

Operating System Concepts



LI Wensheng, SCST, BUPT



Hello, everyone!

**My name is Li WenSheng.
Glad to meet you.**

**You can get in touch with me by
E-mail: wenshli@bupt.edu.cn
or Tel.: 010-62282929**

Course program

■ Book:

- **Operating System Concepts(7th Edition)**

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne

■ References

- **Operating Systems Principles**

Lubomir F.Bic, Alan C. Shaw (清华大学出版社影印版)

- **Operating System Internals and Design Principles (4th Edition)**

William Stallings (电子工业出版社)

- **现代操作系统**

汤小丹, 梁红兵等, 电子工业出版社, 2008.4

- **Linux 操作系统内核实习**

Gary Nutt 著, 潘登 冯锐 等译, 机械工业出版社

- **操作系统习题与解答**

J. Archer Harris 著, 机械工业出版社

Course program (Cont.)

- **64 teaching hours: 48 + 16**
- **Form:**
 - Prelection
 - Homework
 - Class discussion
 - Practice
- **Assistant:**
 -
 -
- **Q&A**
 - **time:**
 - **location:**

Contents

Part 1 Overview

- 1 Introduction**
- 2 Operating-System Structures**

Part 2 Process Management

- 3 Processes**
- 4 Threads**
- 5 CPU Scheduling**
- 6 Process Synchronization**
- 7 Deadlocks**

Part 3 Memory Management

- 8 Main Memory**
- 9 Virtual Memory**

Part 4 Storage Management

- 10 File-System Interface**
- 11 File-System Implementation**
- 12 Mass-Storage Structure**
- 13 I/O System**

Part 5 Protection and Security

- 14 Protection**
- 15 Security**

Part 6 Distributed Systems

- 16 Distributed System Structures**
- 17 Distributed File Systems**
- 18 Distributed Coordination**

Requirements

- Preparation
- Take notes
- Review
- homework
- Thinking, discussing, and Practicing
- Q&A

Grade Principles

- | | |
|------------|-----|
| ■ Homework | 20% |
| ■ Midterm | 20% |
| ■ final | 60% |

***Work hard
and
make progress!***

Chapter 1 Introduction



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

the definition and functions of operating system

Major components of an operating system

kinds of operating systems and their features

Chapter Objectives

- **To provide a grand tour of the major operating systems components**
- **To provide coverage of basic computer system organization**

Contents

- 1.1 What Operating Systems Do**
- 1.2 Computer-System Organization**
- 1.3 Computer-System Architecture (*)**
- 1.4 Operating-System Structure**
- 1.5 Operating-System Operations**
- 1.6 Process Management**
- 1.7 Memory Management**
- 1.8 Storage Management**
- 1.9 Protection and Security**
- 1.10 Distributed Systems (*)**
- 1.11 Special-Purpose Systems (*)**
- 1.12 Computing Environments**

1.1 What Operating Systems do?

- **OS is a system program that acts as an intermediary between a user of a computer and the computer hardware**
 - A program that controls the execution of application programs
 - An interface between applications/users and hardware
- **Operating system's objectives:**
 - **Convenience**
 - Execute user programs and make solving user problems easier.
 - Makes the computer system more convenient to use
 - **Efficiency**
 - use the computer hardware in an efficient manner
 - **Ability to evolve**
 - Permit effective development, testing, and introduction of new system functions without interfering with service

Computer System Components

1. Hardware

provides basic computing resources, such as CPU, memory, I/O devices.

2. Operating system

controls and coordinates the use of the hardware among the various application programs for the various users.

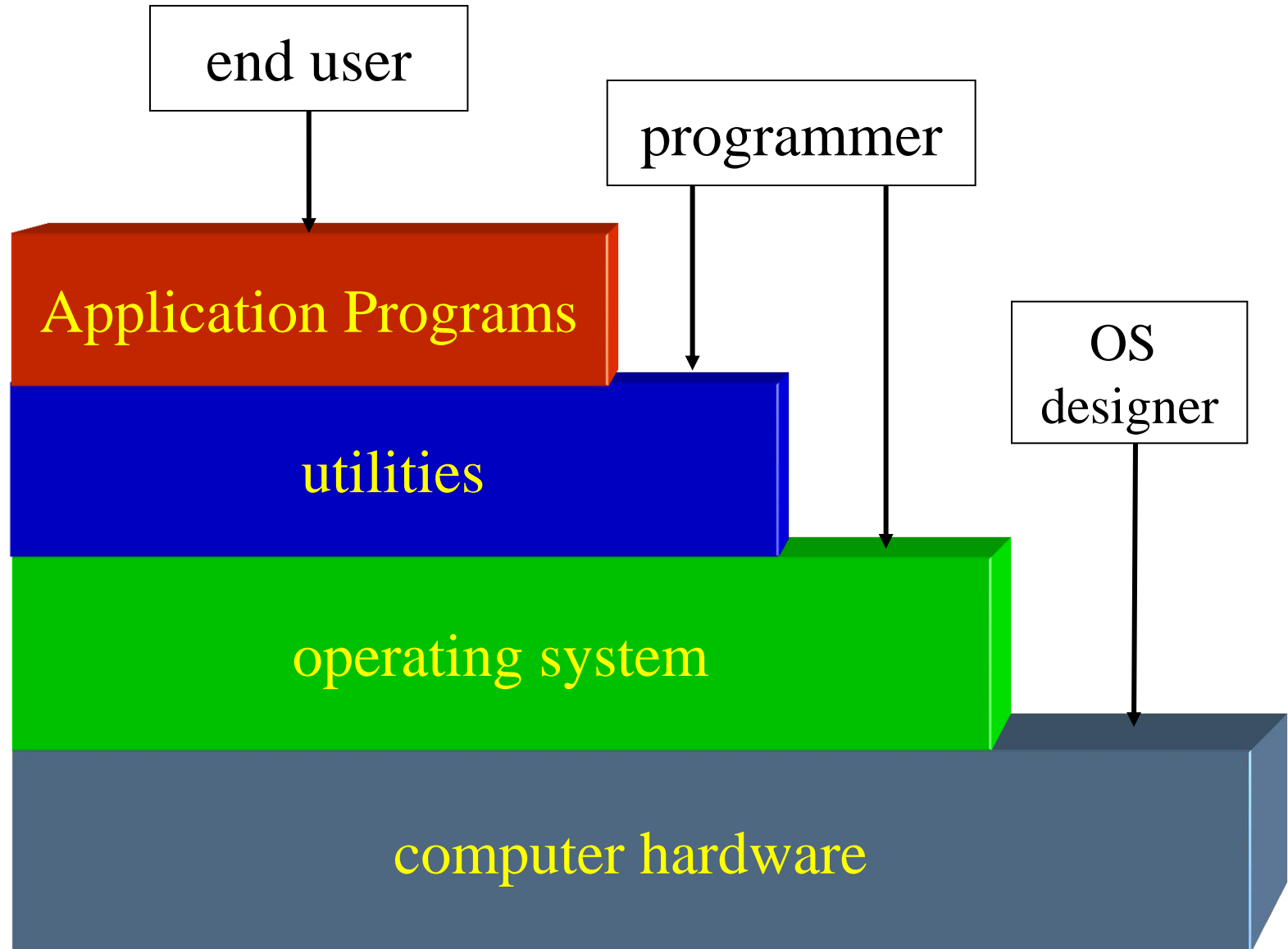
3. Applications programs

**define the ways in which the system resources are used to solve user's computing problems
(compilers, DBMS, database systems, word processors, video games, business programs).**

4. Users

people, machines, other computers

Computer System Structure



1.1.1 User view

- **Varies according to the interface being used.**
- **Designation goal for different OS**
 - **PC**
 - ease to use, performance
 - **Mainframe, minicomputer**
 - Maximize resource utilization
 - **workstation**
 - Usability and resource utilization
 - **Handheld computer**
 - Individual usability, performance per amount of battery life
 - **Embedded computer**
 - Have little or no user view

Services Provided by the OS. (1/2)

- **Program development**
 - Editors and debuggers
- **Program execution**
- **Access to I/O devices**
- **Controlled access to files**
- **System access**

Services Provided by the OS. (2/2)

■ Error detection and response

- internal and external hardware errors
 - memory error
 - device failure
- software errors
 - arithmetic overflow
 - access forbidden memory locations
- operating system cannot grant request of application

■ Accounting

- collect statistics
- monitor performance
- used to anticipate future enhancements
- used for billing users

1.1.2 System view of Operating System

■ Resource allocator

- manages and allocates resources.

■ Control program

- controls the execution of user programs and operations of I/O devices .
- Operating system **relinquishes**(放弃) control of the processor to execute other programs

■ Kernel

- OS is the one program running at all times on the computer (all else being system programs and application programs).
- Portion of operating system that is in main memory
- Contains most-frequently used functions
- Also called the nucleus

操作系统的定义

操作系统是计算机系统中的一个系统软件，
是一些程序模块的集合，
它们能

以尽量有效、合理的方式组织和管理计算机的软硬件资源，

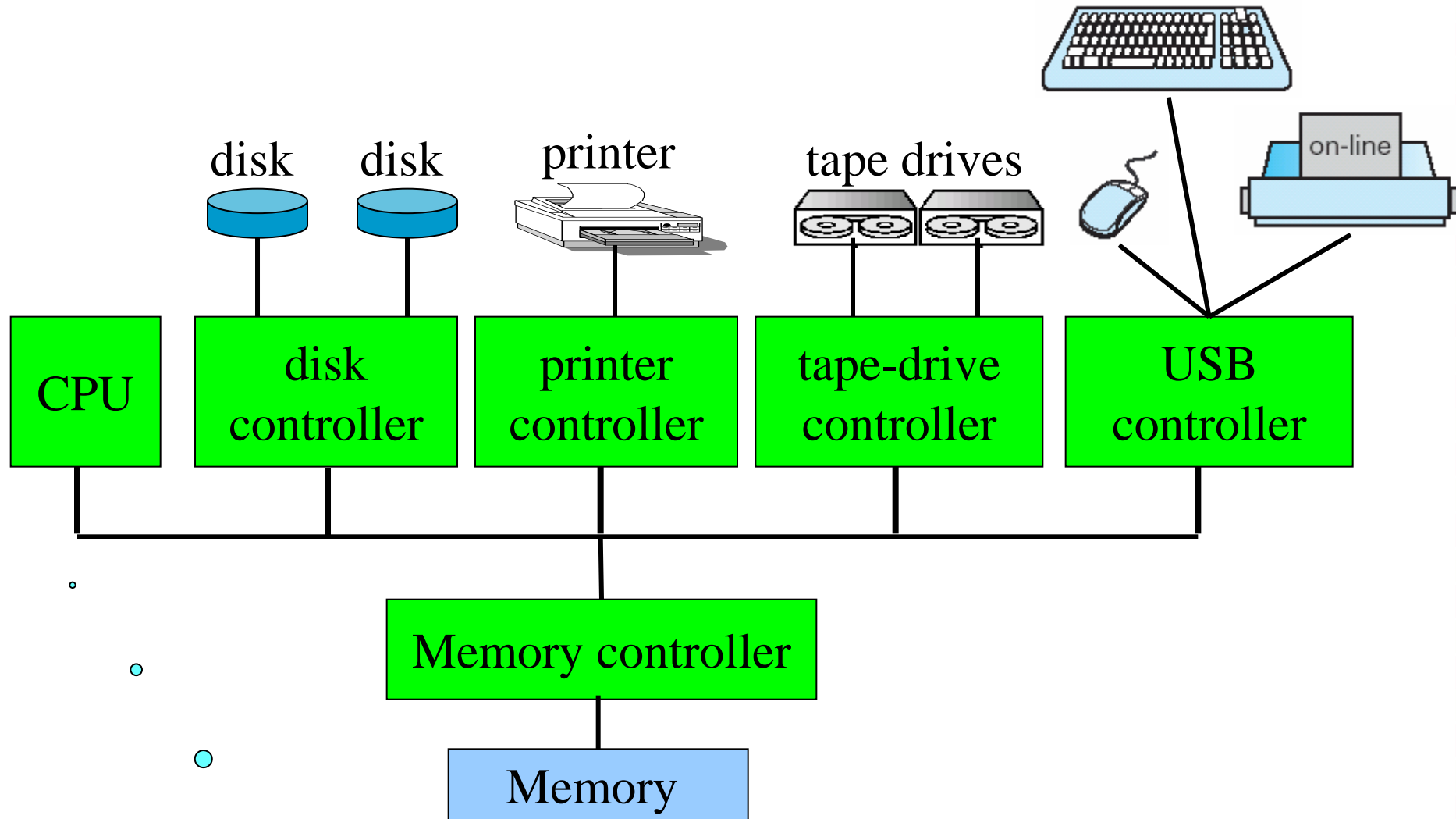
合理地组织计算机的工作流程，控制程序的执行，
向用户提供各种服务功能，

使得用户能够灵活、方便、有效的使用计算机，
使整个计算机系统能高效地运行。

1.2 Computer-System Organization

- **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**

A modern Computer System



Types of registers?

Computer Startup

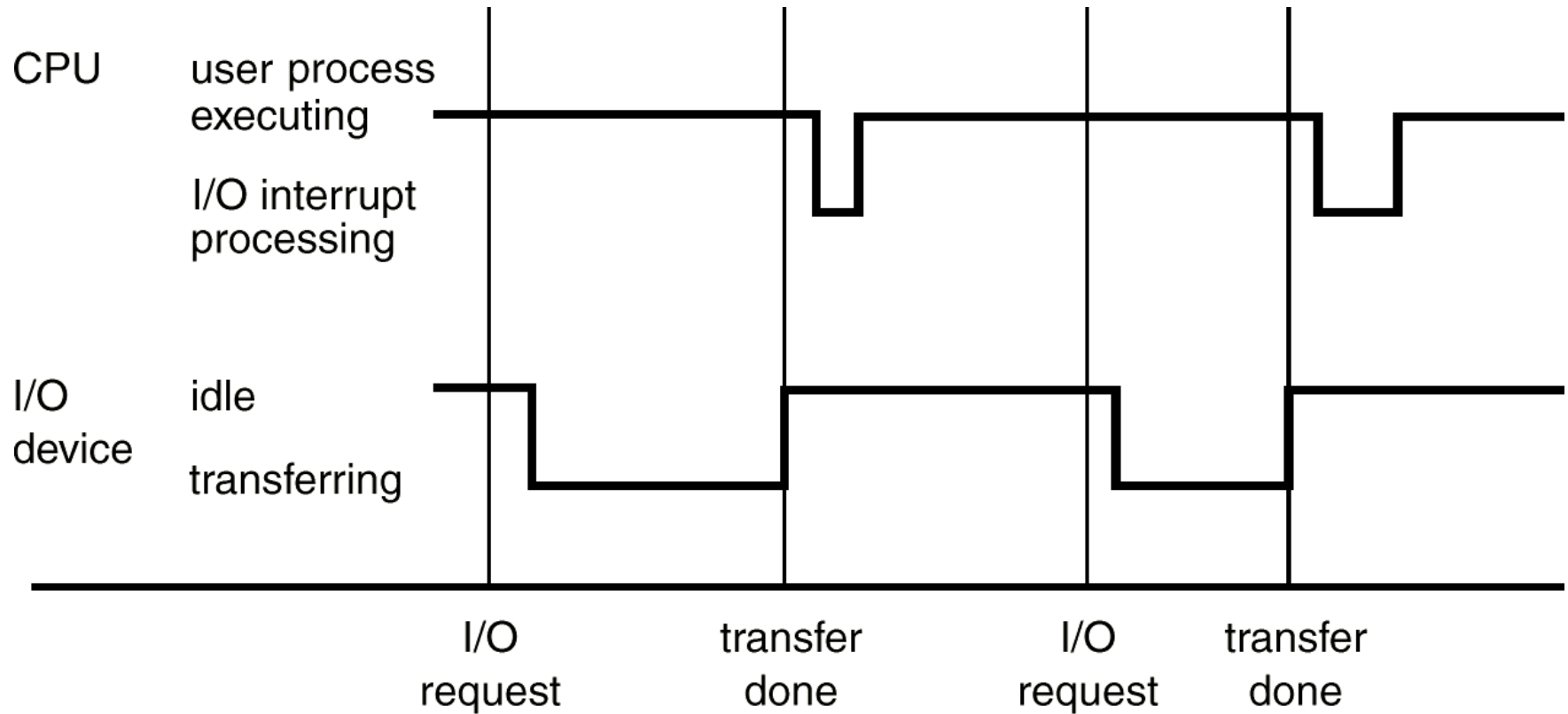
- **bootstrap program** is loaded at power-up or reboot
 - Typically stored in ROM or EEPROM, generally known as **firmware**
 - Initializes all aspects of system
 - From CPU registers to device controllers to memory contents
 - Loads operating system kernel and starts execution
- **OS starts executing the first process, such as “init” and waits for some events to occur**
 - An interrupt from the hardware
 - Sending a signal to CPU by way of the system bus
 - An interrupt from the software
 - Executing a system call

1.2.1 Computer-System Operation

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

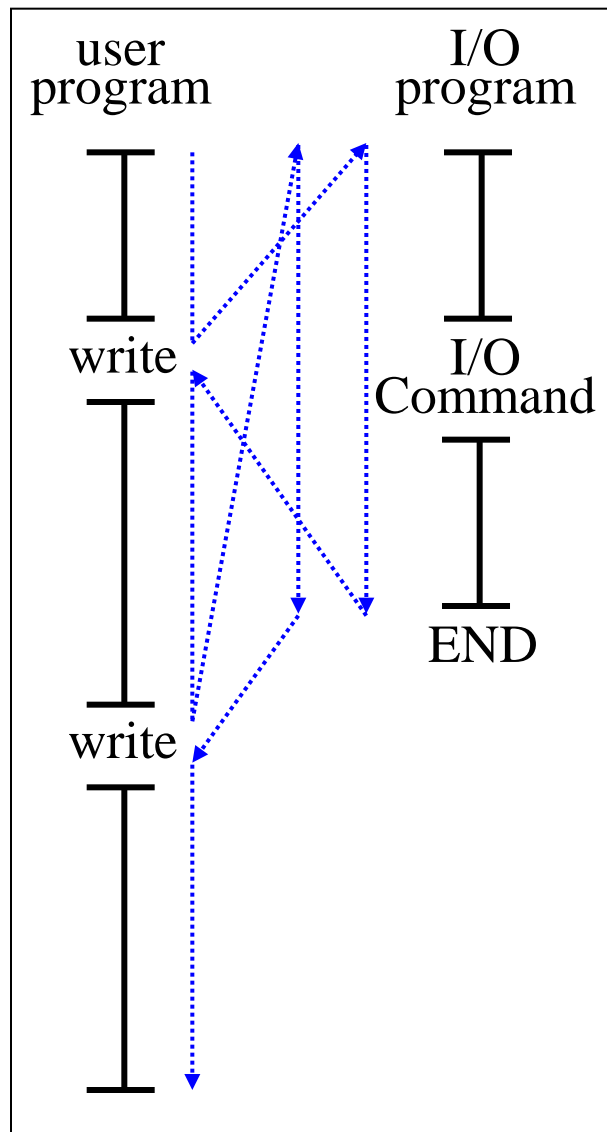


Interrupt Timeline

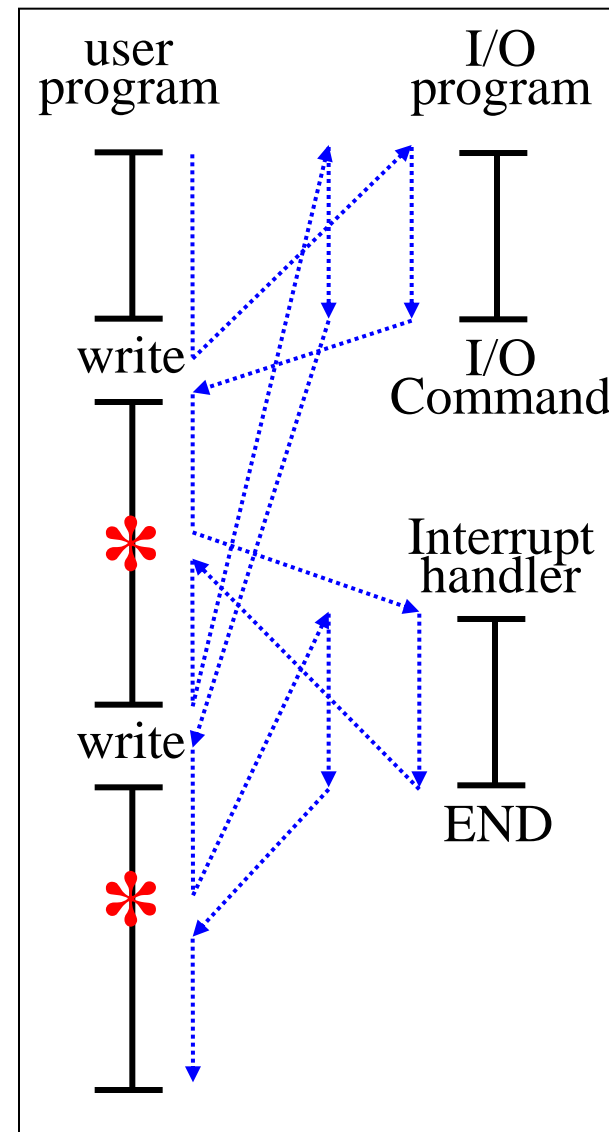


Supplement:

Program Flow of Control Without and With Interrupt



No interrupts



Interrupts, short I/O wait

1.2.2 Storage Structure

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

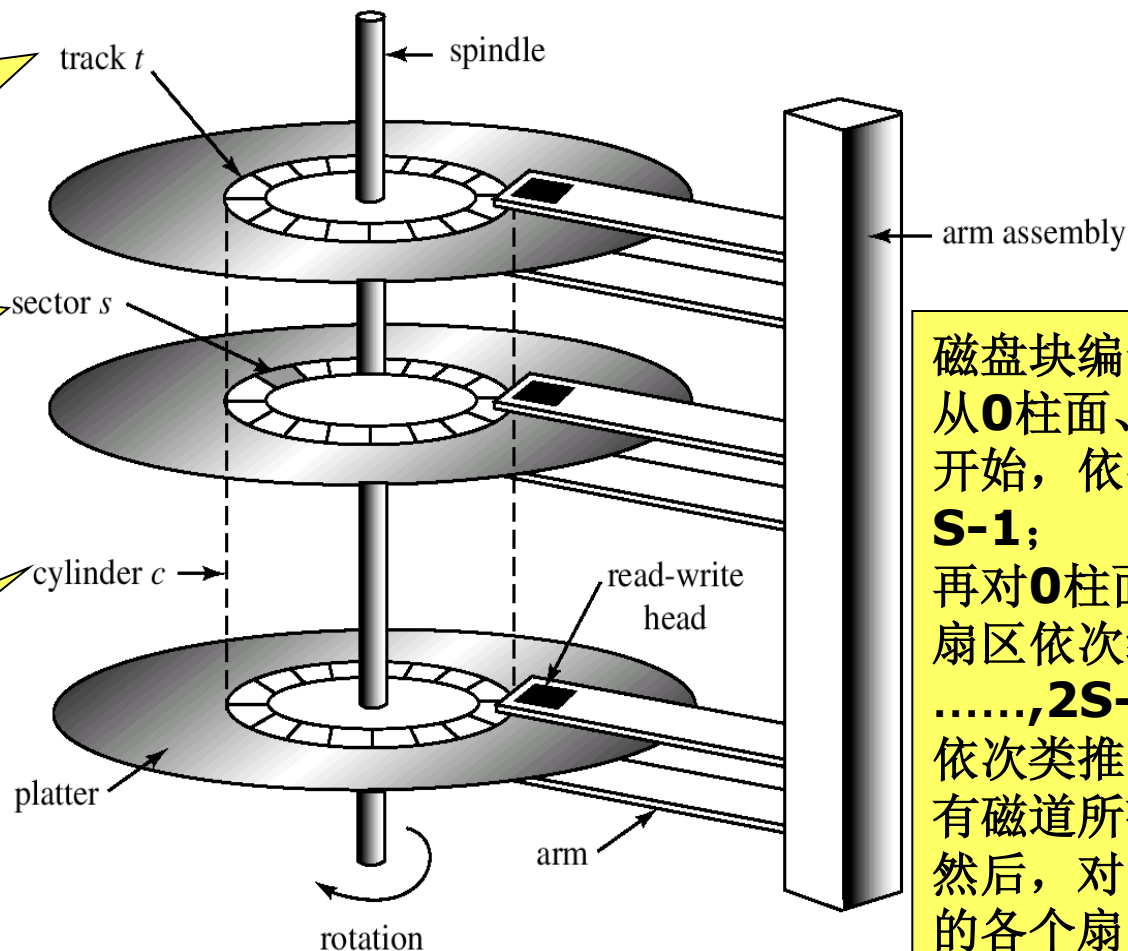
Supplement:

Moving-Head Disk Mechanism

编号:
从上向下
0, 1, ..., T-1

编号:
依次为
0, 1, ..., S-1

编号:
从外向内
0, 1, ..., C-1

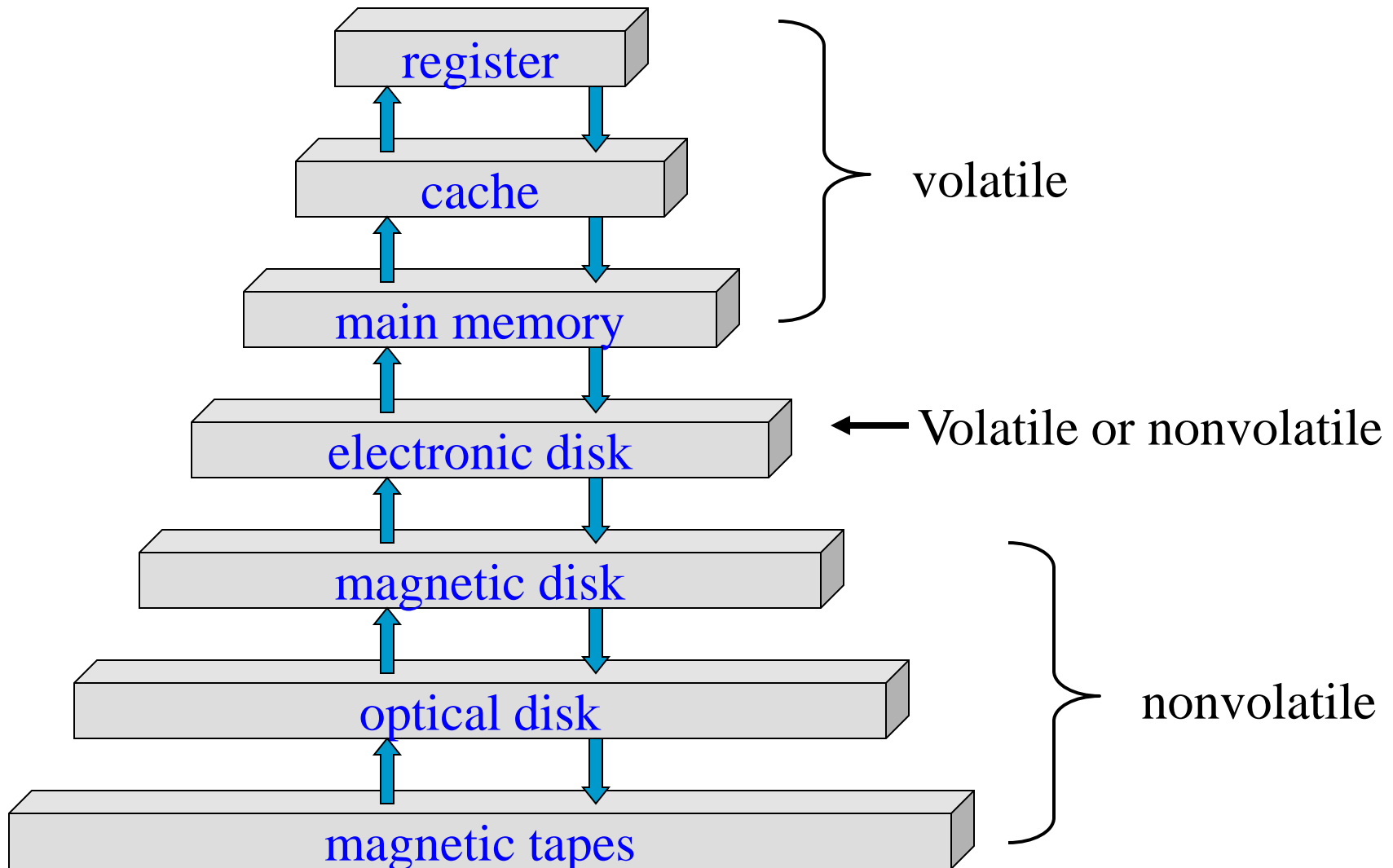


磁盘块编号:
从**0**柱面、**0**磁道的**0**扇区
开始, 依次为**0, 1, ..., S-1**;
再对**0**柱面、**1**磁道的**s**个
扇区依次编号为**S, S+1, ..., 2S-1**;
依次类推, 对柱面**0**的所有
磁道所有扇区进行编号;
然后, 对**1**柱面的**0**磁道的
各个扇区进行编号,...;
直到对所有柱面上所有磁
道的所有扇区编完为止。

Storage Hierarchy

- **Storage systems organized in hierarchy.**
 - **Speed**
 - **Cost**
 - **Volatility**
- **Volatile storage loses its contents when the power to the device is removed.**
- **Principle of design a computer memory system**
 - **uses only as much expensive memory as necessary**
 - **provides as much inexpensive, nonvolatile memory as possible.**

Storage-Device Hierarchy



1.2.3 I/O Structure



Types of I/O?

- **Programmed I/O**
- **Interrupt-Driven I/O**
 - **Synchronous I/O**
 - **Asynchronous I/O**
- **DMA**

I/O operation

■ device controller

- Local buffer storage
- A set of special-purpose registers
- Moving data between device and its local buffer storage

■ device driver

- One for each device controller
- Presents a uniform interface to the device

■ I/O operation

- Device driver loads registers within the controller
- Controller examines the contents of the registers to determine what action to take
- Controller starts the transfer of data between the device and its local buffer
- Once done, controller informs the driver via an interrupt
- Driver returns control to the OS, with data if ‘read’

1.3* Computer-System Architecture

■ Single-processor system

- One main CPU executing a general-purpose instruction set
- special-purpose processors run a limited instruction set and do not run user processes, such controllers, I/O processors

■ Multiprocessor system/parallel system

- Two or more processors in close communication, sharing the computer bus and the clock, memory and I/O devices
- Advantages
 - Increased throughput
 - Economy of scale
 - Increased reliability

Multiprocessor system

- **Symmetric multiprocessing**
 - Each processor performs all tasks within the OS
 - All processors are peers
- **Asymmetric multiprocessing**
 - Each processor is assigned a specific task
 - A master processor controls the system
 - The other slave processors look to the master for instruction or have predefined tasks
 - Master processor schedules and allocate work to the slave processors
- **Multi-core CPUs**
 - Multiple compute cores on a single chip
 - Two-way chips is becoming mainstream

Computer-System Architecture (Cont.)

■ Clustered system

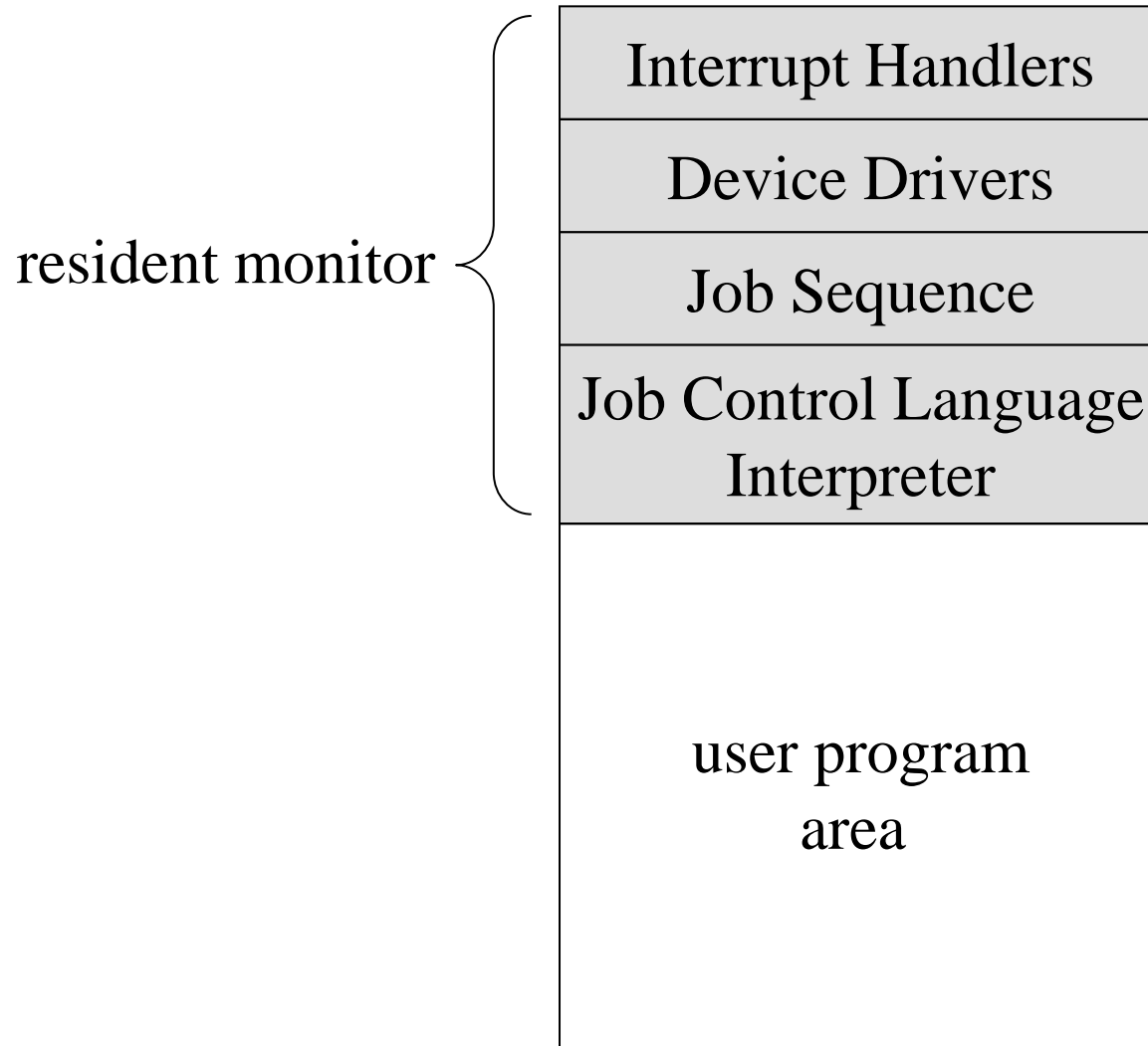
- **Clustered computers share storage and are closely linked via a LAN or a faster interconnect such as InfiniBand.**
- **Used to provide high-availability service**
- **A layer of cluster software runs on the cluster nodes**
- **Asymmetric clustering**
 - **One machine is in hot-standby mode while the other is running the applications**
 - **The hot-standby host machine does nothing but monitor the active server**
- **Symmetric clustering**
 - **Two or more hosts are running applications and are monitoring each other**

1.4 Operating-System Structure

- **Batch systems**
- **Timesharing systems**
- **Real-time systems**

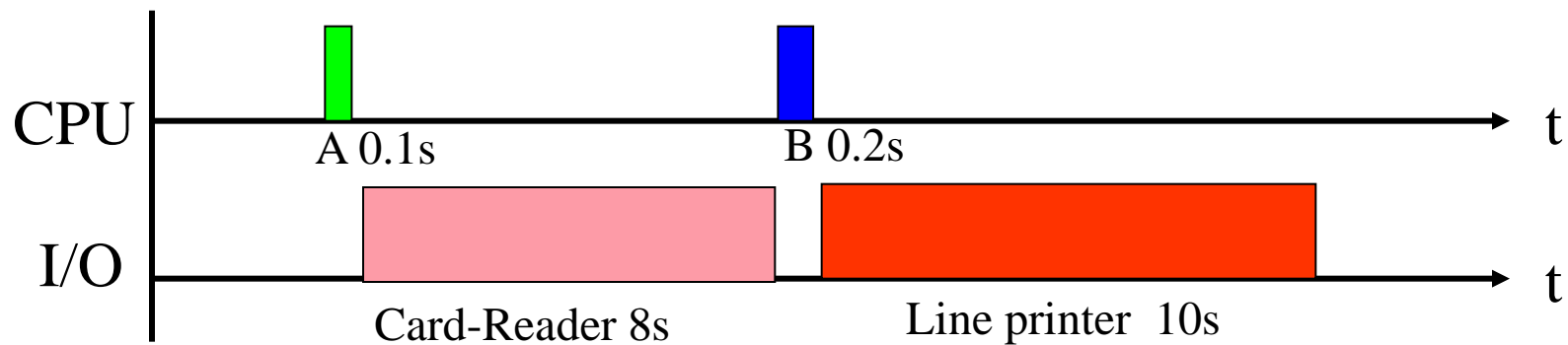
Supplement:

Memory Layout for a Simple Batch System



Supplement: Uniprogramming

- **Problem: low Performance – I/O and CPU could not overlap ; card reader very slow.**
 - Processor must wait for I/O instruction to complete before proceeding



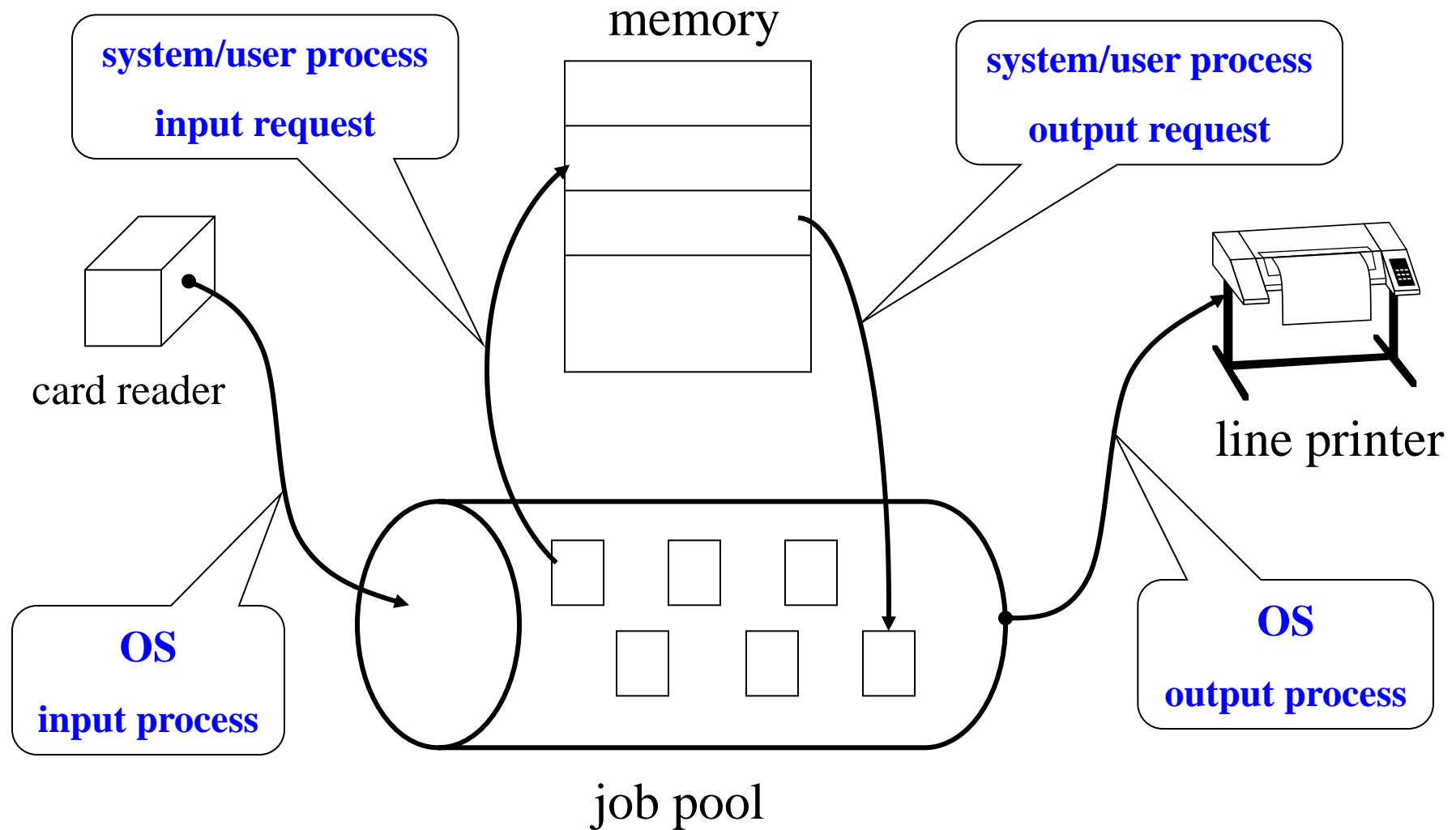
- **solution: Off-line operation – speed up computation by**
 - loading jobs into memory from tapes
 - card reading and line printing done off-line.

