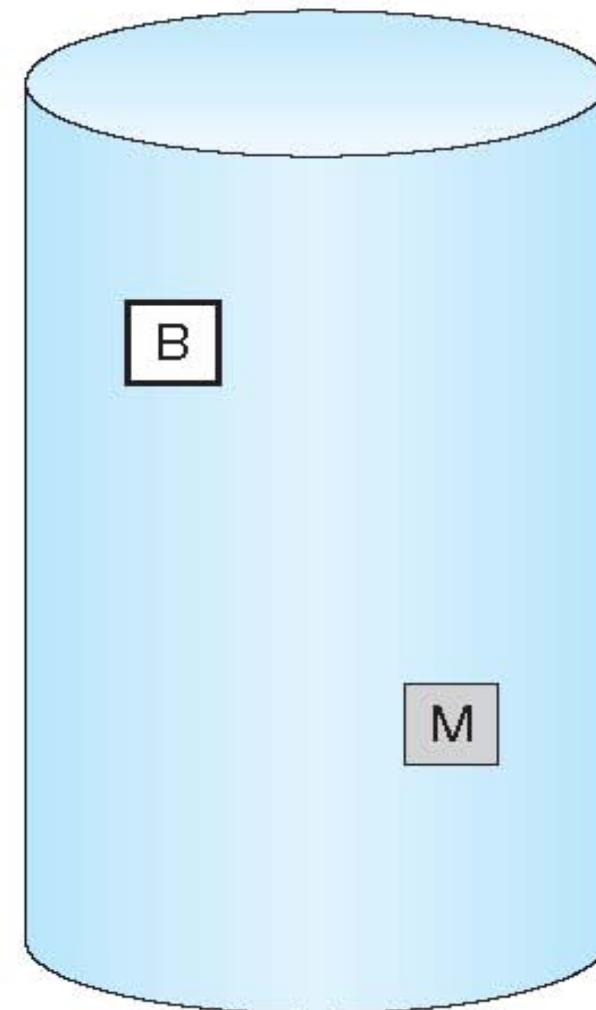
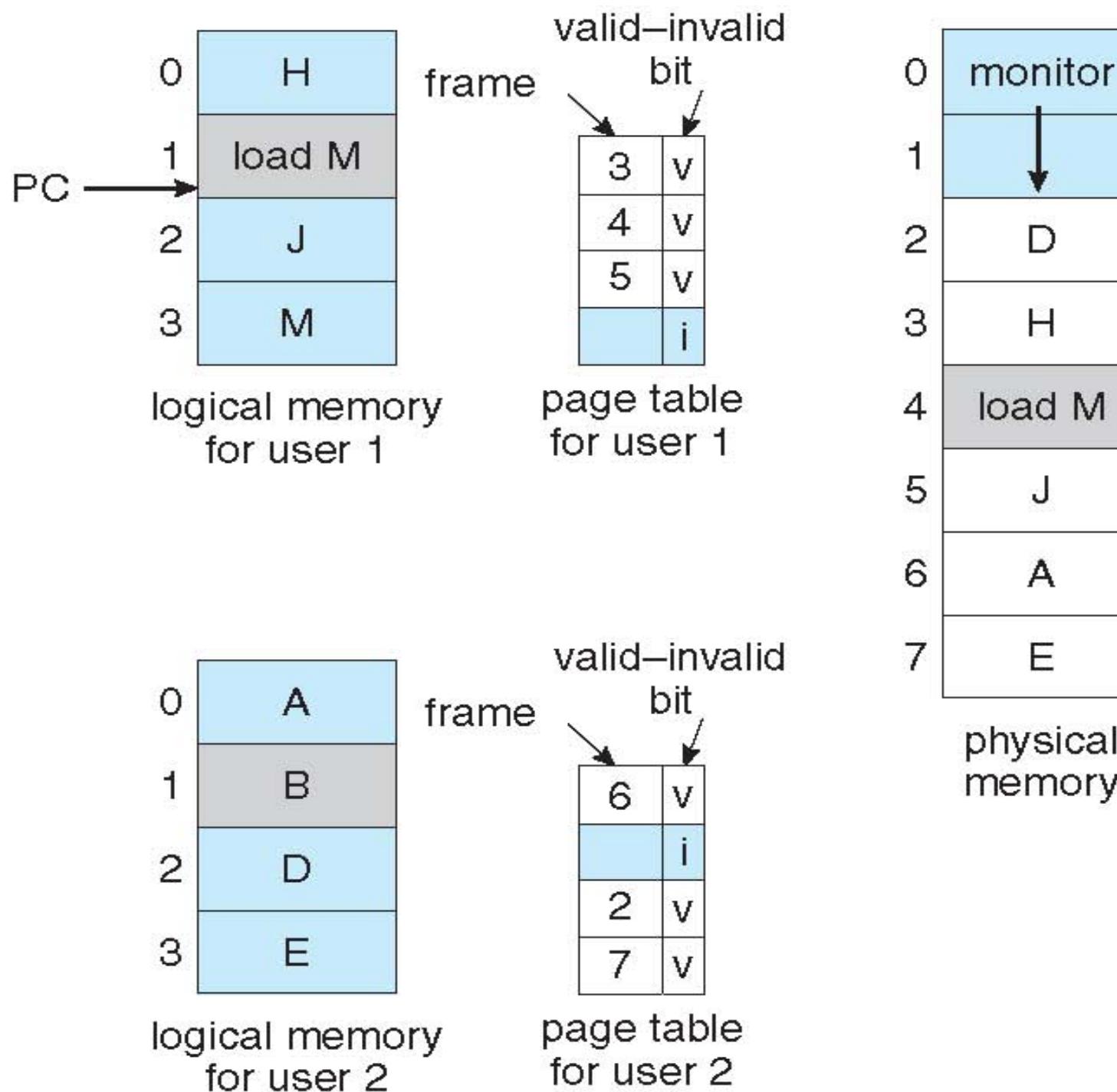
Page Replacement

- 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

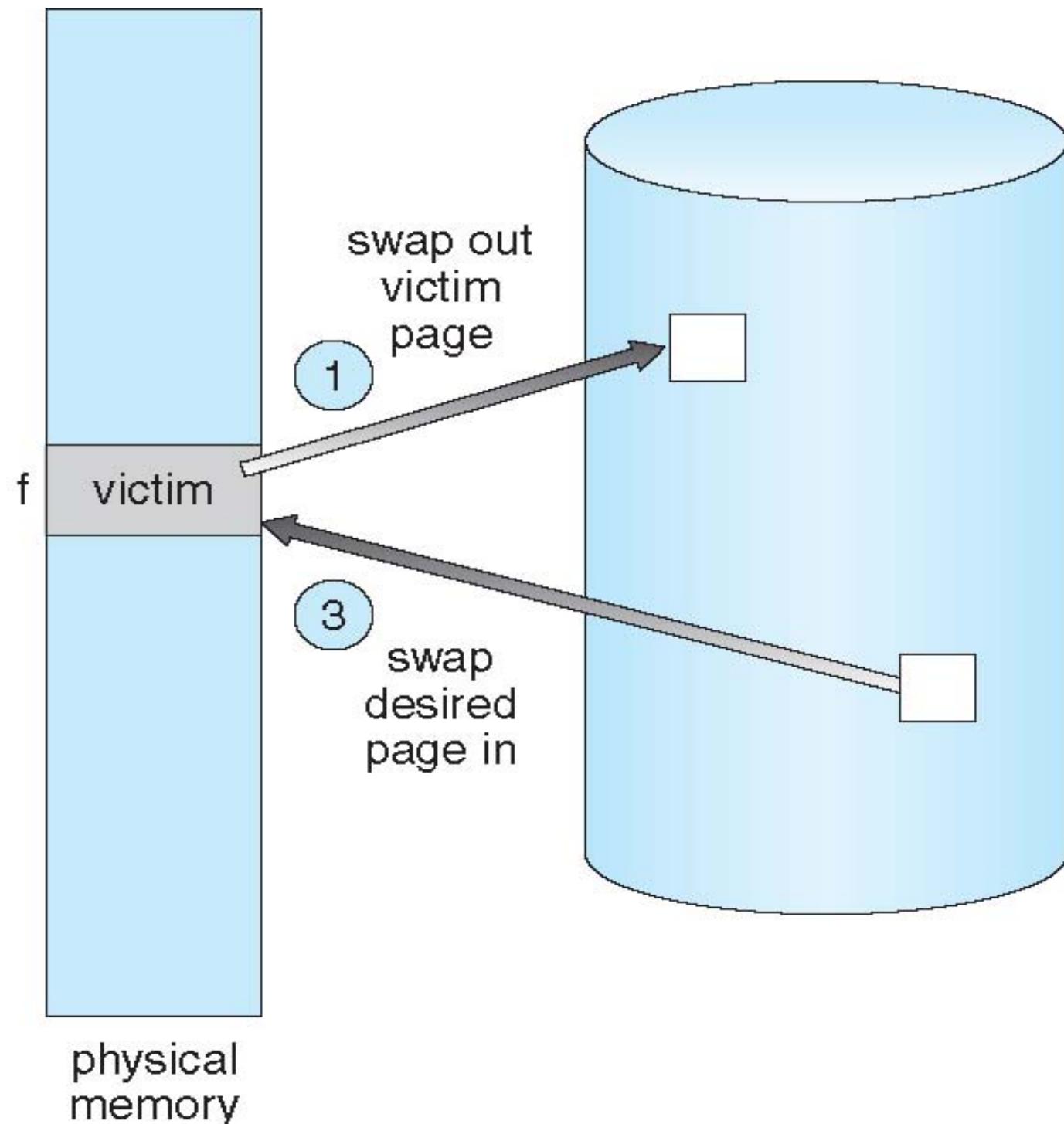
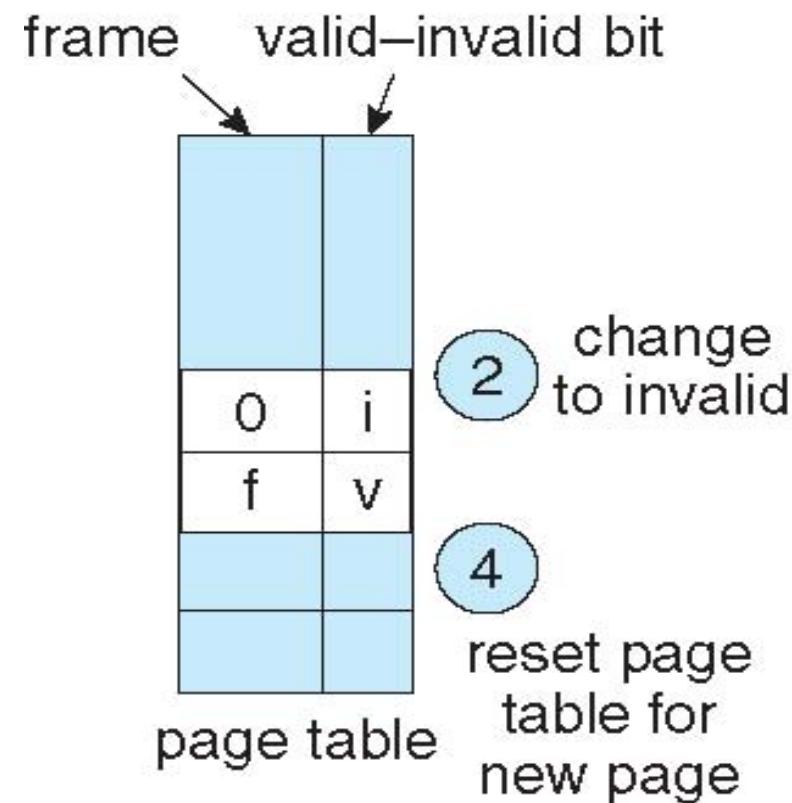
1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the 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



physical
memory

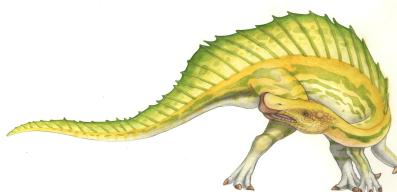


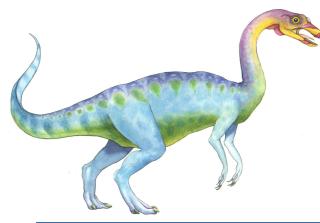


Page and Frame Replacement Algorithms

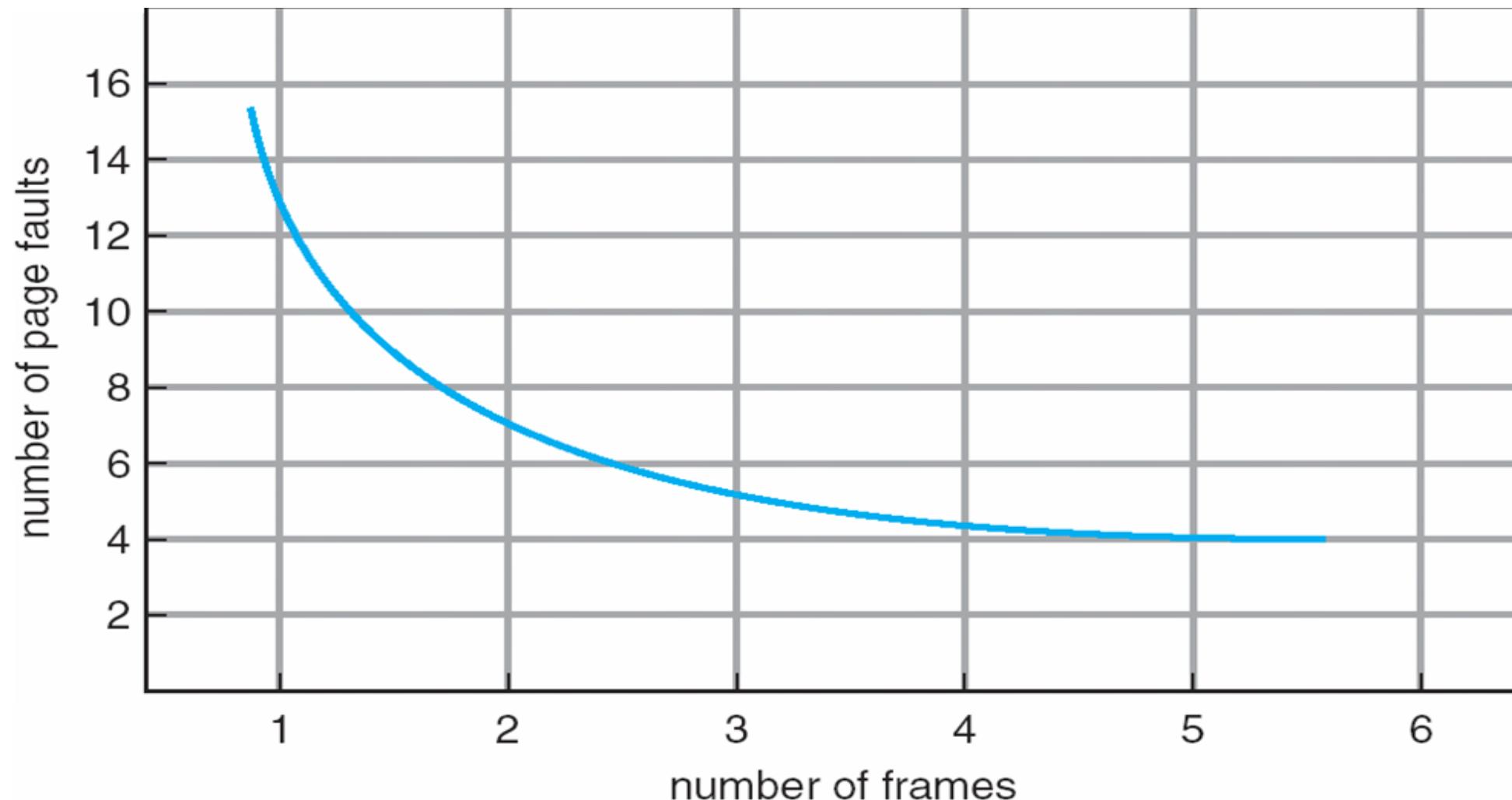
- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) 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
- In all our examples, the reference string is

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





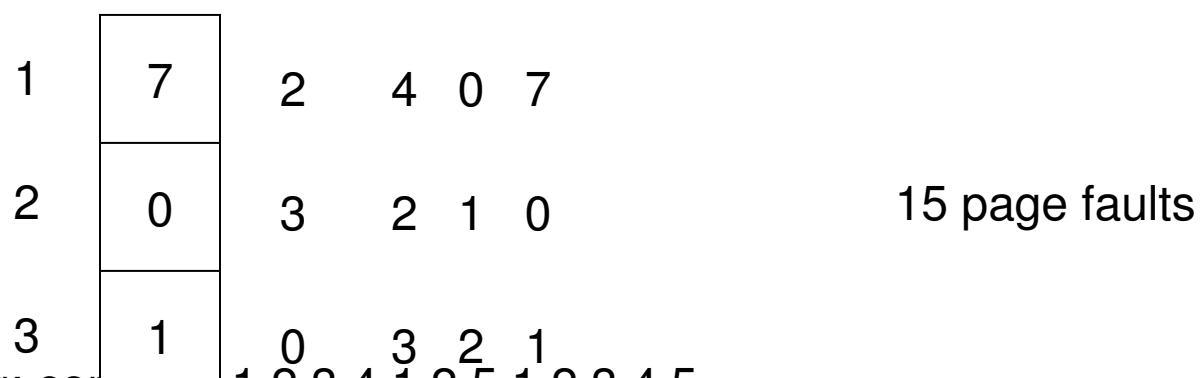
Graph of Page Faults Versus The Number of Frames



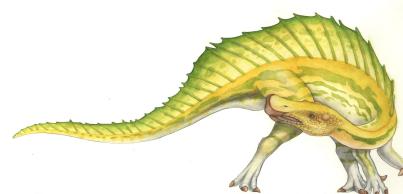


First-In-First-Out (FIFO) Algorithm

- 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)



- Can vary by reference string: consider **1,2,3,4,1,2,5,1,2,3,4,5**
 - Adding more frames can cause more page faults!
 - ▶ **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue

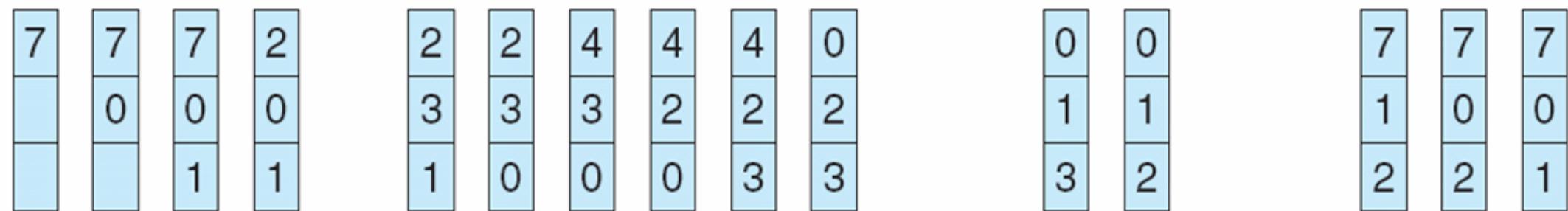




FIFO Page Replacement

reference string

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

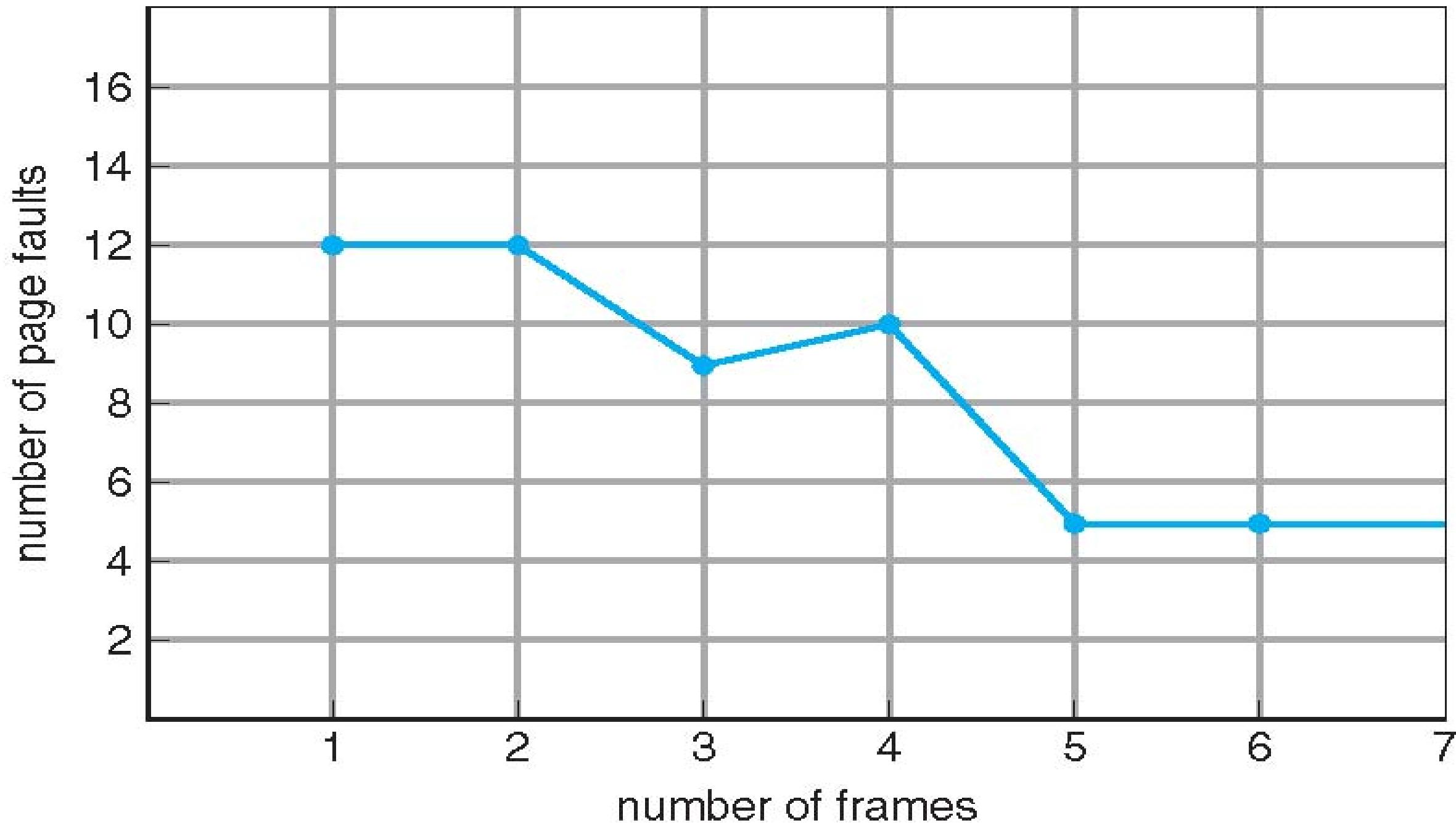


page frames





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 on the next slide
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

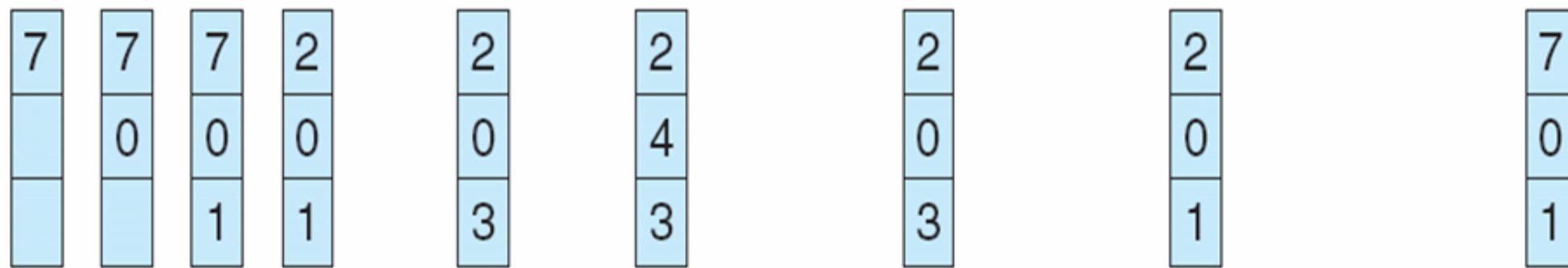




Optimal Page Replacement

reference string

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



page frames



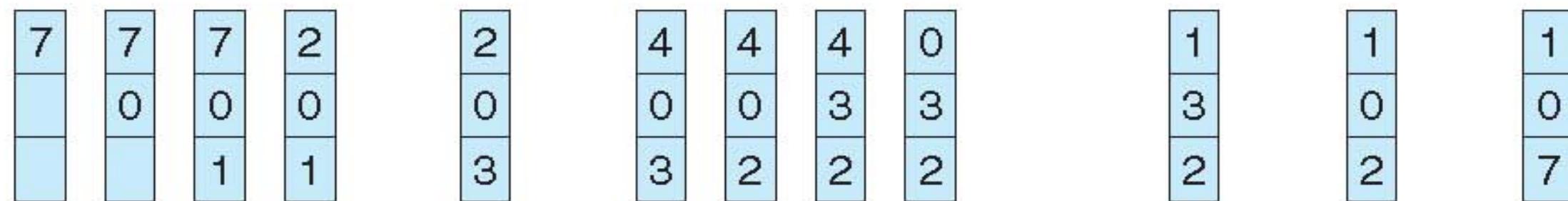


Least Recently Used (LRU) Algorithm

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

reference string

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



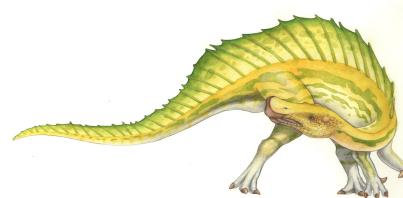
page frames

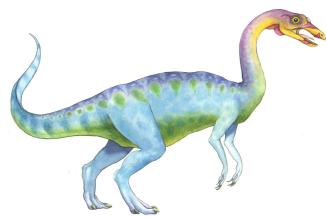




LRU Algorithm (Cont.)

- 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 The Most Recent Page References

reference string

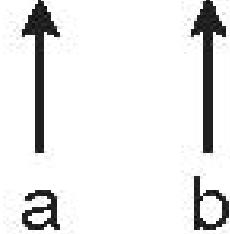
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
a





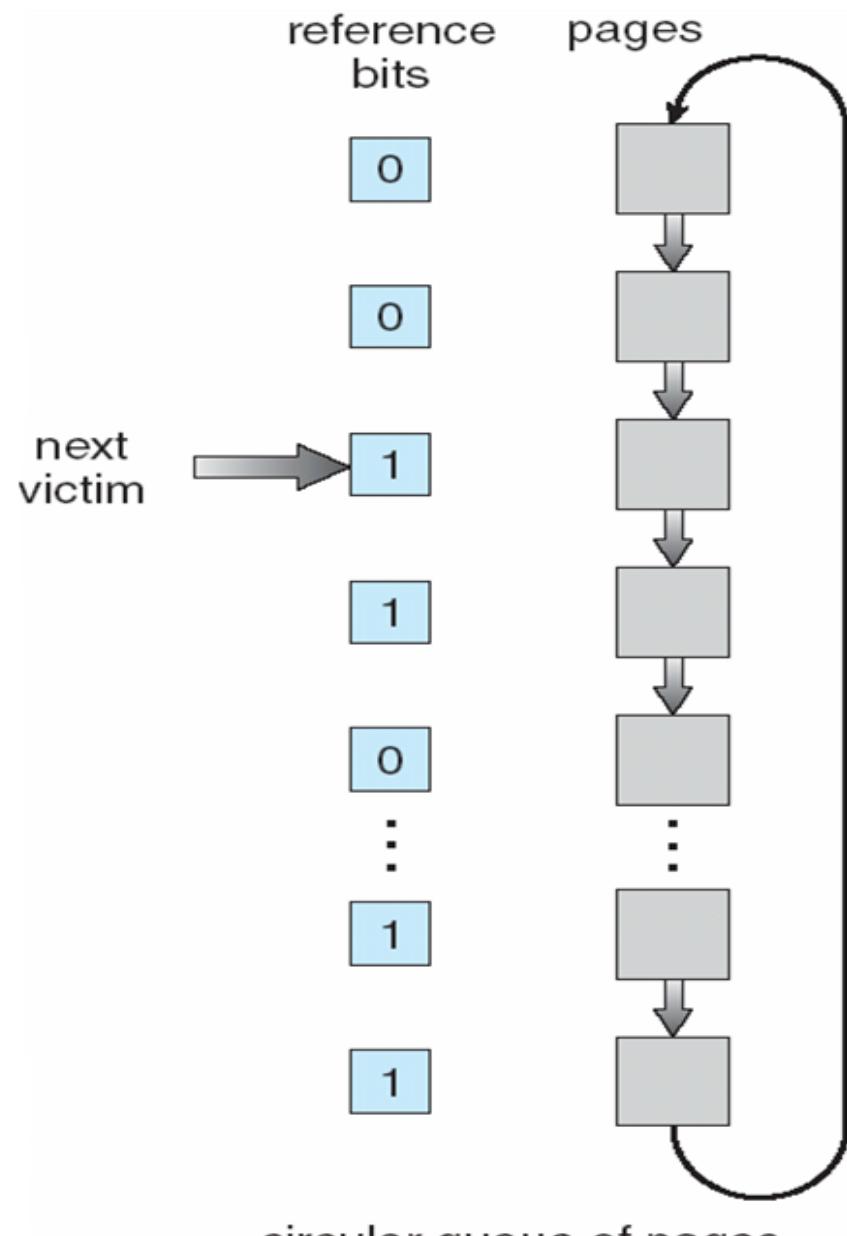
LRU Approximation Algorithms

- 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**
 - 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