Supplement:

problem and solution (cont.): Spooling

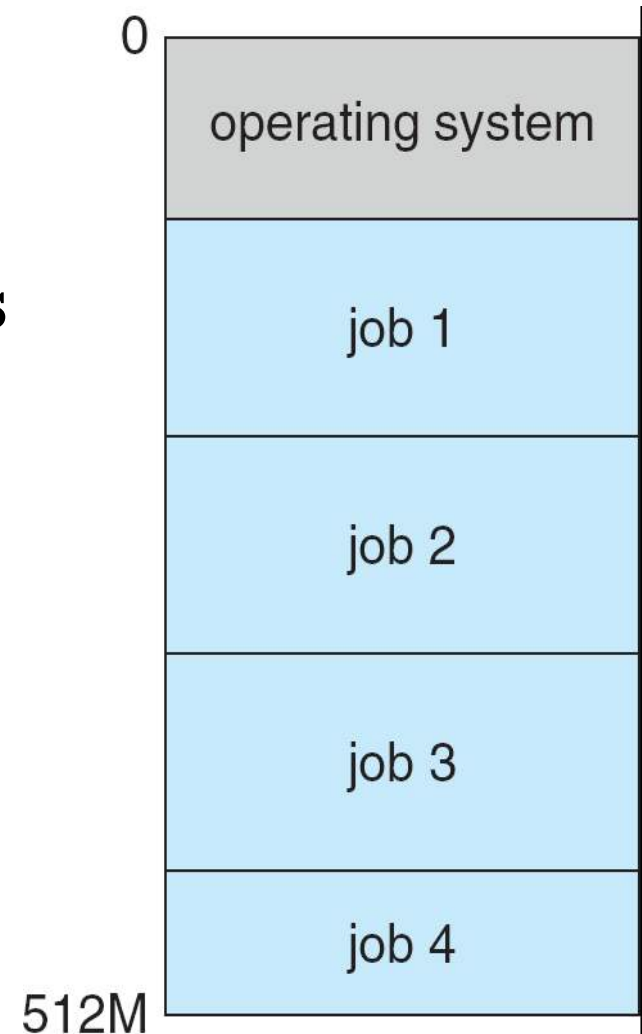
- **Simultaneous Peripheral Operation On Line**
- **Overlap I/O of one job with computation of another job. While executing one job, the OS.**
 - Reads next job from card reader into a storage area on the disk (job queue).
 - Outputs printout of previous job from disk to printer.
- ***Job pool*** – data structure that allows the OS to select which job to run next in order to increase CPU utilization.

Supplement: spooling(cont.)



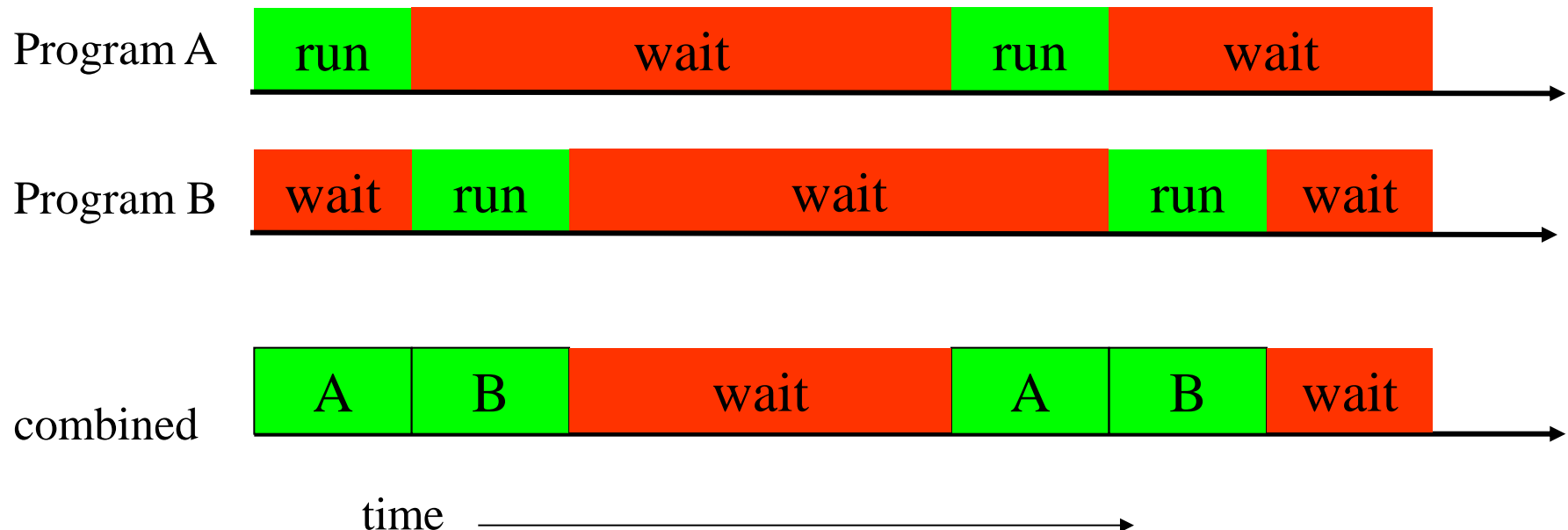
Multiprogramming needed for efficiency

- 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.



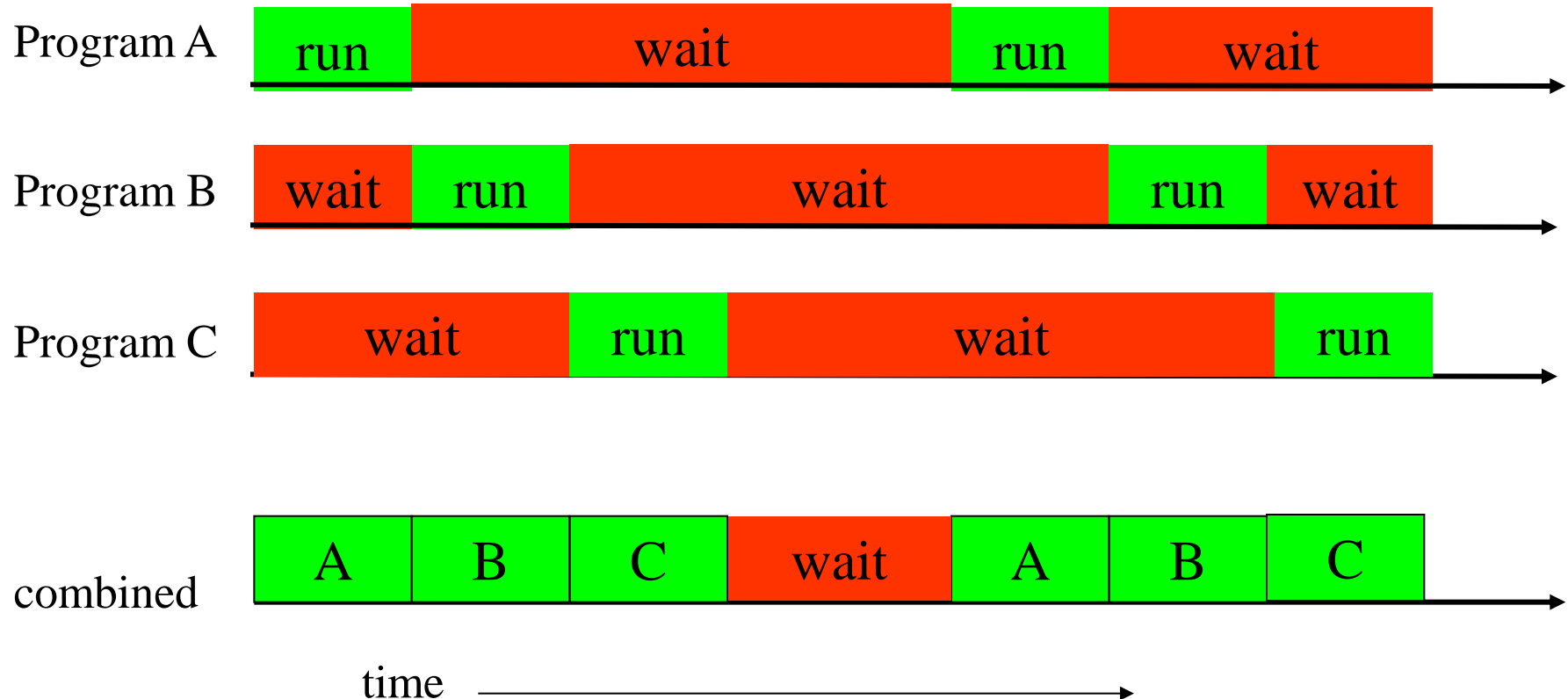
Supplement: Multiprogramming

- When one job needs to wait for I/O, the processor can switch to the other job.
- Multiprogramming with two programs



Supplement: Multiprogramming (Cont.)

■ Multiprogramming with three programs



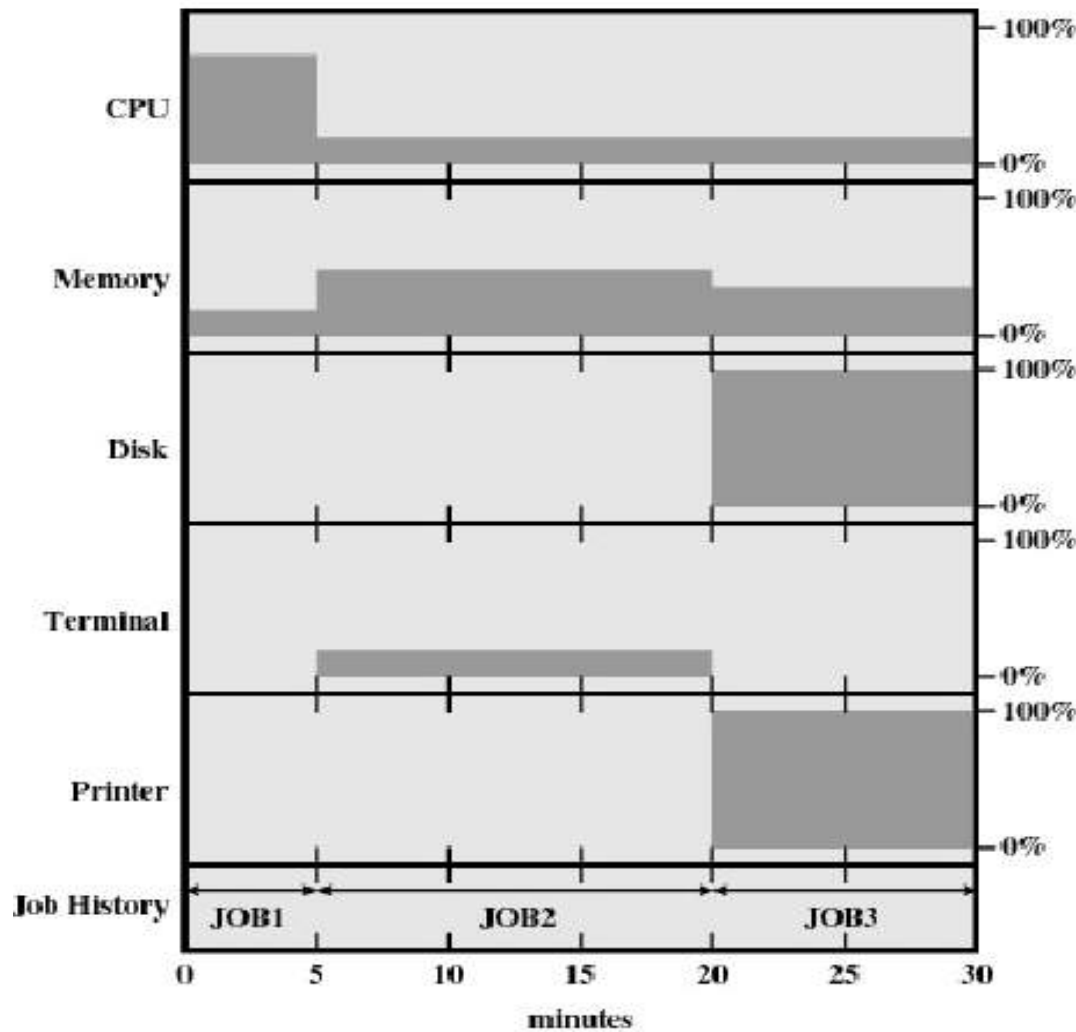
Supplement: Example

A computer:

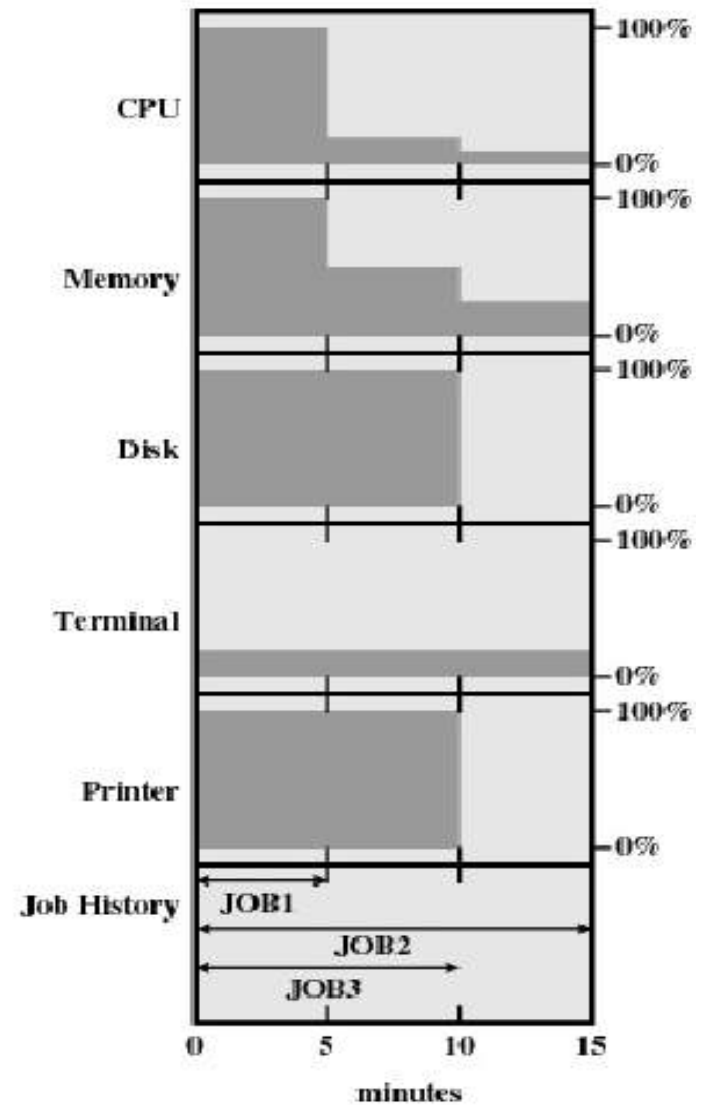
256KB memory, a disk, a terminal, and a printer

	Job1	Job2	Job3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min.	15 min.	10 min.
Memory required	50K	100K	80K
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Supplement: Utilization histograms



(a) Uniprogramming



(b) Multiprogramming

Supplement:

OS Features Needed for Multiprogramming

- I/O routines supplied by the system.
- Memory management – the system must allocate the memory to several jobs.
- CPU scheduling – the system must choose among several jobs ready to run.
- Allocation of devices.

Supplement: Timesharing systems

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
 - **Response time** should be < 1 second
 - Each user has at least one program executing in memory \Rightarrow **process**
 - If several jobs ready to run at the same time \Rightarrow **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

Supplement:

Timesharing systems (Cont.)

- Using multiprogramming to handle multiple interactive jobs
- The CPU is **multiplexed** (复用) among several jobs that are kept in memory and on disk
 - Processor's time is shared among multiple users
 - the CPU is allocated to a job only if the job is in memory.
- A job swapped in and out of memory to the disk.
- Many users share the computer simultaneously.
 - On-line communication between the user and the system is provided;
 - Multiple users simultaneously access the system through terminals;
 - when the operating system finishes the execution of one command, it seeks the next “control statement” not from a card reader, but rather from the user's keyboard.
- On-line system must be available for users to access data and code.

OS Features

Needed for time-sharing system

- I/O routines supplied by the system.
- Memory management – the system must allocate the memory to several jobs.
- Virtual memory – allows the execution of job not be completely in memory.
- Disk management – file system must be provided, which resides on a collection of disks.
- CPU scheduling – mechanisms for concurrent execution, job synchronization and communication, avoiding deadlock.
- Allocation of resources.

Supplement:

Batch Multiprogramming versus Time Sharing

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

1.5 Operating-System Operations

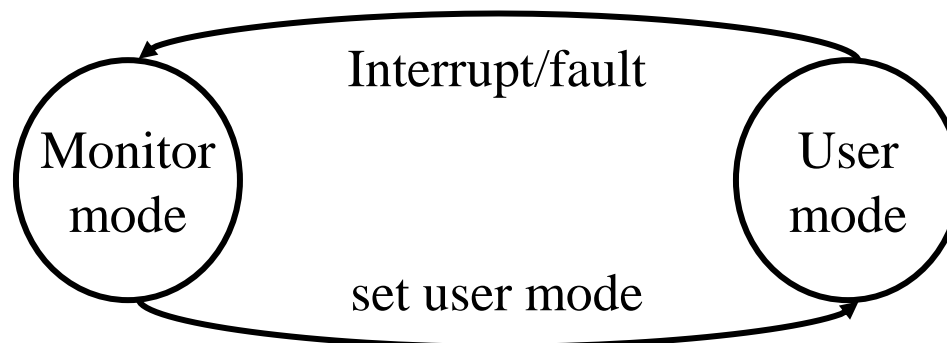
- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
 - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
 - Provide hardware support to differentiate between at least two modes of operations.
 1. **User mode** – execution done on behalf of a user.
 2. **Monitor mode** (also **kernel mode** or **system mode**) – execution done on behalf of operating system.

Dual-Mode Operation

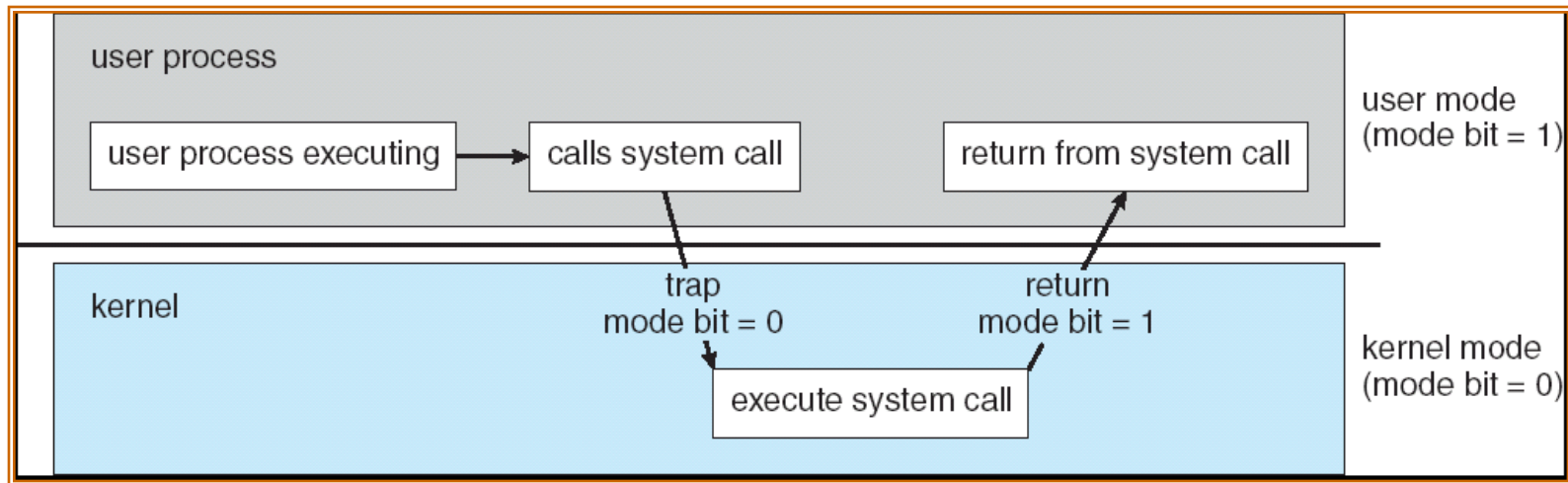
- **Mode bit** provided by hardware

- monitor (0) or user (1)
- 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

- **When an interrupt or fault occurs hardware switches to monitor mode.**



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
 - Instructions that modify the content of the timer are privileged.

1.6 Process Management

- A process is a program in execution. a unit of work within the system. The entity that can be assigned to and executed on a processor
- 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(归还) 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 OS 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**

1.7 Memory Management

- **Central to the operation of a modern computer system**
 - All data in memory before and after processing
 - All instructions in memory in order to execute
- **The only large storage device that CPU is able to address and access directly**
- **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

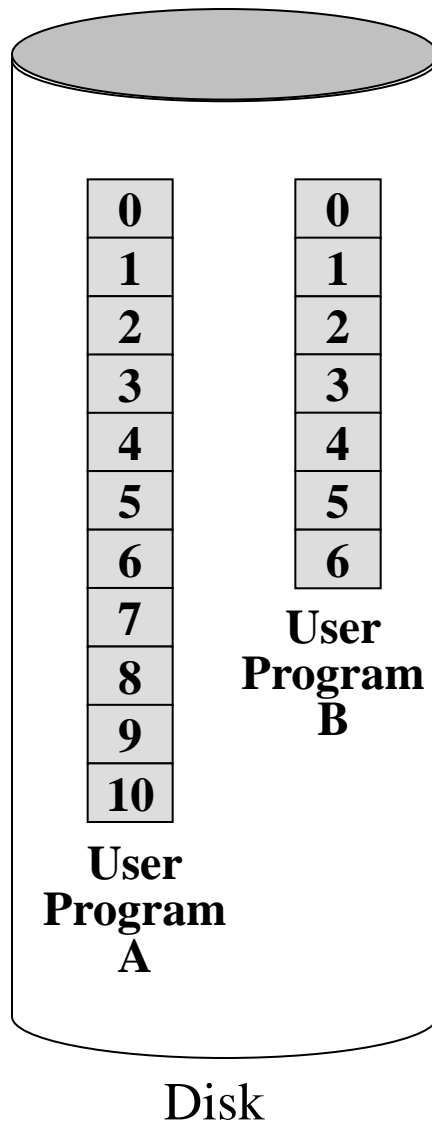
Supplement: Virtual Memory

- Allows programmers to address memory from a logical point of view
- While one process is written out to secondary store and the successor process read in there in no hiatus (裂缝)

Supplement: Paging

- Allows process to be comprised of a number of fixed-size blocks, called pages
- **Virtual address** is a **page number** and an **offset** within the page
- Each page may be located any where in main memory
- Real address or physical address in main memory
- Address mapping

Supplement: Virtual Memory Concept



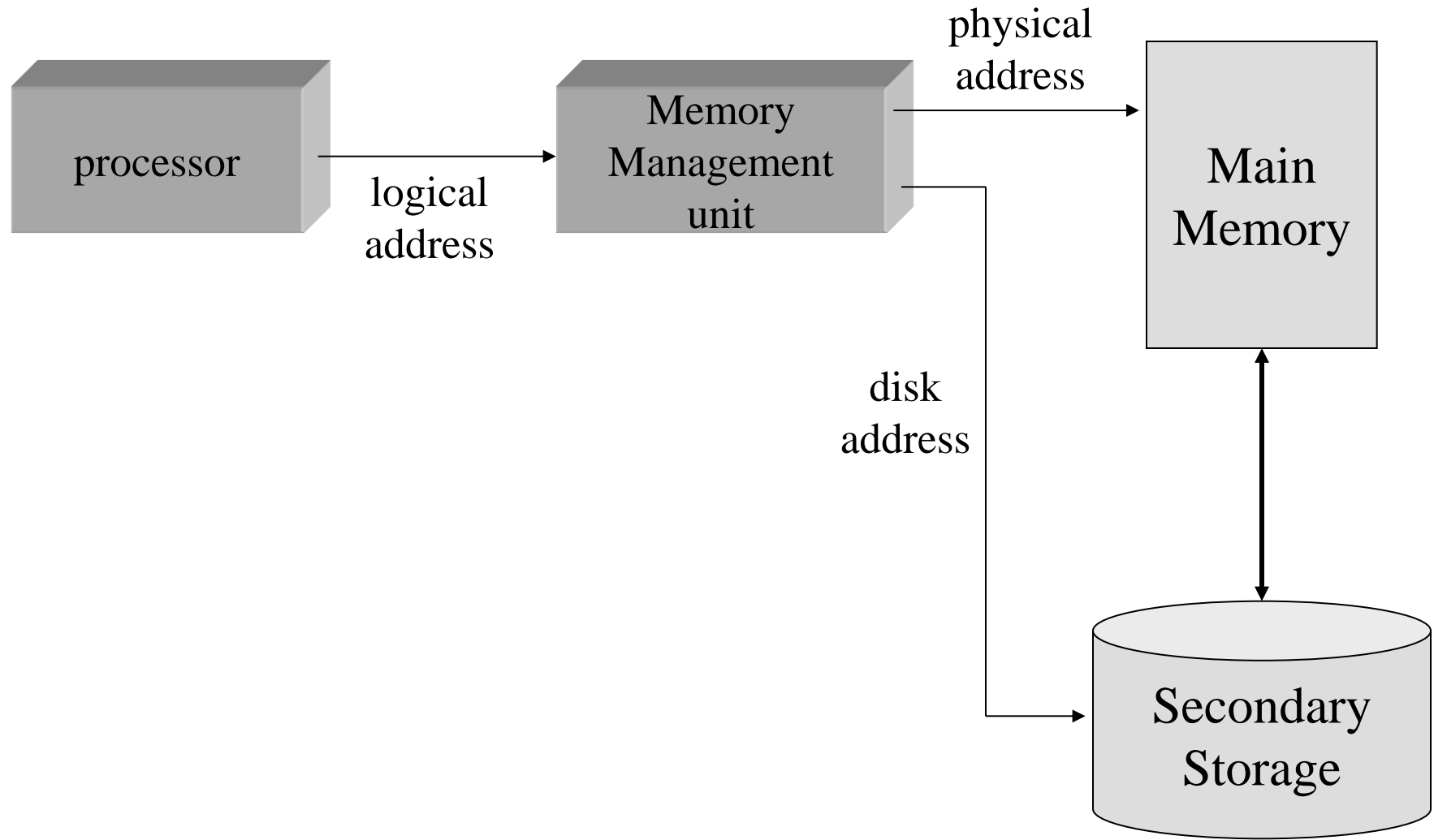
A.1			
	A.0	A.2	
	A.5		
B.0	B.1	B.2	B.3
		A.7	
	A.9		
		A.8	
B.4	B.5	B.6	

Main Memory

Main memory consists of a number of fixed-length frames, equals to the size of a page. For a program to execute, some or all of its pages must be in main memory.

Secondary storage (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages for all programs plus the Operating System are on disk, as are files.

Supplement: Virtual Memory Addressing



1.8 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 (原语) to manipulate files and dirs**
 - **Mapping files onto secondary storage**
 - **Backup files onto stable (non-volatile) storage media**

Mass-Storage Management

- Usually disk is 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
 - Varies between WORM (write-once, read-many-times) and RW (read-write)

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
- *Caching* – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary 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 policy important design problem
 - Cache size and replacement policy

Disk Cache

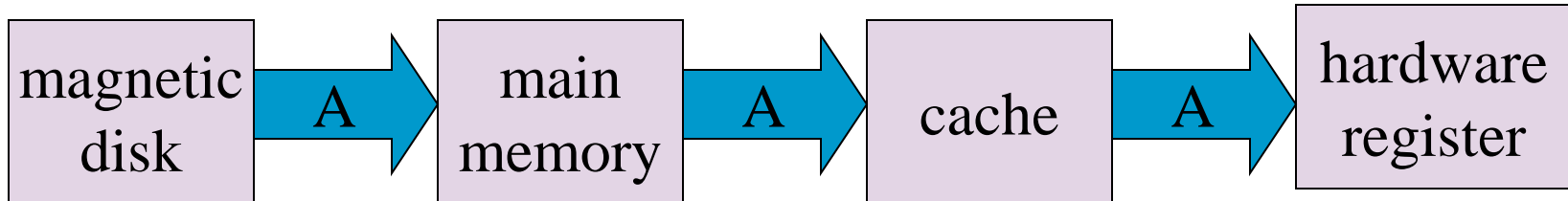
- A portion of main memory used as a buffer to temporarily to hold data for the disk
- Disk writes are clustered (簇)
- Some data written out may be referenced again. The data are retrieved rapidly from the software cache instead of slowly from disk

Cache Memory (高速缓存)

- Invisible to operating system
- Increase the speed of memory
- Processor speed is faster than memory speed

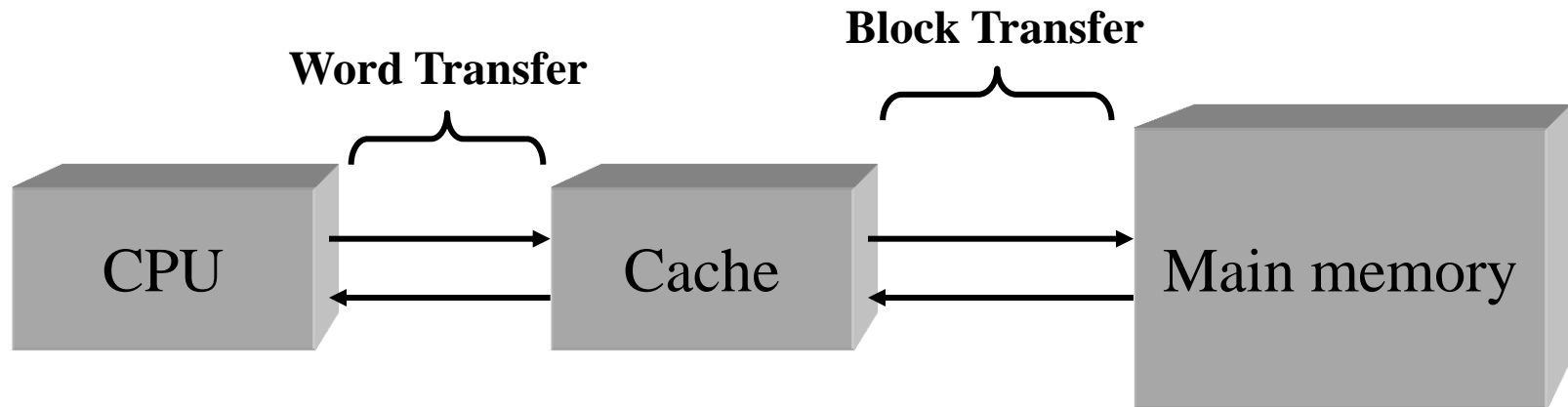
Coherency and Consistency

- The same data may appear in different levels of the storage system.
- Migration of A From Disk to Register



- Computing environment
 - only one process executes at a time
 - multitasking environment
 - multiprocessor environment ---- cache coherency
 - distributed environment

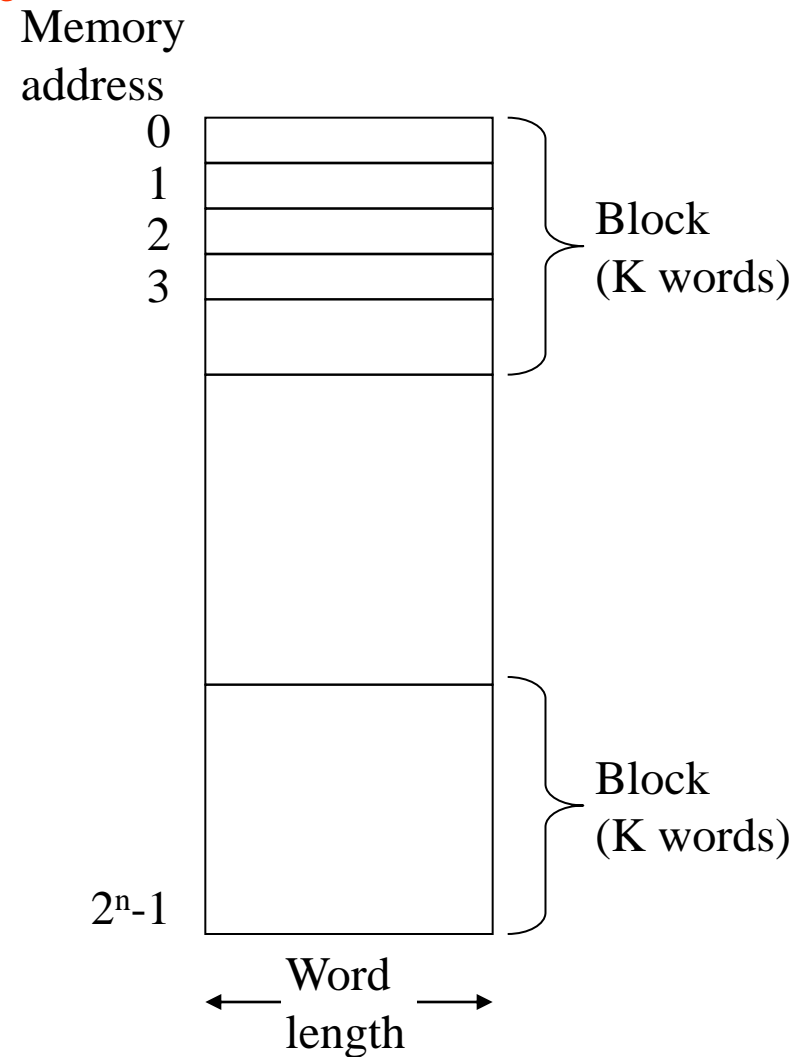
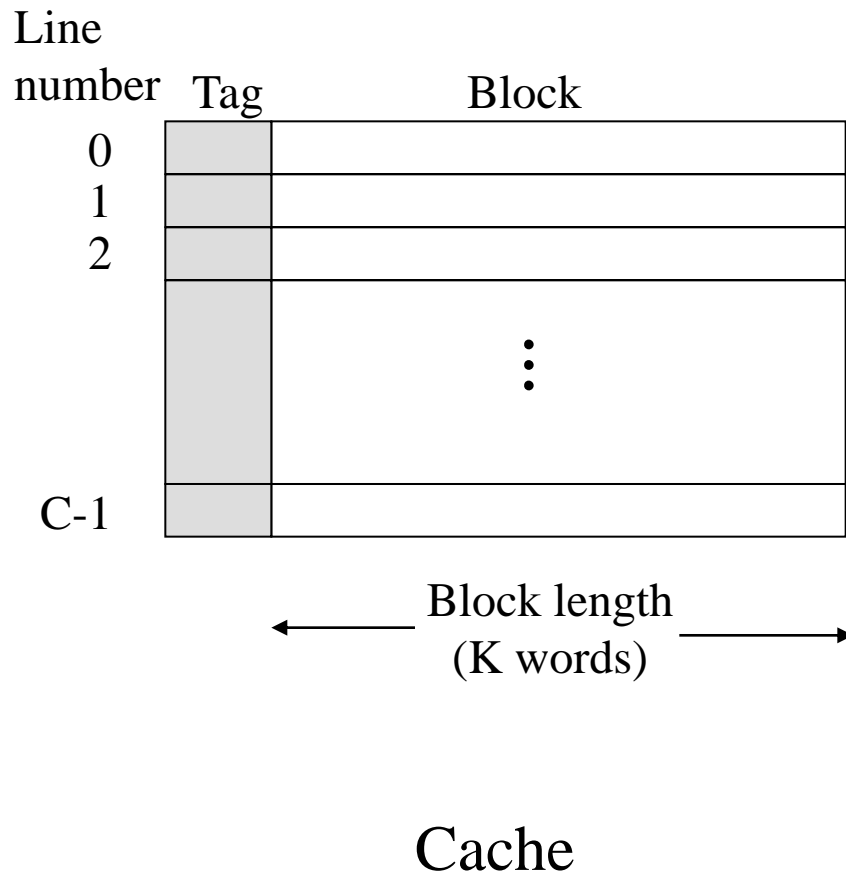
Supplement: Cache Memory



- **Contains a portion of main memory**
- **Processor first checks cache**
- **If not found in cache, the block of memory containing the needed information is moved to the cache**

Supplement:

Cache/Main Memory System Structure



Main memory

Supplement: Cache Design

■ Cache size

- small caches have a significant impact on performance

■ Block size

- the unit of data exchanged between cache and main memory
- hit means the information was found in the cache
- larger block size more hits until probability of using newly fetched data becomes less than the probability of reusing data that has been moved out of cache

Supplement: Cache Design (Cont.)