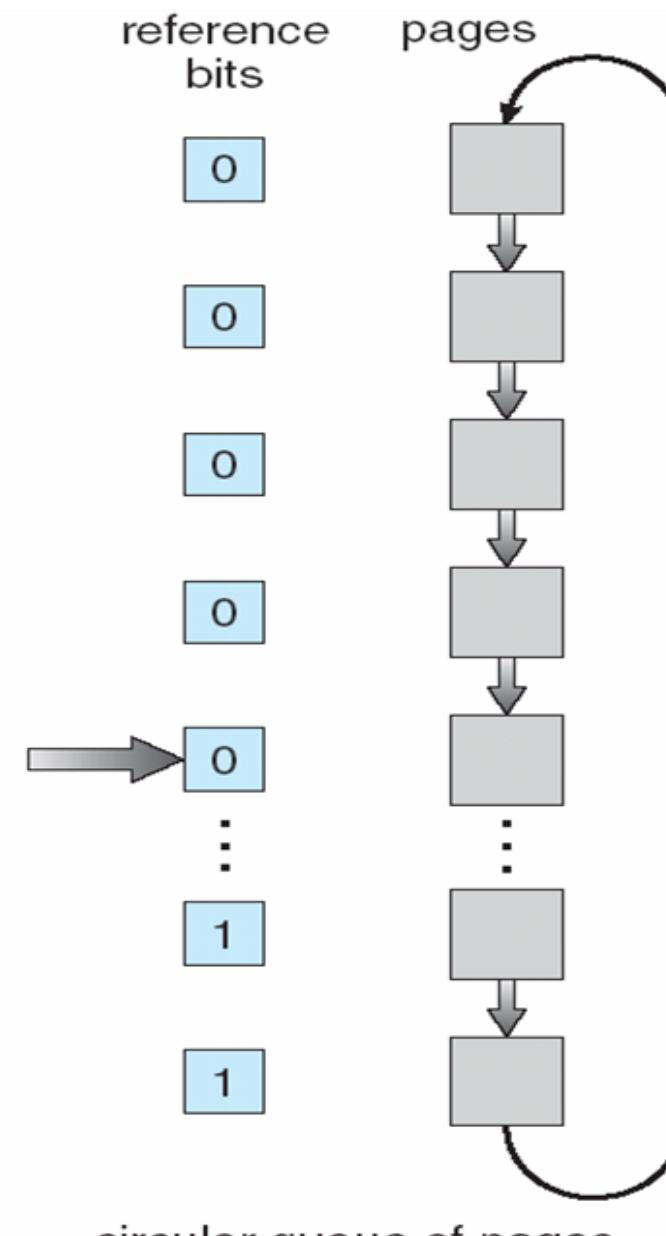




Second-Chance (clock) Page-Replacement Algorithm

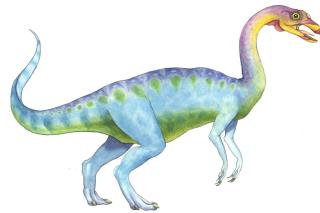


(a)



(b)

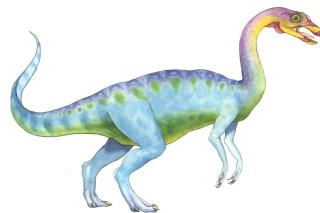




Counting Algorithms

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

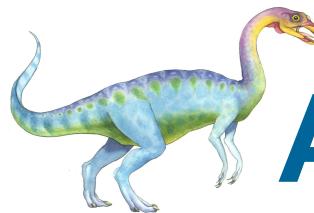




Page-Buffering Algorithms

- 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 give direct access to the disk, getting out of the way of the applications
 - **Raw disk mode**
- Bypasses buffering, locking, etc

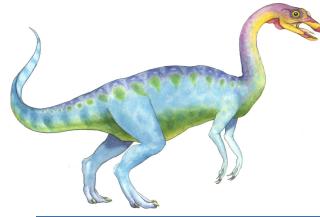




Allocation of Frames

- 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 $m = 64$
— $S = \sum s_i$ $s_1 = 10$
— m = total number of frames $s_2 = 127$
— a_i = allocation for p_i = $\frac{s_i}{S} \times m$ $a_1 = \frac{10}{137} \times 64 \approx 5$ $a_2 = \frac{127}{137} \times 64 \approx 59$





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



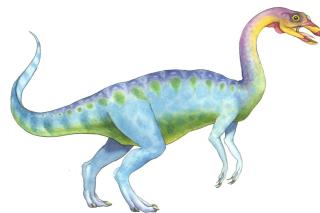


Global vs. Local Allocation

- **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 **Igroups**
 - ▶ 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 Igroup





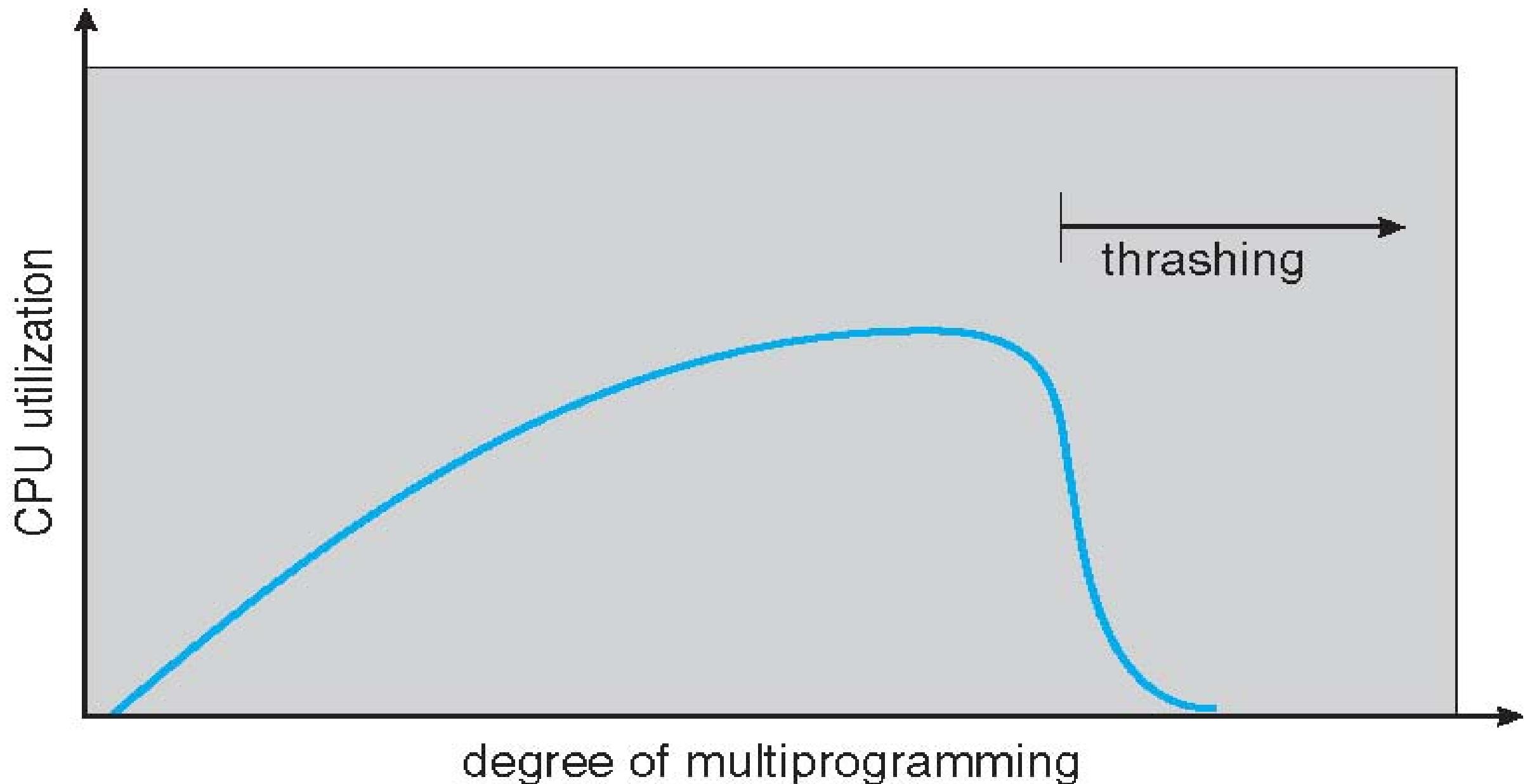
Thrashing

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





Thrashing (Cont.)



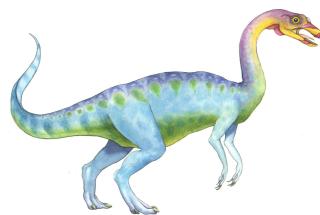


Demand Paging and Thrashing

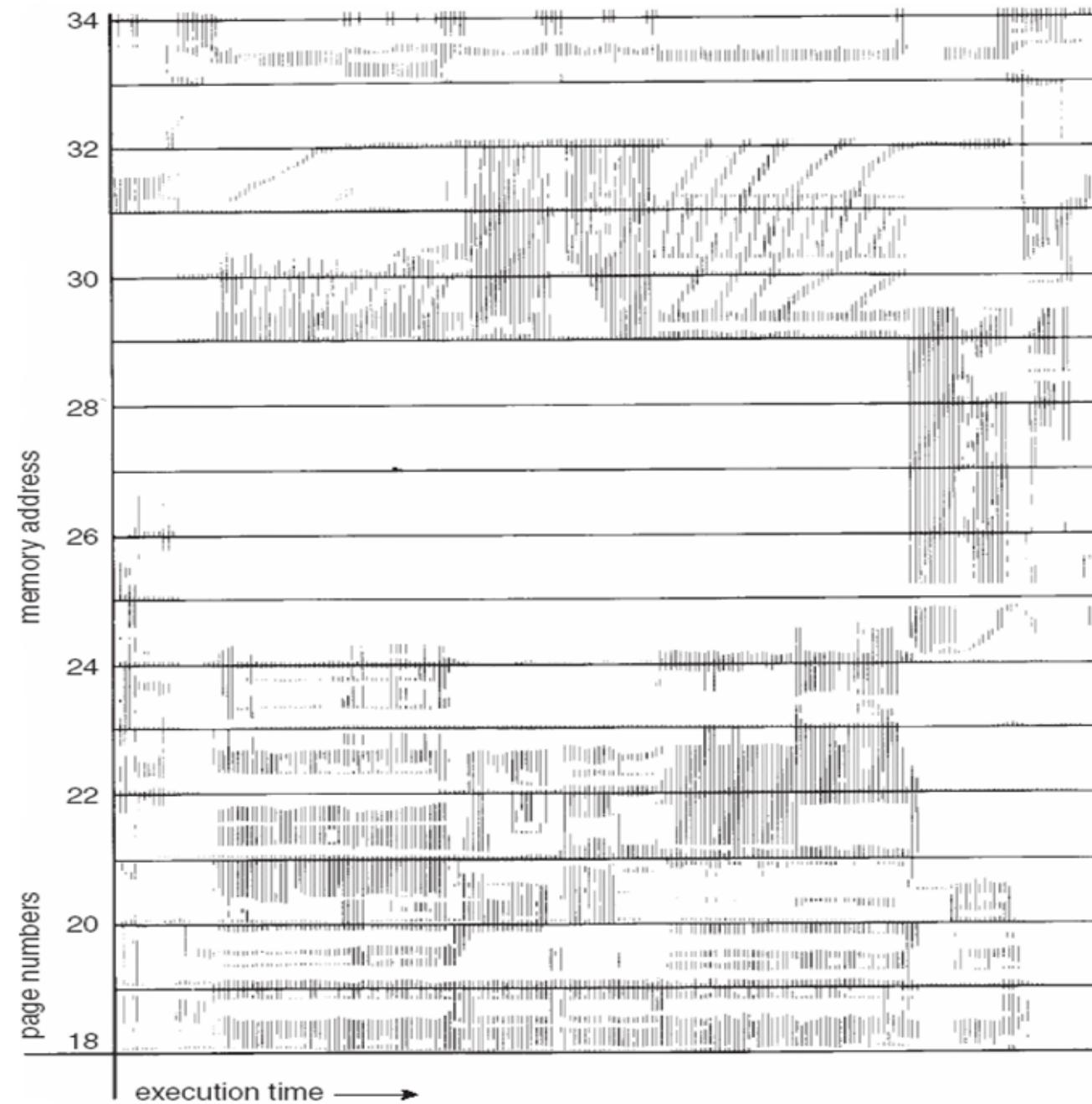
- Why does demand paging work?
Locality model
 - Process migrates from one locality to another
 - Localities may overlap

- Why does thrashing occur?
 Σ size of locality > total memory size
 - Limit effects by using local or priority page replacement





Locality In A Memory-Reference Pattern





Working-Set Model

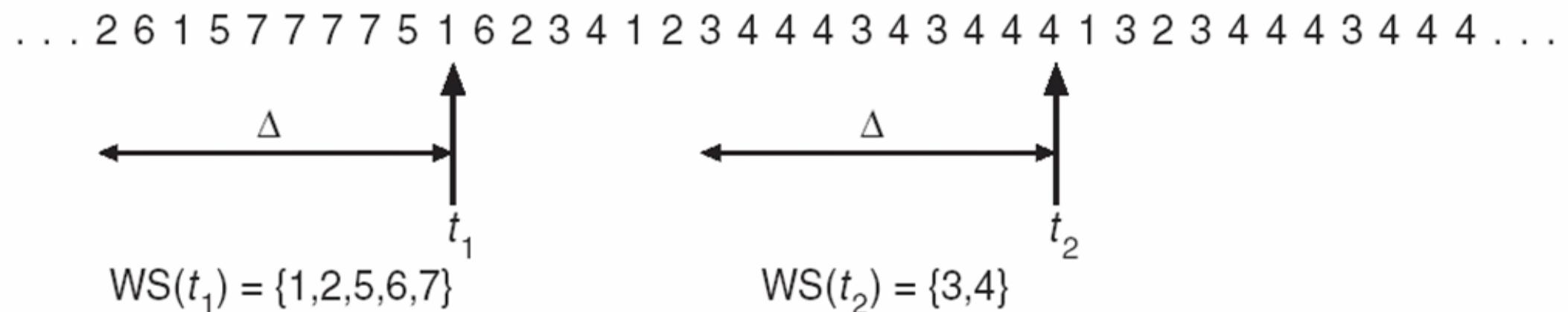
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass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





Working-set model

page reference table





Keeping Track of the Working Set

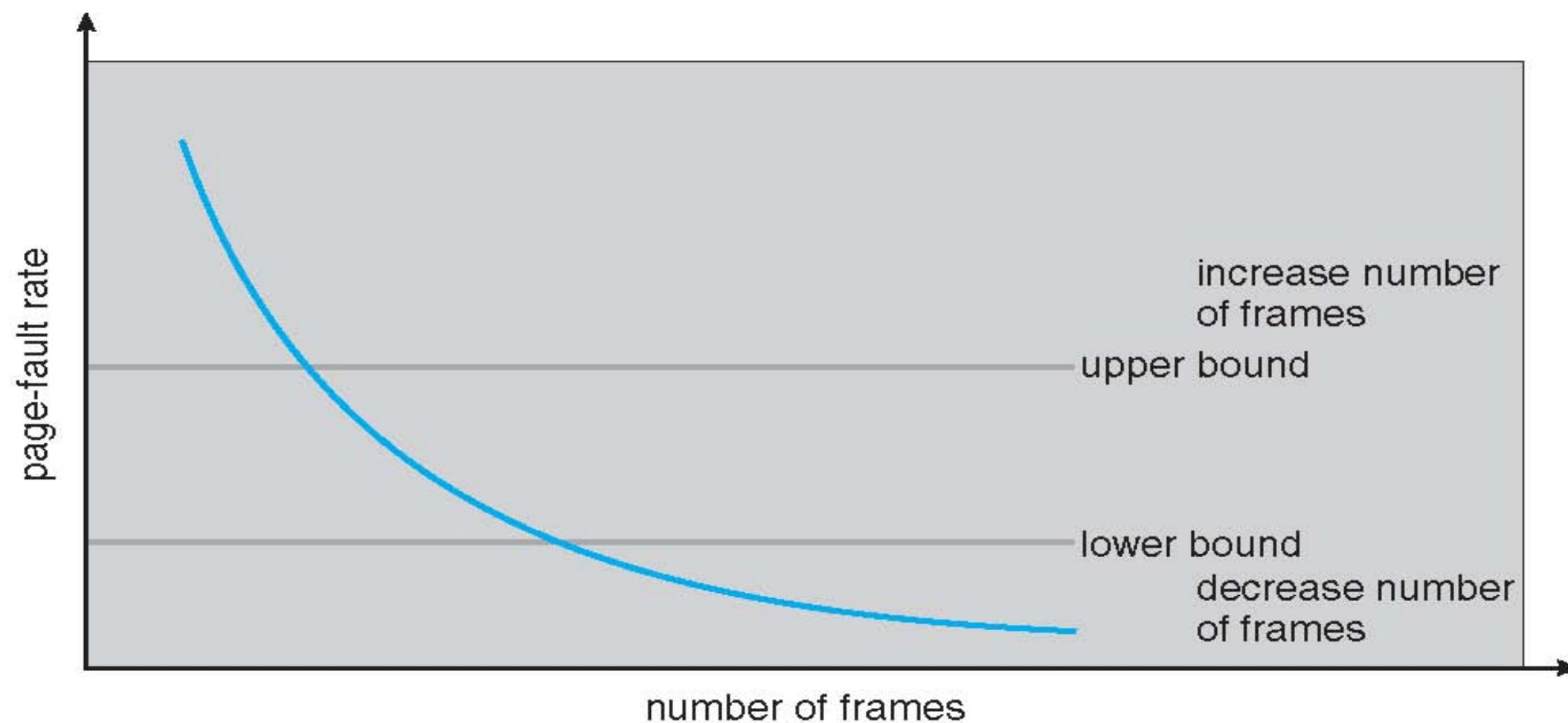
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupt occurs, copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

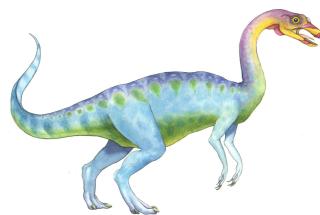




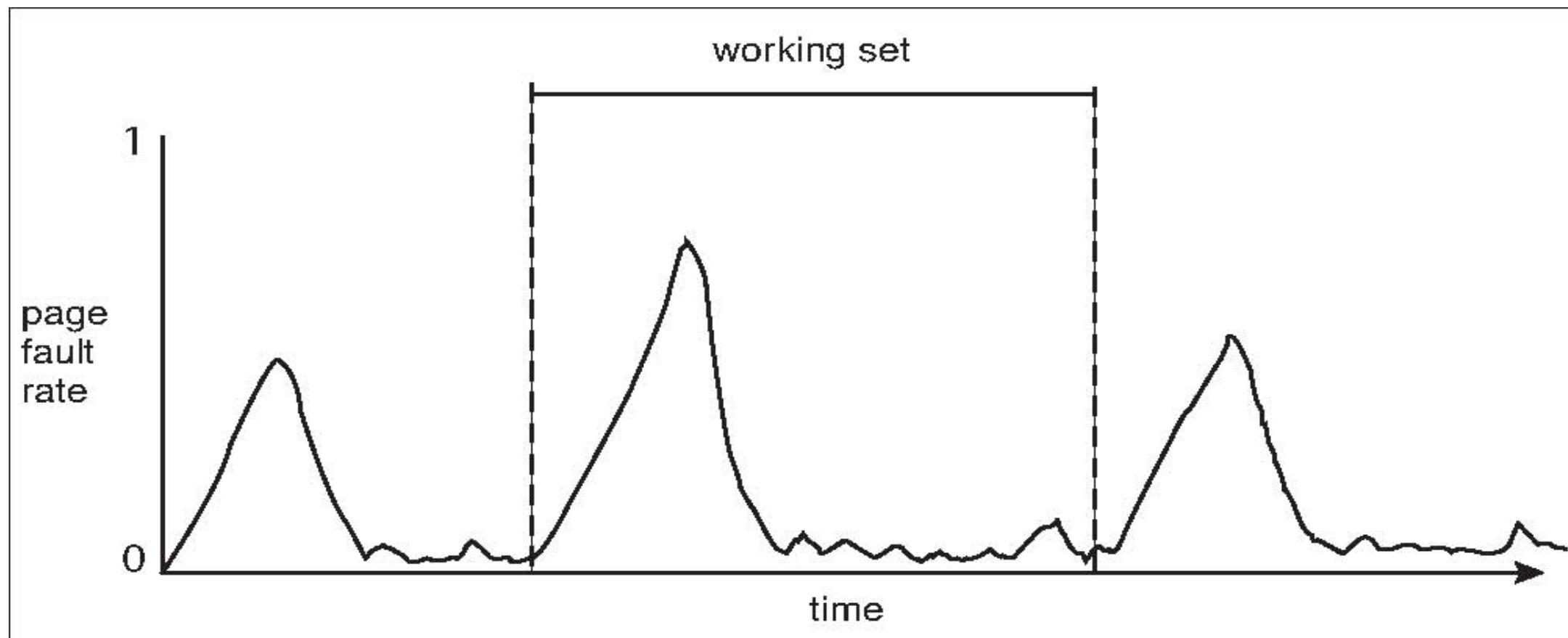
Page-Fault Frequency

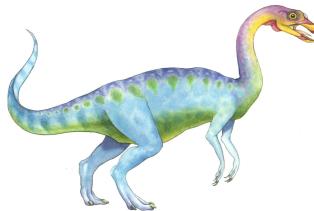
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame





Working Sets and Page Fault Rates

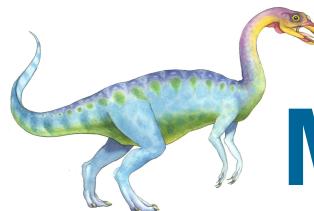




Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
 - A page-sized portion of the file is read from the file system into a physical page
 - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
 - Periodically and / or at file `close()` time
 - For example, when the pager scans for dirty pages





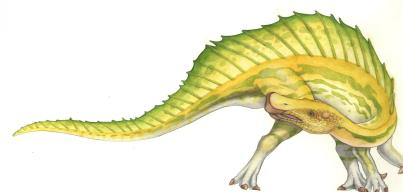
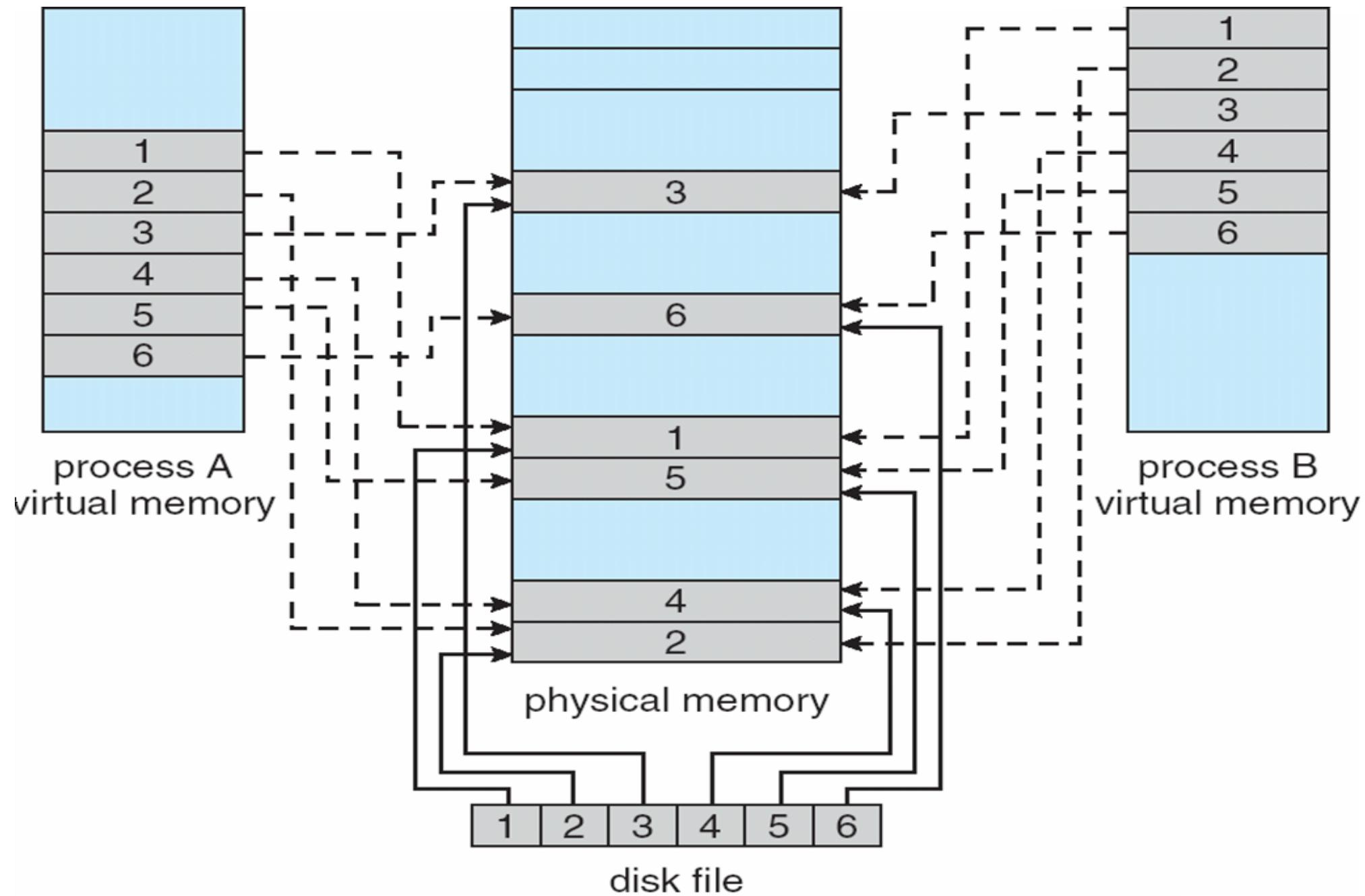
Memory-Mapped File Technique for all I/O

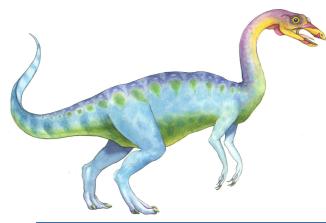
- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
 - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
 - But map file into kernel address space
 - Process still does `read()` and `write()`
 - ▶ Copies data to and from kernel space and user space
 - Uses efficient memory management subsystem
 - ▶ Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)



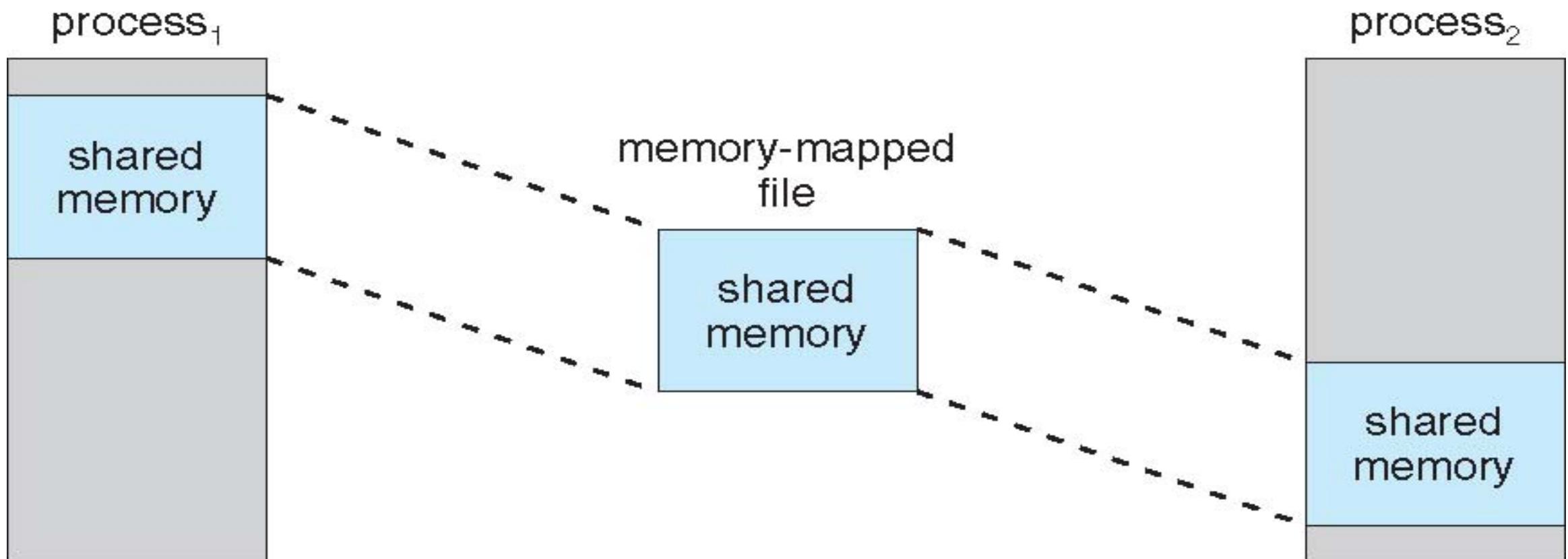


Memory Mapped Files





Memory-Mapped Shared Memory in Windows





Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
 - Kernel requests memory for structures of varying sizes
 - Some kernel memory needs to be contiguous
 - ▶ I.e. for device I/O