- **Mapping function**
 - determines which cache location the block will occupy
- **Replacement algorithm**
 - determines which block to replace
 - Least-Recently-Used (LRU) algorithm
- **Write policy: When the memory write operation takes place**
 - Can occur every time block is updated
 - Can occur only when block is replaced
 - Minimizes memory operations
 - Leaves memory in an obsolete(陈旧的) state

I/O Subsystem

- One purpose of OS is to hide peculiarities(特性) 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

1.9 Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
 - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- **Access control**
 - regulate user access to the system
- **Information flow control**
 - regulate flow of data within the system and its delivery to users
- **Certification**
 - proving that access and flow control perform according to specifications

Protection and Security (Cont.)

- **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

1.10* Distributed Systems

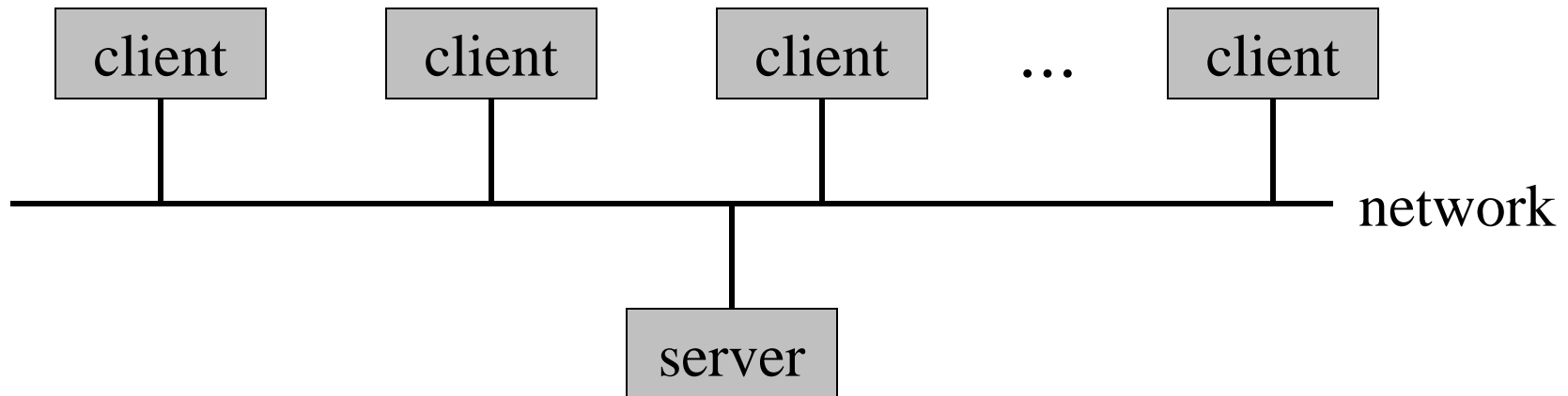
- **Distribute the computation among several physical processors.**
- *Loosely coupled system -- distributed system*
 - each processor has its own local memory;
 - processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.
- **Advantages of distributed systems.**
 - **Resources Sharing**
 - **Computation speed up – load sharing**
 - **Reliability**
 - **Communications**

Distributed Systems (Cont.)

■ Basic concept

- Requires networking infrastructure.
- Local area networks (LAN) or Wide area networks (WAN)
- May be either client-server or peer-to-peer systems.

■ General Structure of Client-Server



Distributed Systems (Cont.)

■ Network Operating System

- provides file sharing across the network
- provides communication scheme allowing different processes on different computers to exchange messages.
- runs independently from other computers on the network

■ Distributed Operating System

- less autonomy between computers
- gives the impression there is a single operating system controlling the network.

1.11* Special-Purpose Systems

- **Real-time systems**
- **Real-time embedded systems**
- **Multimedia systems**
- **Handheld systems**

Real-Time Systems

- Used when there are rigid(严格的) time requirements on the operation of a processor or the flow of data.
- Often used as a control device in a dedicated application, such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.
- Well-defined fixed time constraints.
kernel delays need to be bounded.
- There are two classes of real-time systems
 - hard real-time system
 - soft real-time system.

Real-Time Systems (cont.)

■ Hard real-time system

- goal: to guarantee critical tasks be completed on time.
- Secondary storage limited or absent, data stored in short-term memory, or read-only memory (ROM)
- Conflicts with time-sharing systems, not supported by general-purpose operating systems.

■ Soft real-time system

- a critical real-time task gets priority over other tasks, and retains that priority until it completes.
- Lack of deadline support, limited utility in industrial control or robotics.
- Useful in applications (multimedia, virtual reality, scientific projects) requiring advanced operating-system features.

Real-time embedded systems

- **The OS provide limited features**
 - Have little or no user interface
 - Preferring to spend their time monitoring and managing hardware devices
- **Implementation**
 - General-purpose computers with special-purpose applications
 - Hardware devices with a special-purpose embedded OS
 - Hardware devices with application-specific integrated circuits that perform their tasks without an OS

Multimedia systems

- **Multimedia data consists of audio and video files as well as conventional files**

Handheld systems

- **Personal Digital Assistants (PDAs)**
 - Palm and Pocked-PCs and cellular telephones
 - Use special-purpose embedded OS
- **Issues:**
 - Limited memory size
 - Slow processors
 - Small display screens.

1.12* Computing Environments

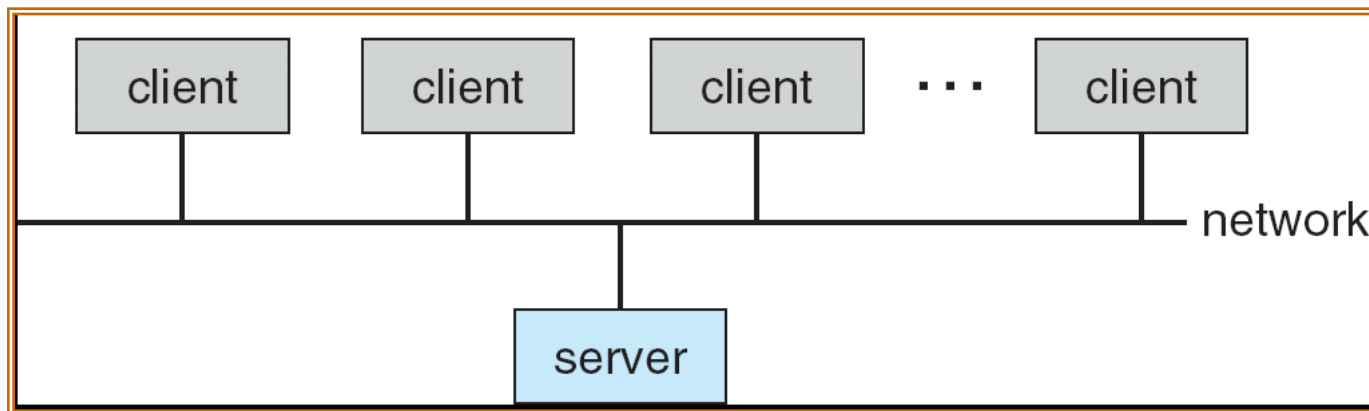
■ Traditional computer

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

Computing Environments (Cont.)

■ Client-Server Computing

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



Peer-to-Peer Computing

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

Web-Based Computing

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

Homework (P.36)

1.17

Thinking:

1.10

1.11

1.13

1.14

1.15

Review: Processor Registers

- **User-visible registers**
 - Enable programmer to minimize main-memory references by optimizing register use
- **Control and status registers**
 - Used by processor to control operating of the processor
 - Used by operating-system routines to control the execution of programs

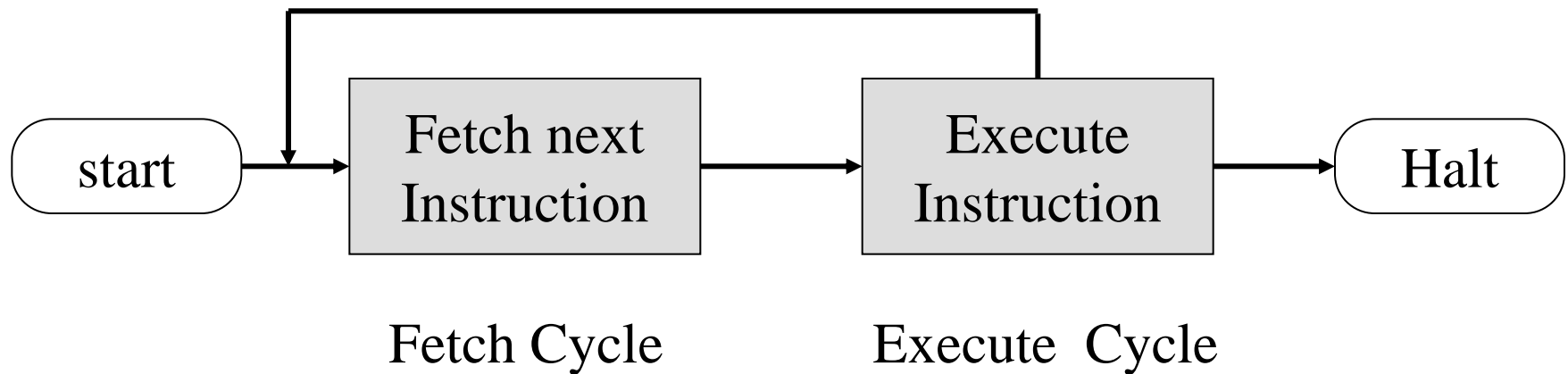
Review: User-Visible Registers

- **May be referenced by machine language**
- **Available to all programs - application programs and system programs**
- **Types of registers**
 - **Data**
 - **Address**
 - **Index:** involves adding an index to a base value to get an address
 - **Segment pointer:** when memory is divided into segments, memory is referenced by a segment and an offset
 - **Stack pointer:** points to top of stack.

Review: Control and Status Registers

- **Program Counter (PC)**
 - Contains the address of an instruction to be fetched
- **Instruction Register (IR)**
 - Contains the instruction most recently fetched
- **Program Status Word (PSW)**
 - condition codes
 - Interrupt enable / disable
 - Monitor (/Supervisor) / user mode
- **Condition Codes or Flags**
 - Bits set by the processor hardware as a result of operations
 - Can be accessed by a program but not altered
 - Examples
 - positive result, negative result, zero
 - Overflow

Review: Basic Instruction Cycle



- **The processor fetches the instruction from memory**
 - Fetched instruction is placed in the instruction register
- **Program counter (PC) holds address of the instruction to be fetched next**
- **Program counter is incremented after each fetch**

Review: Types of instructions

How many types?

- **Processor-memory**
 - transfer data between processor and memory
- **Processor-I/O**
 - data transferred to or from a peripheral device
- **Data processing**
 - arithmetic or logic operation on data
- **Control**
 - alter sequence of execution

Review: Example of Program Execution

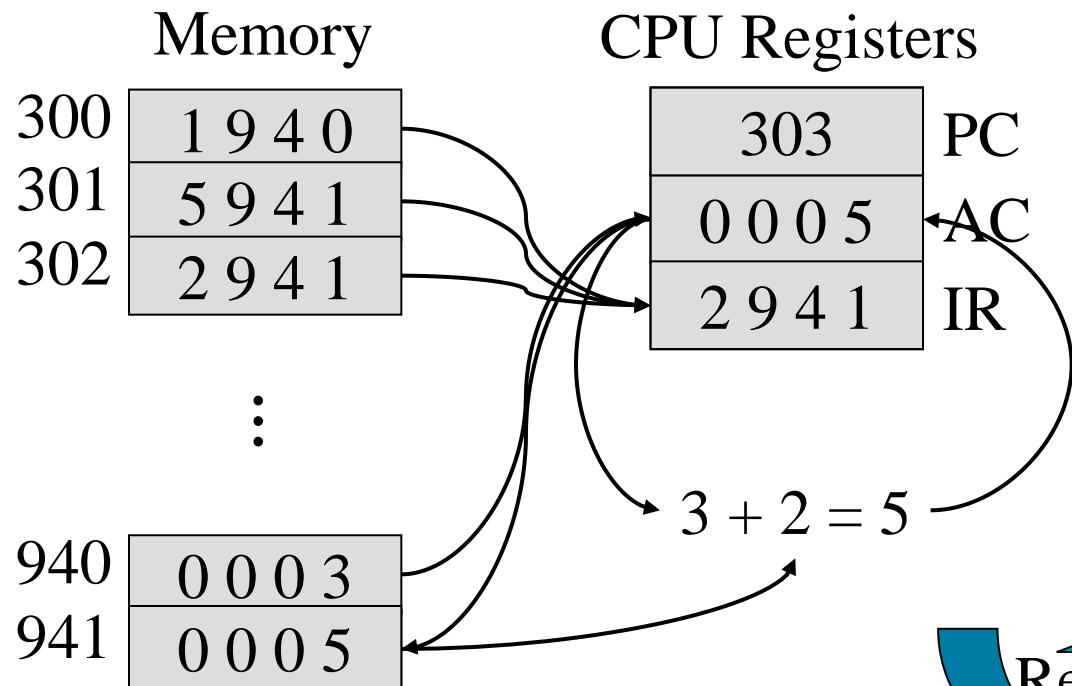
■ Instruction form

Operate Code	Address
--------------	---------

0001: LOAD data from Memory to AC
 0010: MOVE data from AC to Memory
 0101: ADD data in Memory to AC

■ Example

$M1 + M2 \Rightarrow M2$



Review: Interrupts

- **An interruption of the normal sequence of execution**
- **Improves processing efficiency**
- **Allows the processor to execute other instructions while an I/O operation is in progress**
- **A suspension of a process caused by an event external to that process and performed in such a way that the process can be resumed**

Review: Classes of Interrupts

Reasons that can
cause an interrupt?

■ Program

- arithmetic overflow
- divided by zero
- execute illegal instruction
- reference outside user's memory space

trap

■ Timer

■ I/O

■ Hardware failure

interrupt

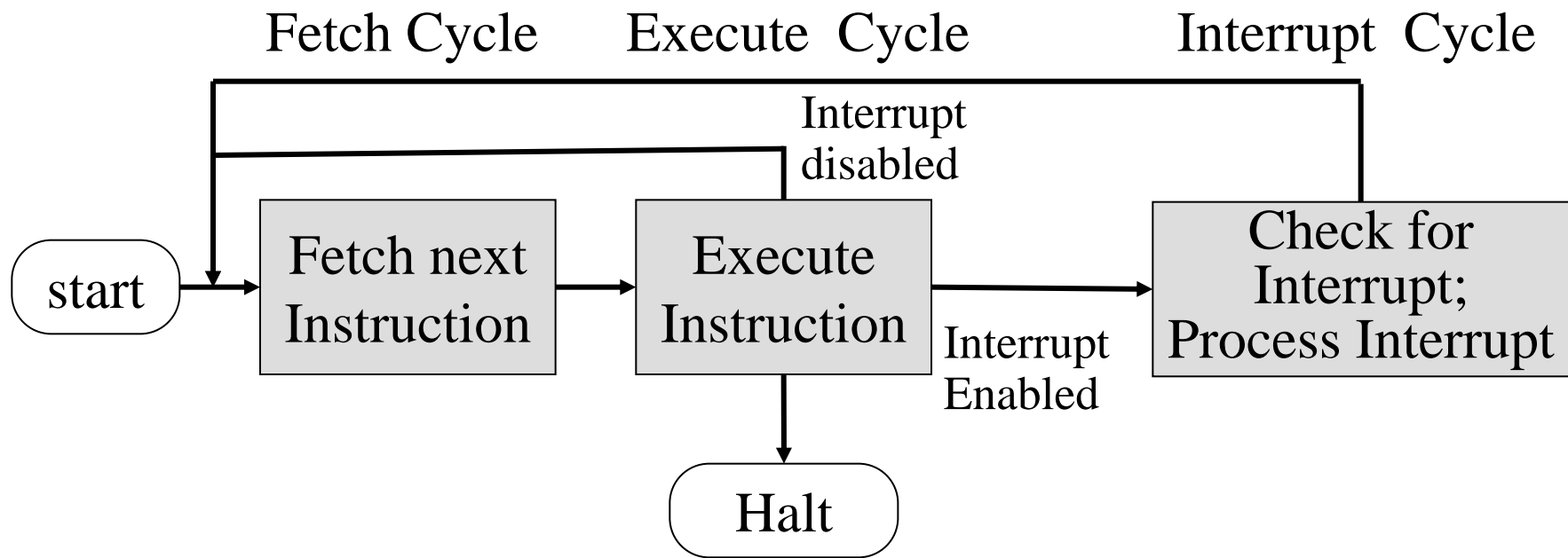
Review: Common Functions of Interrupts

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

Review: 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* (轮询)
 - *vectored* interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Review: Interrupt Cycle

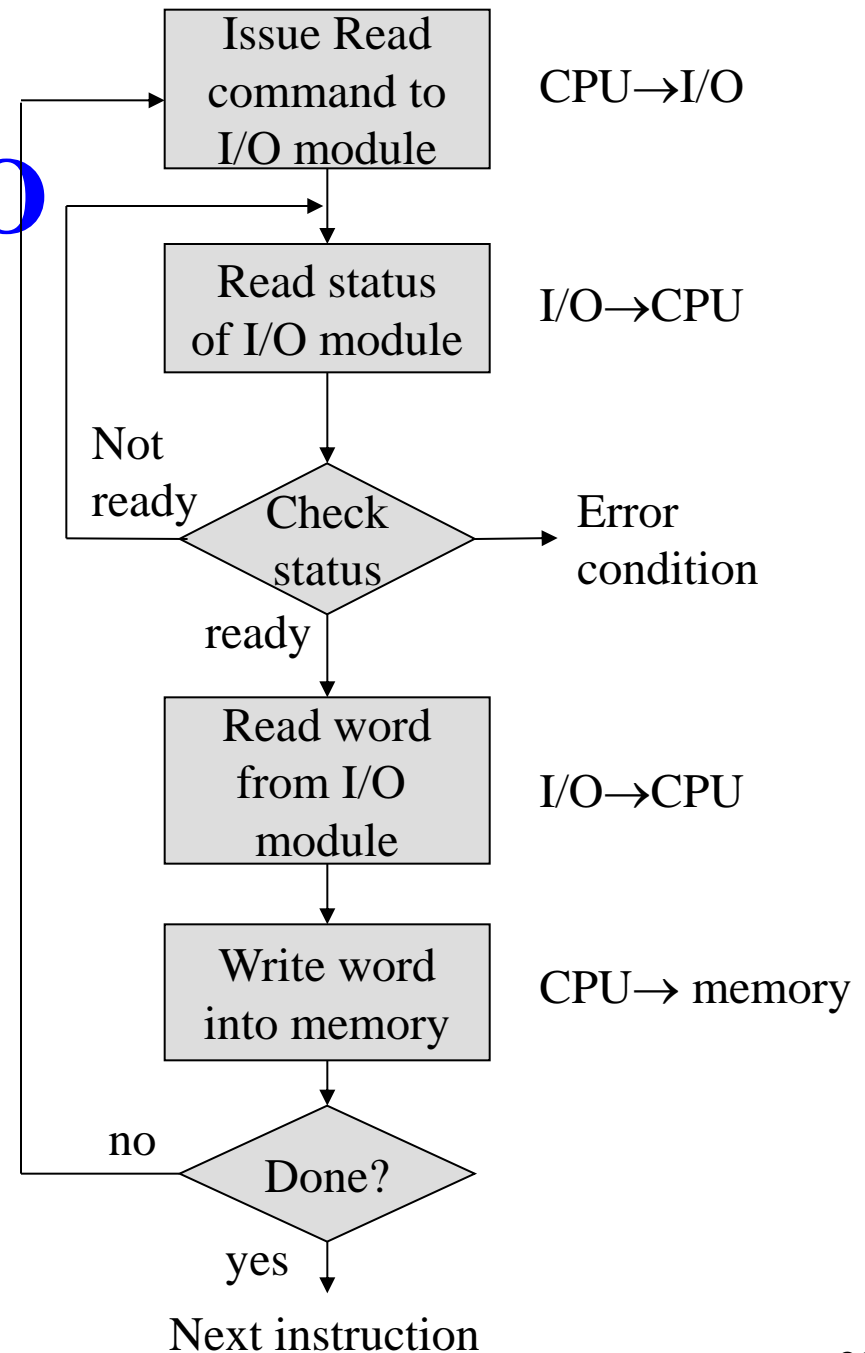


- **Processor checks for interrupts**
- **If no interrupts, fetch the next instruction for the current program**
- **If an interrupt is pending, suspend execution of the current program, and execute the interrupt handler**

Review:

1) Programmed I/O

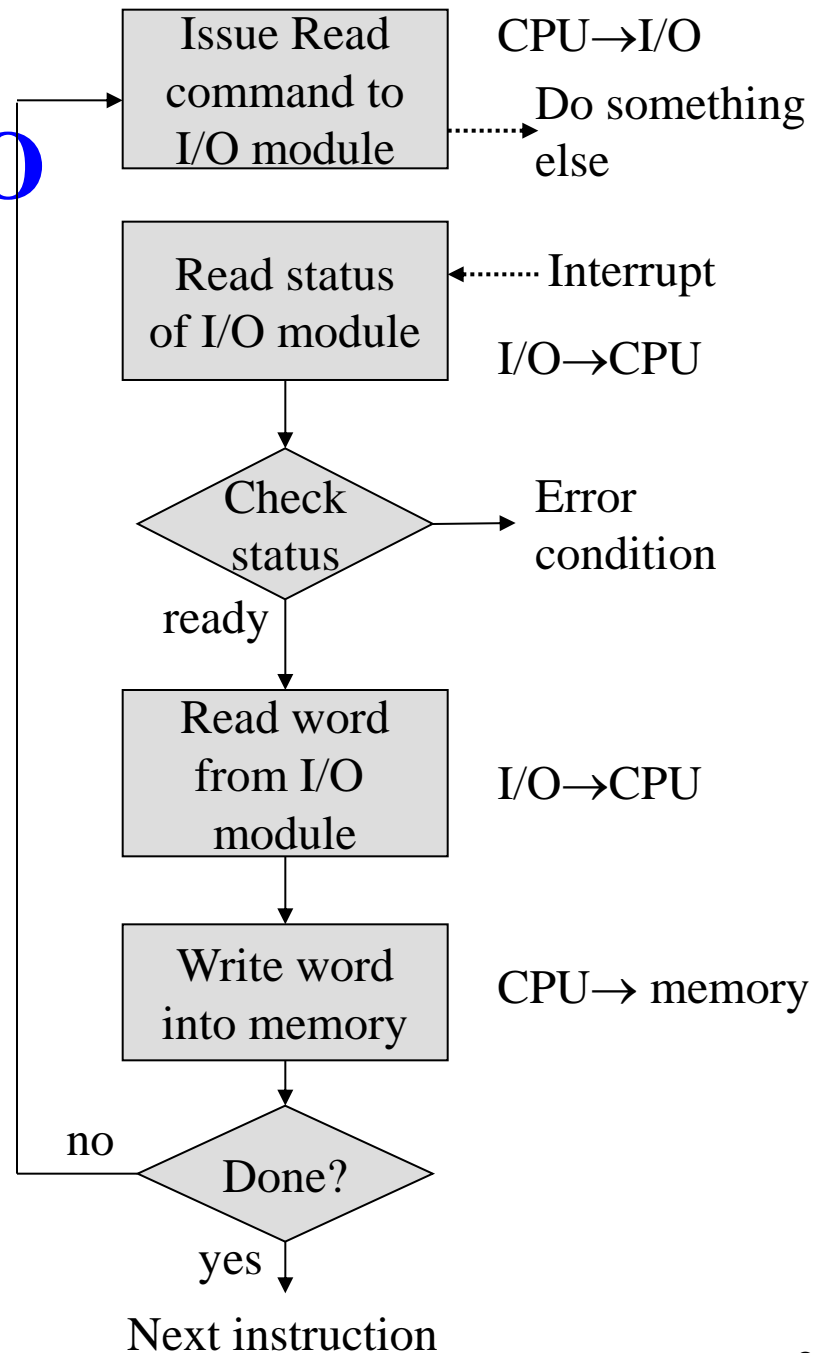
- I/O module performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete



Review:

2) Interrupt-Driven I/O

- Processor is interrupted when I/O module ready to exchange data
- Processor is free to do other work
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor



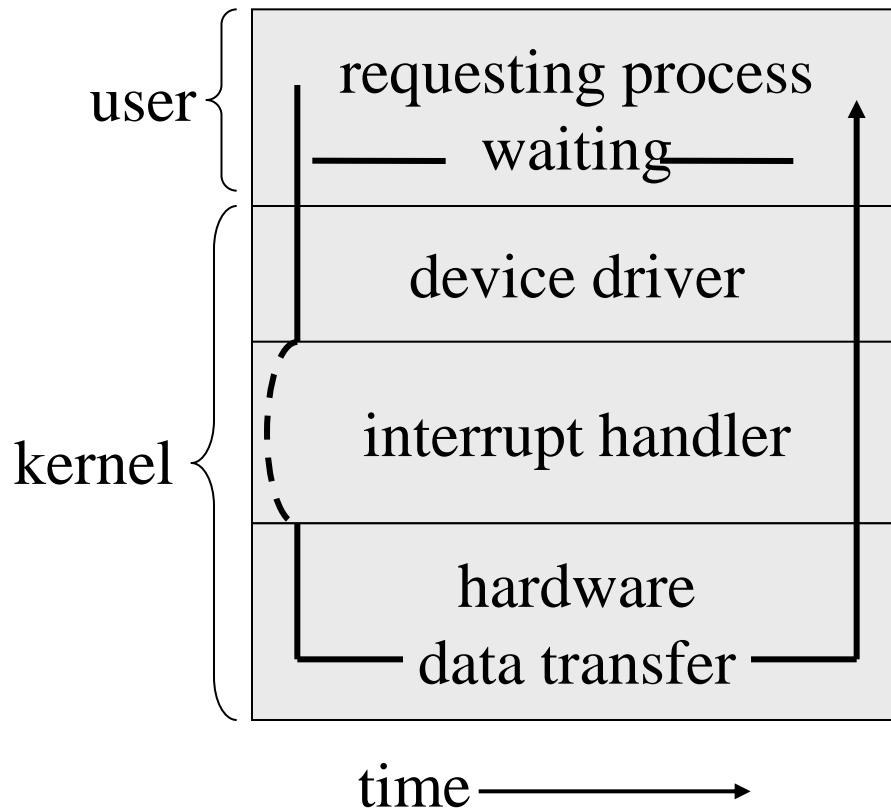
Review: Interrupt-Driven I/O (Cont.)

- **Synchronous I/O**----After I/O starts, control returns to user program only upon I/O completion.
- **Asynchronous I/O**---- After I/O starts, control returns to user program without waiting for I/O completion.
- **For synchronous I/O**, two methods can be used for CPU to wait for I/O completion:
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention (竞争) for memory access).

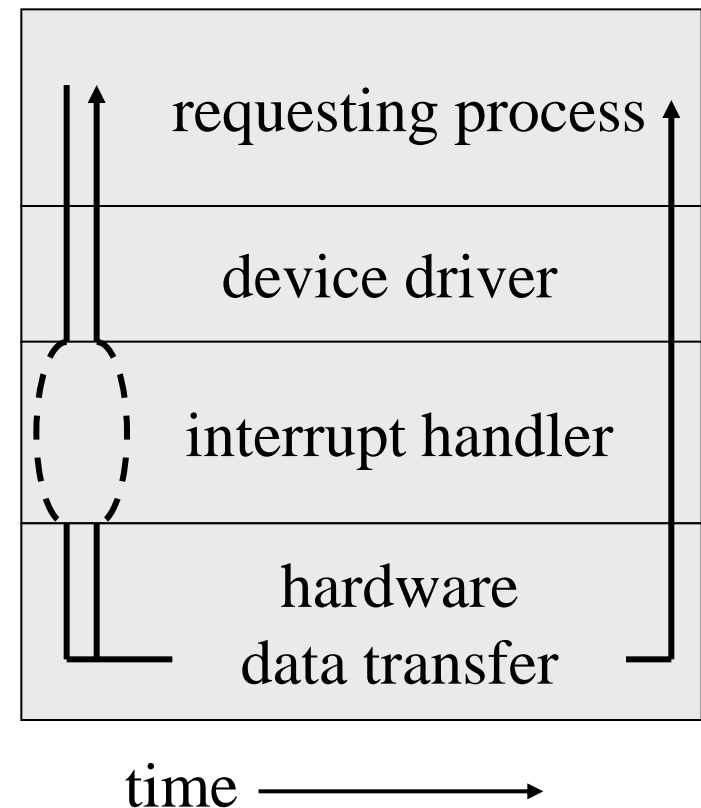
At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- In order to increase the utilization of CPU and I/O device, We need:
 - *System call* – request to the operating system to allow user to wait for I/O completion.
 - *Device-status table* contains entry for each I/O device indicating its type, address, and state.
 - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Review: Two Methods

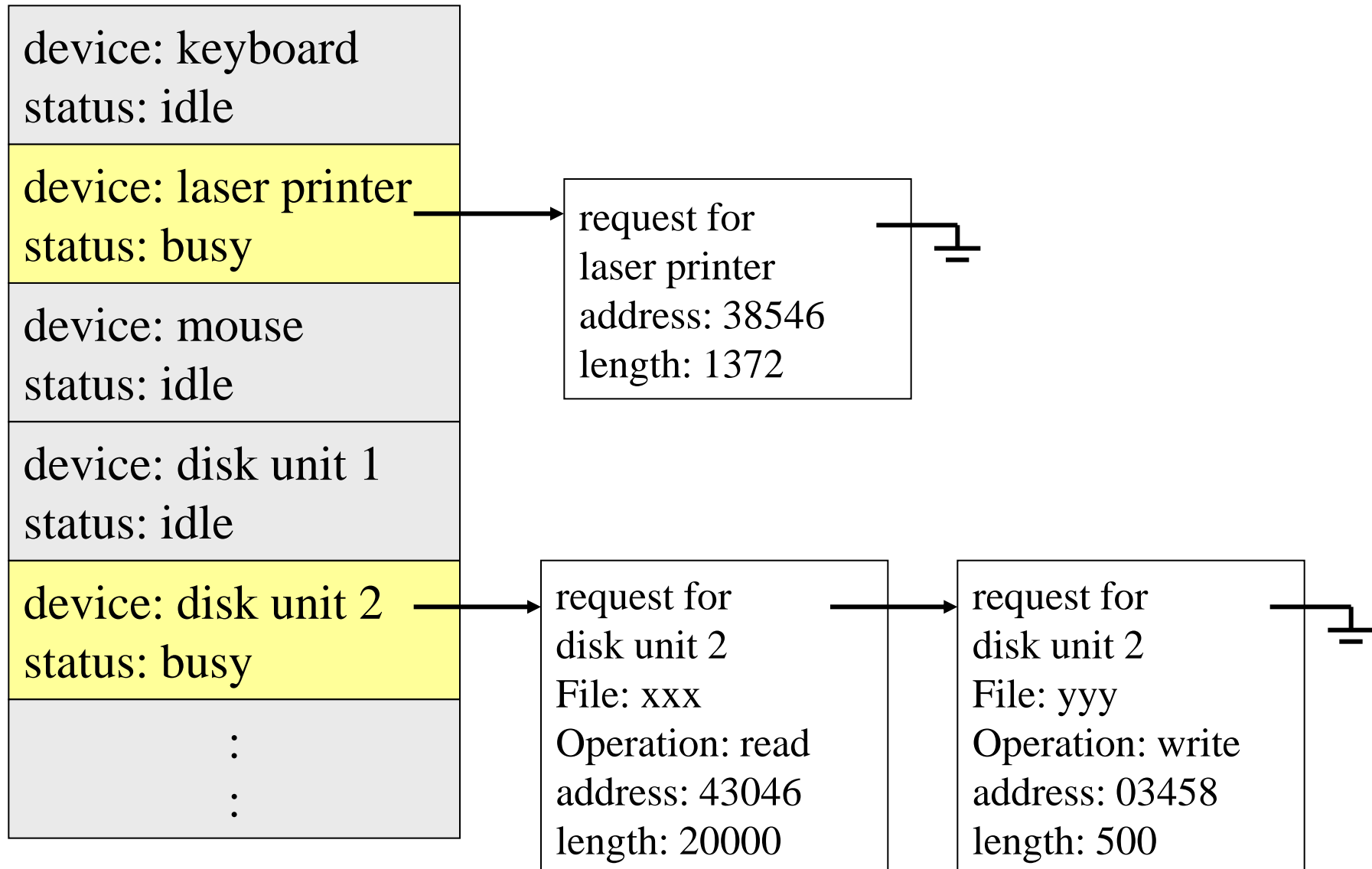
Synchronous



Asynchronous



Review: Device-Status Table

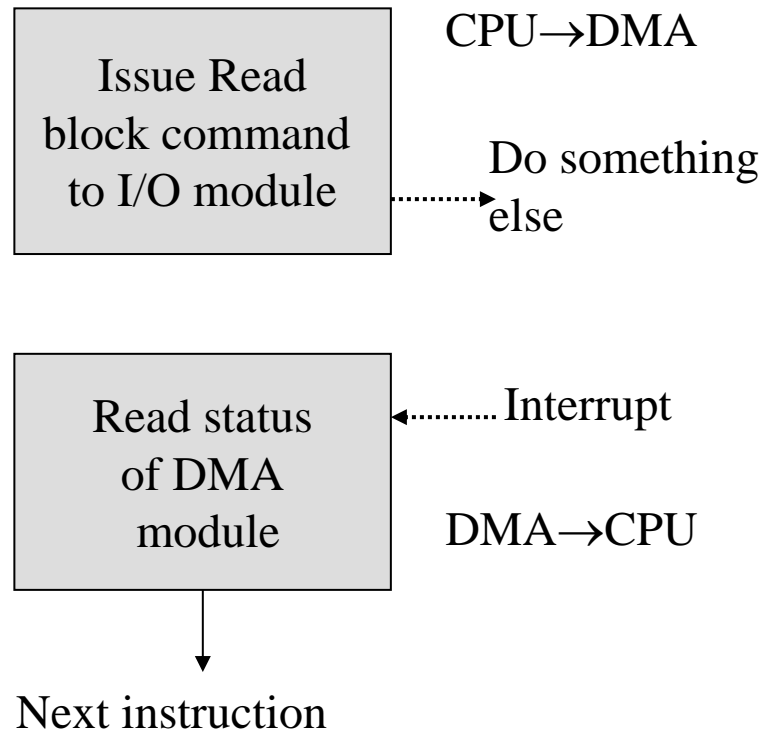


Review:

3) Direct Memory Access Structure

- **Transfers a block of data directly to or from memory.**
- **Used for high-speed I/O devices able to transmit information at close to memory speeds.**
- **After setting up buffers, pointers, and counters for the I/O devices , 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.**
 - **An interrupt is sent when the task is complete**
 - **The processor is only involved at the beginning and end of the transfer**

Review: Direct Memory Access (Cont.)