Buddy System

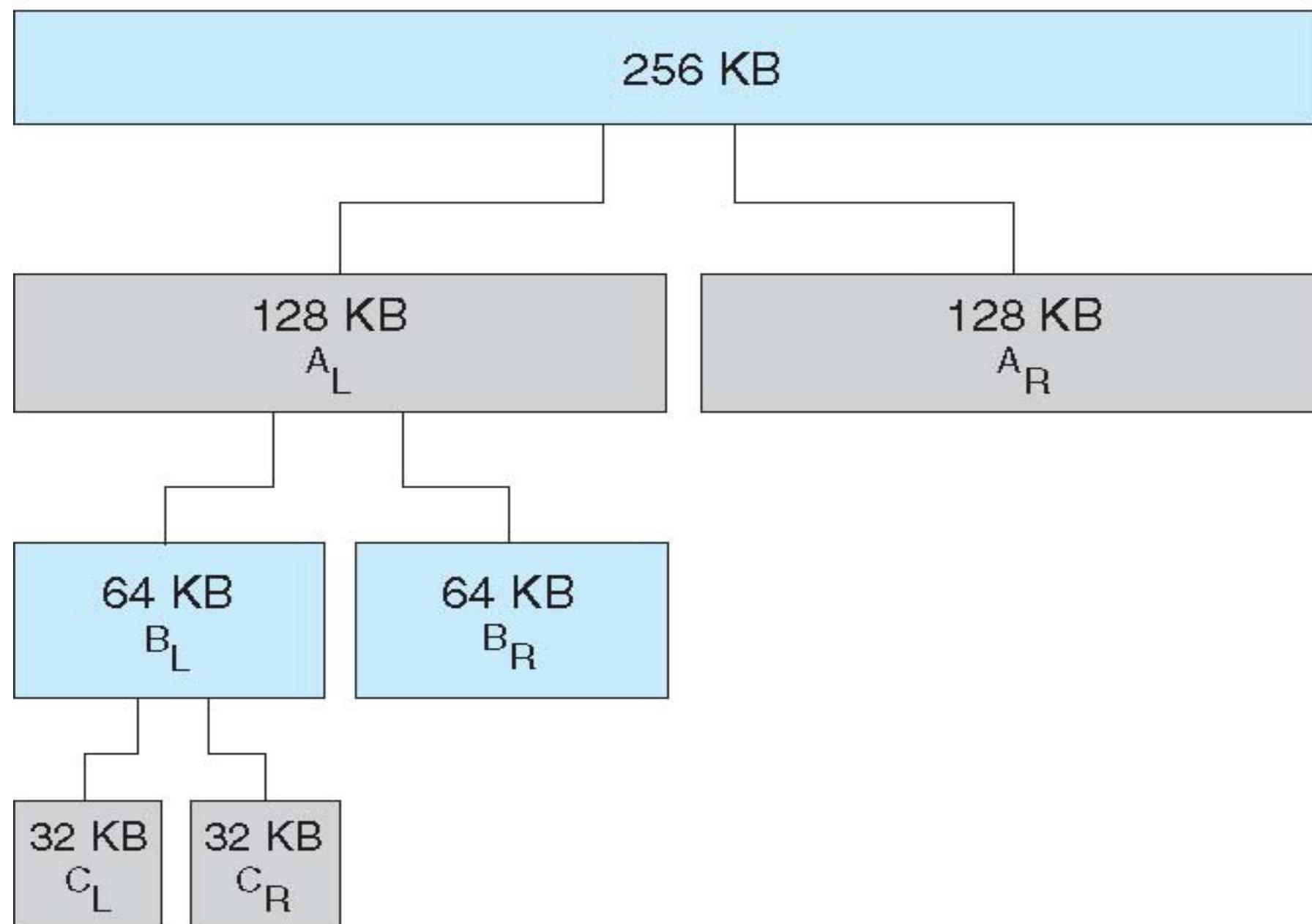
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
 - Satisfies requests in units sized as power of 2
 - Request rounded up to next highest power of 2
 - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
 - ▶ Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
 - Split into A_L and A_R of 128KB each
 - ▶ One further divided into B_L and B_R of 64KB
 - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation





Buddy System Allocator

physically contiguous pages





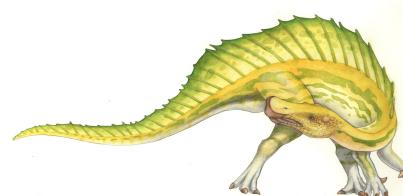
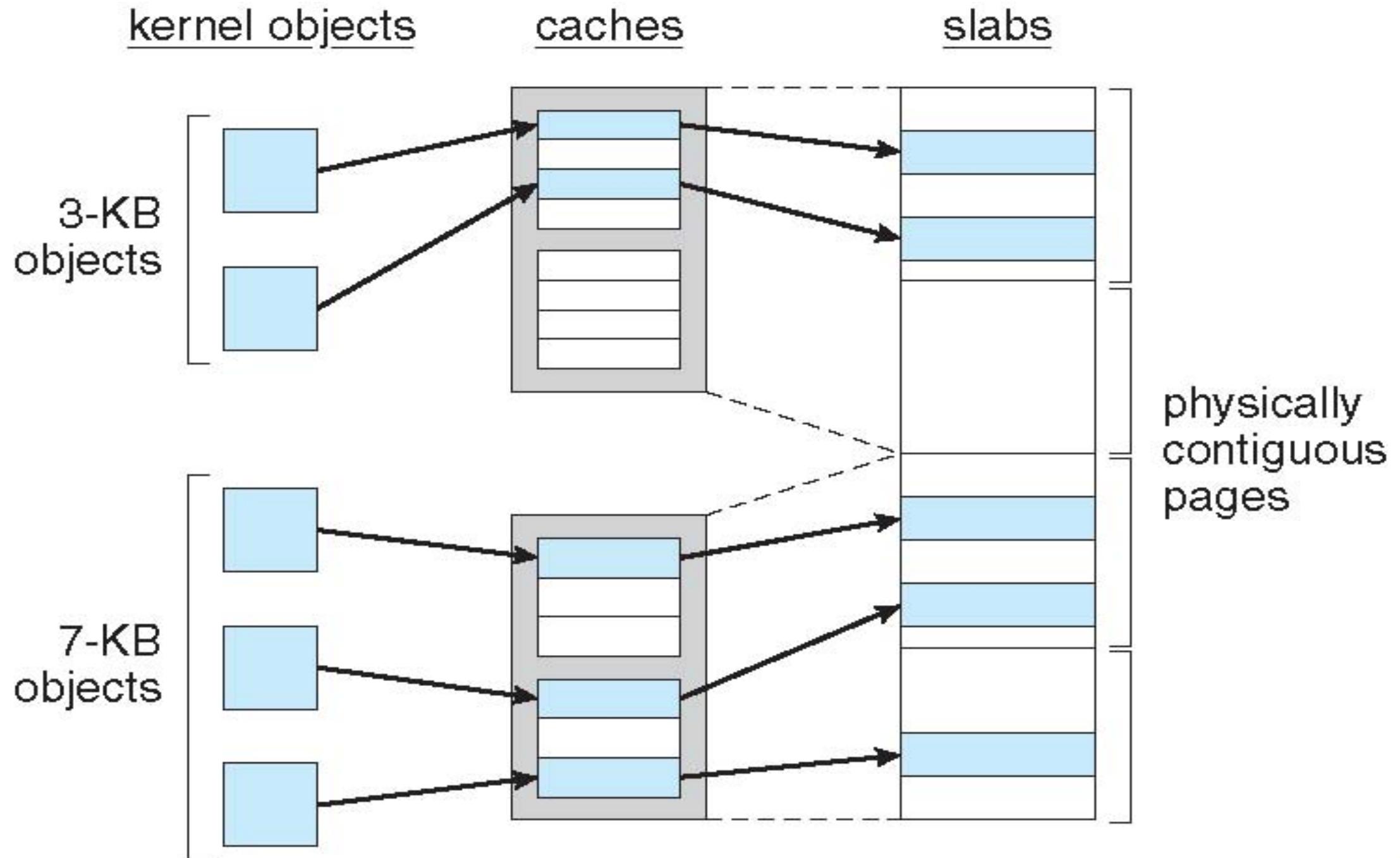
Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
 - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
 - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction





Slab Allocation





Other Considerations -- Prepaging

■ Prepaging

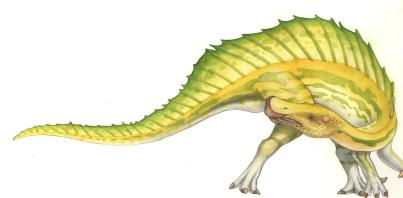
- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used
 - ▶ Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
 - ▶ α near zero \Rightarrow prepaginaing loses

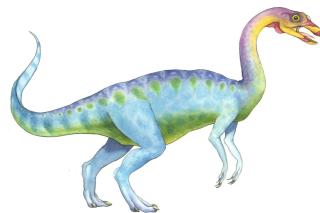




Other Issues – Page Size

- Sometimes OS designers have a choice
 - Especially if running on custom-built CPU
- Page size selection must take into consideration:
 - Fragmentation
 - Page table size
 - **Resolution**
 - I/O overhead
 - Number of page faults
 - Locality
 - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time





Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
 - Otherwise there is a high degree of page faults
- Increase the Page Size
 - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
 - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





Other Issues – Program Structure

■ Program structure

- `Int[128,128] data;`
- Each row is stored in one page
- Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

$128 \times 128 = 16,384$ page faults

- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

128 page faults

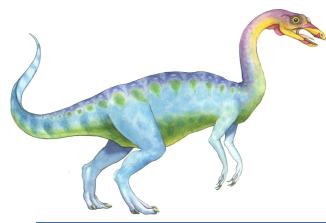




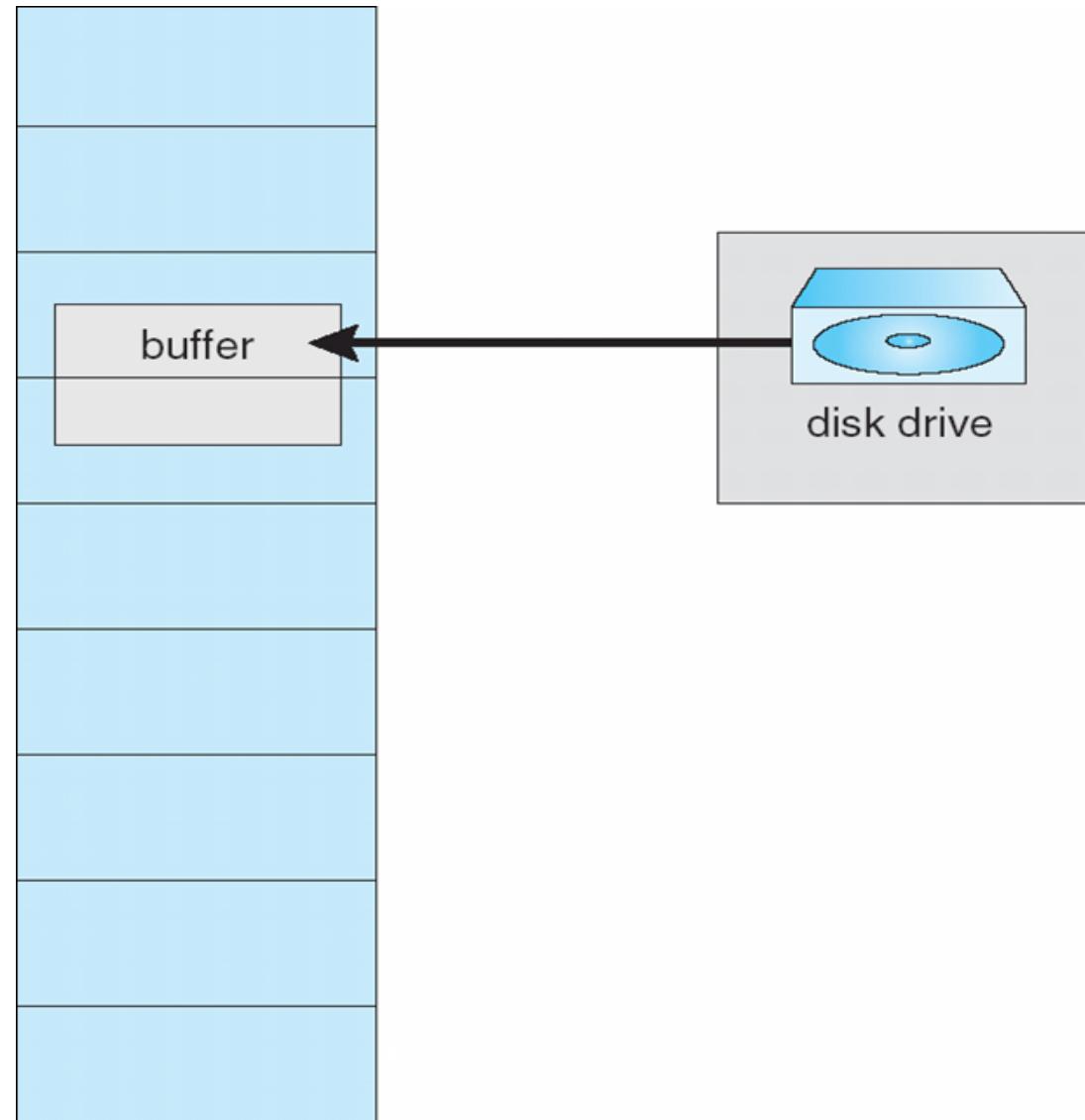
Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm





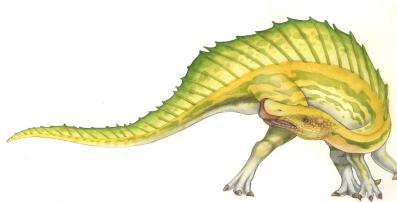
Reason Why Frames Used For I/O Must Be In Memory





Operating System Examples

- Windows XP
- Solaris

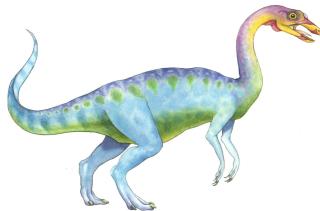




Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum





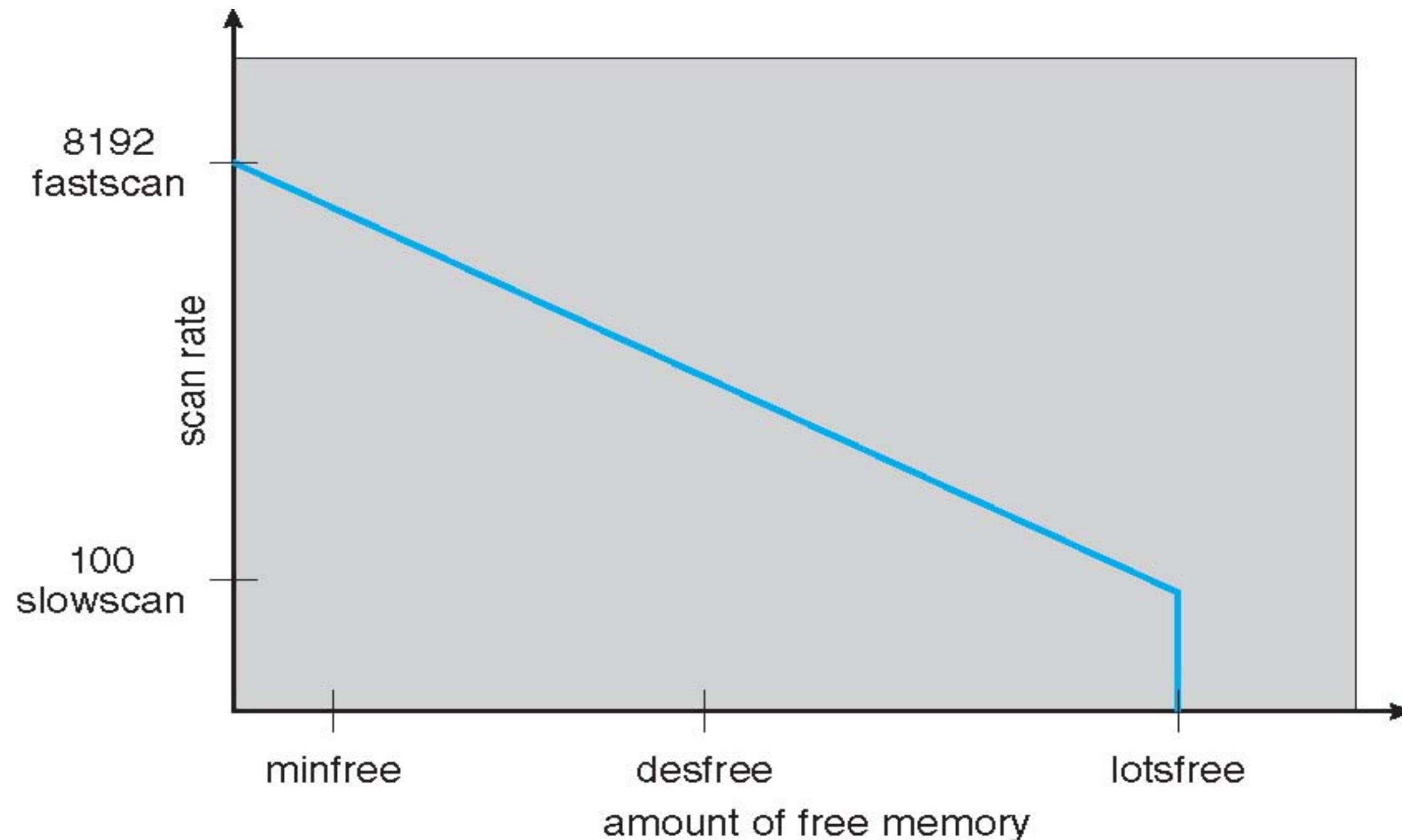
Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to begin swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available
- Priority paging gives priority to process code pages

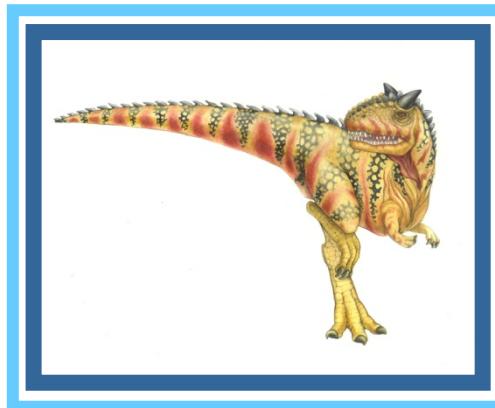




Solaris 2 Page Scanner



Chapter 10: File System





Chapter 10: File System

- 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**





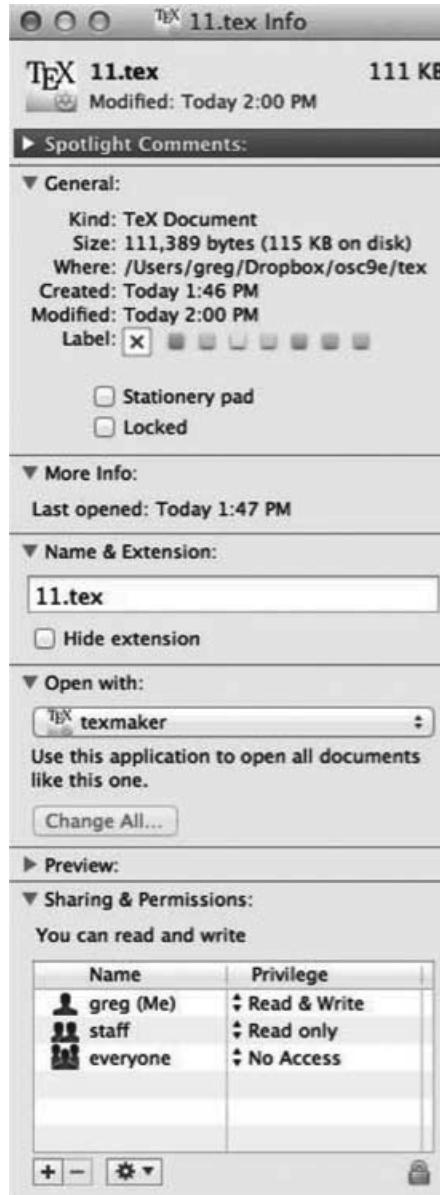
File Attributes

- **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 info Window on Mac OS X





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





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
 - Similar to reader-writer locks
 - **Shared lock** similar to reader lock – several processes can acquire concurrently
 - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
 - **Mandatory** – access is denied depending on locks held and requested
 - **Advisory** – processes can find status of locks and decide what to do





File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
        }
    }
}
```





File Locking Example – Java API (Cont.)