- **CPU is available to accomplish other work while the device controller is performing the DMA transfer**

Review: 4) I/O通道控制方式

- 是DMA方式的发展，进一步减少了CPU的干预。
- 完成一组数据块的读(或写)操作之后产生一次中断。
- 实现了CPU、通道和I/O设备三者的并行操作。
- CPU对通道的管理是通过I/O指令实现的。
 - I/O指令属于特权指令，仅能由操作系统使用。
 - 在I/O指令中需指定通道号、设备号、以及通道程序的内存地址。
- 当CPU要完成一组相关的读(或写)操作及有关控制时，只需向I/O通道发送一条I/O指令，以给出其所要执行的通道程序的首址和要访问的I/O设备，通道接到该指令后，通过执行通道程序便可完成CPU指定的I/O任务。

Review:通道指令

- 通道指令也称为通道命令字 (CCW, Channel Command Word)
 - 对设备进行初始化、读、写、查询、转移等
 - 使用CCW可以编写对各种不同设备进行管理和控制的通道程序
 - 一个通道可以以分时方式同时执行几个通道程序
 - 通道指令在进程要求数据时自动生成
- 指令格式: OP P R 计数 内存地址
 - OP: 读、写、或控制
 - P: 通道程序结束标志
P='1', 本条指令为最后一条指令。
 - R: 记录结束标志
R='0', 本指令与下一条指令所处理的数据属于同一条记录
R='1', 本指令是处理某记录的最后一条指令。
 - 计数: 本条指令所要处理数据的字节数
 - 内存地址:
读操作: 数据要写入的内存地址
写操作: 要写出的数据在内存中的起始地址

Review:通道程序举例

■ OP P R 计数 内存地址

■ read 0 0 250 1850

read 1 1 250 720

■ write 0 0 80 813

write 0 0 140 1034

write 0 1 60 5830

write 0 1 300 2000

write 0 0 200 1650

write 1 1 300 2720

Chapter 2

Operating-System Structures



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

system components

operating system service and system call

system structure

virtual machines

Chapter Objectives

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

Contents

- 2.1 Operating System Services**
- 2.2 User Operating System Interface**
- 2.3 System Calls**
- 2.4 Types of System Calls**
- 2.5 System Programs**
- 2.6 Operating System Design and Implementation(*)**
- 2.7 Operating System Structure**
- 2.8 Virtual Machines**
- 2.9 Operating System Generation(*)**
- 2.10 System Boot(*)**

2.1 Operating System Services

- **User interface:** Almost all OS have a user interface
 - Command-Line (CLI), Graphics User Interface (GUI)
 - APIs (system calls)
- **Program execution:** The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally.
- **I/O operations:** A running program may require I/O, which may involve a file or an I/O device.
- **File-system manipulation:** The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont.)

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

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
 - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
 - **Accounting** - To keep track of which users use how much and what kinds of computer resources

Operating System Services (Cont.)

- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
 - **Protection** involves ensuring that all access to system resources is controlled
 - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
 - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

2.2 User Operating-system Interface

- **Command-Line (CLI)**
- **Graphics User Interface (GUI)**

User OS Interface - CLI

- allows direct command entry
- **Command interpreter** is the interface between the user and the operating system.
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
 - Sometimes commands built-in, sometimes just names of programs
 - If the latter, adding new features doesn't require shell modification

User OS Interface - GUI

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

2.3 System Calls

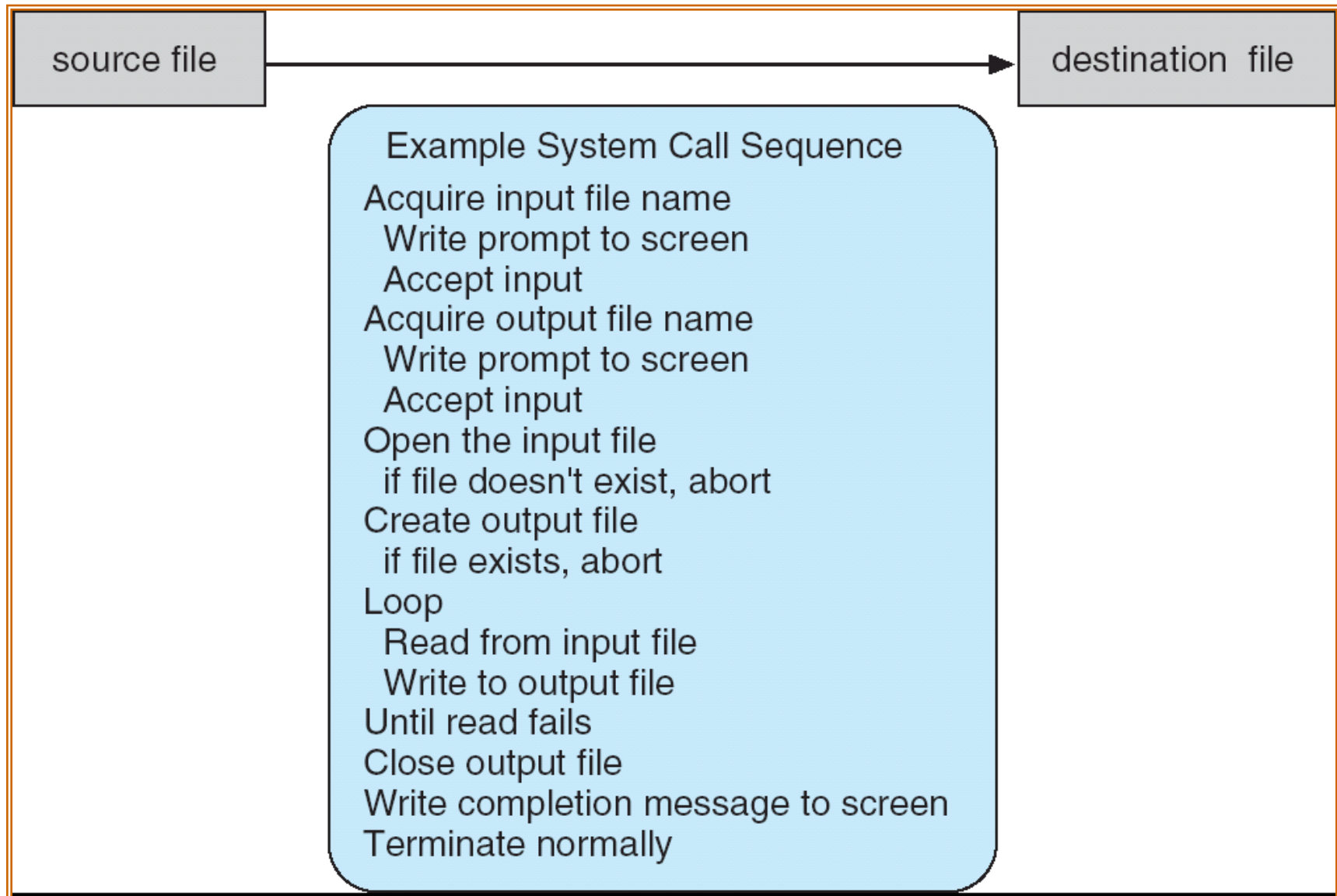
- **Programming interface to the services provided by the OS**
- **Typically written in a high-level language (C or C++)**
- **Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use**
- **Three most common APIs are**
 - **Win32 API for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

POSIX: Portable Operating System Interface for computer environment
 - **Java API for the Java virtual machine (JVM)**
- **Why use APIs rather than system calls?**

Example of System Calls

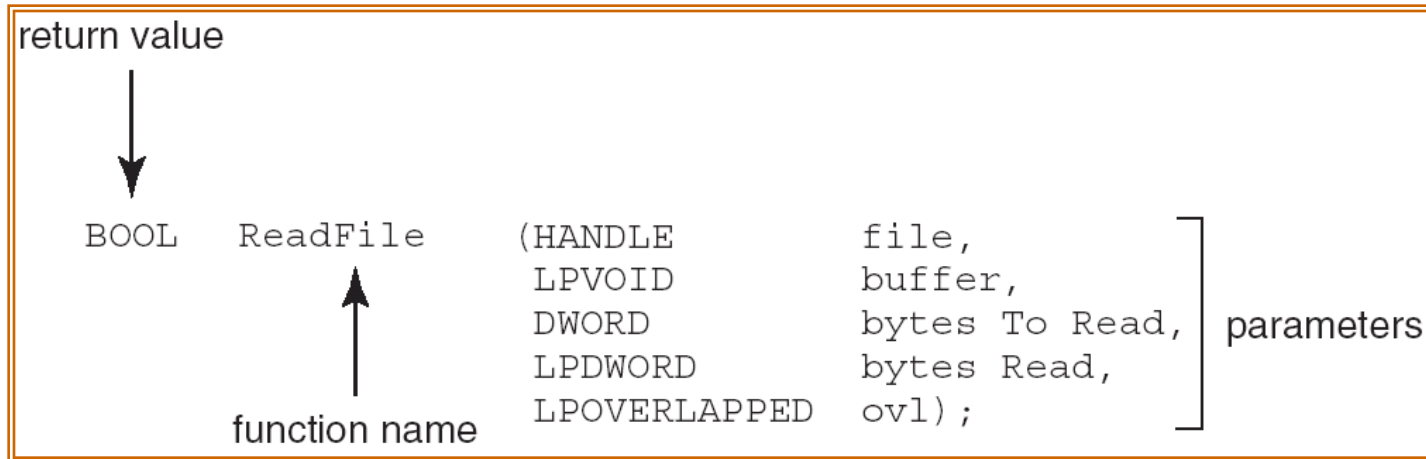
- **writing a simple program to read data from one file and to copy them to another file.**
 - **The input of the names of the input file and output file that the program will need.**
 - **Open the input file, create the output file.**
 - **Loop: reads from the input file and writes to the output file.**
 - **Close both files, write a message to the console, terminate normally.**

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



Example of Standard API

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

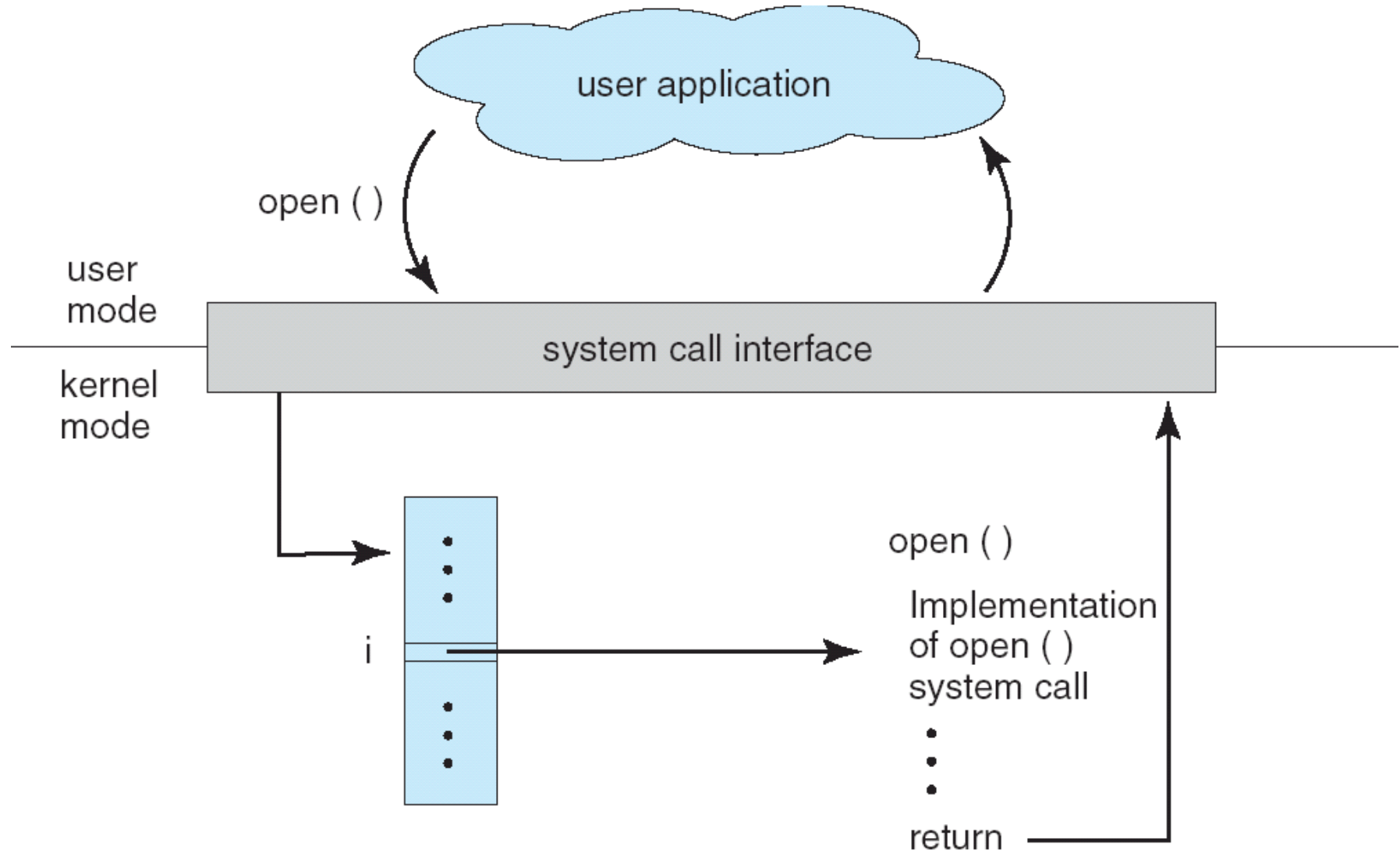


- A description of the parameters passed to `ReadFile()`
 - **HANDLE file**—the file to be read
 - **LPVOID buffer**—a buffer where the data will be read into and written from
 - **DWORD bytesToRead**—the number of bytes to be read into the buffer
 - **LPDWORD bytesRead**—the number of bytes read during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

System Call Implementation

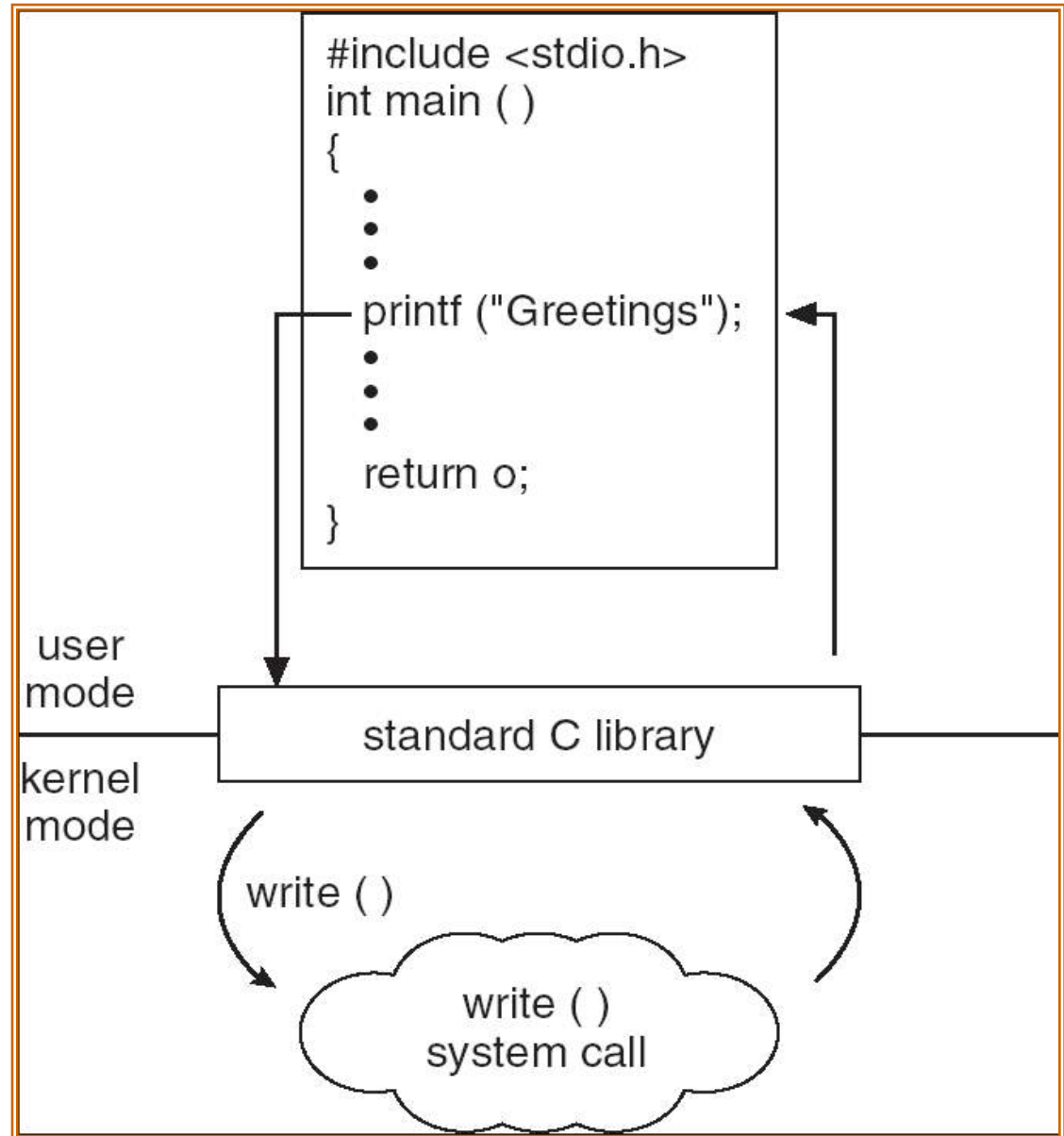
- Typically, a number associated with each system call
 - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how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

API – System Call – OS Relationship



Standard C Library Example

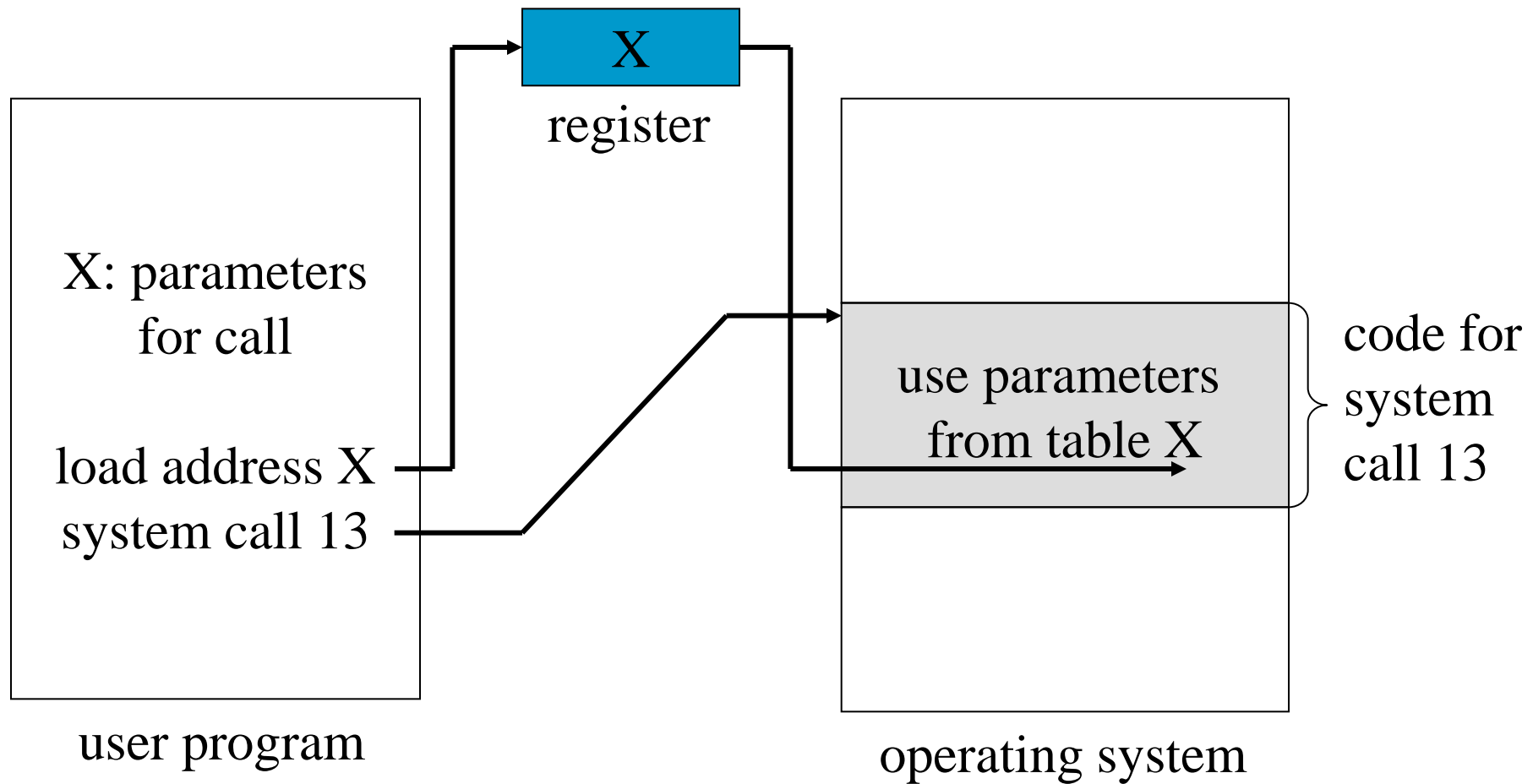
- **C program**
invoking **printf()**
library call, which
calls **write()** system
call



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in *registers*
 - In some cases, may be more parameters than registers
 - Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 - Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
(Block and stack methods do not limit the number or length of parameters being passed)

Passing of Parameters Via a Table



2.4 Types of System Calls

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

Process control

- **end, abort**
- **load, execute**
- **create process, terminate process**
- **get process attributes, set process attributes**
- **wait for time**
- **wait event, signal event**
- **allocate and free memory**

File management

- **Create file, delete file**
- **open, close**
- **read, write, reposition**
- **get file attributes, set file attributes**

Device management

- Request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

Information maintenance

- **Get time or date, set time or date**
- **get system data, set system data**
- **get process, file, or device attributes**
- **set process, file, or device attributes**

Communications

- Create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices
- two common models of communication
 - **message-passing model**: information is exchanged through an interprocess-communication facility (IPC) provided by the operating system.
 - **Shared-memory model**: process use map memory system call to gain access to regions of memory owned by other processes.

2.5 System Programs

- **System programs provide a convenient environment for program development and execution. They can be divided into:**
 - **File management**
 - **Status information**
 - **File modification**
 - **Programming language support**
 - **Program loading and execution**
 - **Communications**
 - **System utilities / Application programs**
- **Most users' view of the operation system is defined by system programs, not the actual system calls.**

System Programs (Cont.)

- Provide a **convenient environment** for program development and execution
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

System Programs (Cont.)

■ Status information

- Some ask the system for info - date, time, amount of available memory, disk space, number of users
- Others provide detailed performance, logging, and debugging information
- Typically, these programs format and print the output to the terminal or other output devices
- Some systems implement a registry - used to store and retrieve configuration information

■ File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

System Programs (Cont.)

- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
 - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

2.6 OS Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
 - *User goals* – operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - *System goals* – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

OS Design and Implementation (Cont.)

- Important principle to separate **Policy** from **Mechanism**
 - **Policy**: What will be done?
 - **Mechanism**: How to do it?
- Mechanisms determine how to do something, policies decide what will be done
 - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

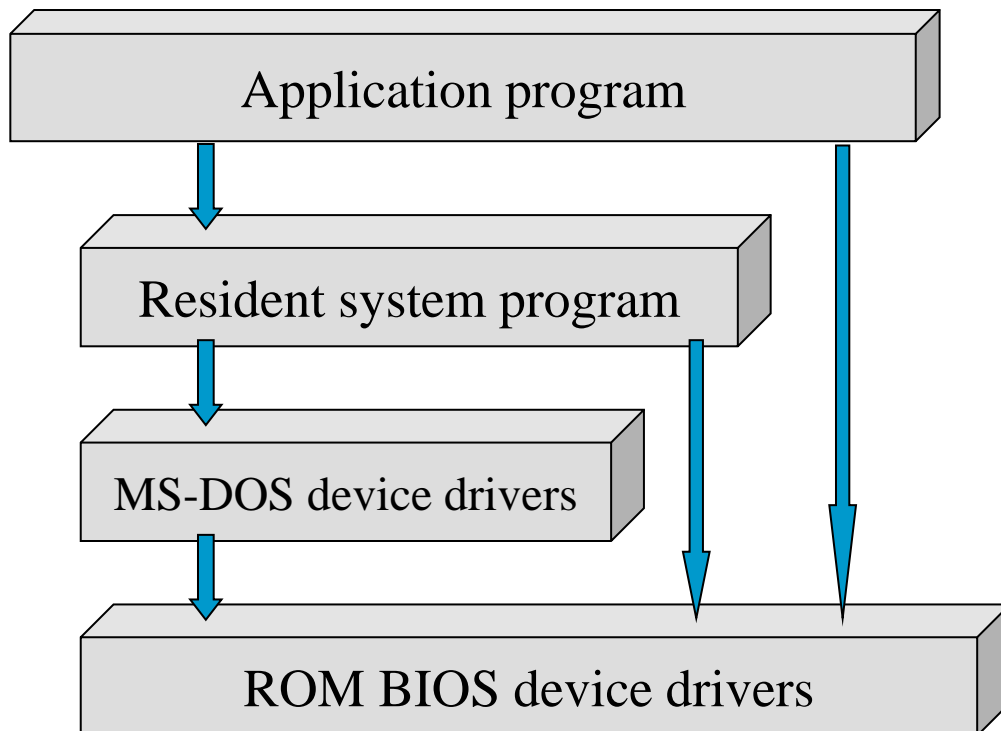
2.7 System Structure

- **simple Structure**
 - MS-DOS system structure
 - UNIX system structure
- **Layered Approach**
- **Microkernels**
- **Modules**

2.7.1 Simple structure

MS-DOS System Structure

- **MS-DOS – written to provide the most functionality in the least space**
 - not divided into modules carefully.



- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

UNIX System Structure

- **UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.**
- **The UNIX OS consists of two separable parts**
 - **System programs**
 - **The kernel, which is further separated into a series of interfaces and device drivers.**
 - **Consists of everything below the system-call interface and above the physical hardware**
 - **Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.**
- **System calls define the API to UNIX, the set of system programs commonly available defines the user interface.**

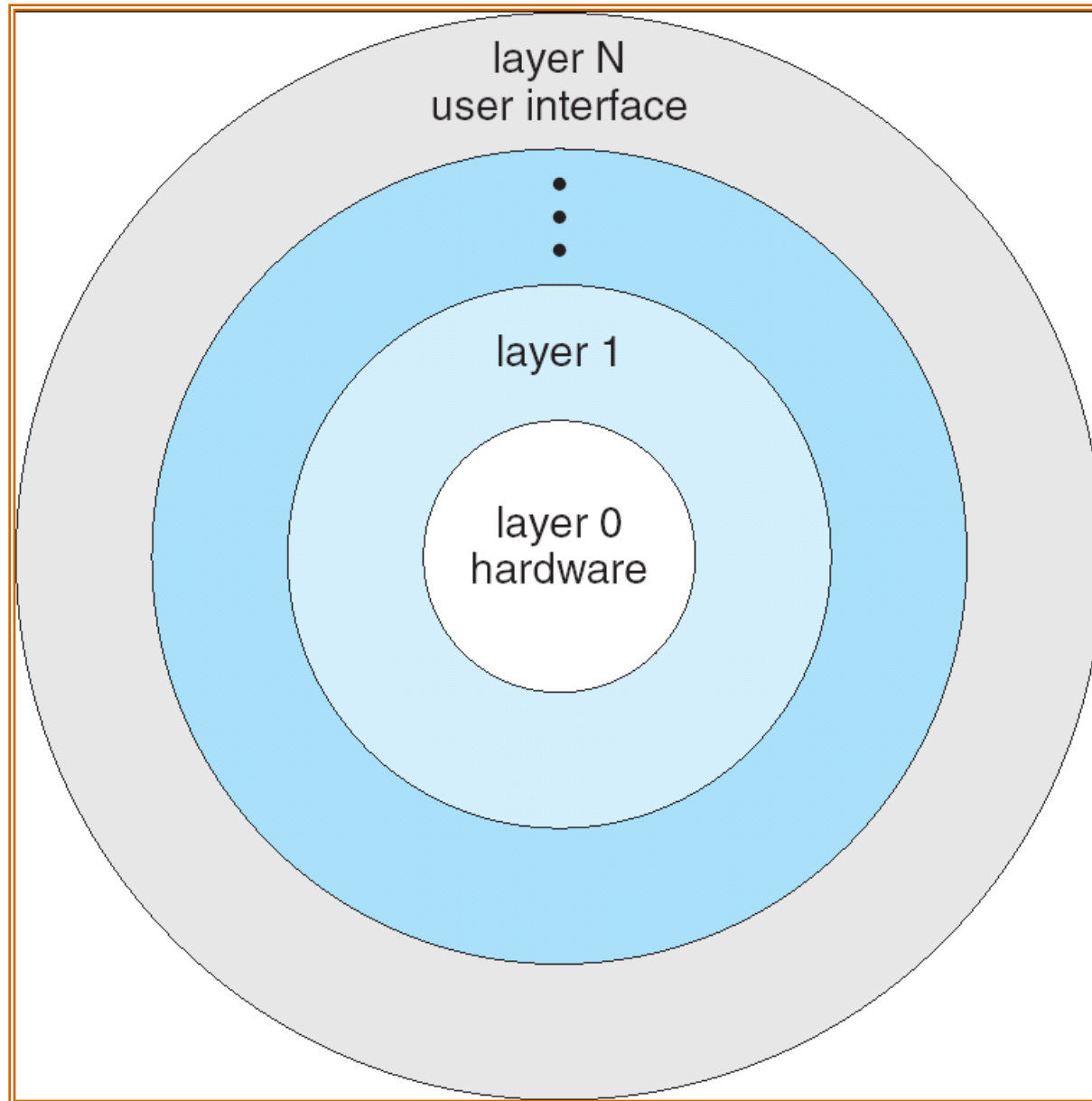
UNIX System Structure

(the user)		
shells and commands compilers and interpreters system libraries		
<i>system-call interface to the kernel</i>		
signals terminal handling character I/O system terminal drivers	file system swapping block I/O system Disk and tape drivers	CPU scheduling page replacement demand paging virtual memory
<i>kernel interface to the hardware</i>		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

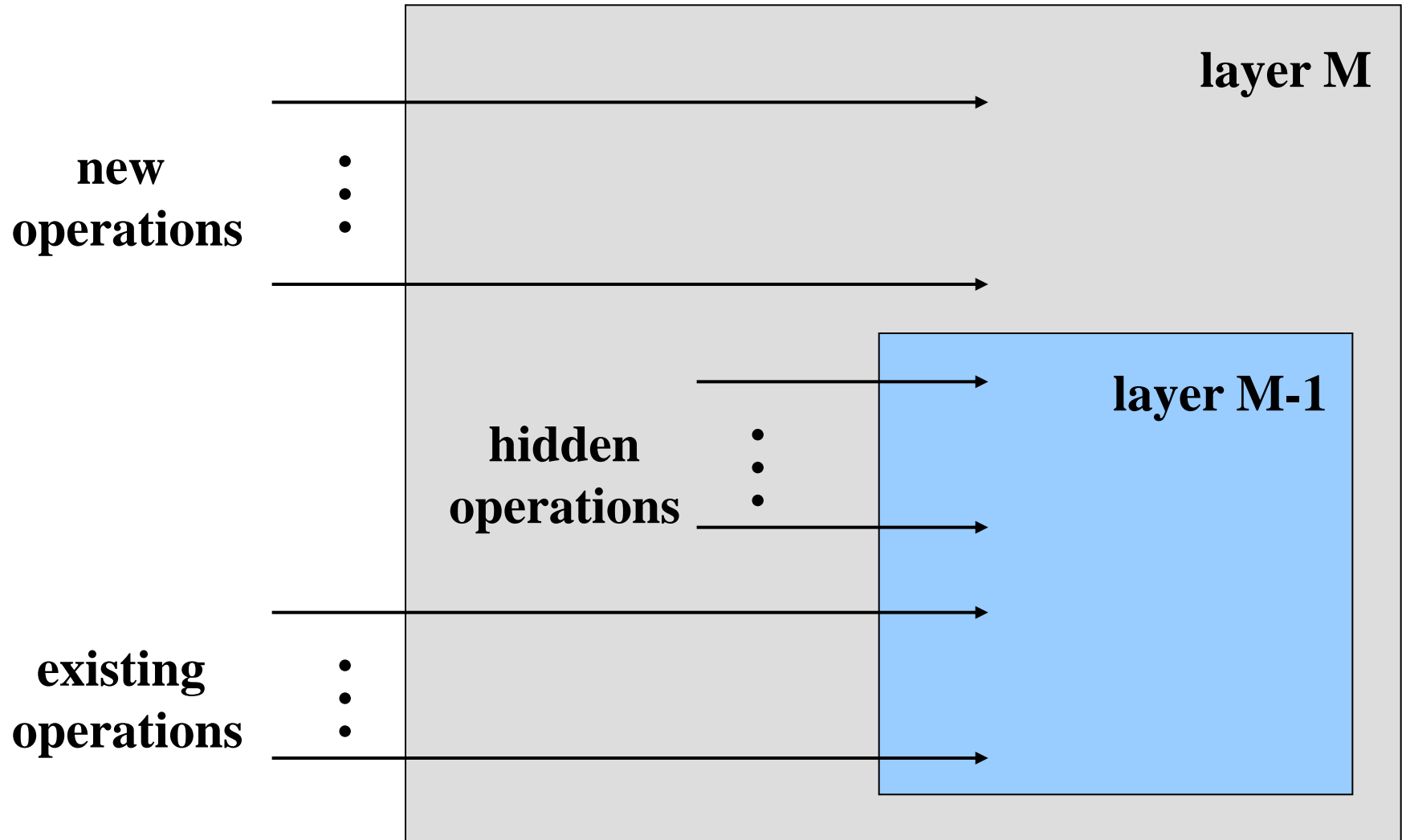
2.7.2 Layered Approach

- **The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.**
- **An operating-system layer is an implementation of an abstract object that is the encapsulation of data, and of the operations that can manipulate those data.**
- **With modularity, layers are selected such that each uses functions (or operations) and services of only lower-level layers.**

Layered Operating System



Supplement: An Operating System Layer



Supplement:

Summary----Layered System Structure

- View the system as a series of levels
- Each level performs a related subset of functions
- Each level relies on the next lower level to perform more primitive functions
- This decomposes a problem into a number of more manageable subproblems

Advantages/disadvantages

- **Simplifies debugging and system verification.**
- **Design and implementation of the system are simplified.**
- **The major difficulty involves the careful definition of the layers.**
- **Tend to be less efficient than other types.**

Supplement: Operating System Design Hierarchy

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printer, displays and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual Memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive process, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction Set	Evaluation stack, micro-program interpreter, scalar and array data	Load, store, add, subtract branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

All info. Needed by managing processes

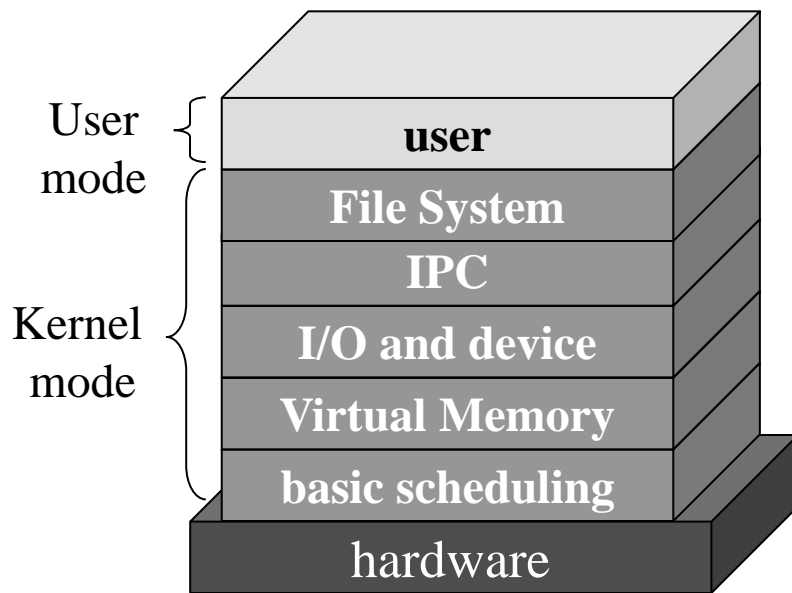
Contents of CPU registers relevant to processes and logic used for scheduling processes

2.7.3 Microkernels

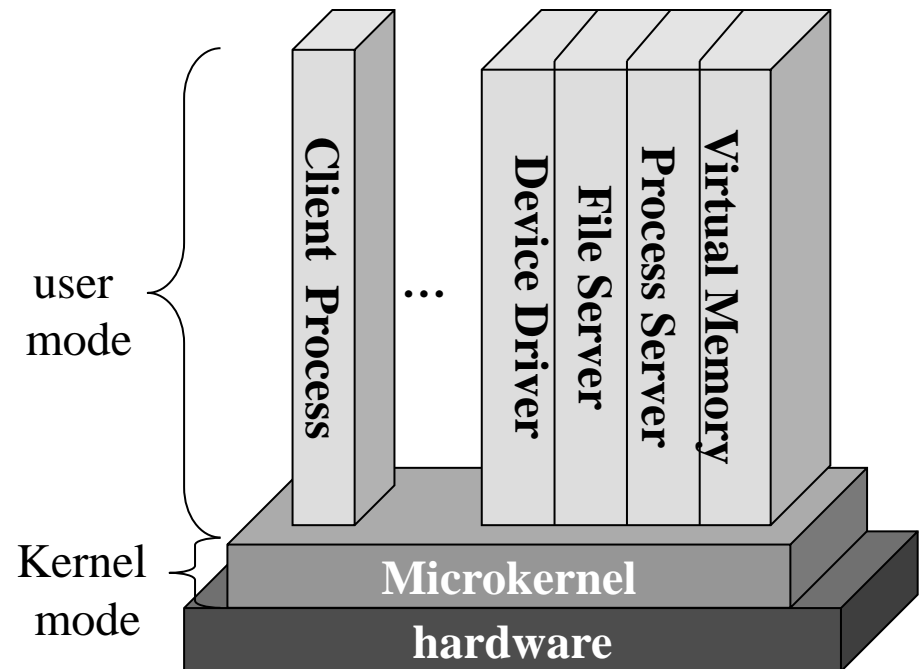
- Moves as much from the kernel into “*user*” space.
 - Small operating system core
 - Contains only essential operating systems functions
 - address space
 - interprocess communication (IPC)
 - basic scheduling
 - Many services traditionally included in the operating system are now external subsystems
 - device drivers
 - file systems
 - virtual memory manager
 - windowing system
 - security services

Microkernels

- The main function of the microkernel is to provide a communication facility between the client program and the various services that are running in user space.
- Communication takes place between user modules using message passing.



Layered Operating System



Microkernel

Benefits of a Microkernel Organization

- **Uniform interface on request made by a process**
 - All services are provided by means of message passing
- **Extensibility**
 - Allows the addition of new services
- **Flexibility**
 - New features added
 - Existing features can be subtracted
- **Portability**
 - Changes needed to port the system to a new processor is changed in the microkernel - not in the other services
- **Reliability**
 - Modular design
 - Small microkernel can be rigorously tested

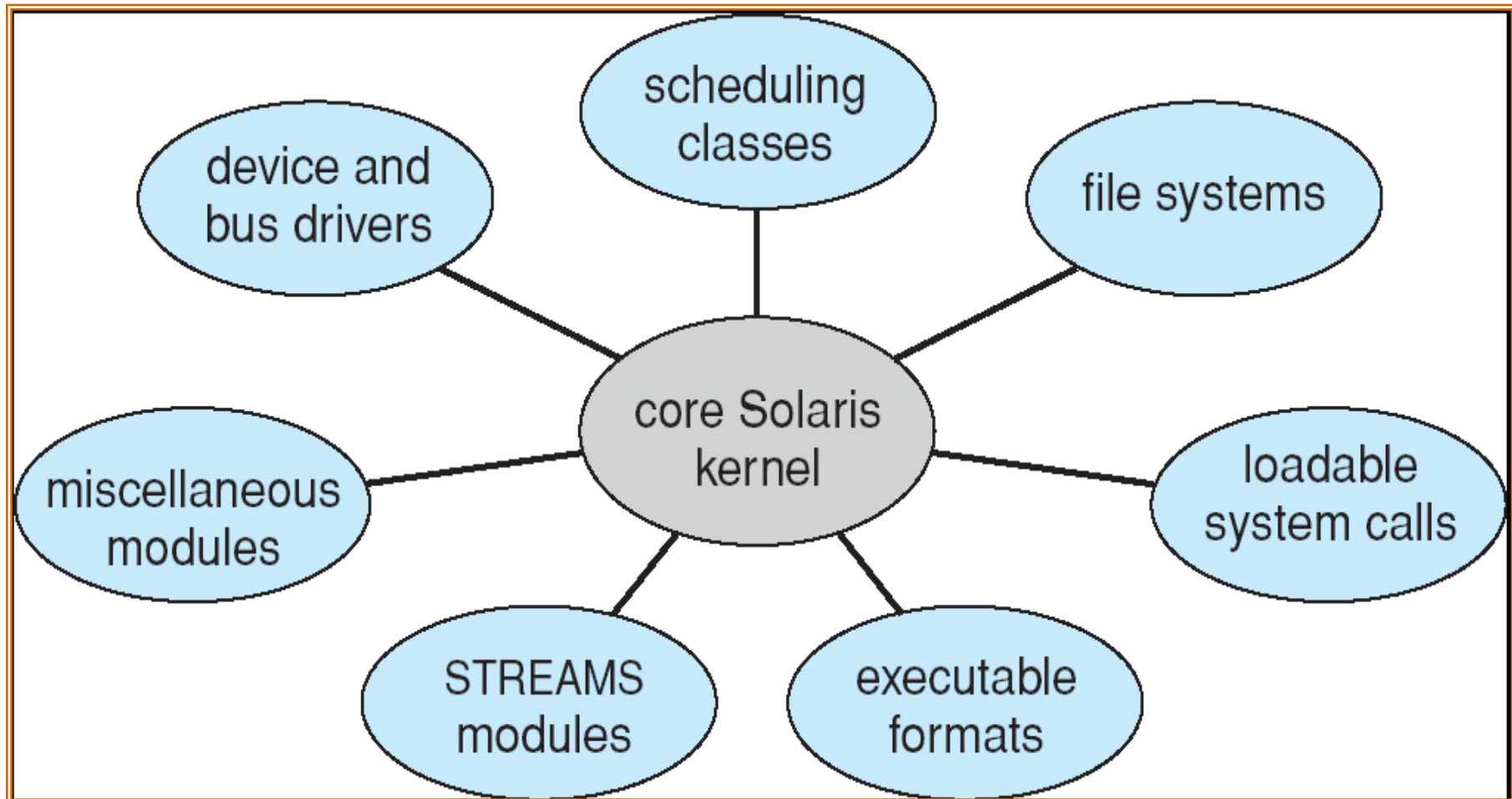
Benefits of a Microkernel Organization

- **Distributed system support**
 - Messages are sent without knowing what the target machine is
- **Object-oriented operating system**
 - Components are objects with clearly defined interfaces that can be interconnected to form software
- **Detriments:**
 - Performance overhead of user space to kernel space communication

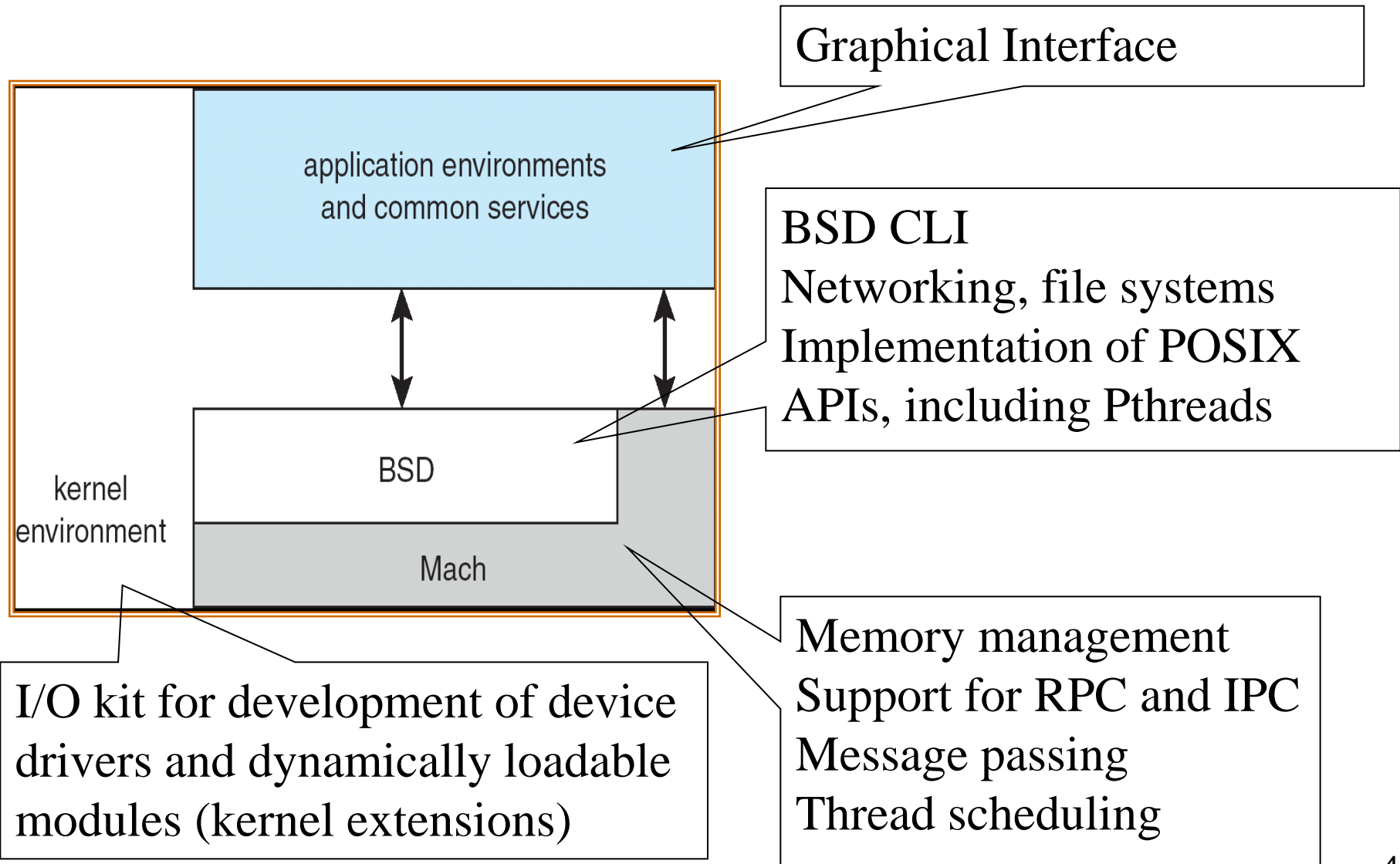
2.7.4 Modules

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

Solaris Modular Approach



A hybrid structure: Mac OS X Structure



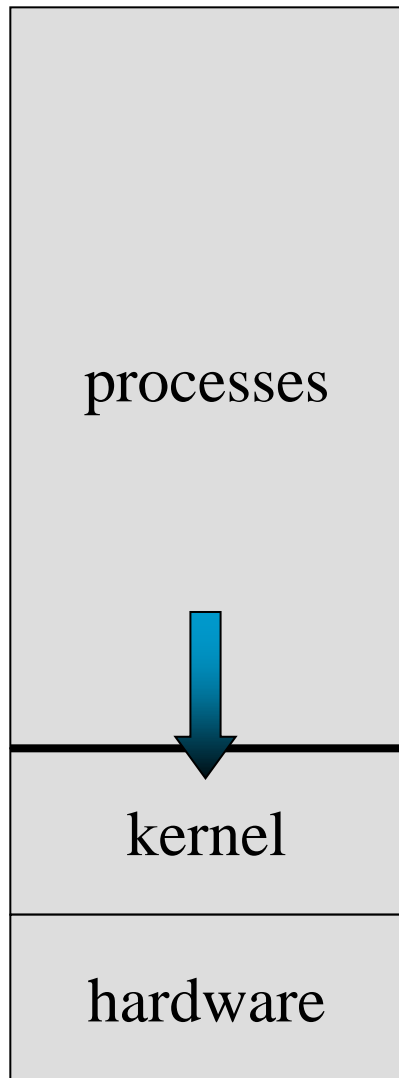
2.8 Virtual Machines (VM)

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.
 - By CPU scheduling
 - By virtual memory

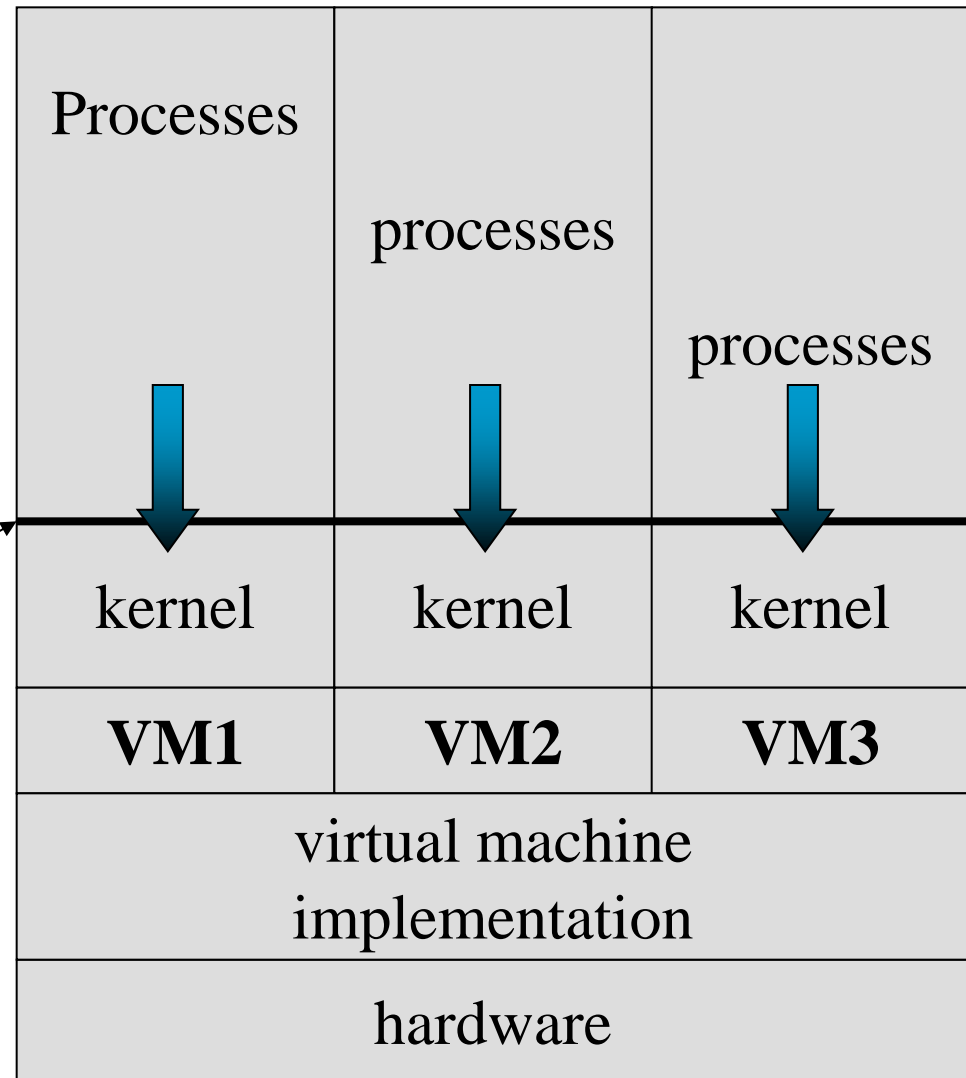
Virtual Machines (Cont.)

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

System Models



Nonvirtual machine



Virtual machine

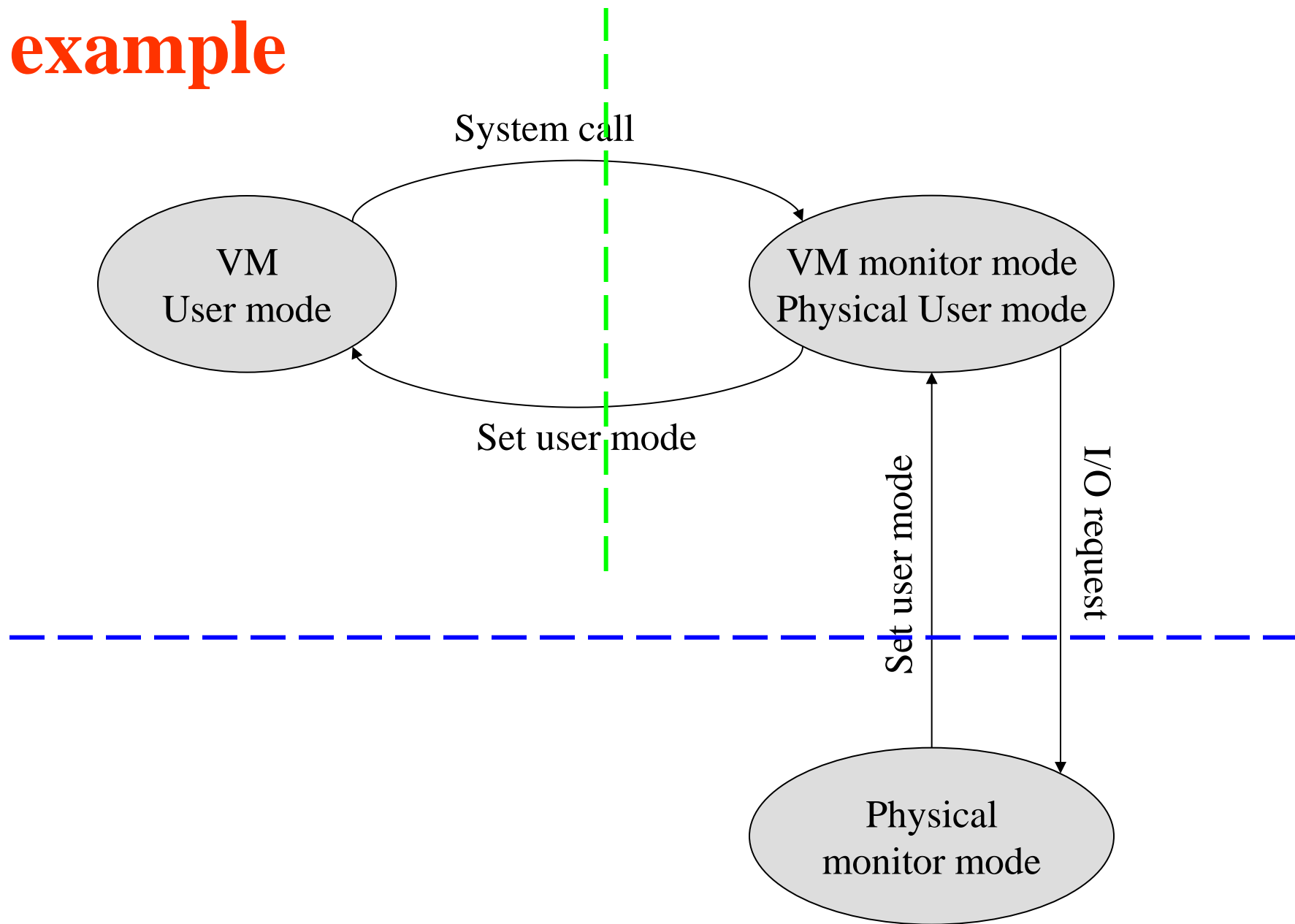
Advantages/Disadvantages of VM (1/2)

- The virtual-machine concept provides complete **protection of system resources** since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect **vehicle for operating-systems research and development**. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- VMs are increasing in popularity as a means of **solving system compatibility problem**.

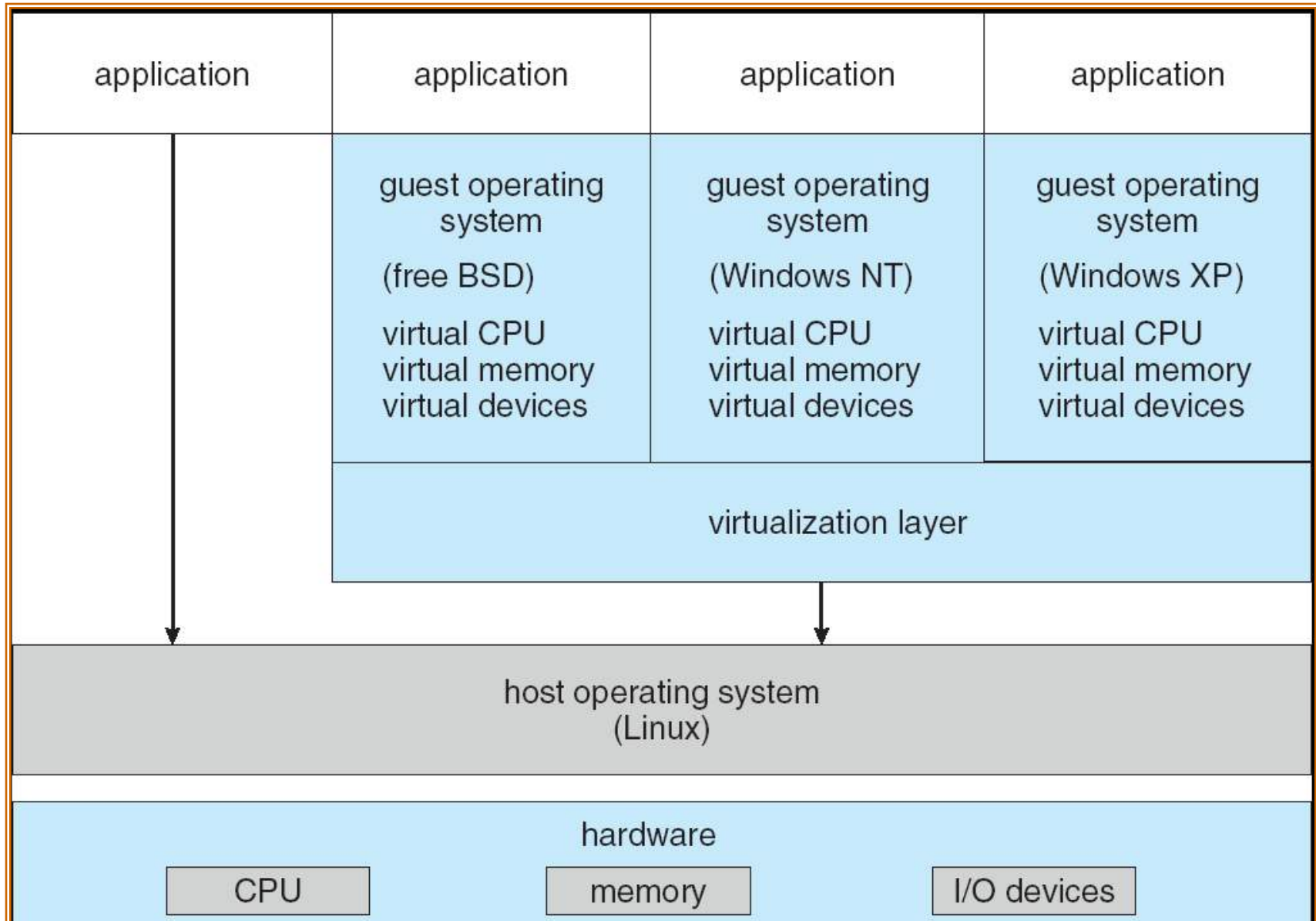
Advantages/Disadvantages of VM (2/2)

- The virtual machine concept is **difficult to implement** due to the effort required to provide an *exact* duplicate to the underlying machine.
 - **Virtual user mode** and **virtual monitor mode**, run in a physical user mode.
 - Actions that cause a transfer from user mode to monitor mode on a real machine must also cause a transfer from virtual user mode to virtual monitor mode on a virtual machine.

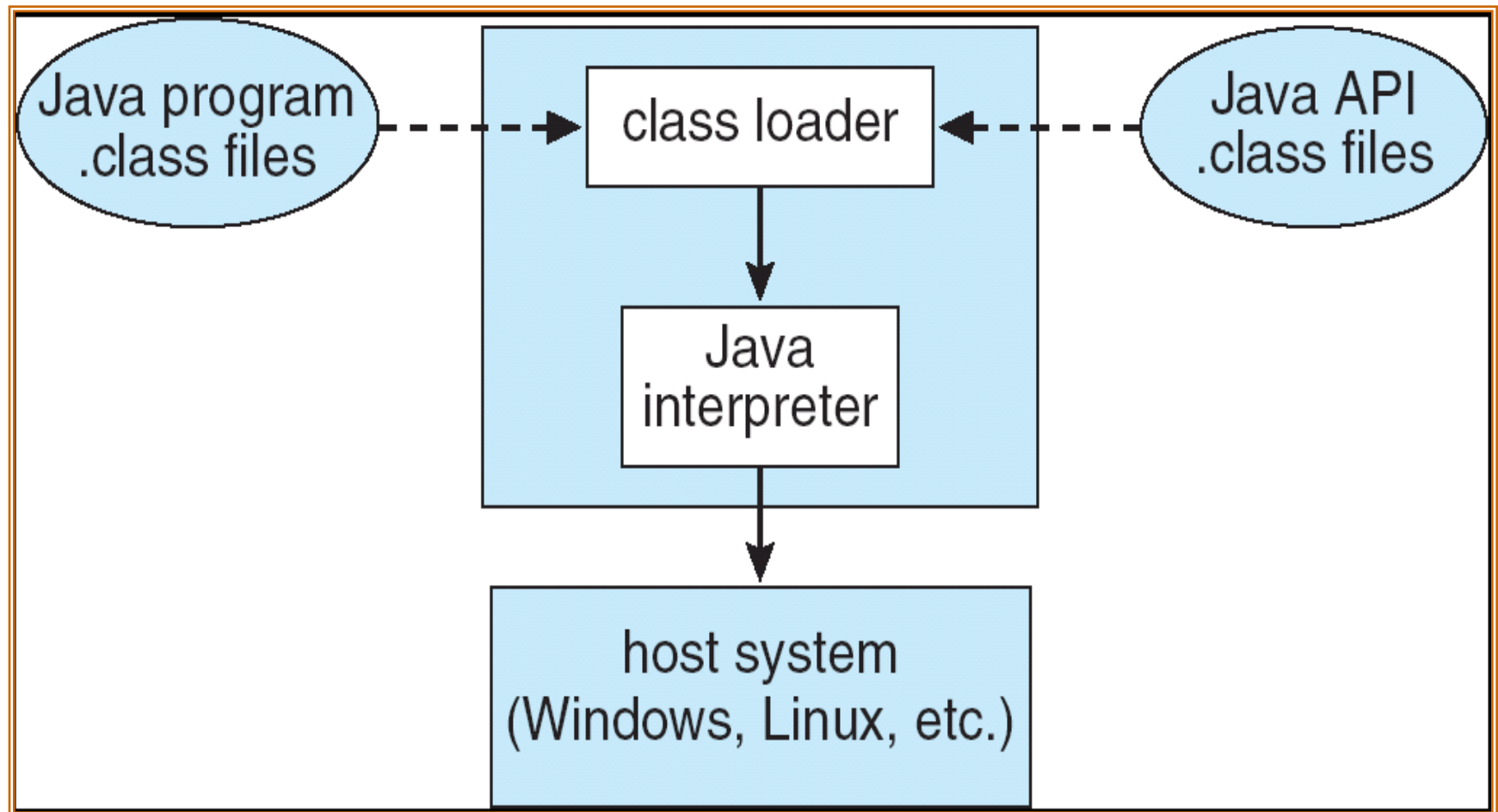
example



VMware Architecture



The Java Virtual Machine



Java Virtual Machine

- **Compiled Java programs are platform-neutral bytecodes executed by a Java Virtual Machine (JVM).**
- **JVM consists of**
 - **class loader**
 - **class verifier**
 - **Java interpreter that executes the architecture-neutral bytecodes.**
- **JVM automatically manages memory by performing garbage collection.**
- **Java interpreter may be a**
 - **software module that interprets the bytecodes one at a time**
 - **Just-In-Time (JIT) compilers turns the architecture-neutral bytecodes into native machine language for the host computer.**

2.9 Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

2.10 System Boot

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

Homework (Page 73)

2.3

2.8

2.15

Thinking:

2.1

2.7

2.11

2.12

2.13

Programing:

2.18

Linux与Windows的命令控制界面

■ 命令控制界面

– 传统的字符界面

- 用户在系统给出的提示符下敲入特定的命令
- 系统接受用户输入的命令，并执行
- 系统向用户报告执行结果，并等待用户输入新的命令

– 图形用户界面（GUI）

- 多窗口
- 命令被开发成用鼠标点击的菜单或者图标
- **Linux**的图形化界面：**X Window**

Linux的命令控制界面

- 命令位置：/user/sbin、user/bin、/sbin、/bin
- Linux命令分类
 - 系统维护及管理：如date, setenv等
 - 文件操作及管理：如ls, find等
 - 进程管理：如kill, at等
 - 磁盘及设备管理：如df, du, mount等
 - 用户管理：如adduser, userdel等
 - 文档操作：如csplit, sort等
 - 网络通信：如netstat, ifconfig等
 - 程序开发：如cc, link等
 - X Window管理：如startx, XE86Stup等
- man命令：显示某命令的联机帮助

Linux shell

- 交互型命令解释程序、shell程序解释系统
- Shell程序：带形参的批命令文件
- Shell程序的组成
 - 命令或其他shell程序
 - 位置参数
 - 变量及特殊字符
 - 表达式比较
 - 控制流语句
 - 函数

mkdir backup 在当前目录下创建子目录

for file in ‘ls’

begin

cp \$ file backup/\$ file

if [\$>-ne 0] **then**

echo “copying \$ file error”

endif

end

Windows的命令控制界面

- **Windows命令控制界面分两部分**
 - 命令解释程序cmd.exe: 接受键盘输入的命令
 - 窗口: 通过鼠标或键盘进行操作
- **命令: Dos基本命令+Windows自有命令**
 - 系统信息命令, 如time, date, mem
 - 系统操作命令, 如shutdown, taskkill
 - 文件系统命令, 如copy, del, mkdir
 - 网络通信命令, 如ping, netstat, route

命令组合运算

- **& 命令分隔符**
 - **Cmd1 & cmd2**
- **&& 若前面的命令成功，则执行后面的命令**
 - **Cmd1 && cmd2**
- **|| 若前面的命令失败，则执行后面的命令**
 - **Cmd1 || cmd2**
- **() 命令分组或嵌套**
 - **Cmd1 || (cmd2 & cmd3)**
- **; 命令参数分隔符**
 - **Cmd param1;param2**

命令的使用

- 直接在命令行输入命令
 - 运行cmd.exe进入命令行界面
 - 在命令行提示符下输入命令
- 编制批处理文件
 - 扩展名为.bat或.cmd的文本文件
 - 在命令行提示符下输入批处理文件名
- 批处理文件可以带有参数
 - % 1 表示第一个参数

批处理文件示例

■ Example1.bat

@echo off

mem>%1\meminfo.txt

Echo generate memoryinfo ok!

■ Example2.bat

@echo off

Type %1*.txt

Echo type ok!

■ Example3.bat

@echo off

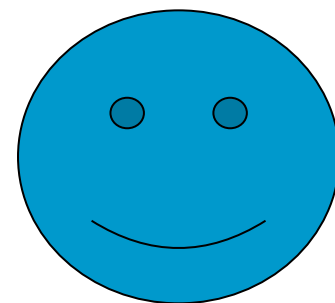
Mkdir batexample

Call exam1.bat test

Call exam2.bat test

Echo call ok!

pause



Chapter 3 Processes



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

Process Concept

Operations on Processes

Cooperating Processes

Interprocess Communication

Chapter Objectives

- **To introduce the notion of a process – a program in execution, which forms the basis of all computer**
- **To describe the various features of processes, including**
 - Scheduling
 - creation and termination
 - communication
- **To describe communication in client-server systems**

Contents

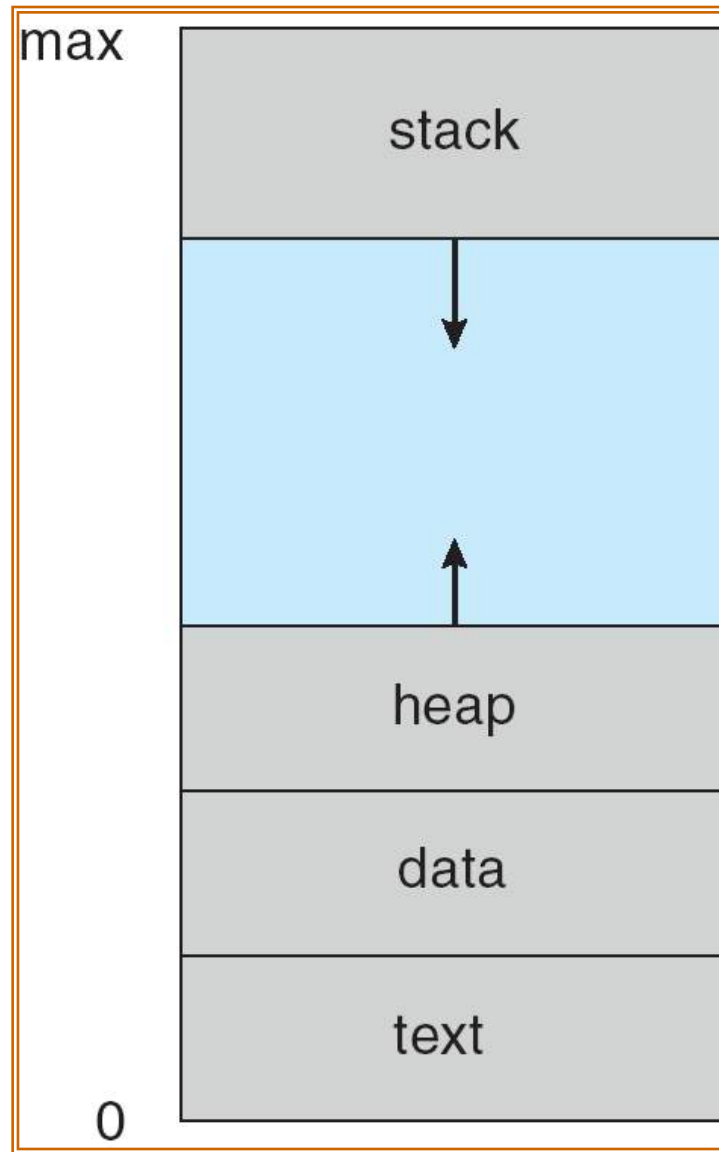
- 3.1 Process Concept**
- 3.2 Process Scheduling**
- 3.3 Operations on Processes**
- 3.4 Interprocess Communication**

- 3.5 example of IPC Systems**
- 3.6 Communication in Client-Server Systems**

3.1 Process Concept

- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- terms *job* and *process*
- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes:
 - program code, known as text section
 - data section, contains global variables.
 - current activity, represented by the value of the program counter and the contents of the processor's register
 - process stack, contains temporary data