```
// this locks the second half of the file - shared  
sharedLock = ch.lock(raf.length()/2+1, raf.length(),  
    SHARED);  
/** Now read the data . . . */  
// release the lock  
sharedLock.release();  
} catch (java.io.IOException ioe) {  
    System.err.println(ioe);  
}finally {  
    if (exclusiveLock != null)  
        exclusiveLock.release();  
    if (sharedLock != null)  
        sharedLock.release();  
}  
}
```





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 compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information





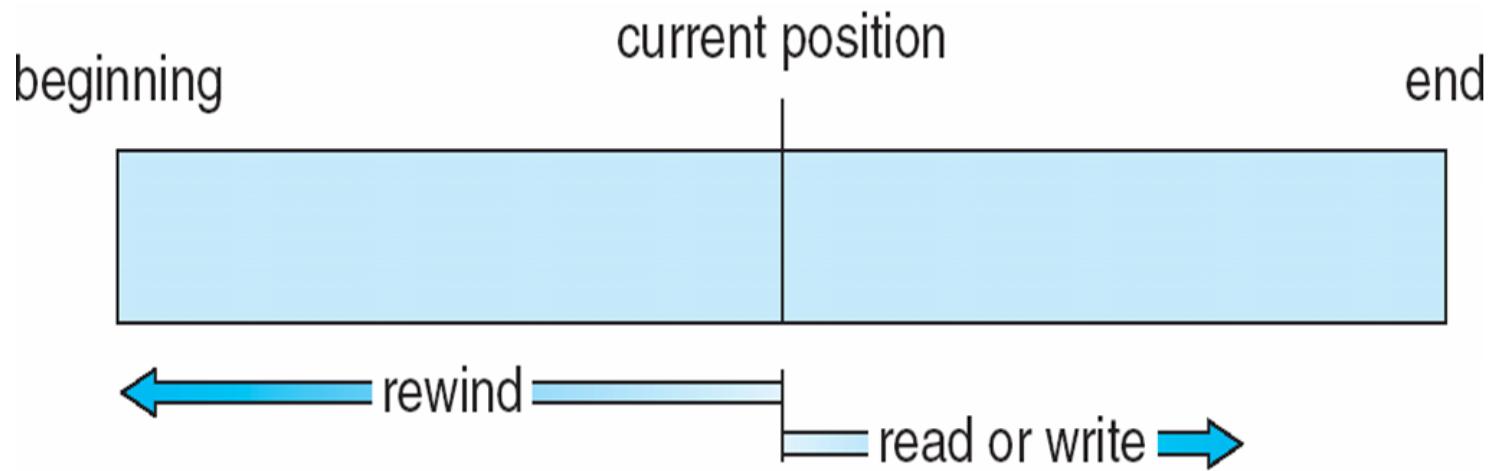
File Structure

- None - 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





Sequential-access File





Access Methods

- **Sequential Access**

- read next**

- write next**

- reset**

- no read after last write

- (rewrite)

- **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

- Relative block numbers allow OS to decide where file should be placed

- See [allocation problem](#) in Ch 11





Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	<i>read cp;</i> $cp = cp + 1;$
<i>write next</i>	<i>write cp;</i> $cp = cp + 1;$





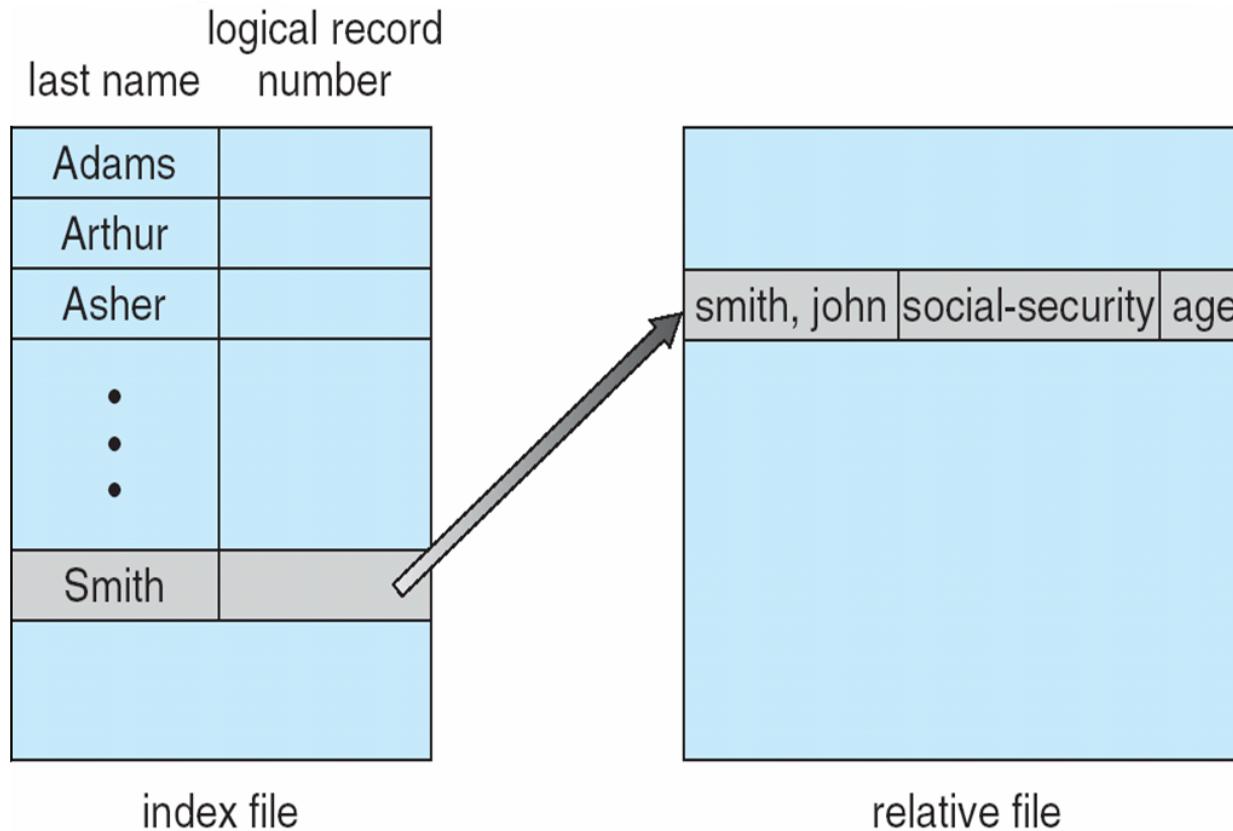
Other Access Methods

- 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)





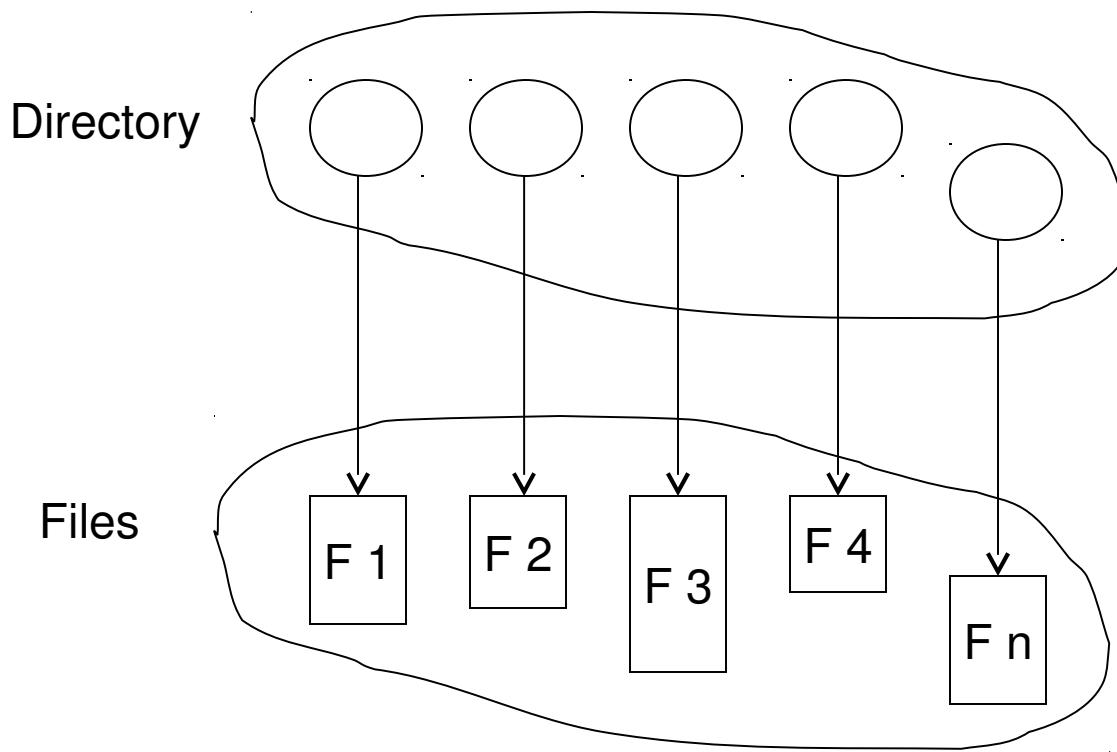
Example of Index and Relative Files





Directory Structure

- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk





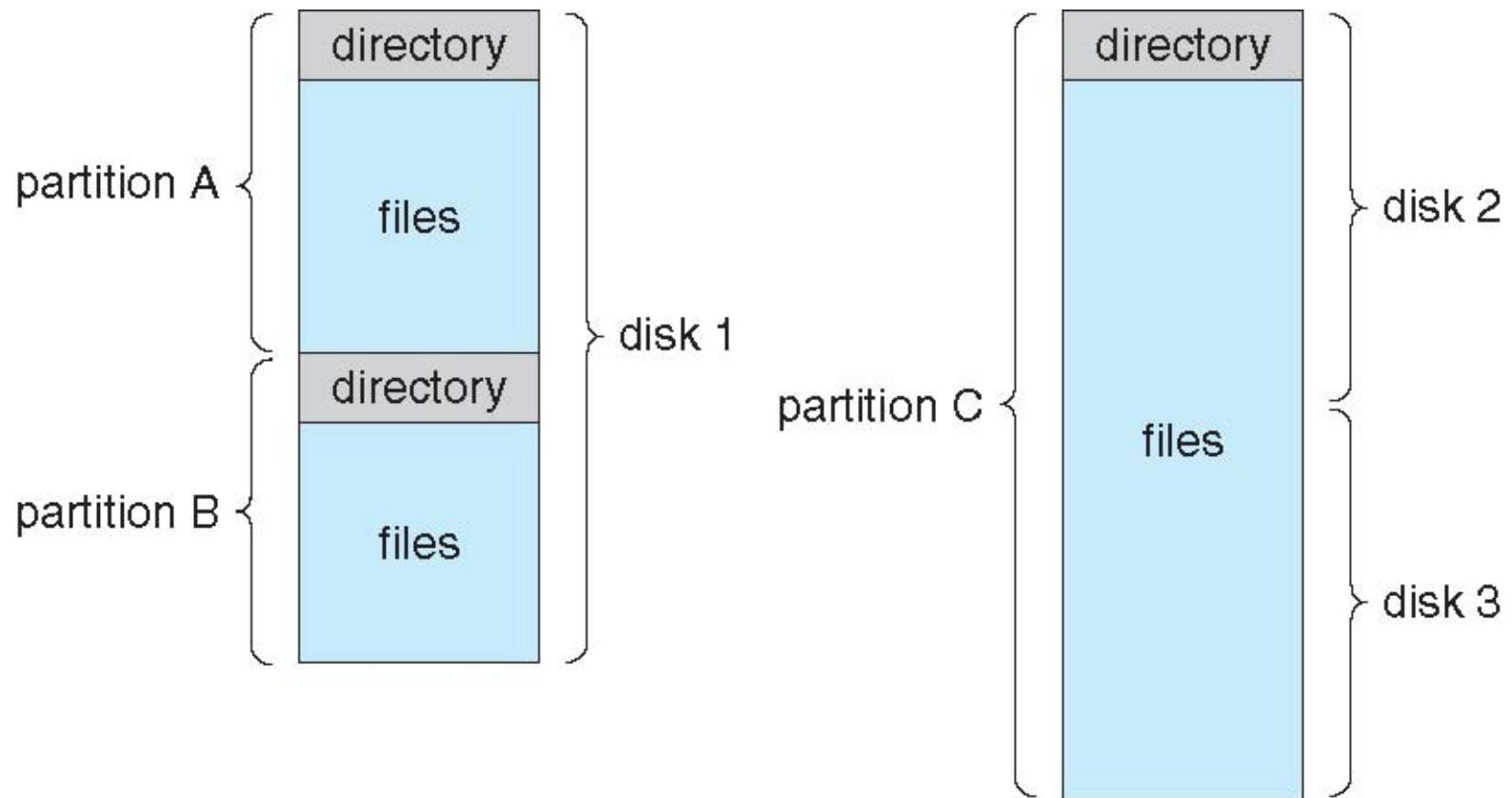
Disk Structure

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- 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





Types of File Systems

- We mostly talk of general-purpose file systems
- But systems frequently have many file systems, some general- and some special- purpose
- Consider Solaris has
 - tmpfs – memory-based volatile FS for fast, temporary I/O
 - objfs – interface into kernel memory to get kernel symbols for debugging
 - ctfs – contract file system for managing daemons
 - lofs – loopback file system allows one FS to be accessed in place of another
 - procfs – kernel interface to process structures
 - ufs, zfs – general purpose file systems





Operations Performed on Directory

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





Organize the Directory (Logically) to Obtain

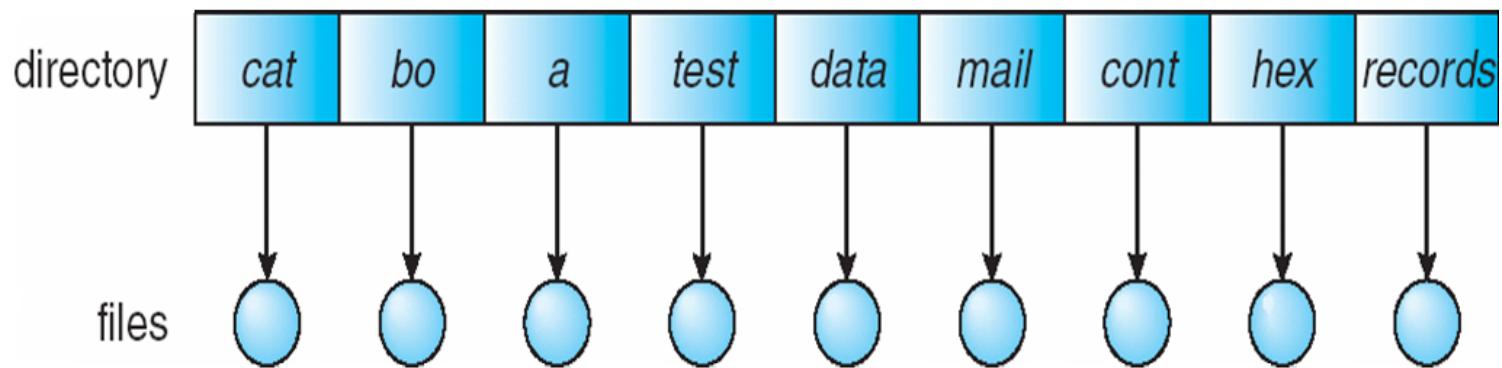
- Efficiency – locating a file quickly
- Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)





Single-Level Directory

- A single directory for all users



Naming problem

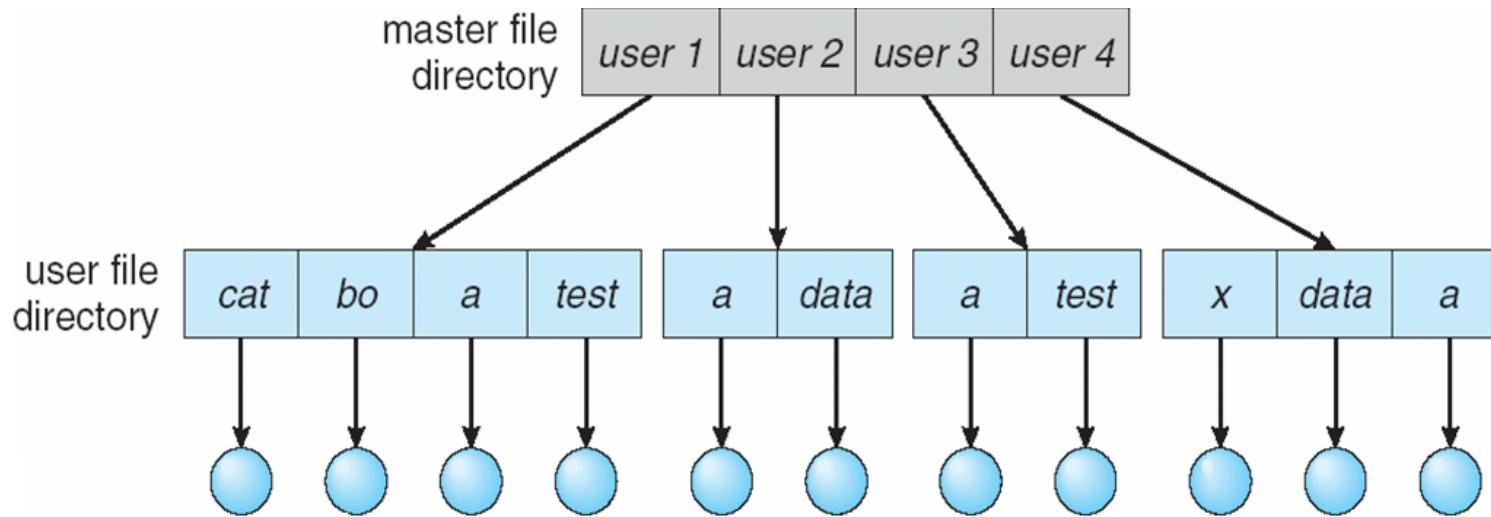
Grouping problem





Two-Level Directory

- Separate directory for each user

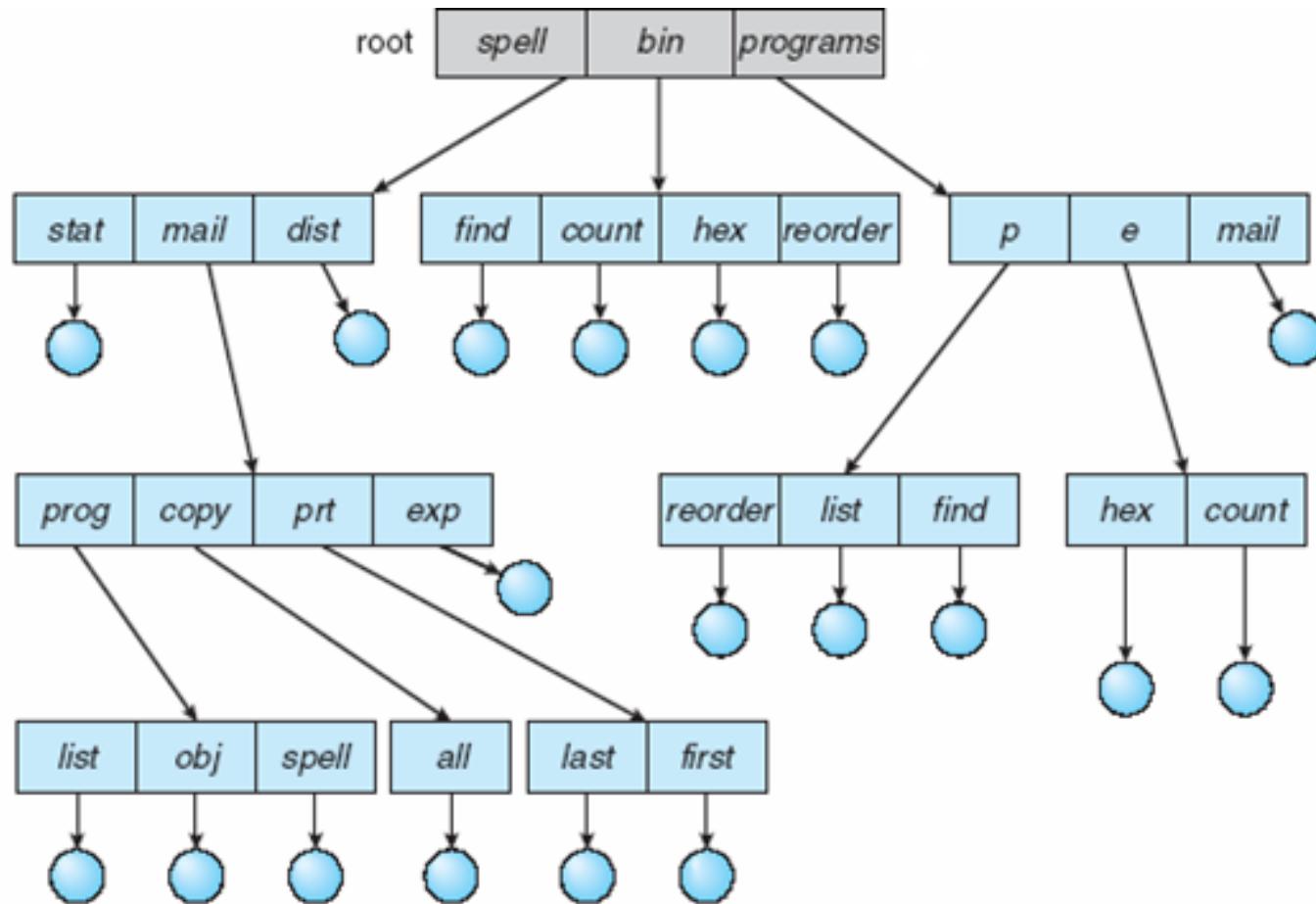


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability





Tree-Structured 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

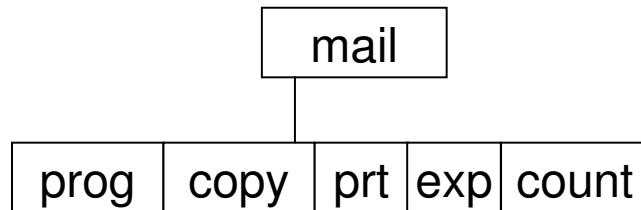
`rm <file-name>`

- Creating a new subdirectory is done in current directory