Process in Memory

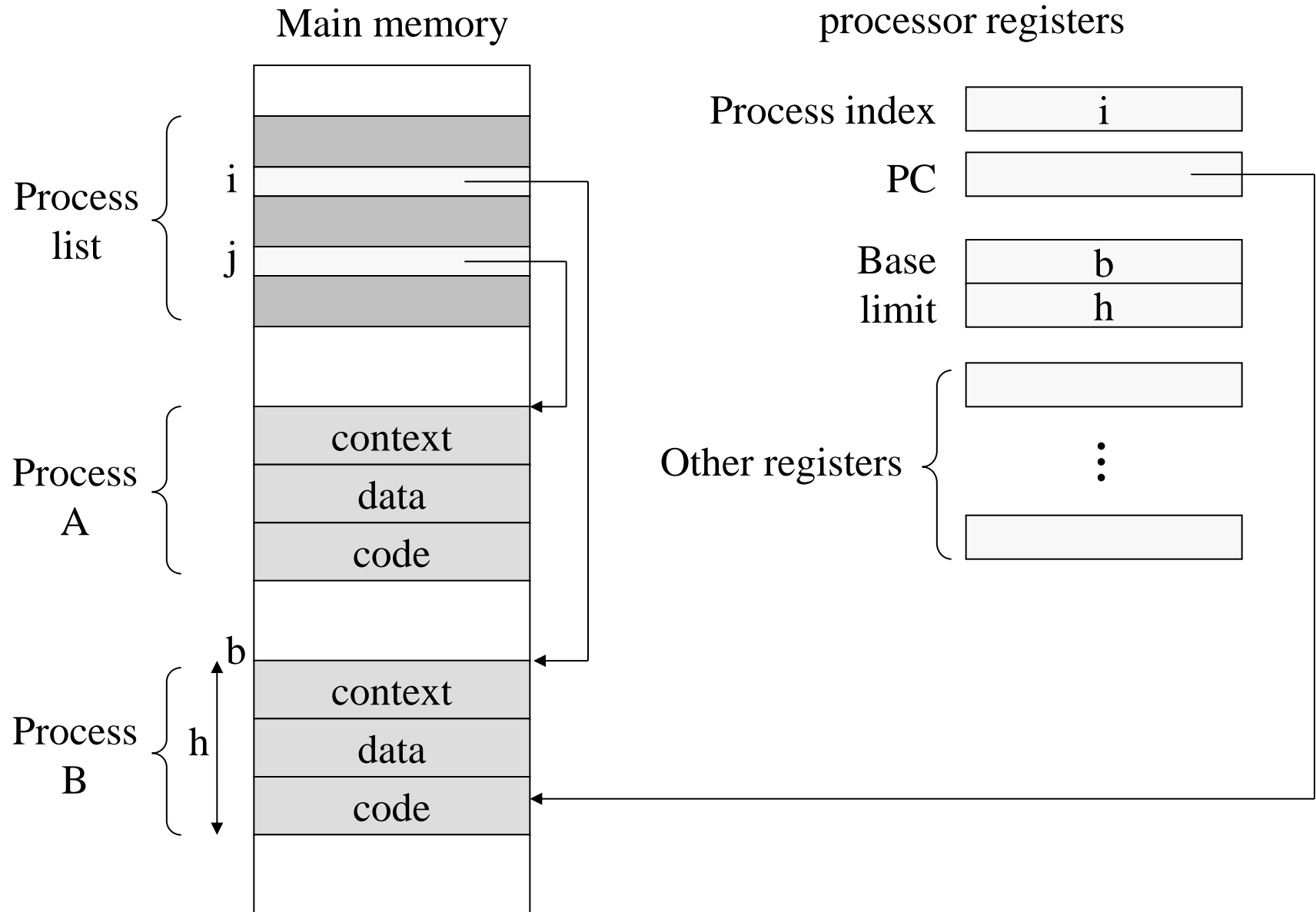


Supplement: Components of a Process

- **Consists of three components**
 - An executable program
 - Associated data needed by the program
 - **Execution context (上下文环境) of the program**
 - All information the operating system needs to manage the process
- **PCB-Process Control Block**
 - Data structure used by OS to control/manage processes

Supplement:

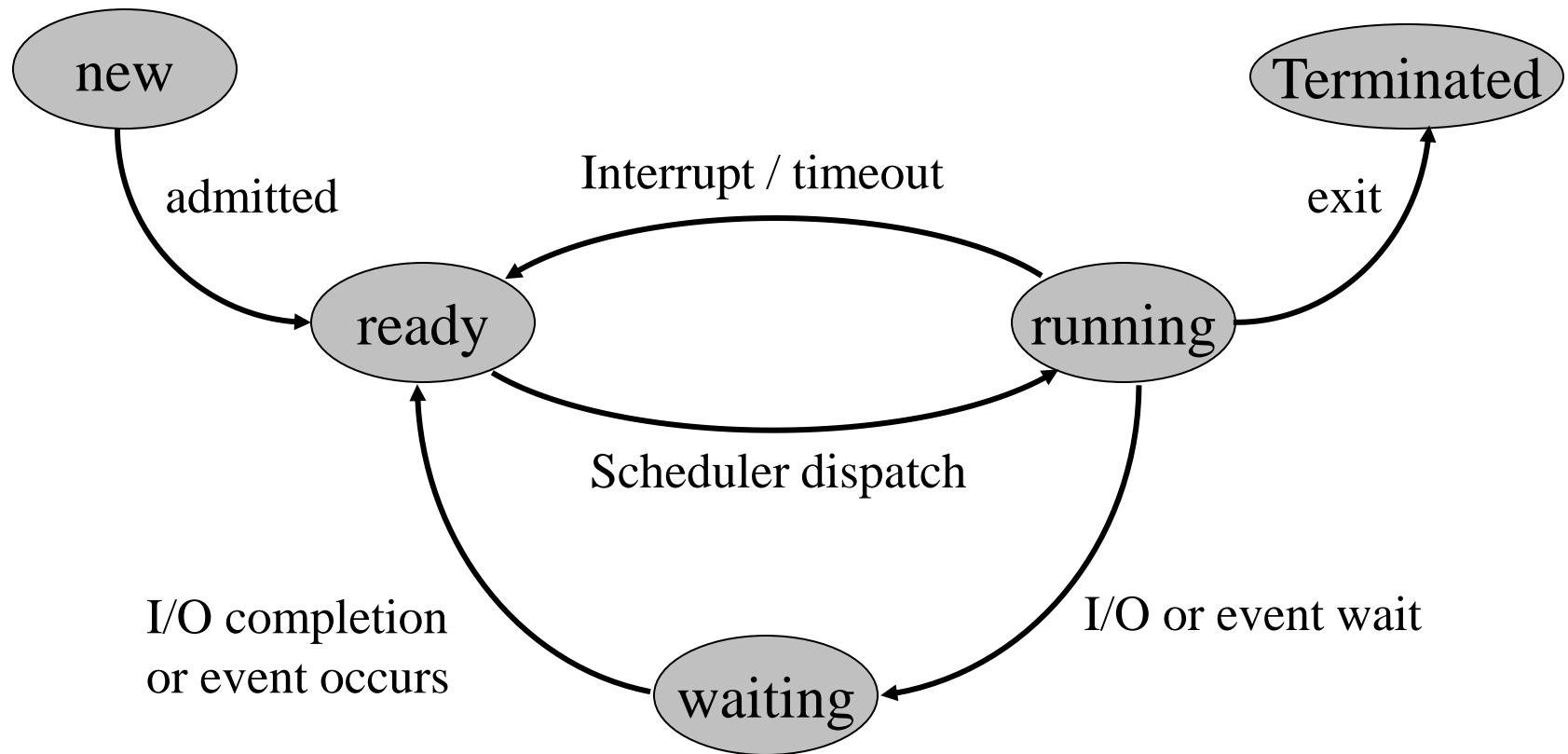
Typical Process Implementation



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.
- These state names are arbitrary, and they vary across operating systems.
- Only one process can be running on any processor at any time, although many processes may be *ready* and *waiting*.

Diagram of Process State



Process Control Block (PCB)

- PCB is also called *task control block*
- PCB contains many pieces of information associated with each process.

pointer	process state
process number	
program counter	
CPU registers	
memory limits	
list of open files	
⋮	

PCB (1/5)

■ Process identification

- **Identifiers, Numeric identifiers that may be stored with the process control block include:**
 - Identifier of this process
 - Identifier of the process that created this process (parent process)
 - User identifier

■ Processor State Information

- **User-Visible Registers**
 - A user-visible register is one that may be referenced by means of the machine language that the processor executes. Typically, there are from 8 to 32 of these registers, although some RISC (Reduced Instruction Set Computer) implementations have over 100.

PCB (2/5)

■ Processor State Information

– Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor. These include

- *Program counter*: Contains the address of the next instruction to be fetched
- *Condition codes*: Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- *Status information*: Includes interrupt enabled/disabled flags, execution mode

– Stack Pointers

- Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

PCB (3/5)

■ Process Control Information

– Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- ***Process state***: defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, terminated).
- ***Priority***: One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
- ***Scheduling-related information***: This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- ***Event***: Identity of event the process is awaiting before it can be resumed

PCB (4/5)

■ Process Control Information

– Data Structuring

- A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

– Interprocess Communication

- Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

– Process Privileges

- Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

PCB (5/5)

■ Process Control Information

– Memory Management

- This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
- the value of the base and limit registers

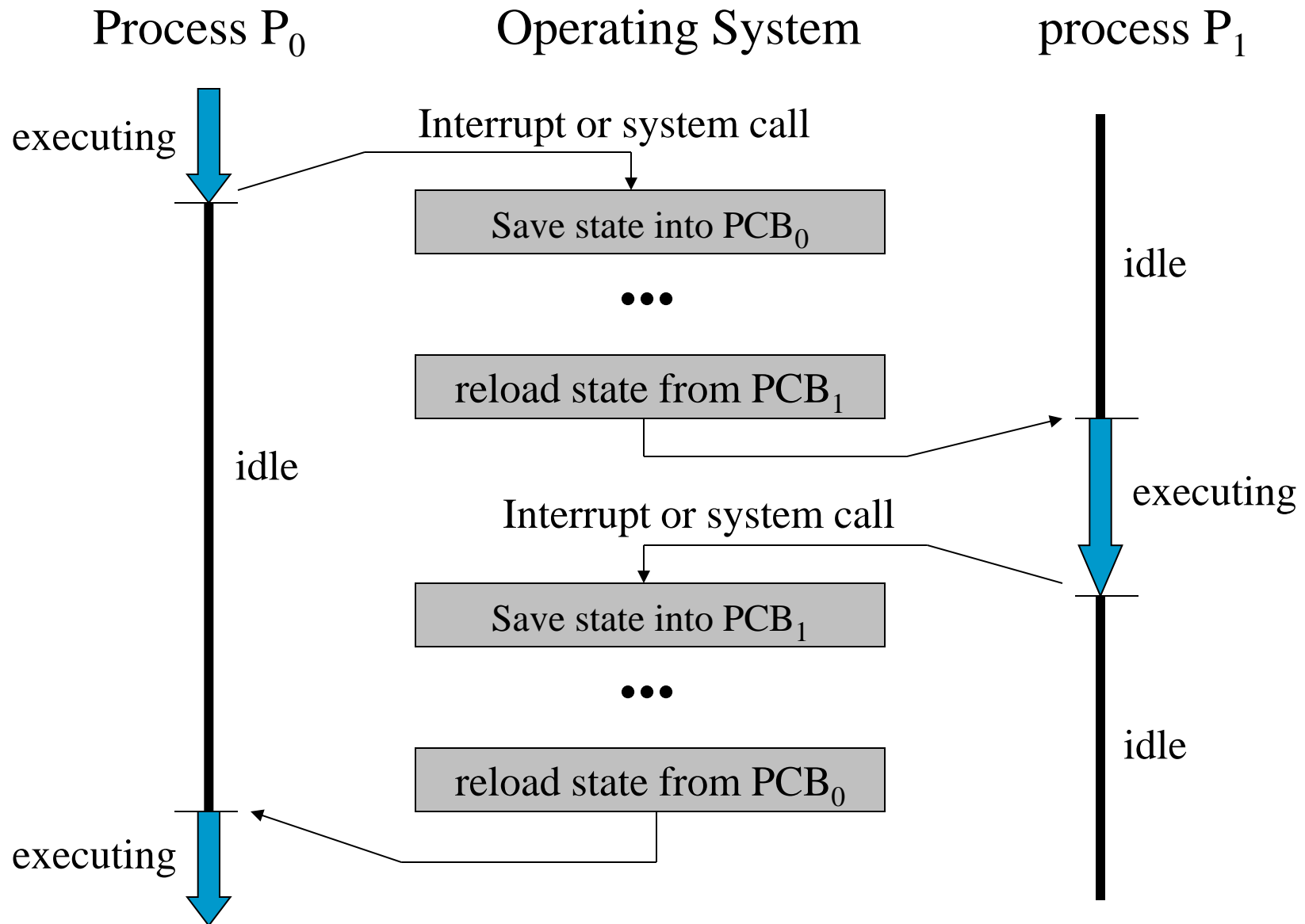
– Resource Ownership and Utilization

- Resources controlled by the process may be indicated, such as opened files.
- A history of utilization of the processor or other resources may also be included;
- this information may be needed by the scheduler.

Supplement: When to Switch a Process

- **Clock interrupt**
 - process has executed for the maximum allowable time slice
- **I/O interrupt**
- **Memory fault**
 - memory address is in virtual memory so it must be brought into main memory
- **Trap**
 - error occurred
 - may cause process to be moved to terminated/exit state
- **System call**
 - such as file open

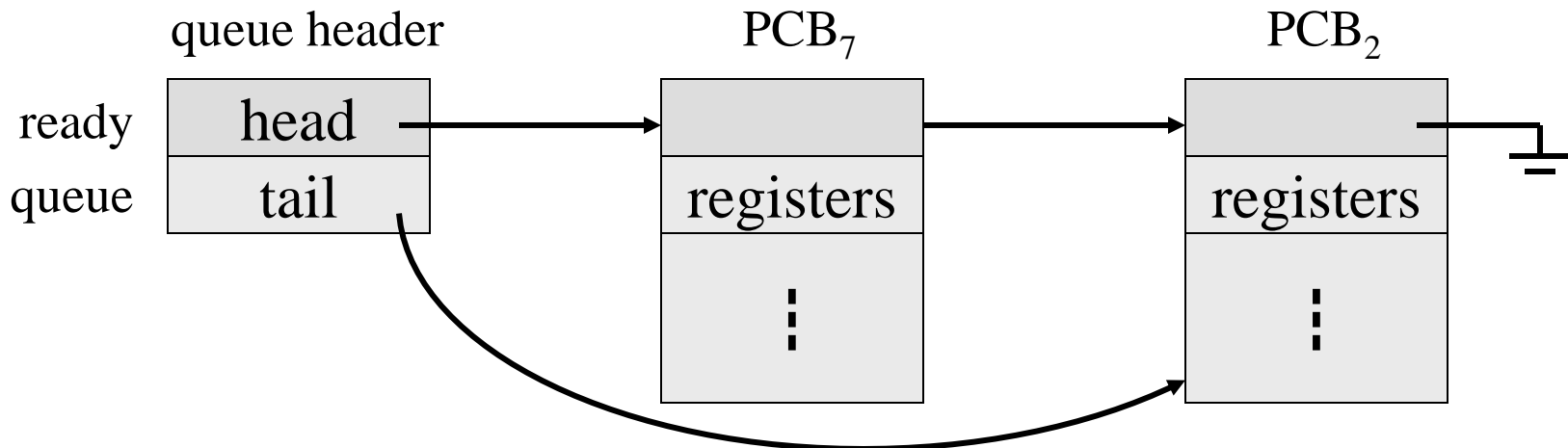
CPU Switch From Process to Process



3.2 Process Scheduling

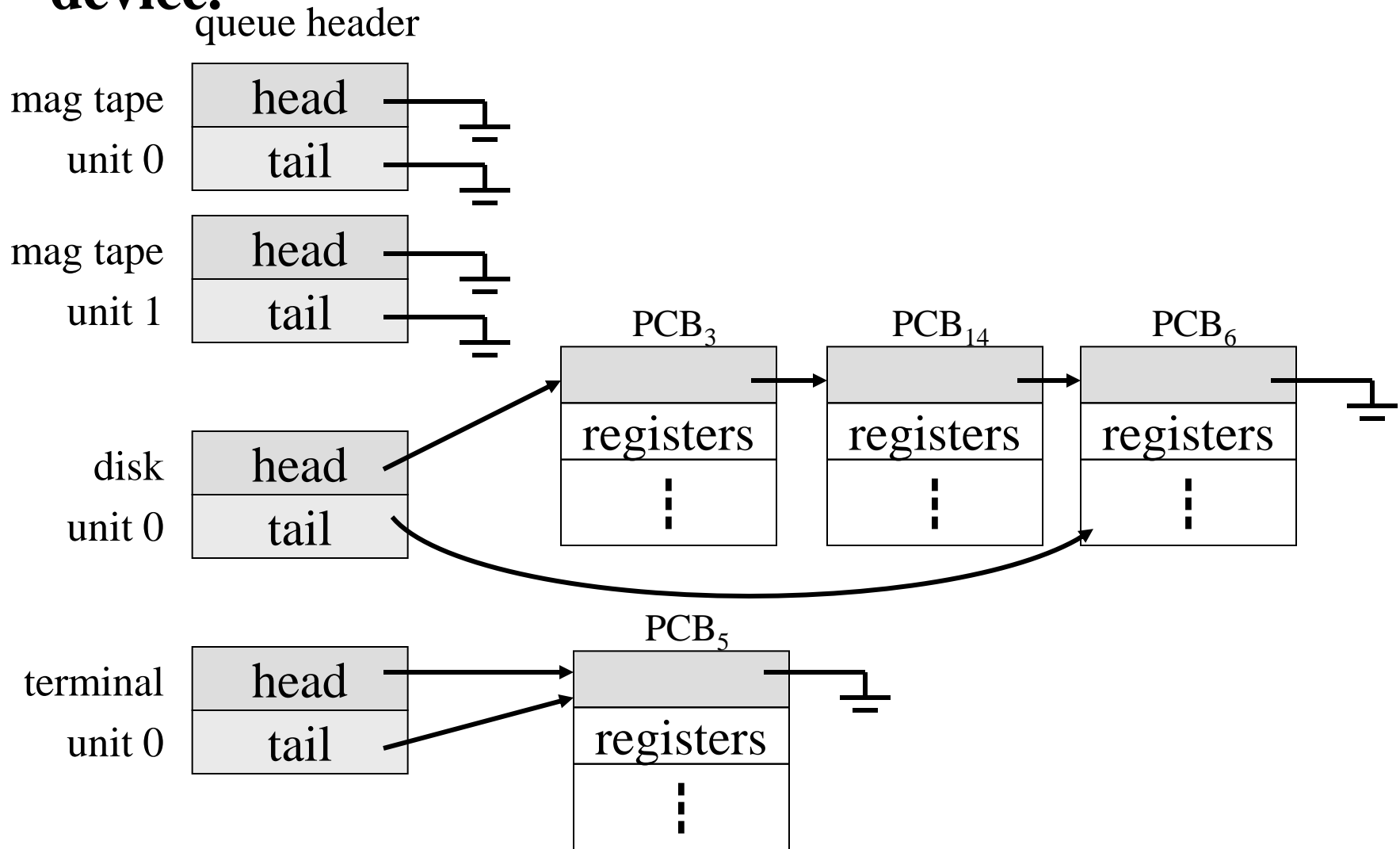
3.2.1 Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
 - generally stored as a linked list
 - header contains pointers to the first and final PCBs in the list



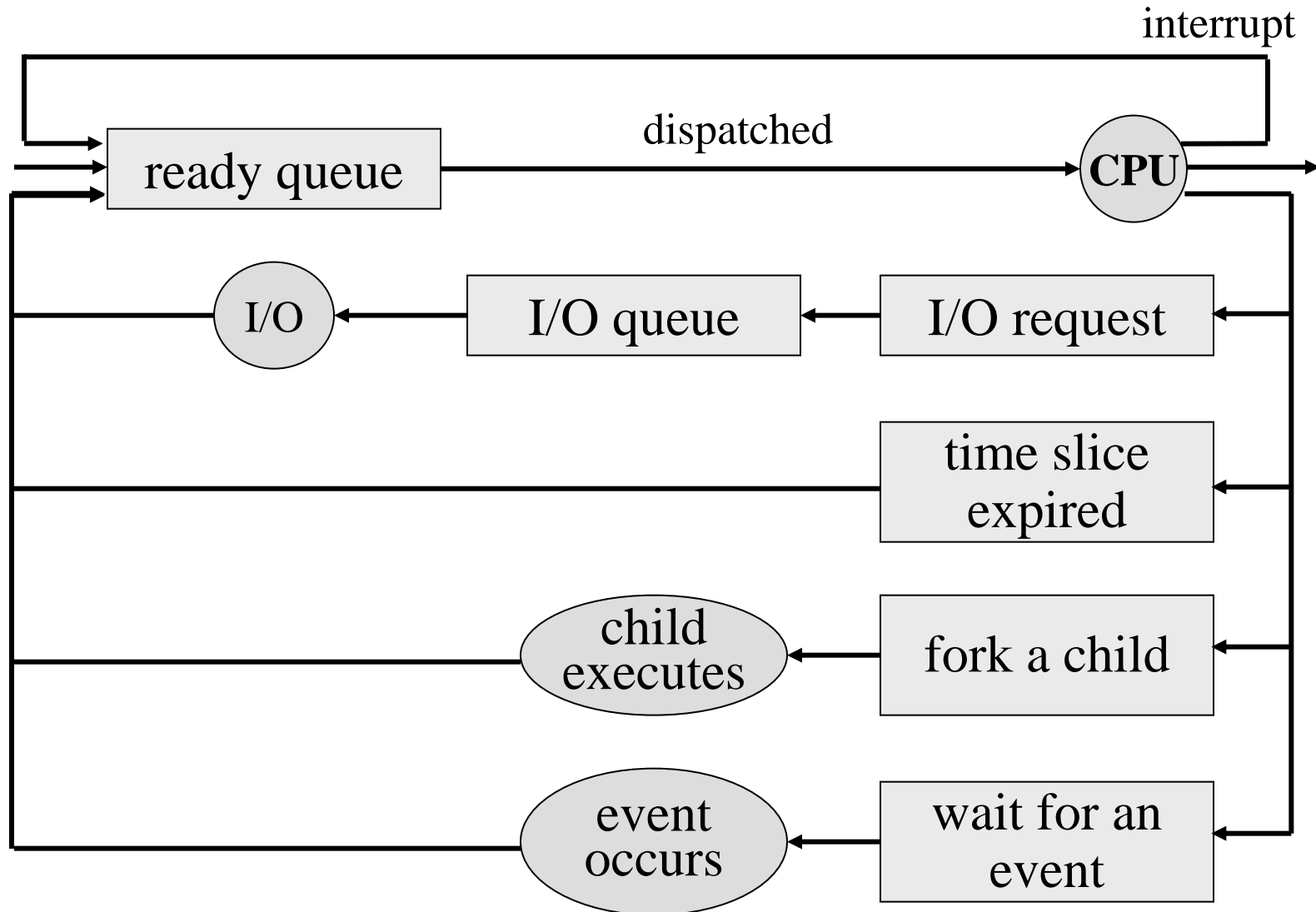
Scheduling Queues(Cont.)

- **Device queues** – set of processes waiting for an I/O device.



Representation of Process Scheduling

- Process migration between the various queues.



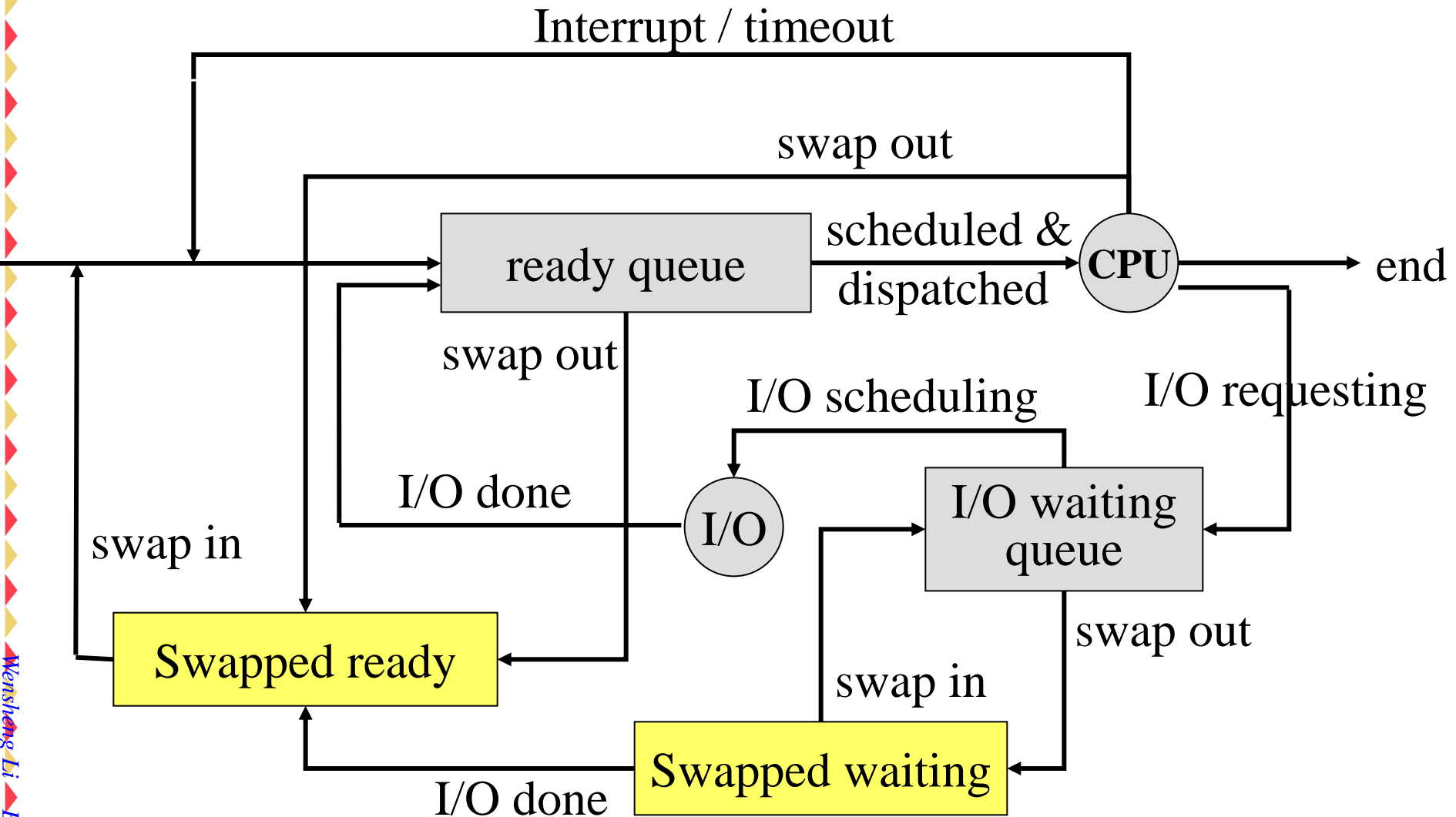
3.2.2 Schedulers

- Long-term scheduler (or **job scheduler**) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or **CPU scheduler**) – selects which process should be executed next and allocates CPU.
- Primary distinction: execution frequency.
 - Short-term scheduler is invoked very frequently (milliseconds) \Rightarrow (scheduler must be fast).
 - Long-term scheduler is invoked very infrequently (seconds, minutes) \Rightarrow (scheduler may be slow).
 - The long-term scheduler controls the *degree of multiprogramming*. the number of processes in memory.

Schedulers (Cont.)

- **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.
- **Select a good process mix.**
 - If all processes are I/O-bound, the ready queue will almost always be empty, and CPU scheduler will have little to do.
 - If all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused.

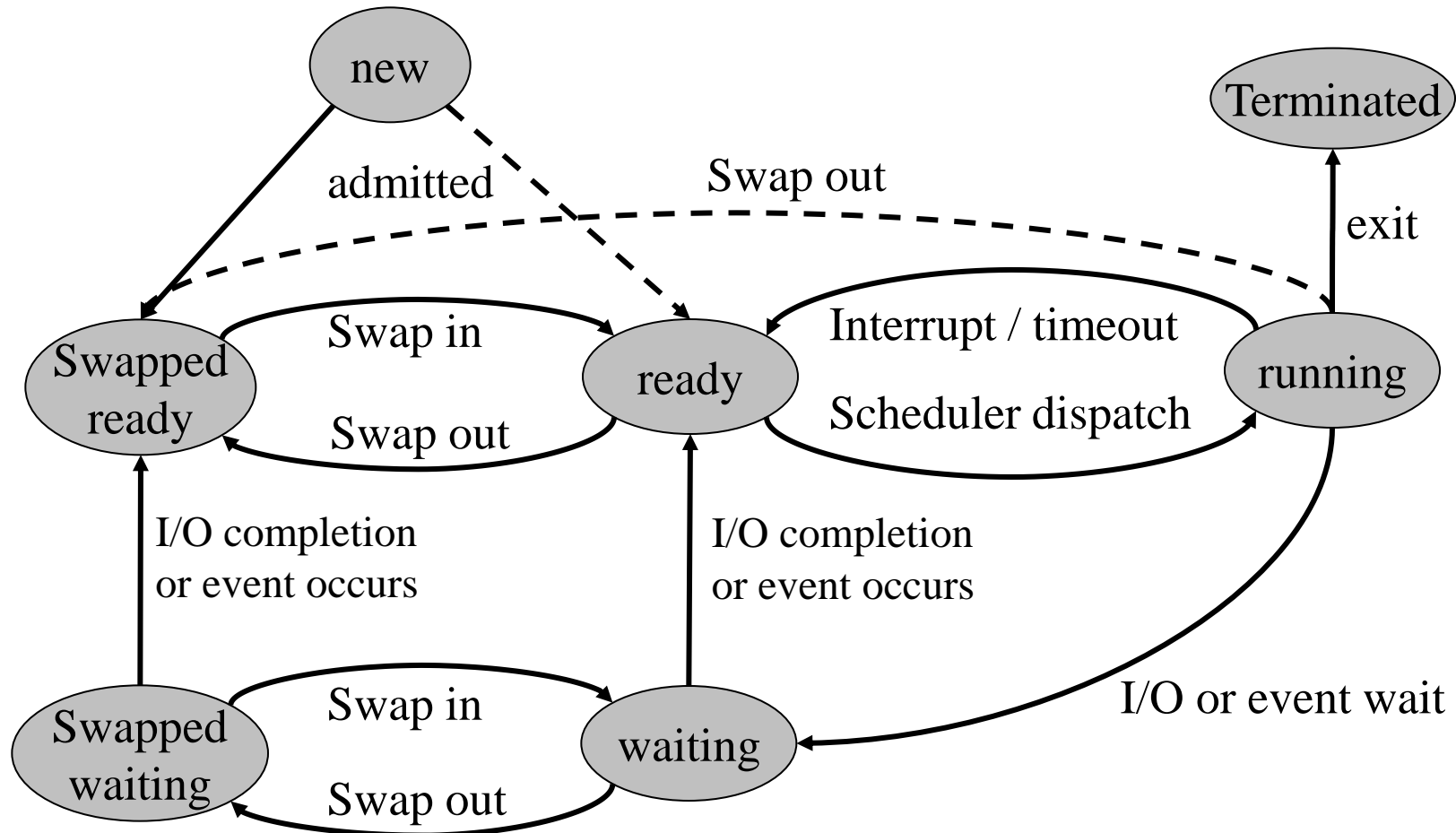
Addition of Medium Term Scheduling



Supplement: Suspended Processes

- Processor is faster than I/O so all processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Waiting state becomes swapped waiting when swapped to disk
- Two new states
 - **swapped waiting** (suspended waiting/blocked)
 - **Swapped ready** (suspended ready)

Supplement: Two swapped State



Supplement: Reasons for Process swapped out

Swapping	The operating system needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The operating system may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purpose of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendant to examine or modify the suspended process, or to coordinate the activity of various descendants.

3.2.3 Context Switch

- **When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. Known as a **context switch**.**
 - Save context of processor including program counter and other registers
 - Update the process control block (PCB) of the process that is currently running
 - Move process control block (PCB) to appropriate queue - ready, waiting
 - Select another process for execution (--scheduling)
 - Update the process control block (PCB) of the process selected
 - Update memory-management data structures
 - Restore context of the selected process

Context Switch

- **The context of a process is represented in the PCB of the process.**
- **Context-switch time is overhead; the system does no useful work while switching.**
- **Context-switch times dependent on hardware support.**

3.3 Operations on Processes

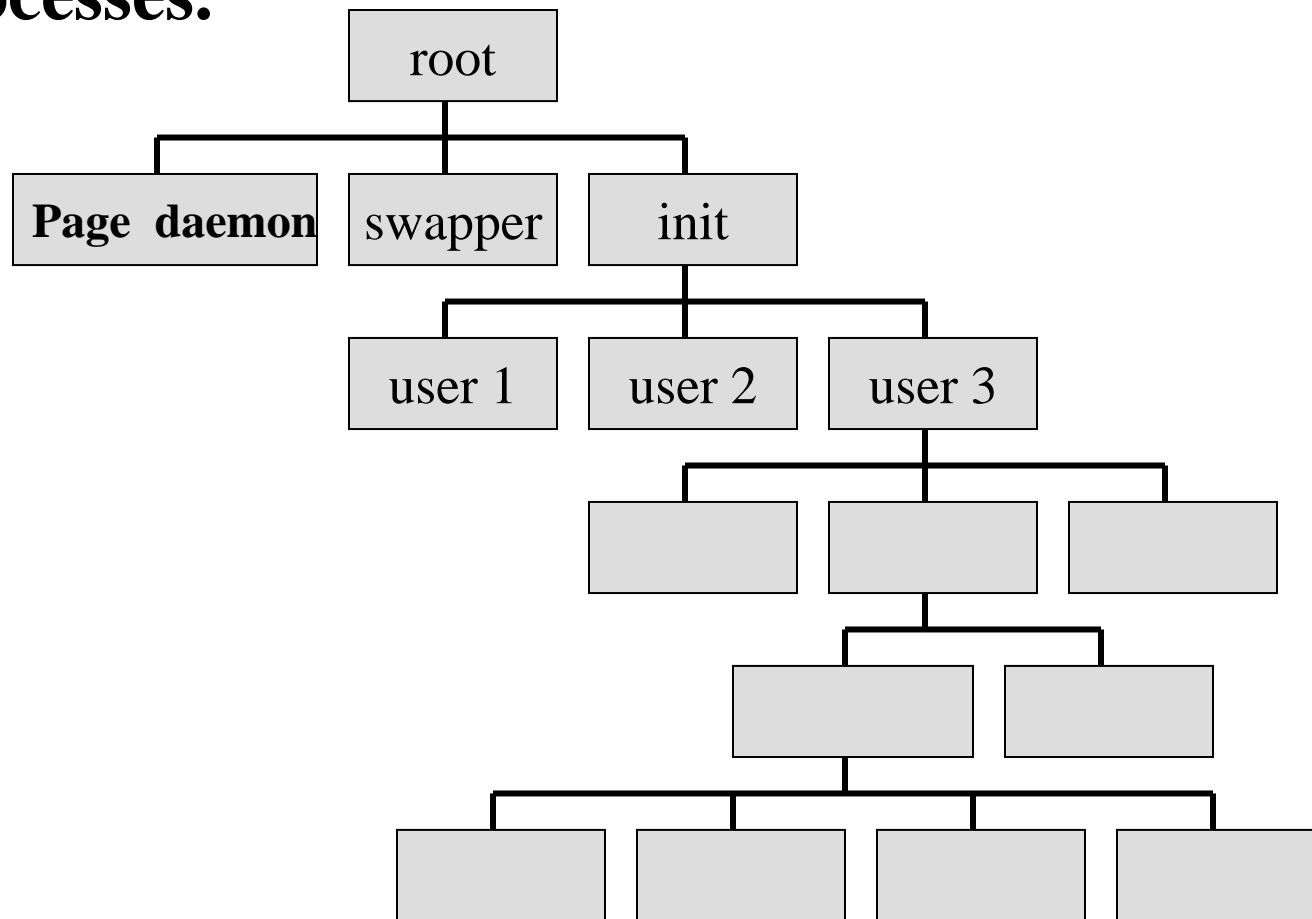
Process Creation & Process Termination

■ Reason for Process Creation

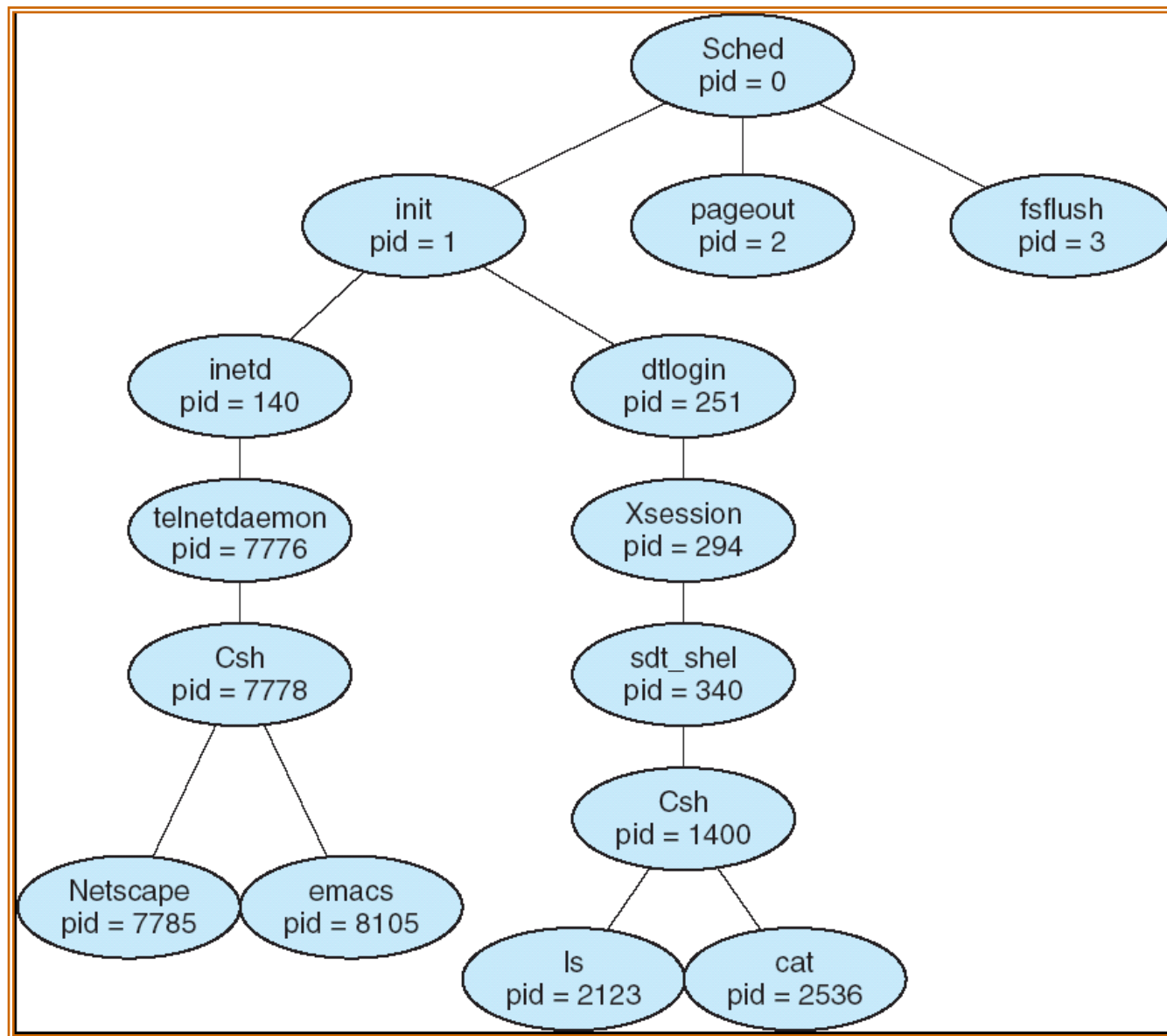
- Submission of a batch job
- User logs on
- Created to provide a service such as printing
- Process creates another process

Process Creation

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



A tree of processes on a typical Solaris



Process Creation (1/3)

- **Assign a unique process identifier**
- **Allocate space for the process**
- **Initialize process control block (PCB)**
- **Set up appropriate linkages**
 - e.g. add new process to linked list used for scheduling queue
- **Create or expand other data structures**
 - e.g. maintain an accounting file

Process Creation (2/3)

■ Resource sharing

- Parent and children share all resources.
- Children share subset of parent's resources.
- Parent and child share no resources.

■ When a process is created, it obtains initialization data that may be passed along from the parent process to the child process.

■ Execution

- Parent and children execute concurrently.
- Parent waits until some or all of its children terminate.

Process Creation (3/3)

■ Address space

- Child is a duplicate of the parent.
- Child has a program loaded into it.

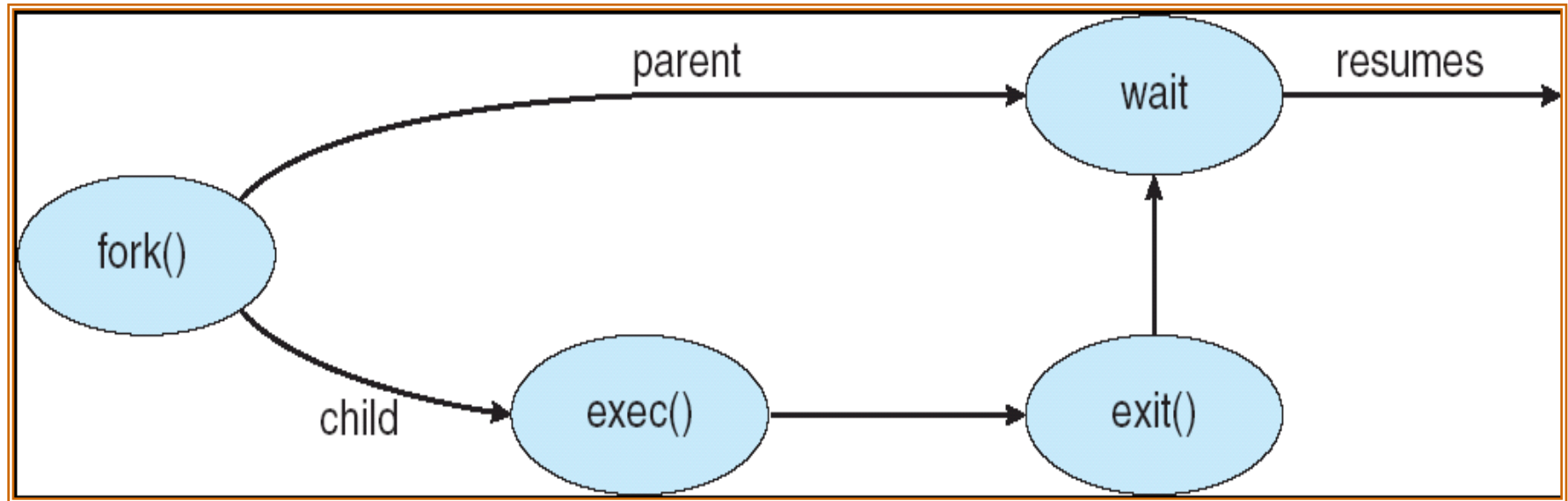
■ UNIX examples

- each process is identified by its process identifier, an unique integer
- system call **fork** creates new process
- system call **exec** used after a fork to replace the child process' memory space with a new program.

C program forking a separate process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
void main( )
{
    pid_t pid;
    pid=fork();    // fork another process
    if (pid<0) {    // error occurred
        fprintf(stderr, "Fork failed");
        exit(-1);
    }
    else if (pid==0) {    // child process
        execlp("/bin/ls", "ls", null);
    } else {    // parent process
        wait(null); /* parent process will wait for the child to
                       complete */
        printf("Child Complete");
        exit(0);
    }
}
```


Process Creation



Creating a separate process using the win32 API

```
#include <windows.h>
#include <stdio.h>
int main( VOID )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) ); // allocate memory
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
```

// Start the child process.

```

if( !CreateProcess( NULL, // No module name (use command line).
    "C:\\WINDOWS\\system32\\mspaint.exe", // Command line.
    NULL, // Process handle not inheritable.
    NULL, // Thread handle not inheritable.
    FALSE, // Set handle inheritance to FALSE.
    0, // No creation flags.
    NULL, // Use parent's environment block.
    NULL, // Use parent's starting directory.
    &si, // Pointer to STARTUPINFO structure.
    &pi ) // Pointer to PROCESS_INFORMATION structure. )
{
    printf( "CreateProcess failed (%d).\n", GetLastError() );
    return -1;
}

// Wait until child process exits.
WaitForSingleObject( pi.hProcess, INFINITE );
printf("parent exit");
// Close process and thread handles.
CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );}

```

Process Termination

- Batch job issues *Halt* instruction
- User logs off
- Quit an application
- Error and fault conditions

Supplement:

Reasons for Process Termination (1/2)

- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation
- Protection error
 - example write to read-only file
- Arithmetic error
- Time overrun
 - process waited longer than a specified maximum for an event

Supplement:

Reasons for Process Termination(2/2)

- **I/O failure (e.g. file unfinded)**
- **Invalid instruction**
 - happens when try to execute data
- **Privileged instruction**
- **Data misuse (e.g. type error)**
- **Operating system intervention**
 - such as when deadlock occurs
- **Parent terminates so child processes terminate**
- **Parent request**

Process Termination

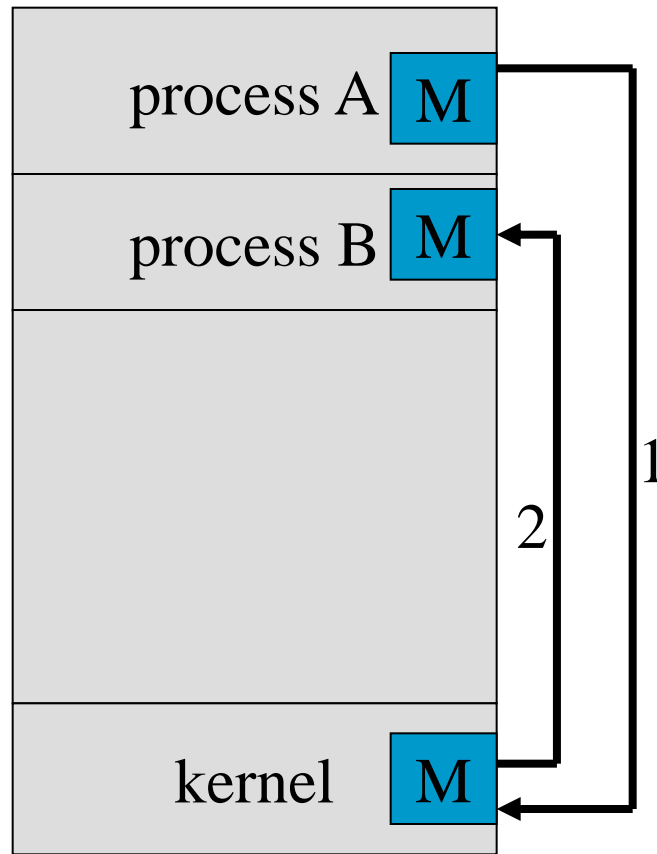
- **Process executes last statement and asks the operating system to delete it (via **exit**).**
 - **Output data from child to parent (via **wait**).**
 - **Process' resources are deallocated by operating system.**
- **Parent may terminate execution of its children processes (via **abort**) for a variety of reasons:**
 - **Child has exceeded its usage of allocated resources.**
 - **Task assigned to child is no longer required.**
 - **Parent is exiting.**
 - **Operating system does not allow child to continue if its parent terminates.**
 - **Cascading termination, normally initiated by the operating system.**

3.4 Interprocess Communication

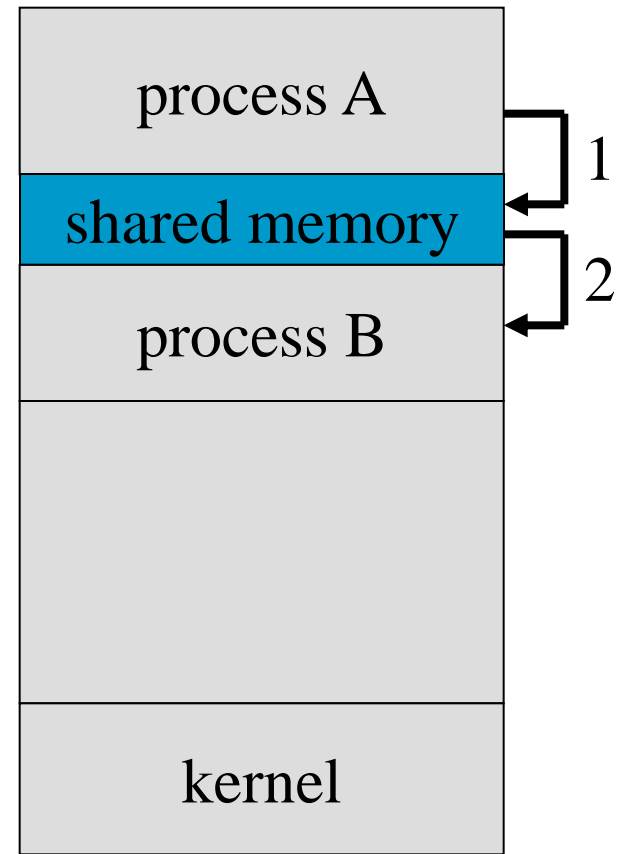
- ***Independent*** process cannot affect or be affected by the execution of another process.
- ***Cooperating*** process can affect or be affected by the execution of another process
- **Advantages of process cooperation**
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience
- **Two fundamental models**
 - Shared-memory
 - Message passing

Communication Models (1/2)

- Communication may take place using either message passing or shared memory.



Message Passing



Shared Memory

Communication Models (2/2)

- **Message passing is**
 - useful when smaller numbers of data need to be exchanged, because no conflicts need to be avoided.
 - Easier to implement for intercomputer communication.
- **Shared memory allows maximum speed and convenience of communication.**
 - It can be done at memory speeds when within a computer.

3.4.1 Shared-memory systems

Producer-Consumer Problem

- A common paradigm for cooperating processes: *producer* process produces information that is consumed by a *consumer* process.
- we must have available a buffer of items that can be filled by the *producer* and emptied by the *consumer*.
 - A *producer* can produce one item while the *consumer* is consuming another item.
 - The *producer* and *consumer* must be **synchronized**.
- Two types of buffer used
 - ***unbounded-buffer*** places no practical limit on the size of the buffer.
 - ***bounded-buffer*** assumes that there is a fixed buffer size.

Shared-Memory Solution to Bounded-Buffer problem

■ Shared data

```
#define BUFFER_SIZE 10
```

```
Typedef struct {
```

```
    ...
```

```
} item;
```

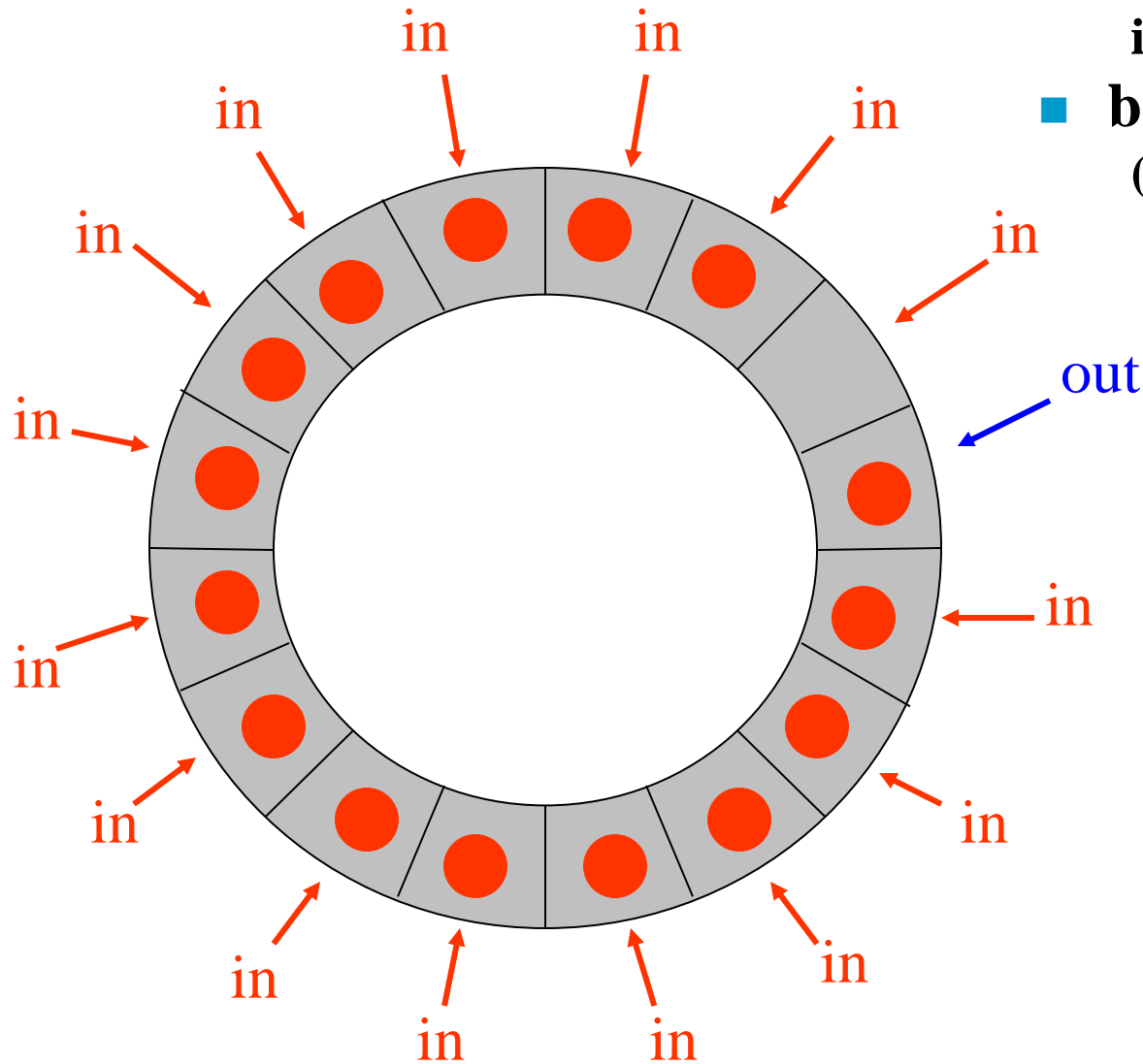
```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

- **in** points to the next free position in the buffer;
- **out** points to the full position in the buffer.
- the buffer is empty when **in==out**;
- the buffer is full when **((in+1) % buffer_size)==out**.
- solution is correct, but can only use **BUFFER_SIZE-1** elements

Shared Buffer--Circular array



- buffer is empty when $in == out$;
- buffer is full when $(in + 1) \% \text{buffer-size} == out$.

**solution is correct,
but can only use
BUFFER_SIZE-1
elements**

Bounded-Buffer – *Producer* Process

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

Bounded-Buffer – *Consumer* Process

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

3.4.2 Message-passing systems

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message passing 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 aspects (e.g., shared memory, hardware bus, network)
 - logical aspects (e.g., logical properties)

Implementation Questions

- **Direct / indirect communication?**
- **Synchronous / asynchronous communication?**
- **Automatic / explicit buffering?**
- **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--naming

- **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.
- **disadvantages**
 - Limited modularity of the resulting process definitions

Indirect Communication(1/3)

- Messages are sent and received from mailboxes (also referred to as ports).
 - Each mailbox has a unique id.
 - Processes can communicate only if they share a mailbox.
- Primitives are defined as:
 - send**(*A*, *message*) – send a message to mailbox A
 - receive**(*A*, *message*) – receive a message from mailbox A
- 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(2/3)

■ 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.

Indirect Communication(3/3)

- **If the mailbox is owned by a process**
 - owner can only receive messages through this mailbox
 - user can only send messages to the mailbox
- **if the mailbox is owned by the operating system**
 - the mailbox is independent
 - not attached to any particular process
- **OS provides mechanism allowing a process to do following operations**
 - create a new mailbox
 - send and receive messages through mailbox
 - delete a mailbox

Synchronization

- Message passing may be either blocking or non-blocking.
- Blocking is considered **synchronous**
 - **Blocking send**: sending process is blocked until the message is received by the receiving process or by the mailbox.
 - **Blocking receive**: receiver blocks until a message is available.
- Non-blocking is considered **asynchronous**
 - **Nonblocking send**: sending process sends the message and resumes operation.
 - **Nonblocking receive**: receiver retrieves either a valid message or a null.

Buffering

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

Supplement: pipeline(管道)

- 管道是一条在进程间以字节流方式传送数据的通信通道。
- 由**OS**核心的缓冲区（通常几十**KB**）来实现，是单向的；
- 常用于命令行所指定的输入输出重定向和管道命令。
- 使用前要建立相应的管道，然后才可使用。
- 管道逻辑上可以看作管道文件，物理上则是由文件系统的高速缓存区构成。
- 管道分为有名管道和无名管道
- 管道按先进先出（**FIFO**）的方式传送消息，且只能单向传送消息。
- 管道操作
 - 发送进程利用文件系统的系统调用 **write (fd[1], buf, size)** 把**buf**中长度为**size**字符的消息送入管道入口**fd[1]**；
 - 接收进程利用文件系统的系统调用 **read (fd[0], buf, size)** 从管道出口**fd[0]**读出**size**字符的消息放入**buf**中。

Supplement: 管道应用示例

```
#include<stdio.h>
main()
{
    int pid,fd[2];
    char  buf[30], s[30];
    pipe(fd);
    while((pid=fork())!=-1);
    if (pid==0) {
        sprintf(buf, "this is an example\n");
        write(fd[1],buf,30);
        exit(0);
    }
    else {
        wait(0);
        read(fd[0],s,30);
        printf("%s",s);
    }
}
```


3.5 example of IPC systems

C program illustrating POSIX shared-memory API

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main()
{
    int segment_id;           // the id for the shared memory segment
    char* shared_memory;      // a pointer to the shared memory segment
    const int segment_size = 4096; // the size of the shared memory segment

    /** allocate a shared memory segment */
    segment_id = shmget (IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);

    /** attach the shared memory segment */
    shared_memory = (char *) shmat (segment_id, NULL, 0);

    printf("shared memory segment %d attached at address %p\n", segment_id,
        shared_memory);
}
```

```
/** write a message to the shared memory segment */  
sprintf(shared_memory, "Hi there!");
```

```
/** now print out the string from shared memory */  
printf("*%s*\n", shared_memory);
```

```
/** now detach the shared memory segment */  
if ( shmdt(shared_memory) == -1)  
{  
    fprintf(stderr, "Unable to detach\n");  
}
```

```
/** now remove the shared memory segment */  
shmctl(segment_id, IPC_RMID, NULL);
```