`mkdir <dir-name>`

Example: if in current directory /mail

`mkdir count`



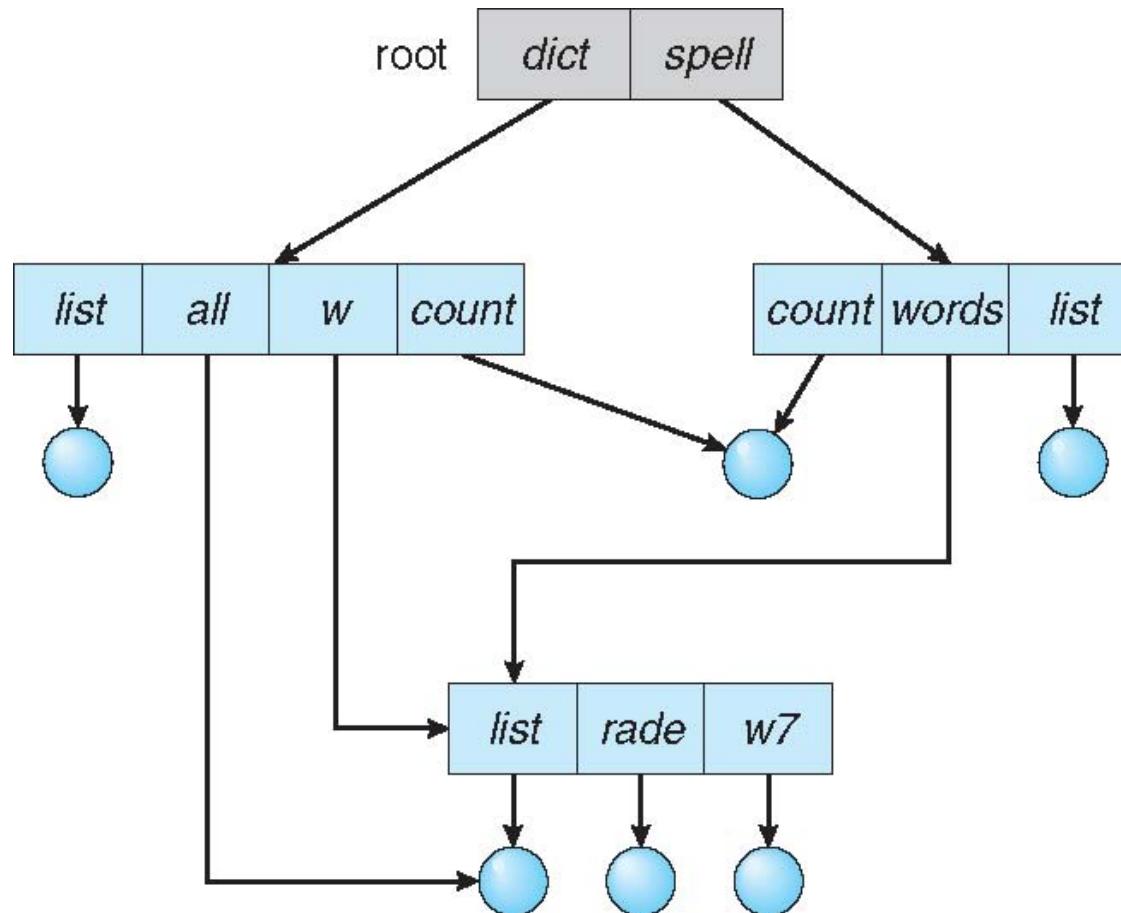
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”





Acyclic-Graph Directories

- Have shared subdirectories and files





Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If ***dict*** deletes ***list*** ⇒ dangling pointer

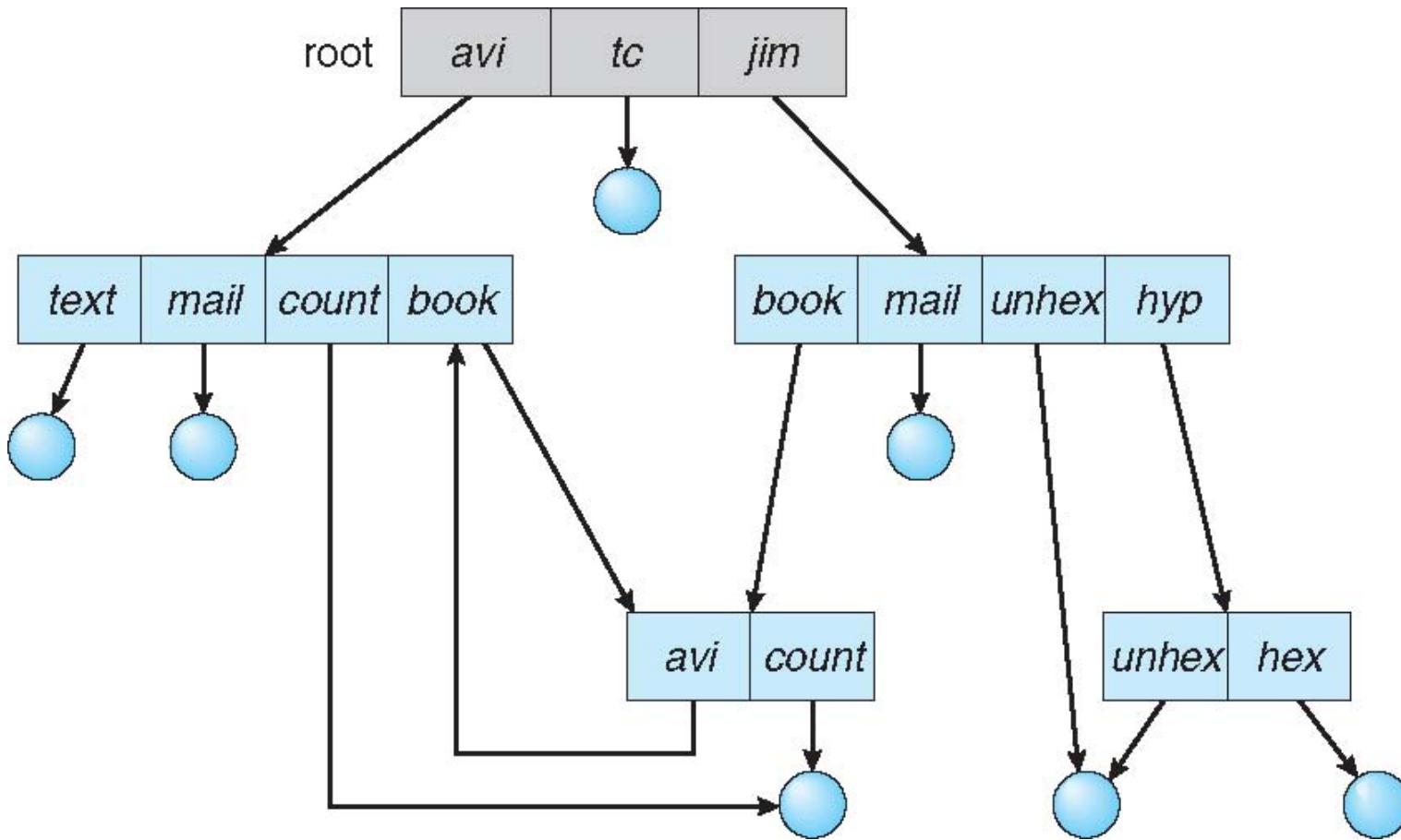
Solutions:

- Backpointers, so we can delete all pointers
Variable size records a problem
 - Backpointers using a daisy chain organization
 - Entry-hold-count solution
-
- New directory entry type
 - **Link** – another name (pointer) to an existing file
 - **Resolve the link** – follow pointer to locate the file





General Graph Directory





General Graph Directory (Cont.)

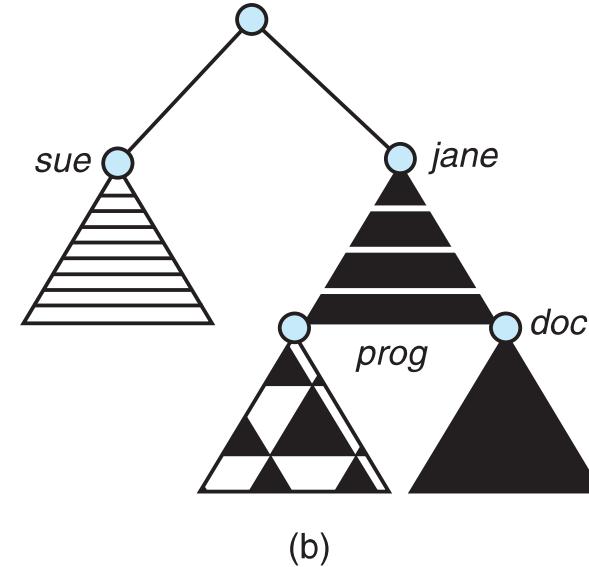
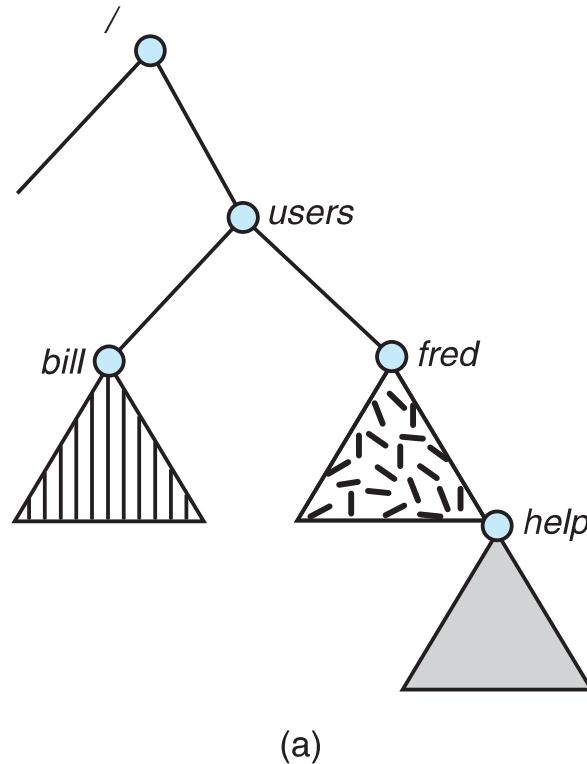
- How do we guarantee no cycles?
 - Allow only links to file not subdirectories
 - **Garbage collection**
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK





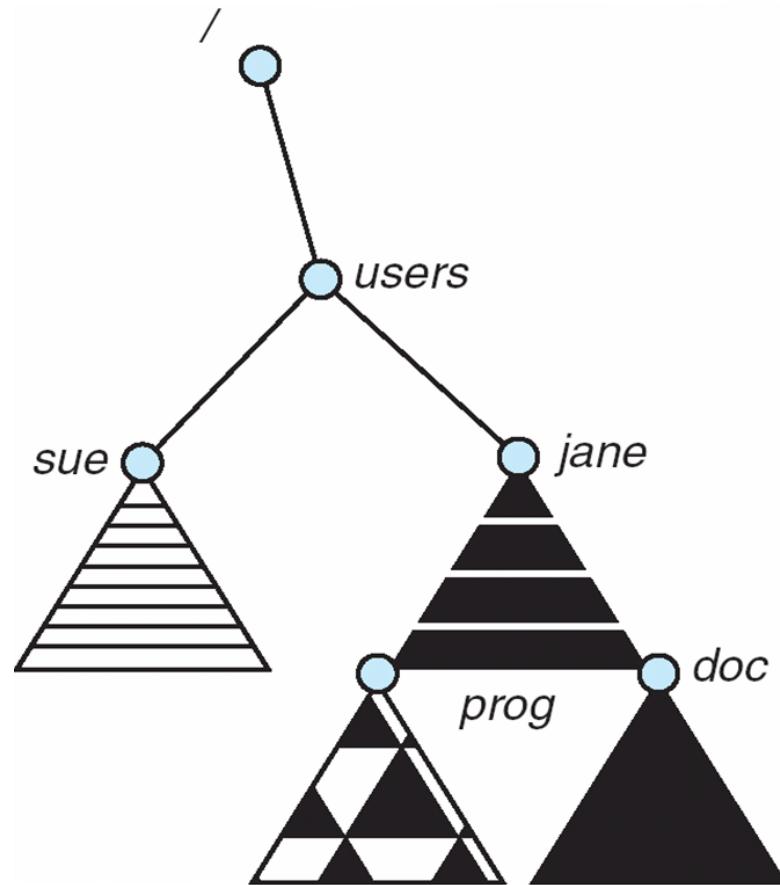
File System Mounting

- A file system must be **mounted** before it can be accessed
- An unmounted file system (i.e., Fig. 10-11(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
 - **NFS** is standard UNIX client-server file sharing protocol
 - **CIFS** is standard Windows protocol
 - Standard operating system file calls are translated into remote calls
- 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
 - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- 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 Semantics

- Specify how multiple users are to access a shared file simultaneously
 - Similar to Ch 6 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
 - 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





Protection

- File owner/creator should be able to control:
 - what can be done
 - by whom

- Types of access
 - **Read**
 - **Write**
 - **Execute**
 - **Append**
 - **Delete**
 - **List**



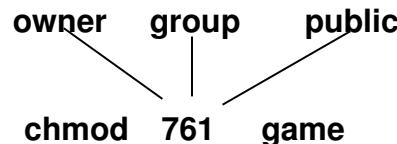


Access Lists and Groups

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

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

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game





Windows 7 Access-Control List Management

ListPanel.java Properties

General Security Details Previous Versions

Object name: H:\DATA\Patterns Material\Src\ListPanel.java

Group or user names:

- SYSTEM
- Gregory G. Gagne (ggagne@wcusers.int)
- Guest (WCUSERS\Guest)**
- FileAdmins (WCUSERS\FileAdmins)
- Administrators (FILES\Administrators)

To change permissions, click Edit.

Edit...

Permissions for Guest	Allow	Deny
Full control	✓	
Modify	✓	
Read & execute	✓	
Read	✓	
Write	✓	
Special permissions		

For special permissions or advanced settings, click Advanced.

Advanced

[Learn about access control and permissions](#)

OK Cancel Apply





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/