```
return 0;
```

```
}
```

message passing facility in Windows XP-- LPC

- **LPC Communicates between two processes on the same machine.**
- **a port object is used to establish and maintain a connection between two processes.**
- **two types of ports:**
 - **Connection ports**
named objects, visible to all processes;
used to set up communication channels.
 - **Communication ports**
- **Communication works**
 - **Client opens a handle to the subsystem's connection port object**
 - **Client sends a connection request**
 - **Server creates two private communication ports and returns the handle to one of them to the client**
 - **Client and server use the corresponding port handle to send messages or callbacks and to listen for replies**

■ Two types of message-passing techniques

- Used for small messages (up to 256 bytes)
simplest.

uses the port's message queue as intermediate storage and copies the message from one process to the other.

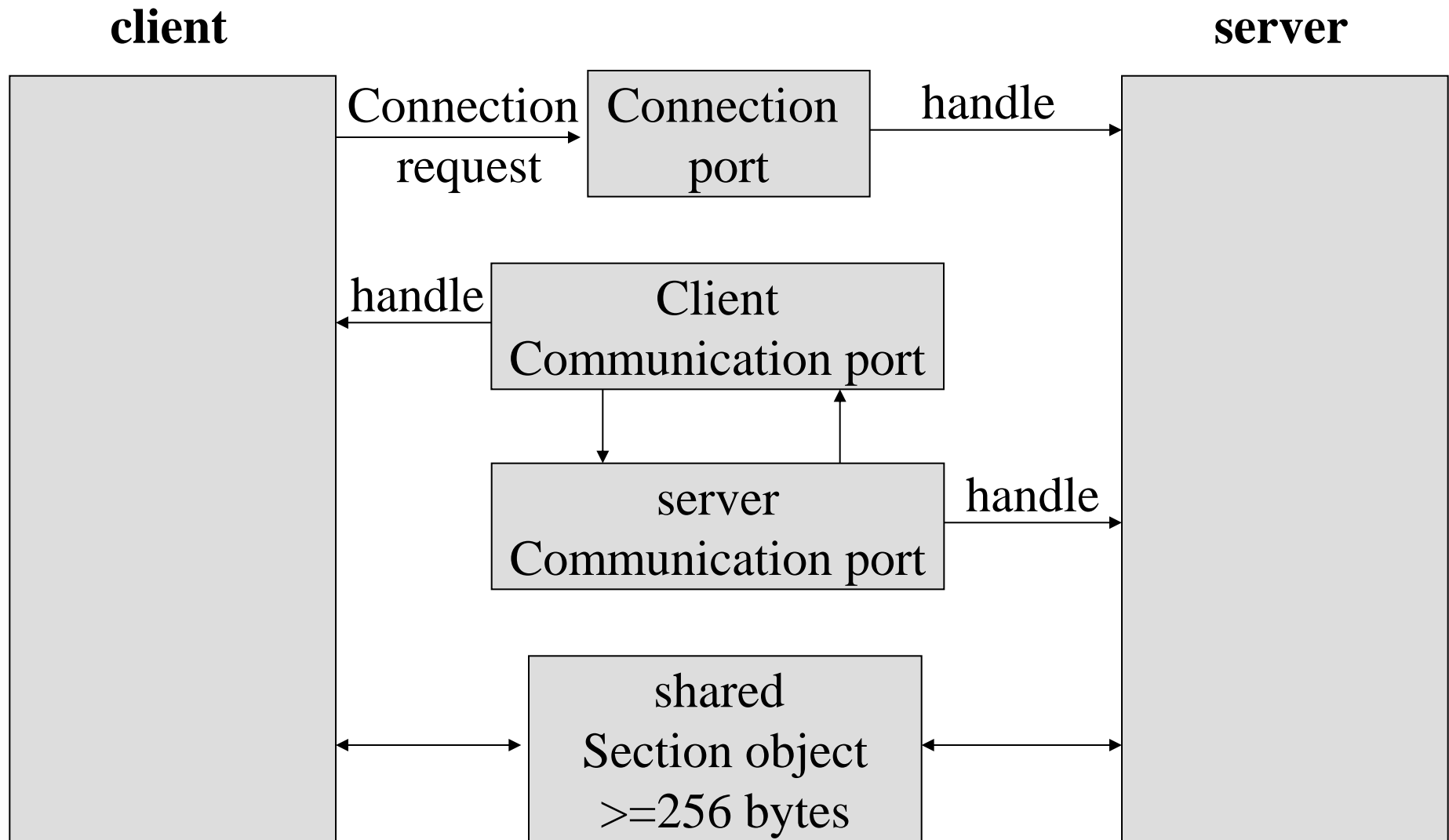
- A section object (for large messages)
sets up a region of shared memory.

A small message is sent that contains a pointer and size information about the section object.

■ callback mechanism

- Can be used when either the client or the server cannot respond immediately to a request.
- Allows to perform asynchronous message handling.

LPC in Windows XP



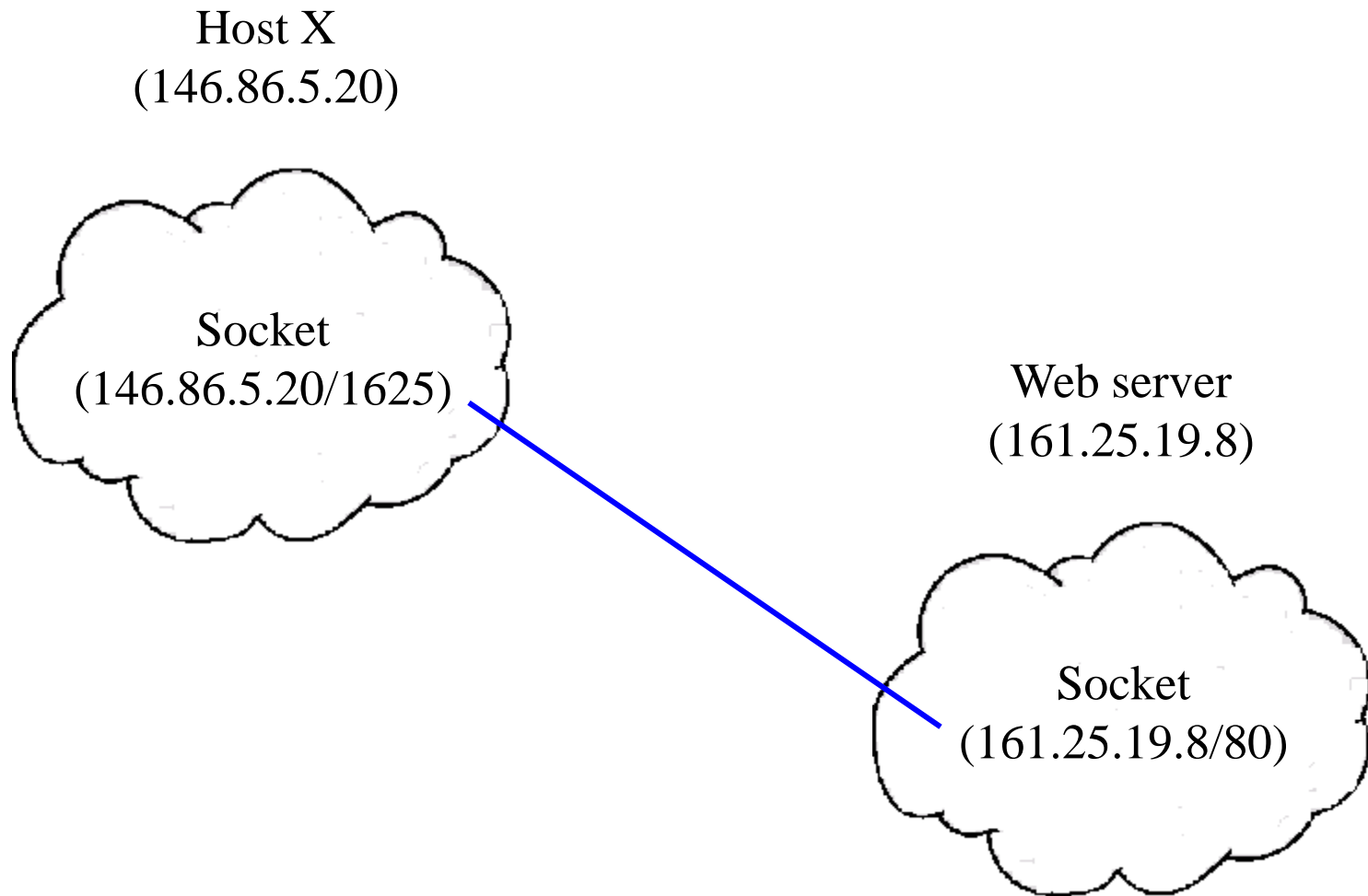
3.6 Communication in Client-Server systems

- **Sockets**
- **Remote Procedure Calls (RPC)**
- **Remote Method Invocation (RMI, Java)**

Sockets

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

Communication using sockets



All connections must be unique.

RPCs -- Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally.

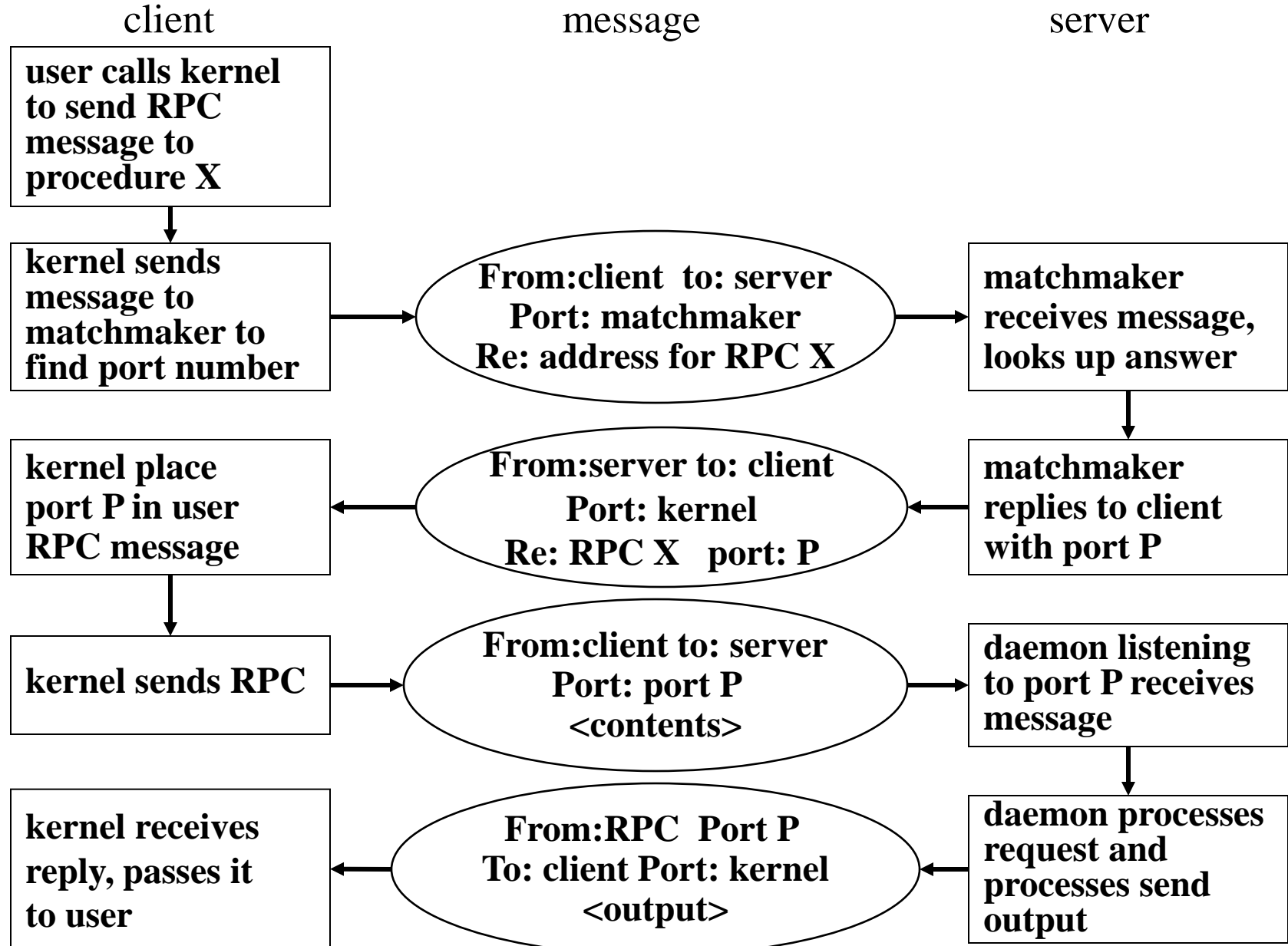
By providing a stub on the client side

- **Stubs** – client-side proxy for the actual procedure on the server.
- **The client-side stub** locates port on the server and *marshals* the parameters.
- **The server-side stub** receives this message, unpacks the marshaled parameters, and performs the procedure on the server. Return values are passed back to the client using the same technique.

Remote Procedure Calls (Cont.)

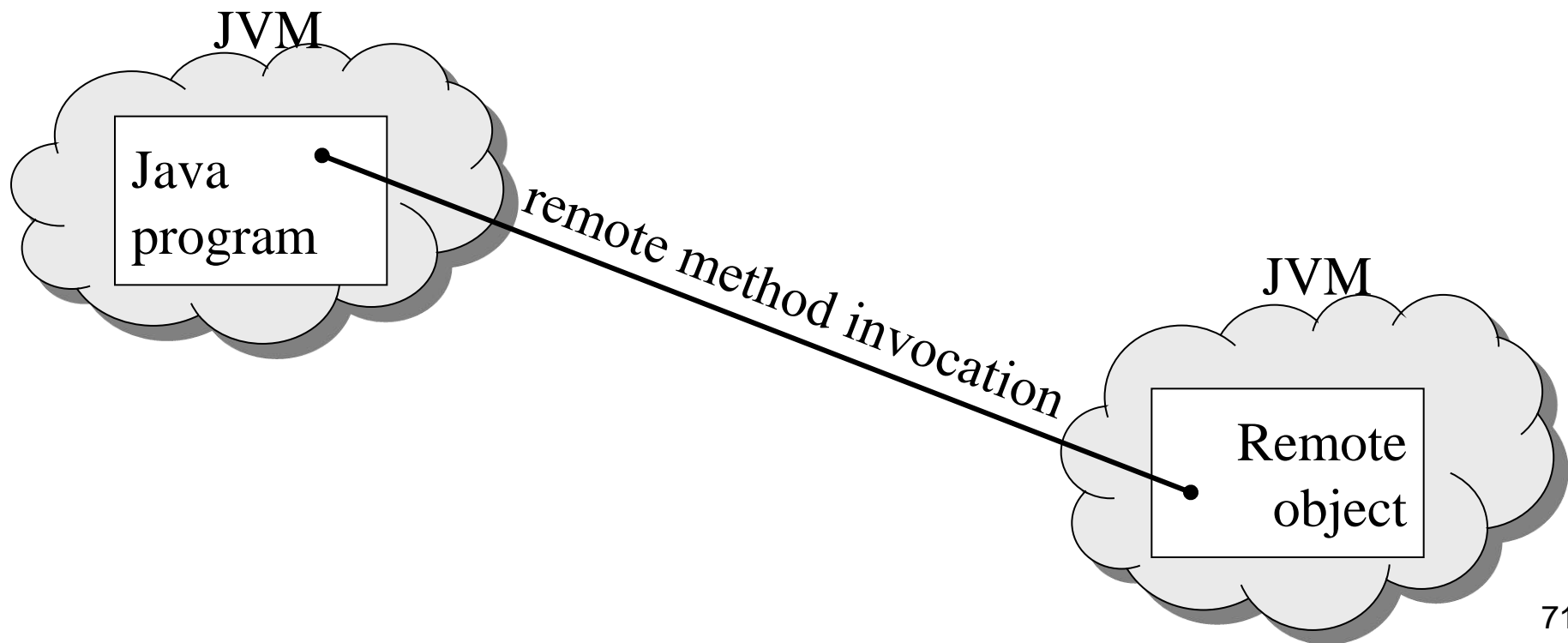
- **machine-independent representation of data, external data representation(XDR).**
 - Client, converting machine-dependent data into XDR
 - server, XDR data is unmarshalled and converted into the machine-dependent representation.
- **The semantics of a call**
 - attaching to each message a timestamp.
 - the server must keep a history of all the timestamps of messages it has already processed
- **the communication between a server and a client**
 - binding information may be predetermined, fixed port address
 - binding can be done dynamically by a rendezvous mechanism

Execution of RPC

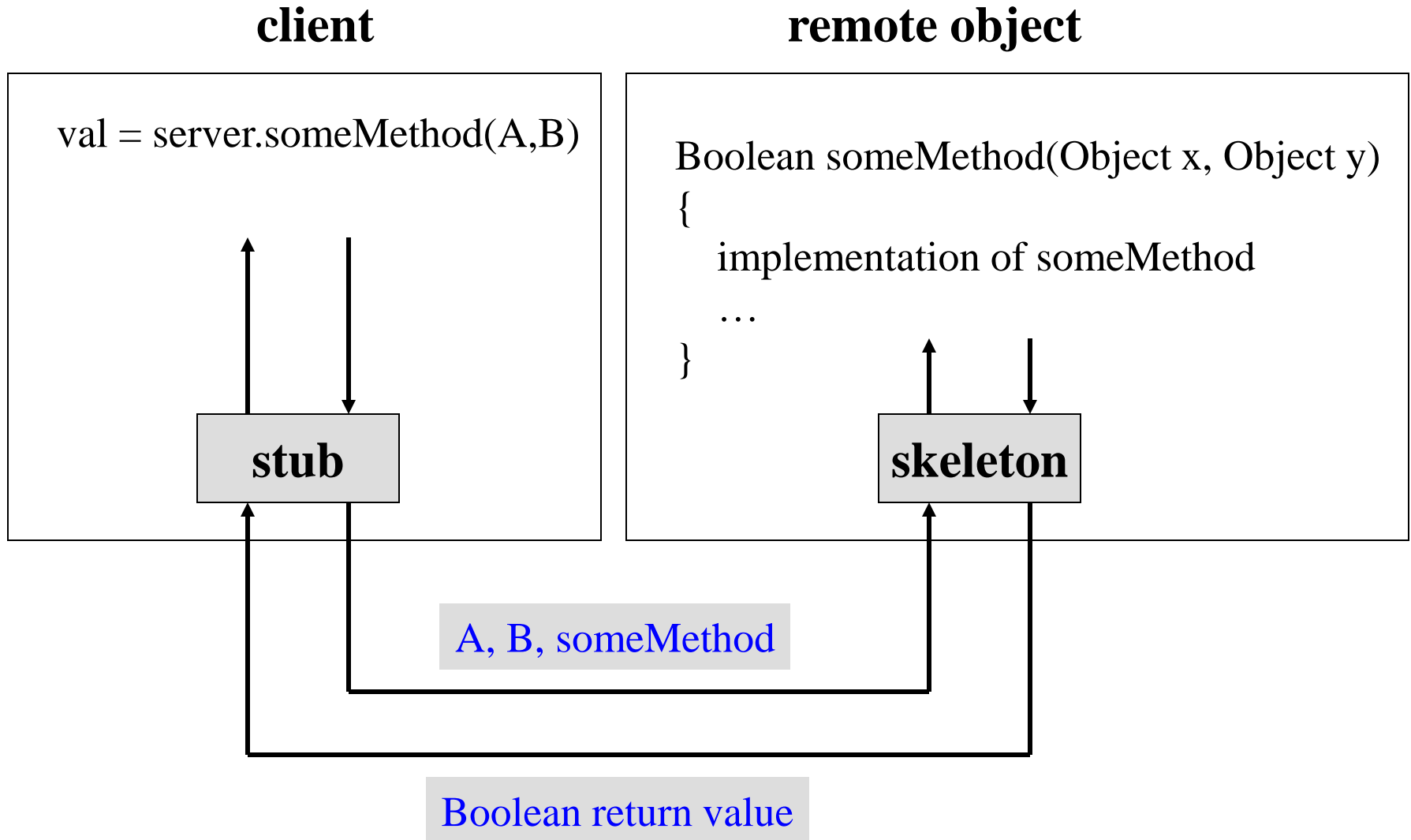


RMI -- Remote Method Invocation

- **Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.**
- **RMI allows a Java program on one machine to invoke a method on a remote object.**



Marshalling Parameters



rules about parameter passing behavior

- If the marshalled parameters are local objects, they are passed by copy using a technique known as object serialization.

If the parameters are also remote objects, they are passed by reference.

- If local objects are to be passed as parameters to remote objects, they must implement the interface `java.io.Serializable`.

Differences between RPCs and RMI

- **RPCs support procedural programming whereby only remote procedures or functions may be called.**
RMI is object-based, it supports invocation of methods on remote objects.
- **The parameters to remote procedures are ordinary data structures in RPC;**
with RMI it is possible to pass objects as parameters to remote methods.
- **RMI implements the remote object using stubs and skeletons, so remote methods are transparent to both the client and the server**
 - **A stub is a proxy for the remote object, resides with the client**
 - **The skeleton is responsible for unmarshalling the parameters and invoking the desired method on the server**

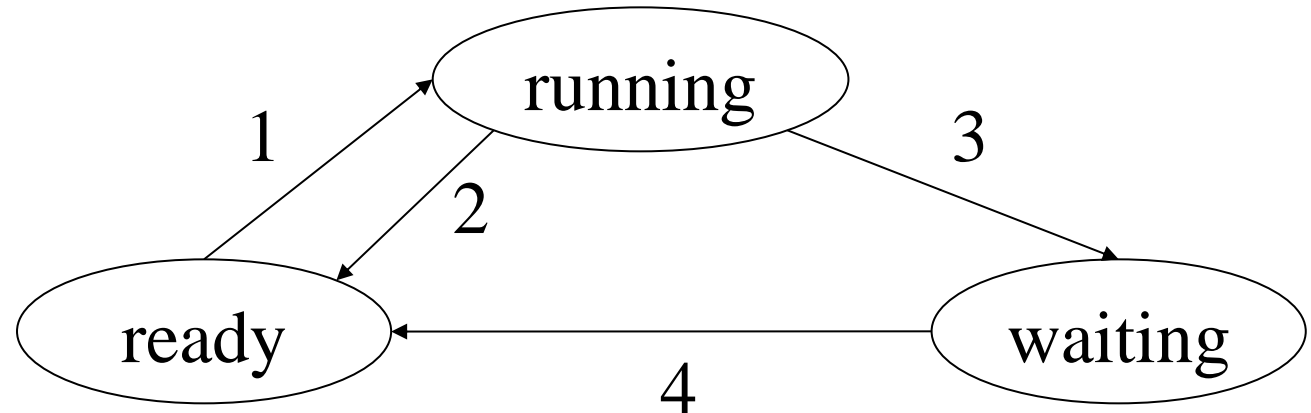
Homework (page 116)

3.1 3.2 3.4

1. On a system with n CPUs,
 - a. what is the maximum number of processes that can be in the ready, running, and waiting states?
 - b. what is the minimum number of processes that can be in the ready, running, and waiting states?
2. For each of the following transitions between process states, indicate whether the transition is possible. If it is possible, give an example of one thing that would cause it.
 - a. Running \rightarrow ready
 - b. Running \rightarrow waiting
 - c. Running \rightarrow swapped waiting
 - d. Waiting \rightarrow running
 - e. Running \rightarrow terminated

3. See the following diagram of process state transition. Can the following cause and effect transitions be possible? If yes, give an example.

- a. 2 1
- b. 3 2
- c. 4 1
- d. 1 3



Chapter 4 Threads



LI Wensheng, SCS, BUPT

Teaching hours: 2h

Strong point:

concepts of thread

Multithreading Models

Threading Issues

Chapter 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 Pthreads, Win32, and Java thread libraries**

Contents

- 4.1 Overview**
- 4.2 Multithreading Models**
- 4.3 Threads Libraries**
- 4.4 Threading Issues**
- 4.5 Operating-System examples**

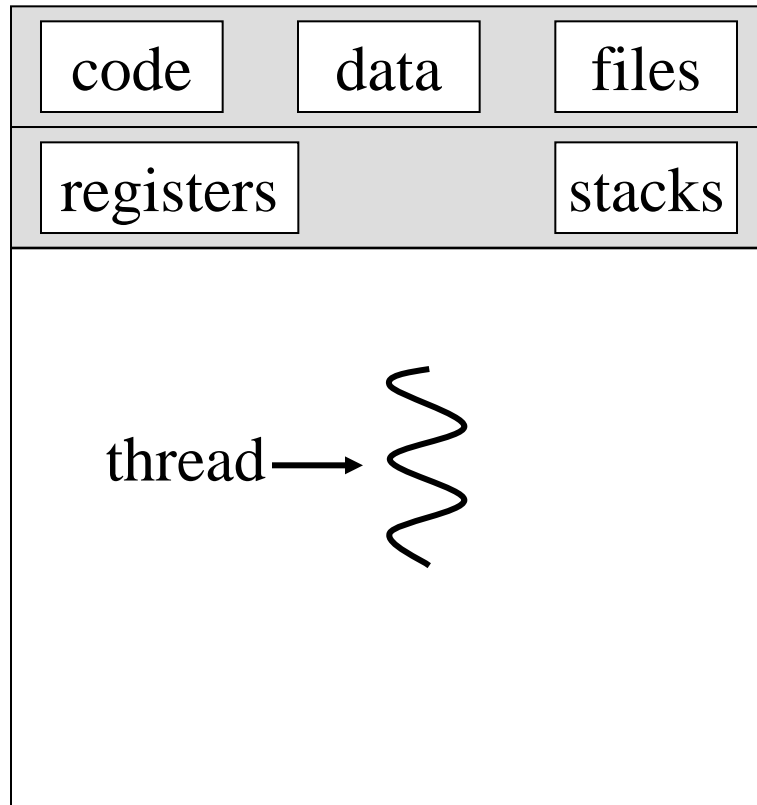
4.1 Overview

- **process**
 - **Resource ownership** - process is allocated a virtual address space to hold the process image
 - **Scheduling/execution**- follows an execution path that may be interleaved(交叉) with other processes
 - These two characteristics are treated independently by the operating system
- **Dispatching** is referred to as a thread
- **Resource of ownership** is referred to as a process or task

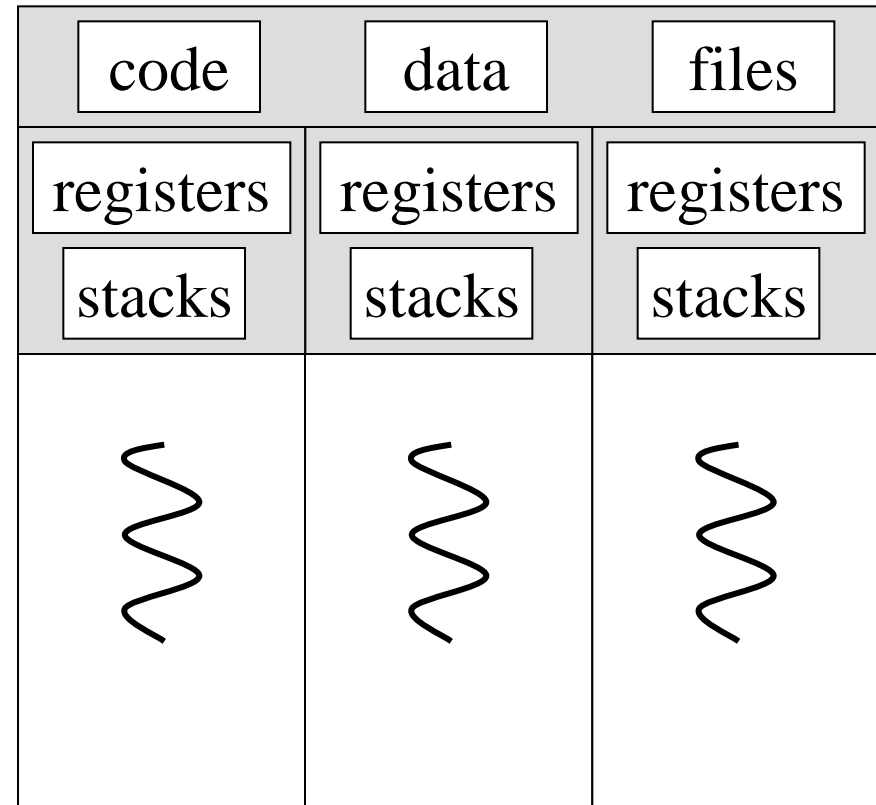
Overview (Cont.)

- **A thread, called a lightweight process (LWP), is a basic unit of CPU utilization.**
 - **Comprises a thread ID, a program counter, a register set, and a stack.**
 - **Shares with other threads belongs to the same process its code section, data section, and other operating-system resources, such as open files and signals.**
- **A traditional (or heavyweight) process has a single thread of control.**
- **If the process has multiple threads of control, it can do more than one task at a time.**

Single and Multithreaded Processes



single-threaded



multithreaded

- An application typically is implemented as a separate process with several threads of control.

Example?

Benefits of Threads

- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Utilization of MP Architectures**

Supplement: Example of Uses of Threads in a Single-User Multiprocessing System

- **Foreground to background work**
 - Web Browser
- **Asynchronous processing**
 - Word, automatically save
- **Speed execution**
 - Data accepting, manipulating, saving, outputting
- **Modular program structure**

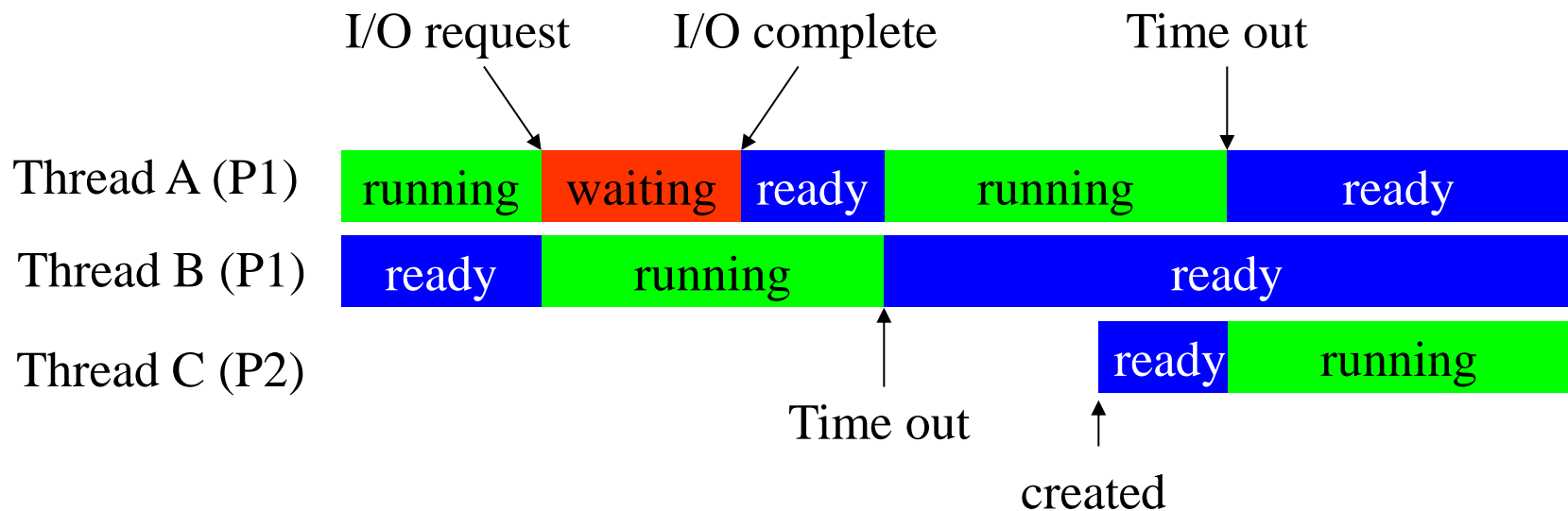
Supplement: Threads

- **Swapping a process involves swapping all threads of the process**
- **Termination of a process, terminates all threads within the process**

since all threads share the same address space

Supplement: Thread States

- Running, ready, waiting
- operations associated with a change in thread state
 - Spawn: Spawn another thread
 - Block
 - Unblock
 - Finish: Deallocate register context and stacks



4.2 Multithreading Models

- **User threads**
- **Kernel threads**

- **Three common types of threading implementation**
 - **Many-to-One**
 - **One-to-One**
 - **Many-to-Many**

Supplement: User Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads
- User threads are supported above the kernel and are implemented by a thread library at the user level.
- The library provides support for thread creation, scheduling, and management with no support from the kernel.
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads

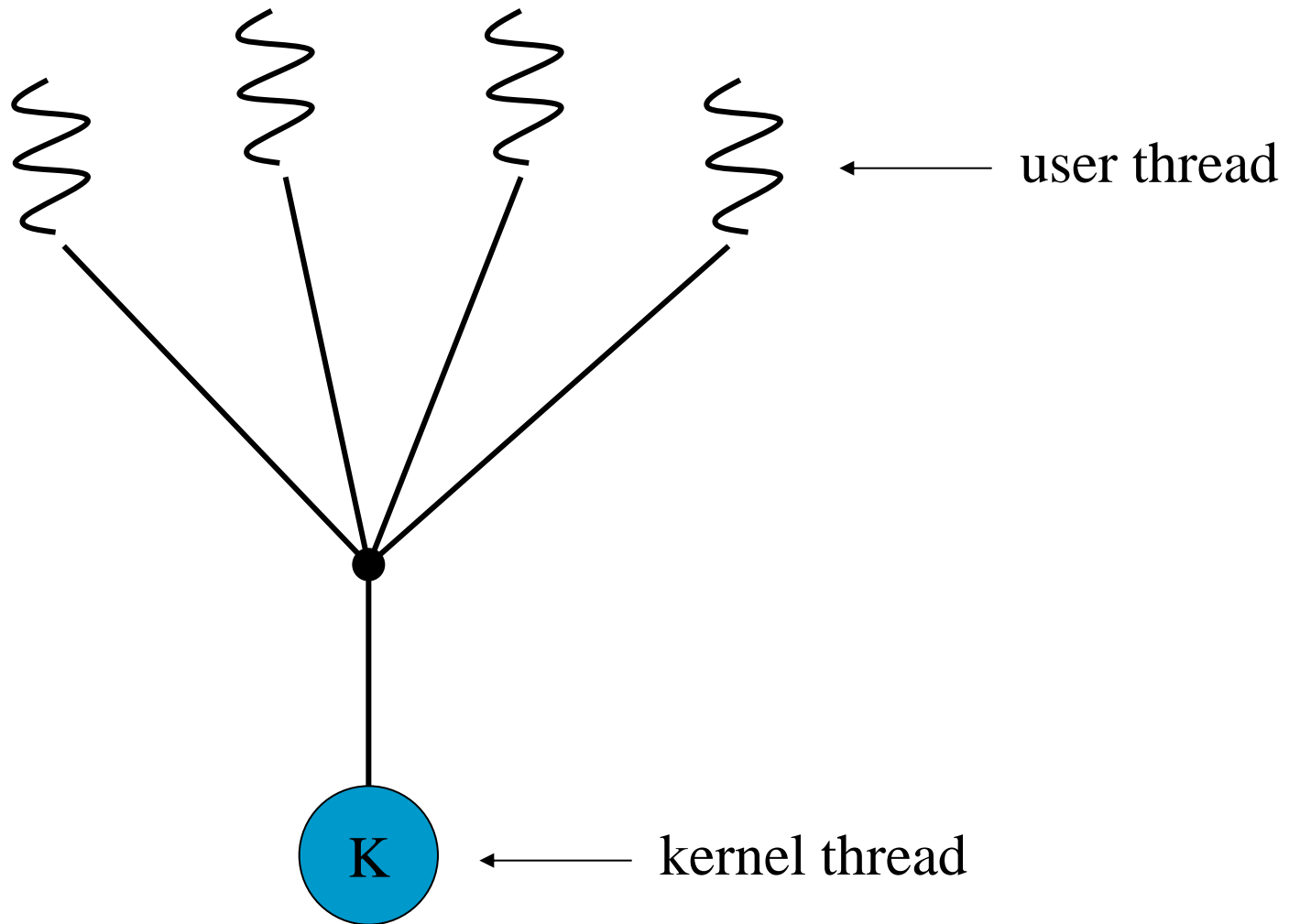
Supplement: Kernel Threads

- Supported directly by the operating system.
- The Kernel performs thread creation, scheduling, and management in kernel space.
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis
- Examples of systems supporting kernel threads
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

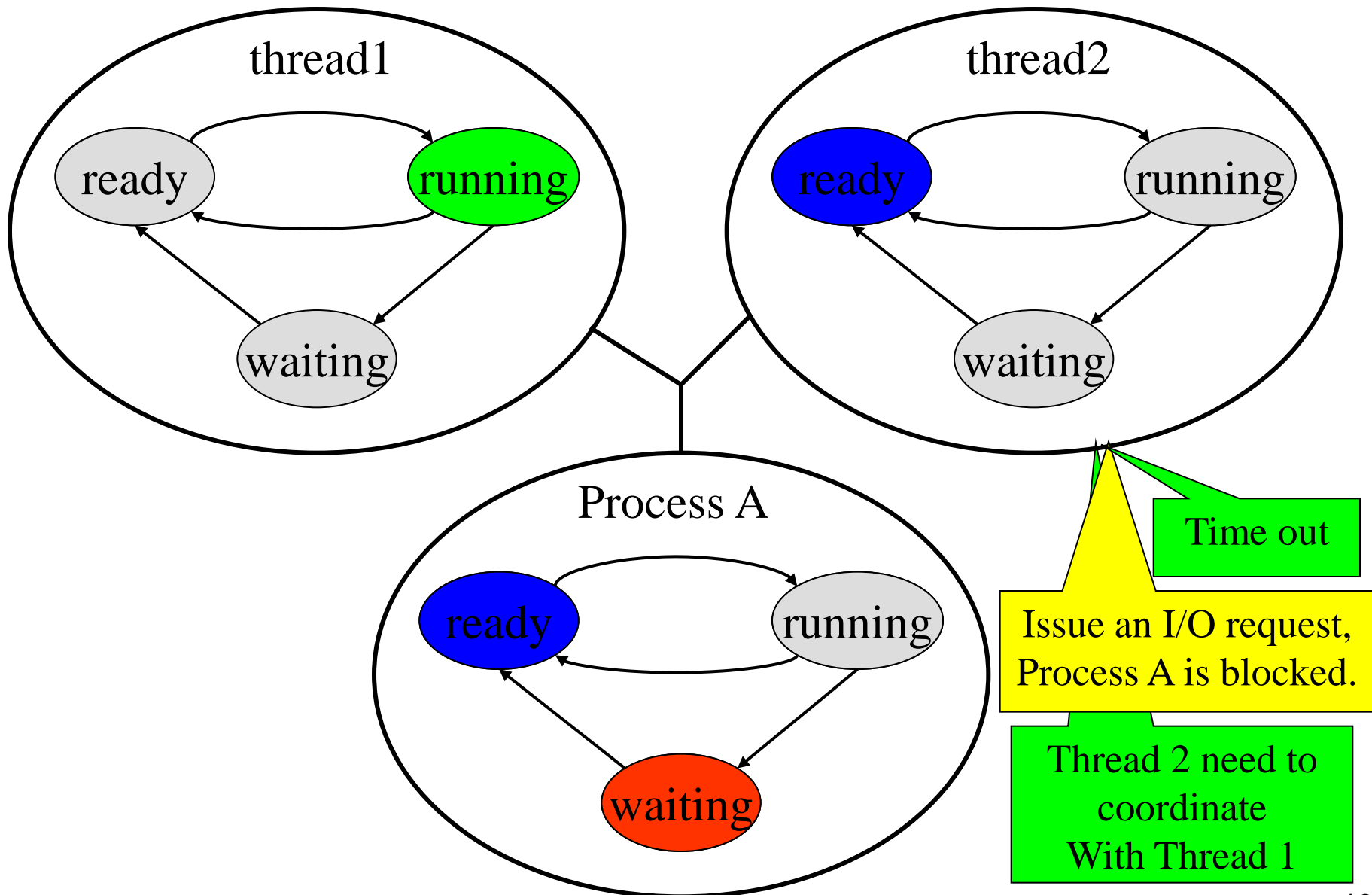
Many-to-One Model

- **Many user-level threads mapped to single kernel thread.**
- **Thread management is done in user space.**
- **The entire process will block if a thread makes a blocking system call.**
- **Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.**
- **Used on systems that do not support kernel threads.**
- **Example of system uses this model**
 - **Solaris Green Threads, a thread library available for Solaris 2**
 - **GNU Portable Threads**

Many-to-One Model



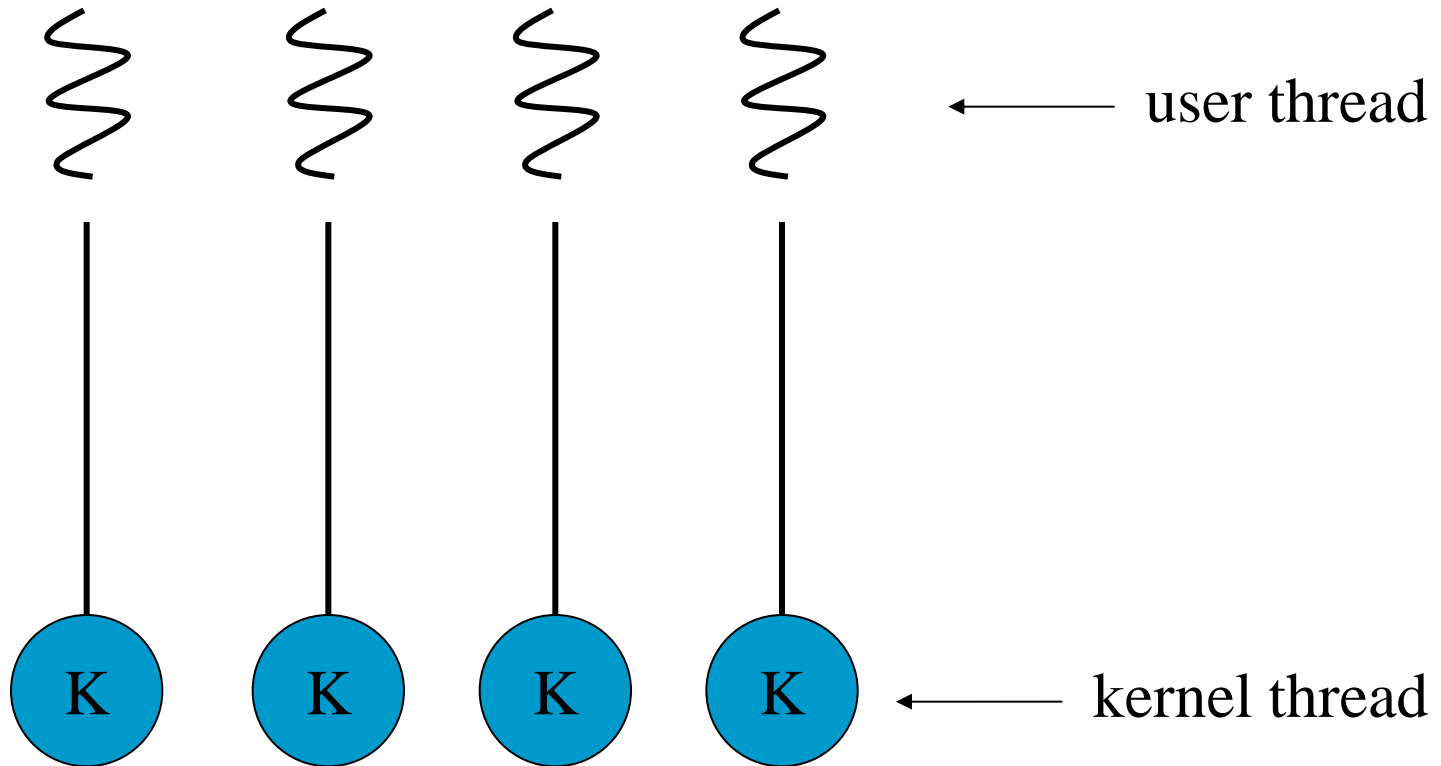
Supplement: ULT States vs. Process States



One-to-One Model

- Each user-level thread maps to a kernel thread.
- Allows another thread to run when a thread makes a blocking system call.
- Allows multiple threads to run in parallel on multiprocessors.
- Creating a user thread requires creating the corresponding kernel thread.
- Restrict the number of threads supported by the system.
- Examples of systems using this model
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

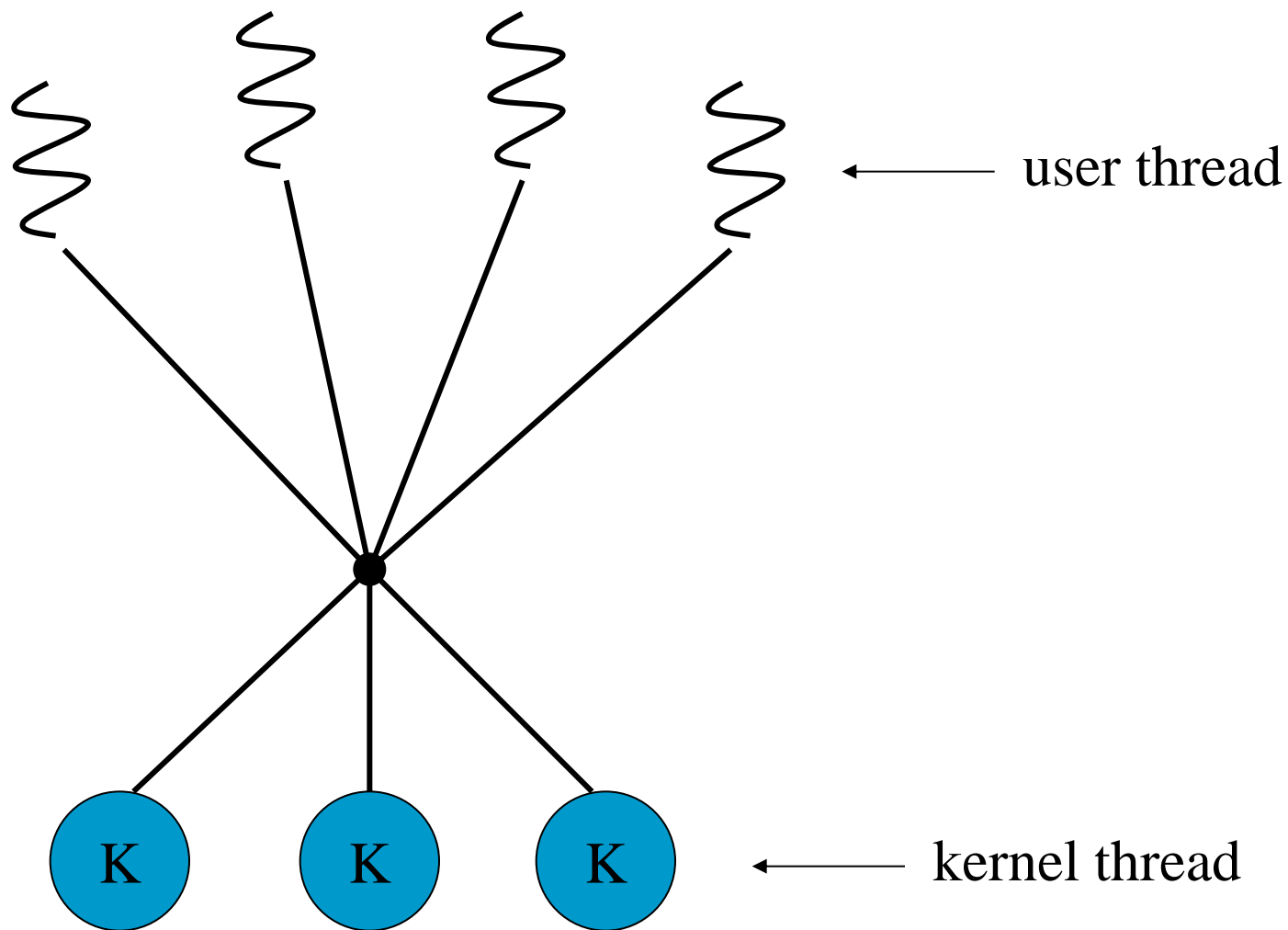
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Examples of systems supporting this model
 - Solaris prior to version 9
 - Windows NT/2000 with the *ThreadFiber* package

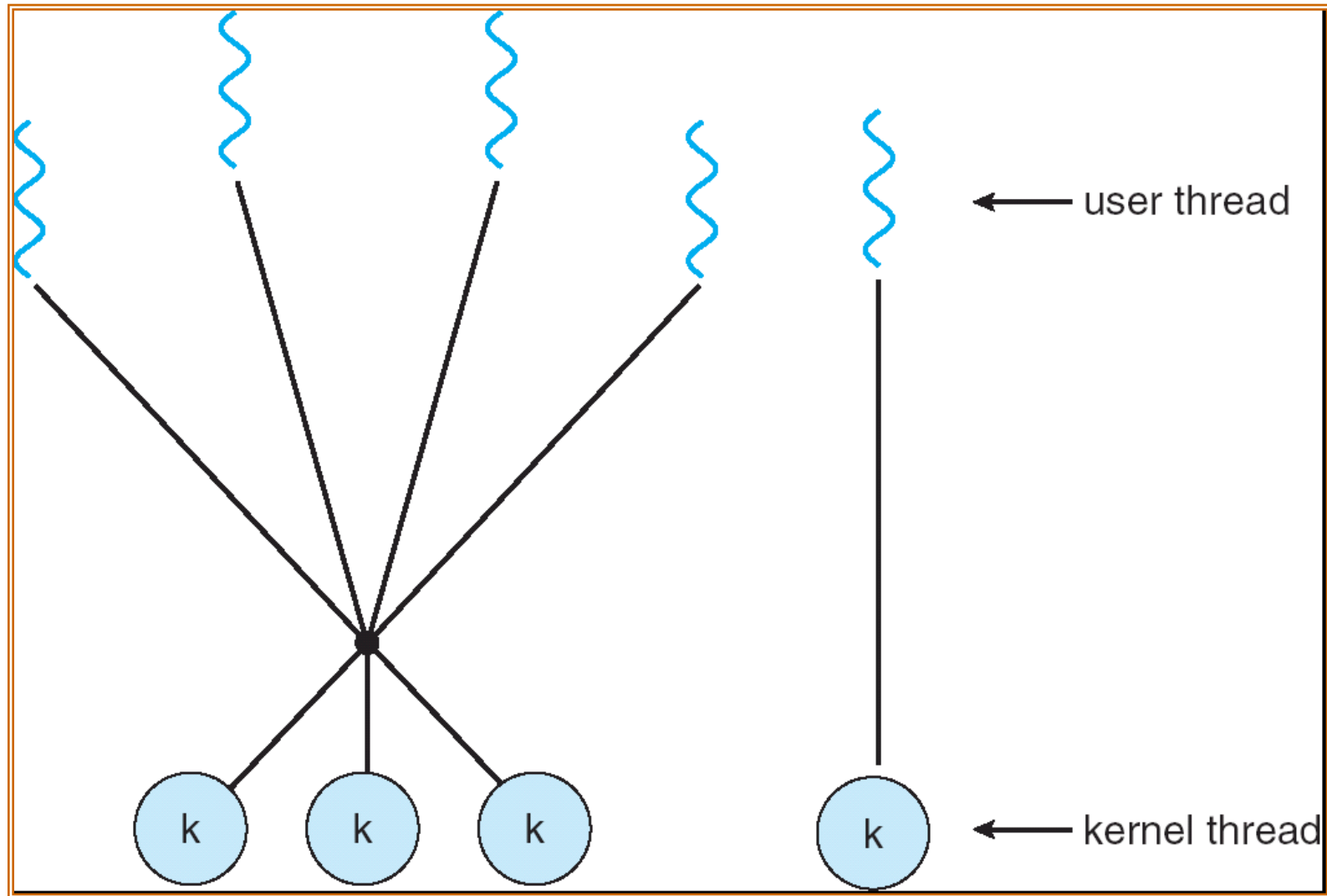
Many-to-Many Model



Two-level Model

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

Two-level Model



Comparison of these three models

- **Many-to-one model allows the developer to create as many user threads as he wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.**
- **One-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application.**
- **Many-to-many model allows the developer to create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.**

4.3 Thread Libraries

- **Provides the programmer an API for creating and managing threads**
- **Two implementing ways**
 - Provides a library entirely in user space with no kernel support. (-- local function call)
code and data structures exist in user space
 - Implement a kernel-level library supported directly by the OS. (-- system call)
code and data structures exist in kernel space
- **Main thread library**
 - POSIX Pthreads
 - Win32
 - java

POSIX Pthreads

- **POSIX:**

- Portable Operating System Interface for computer environment**

- **a POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.**
- **API specifies behavior of the thread library, implementation is up to development of the library.**
- **Common in UNIX and Linux operating systems.**
- **Referred to user-level library, because no distinct relationship exists between a thread created using the Pthread API and any associated kernel threads.**

Multithreaded C program using the Pthreads API

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

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

```
int sum; /* this data is shared by the thread(s) */
```

```
void *runner(void *param); /* the thread */
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    pthread_t tid; /* the thread identifier */
```

```
    pthread_attr_t attr; /* set of attributes for the thread */
```

```
    if (argc != 2) {
```

```
        fprintf(stderr, "usage: a.out <integer value>\n");
```

```
        return -1;
```

```
    }
```

```
    if (atoi(argv[1]) < 0) {
```

```
        fprintf(stderr, "Argument %d must be non-negative\n", atoi(argv[1]));
```

```
        return -1;
```

```
    }
```

Multithreaded C program using the Pthreads API

```
pthread_attr_init(&attr);  /* get the default attributes */
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* now 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);
}
```

Win32 threads

- **A kernel-level threads library available on Windows systems**

Multithreaded C program using the Win32 API

```
#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(PVOID Param)
```

```
{
```

```
    DWORD Upper = *(DWORD *)Param;
```

```
    DWORD i;
```

```
    for ( i = 0; i <= Upper; i++)
```

```
        Sum += i;
```

```
    return 0;
```

```
}
```

Multithreaded C program using the Win32 API

```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    // do some basic error checking
    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }

    Param = atoi(argv[1]);

    if (Param < 0) {
        fprintf(stderr, "an integer >= 0 is required \n");
        return -1;
    }
}
```

Multithreaded C program using the Win32 API

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

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

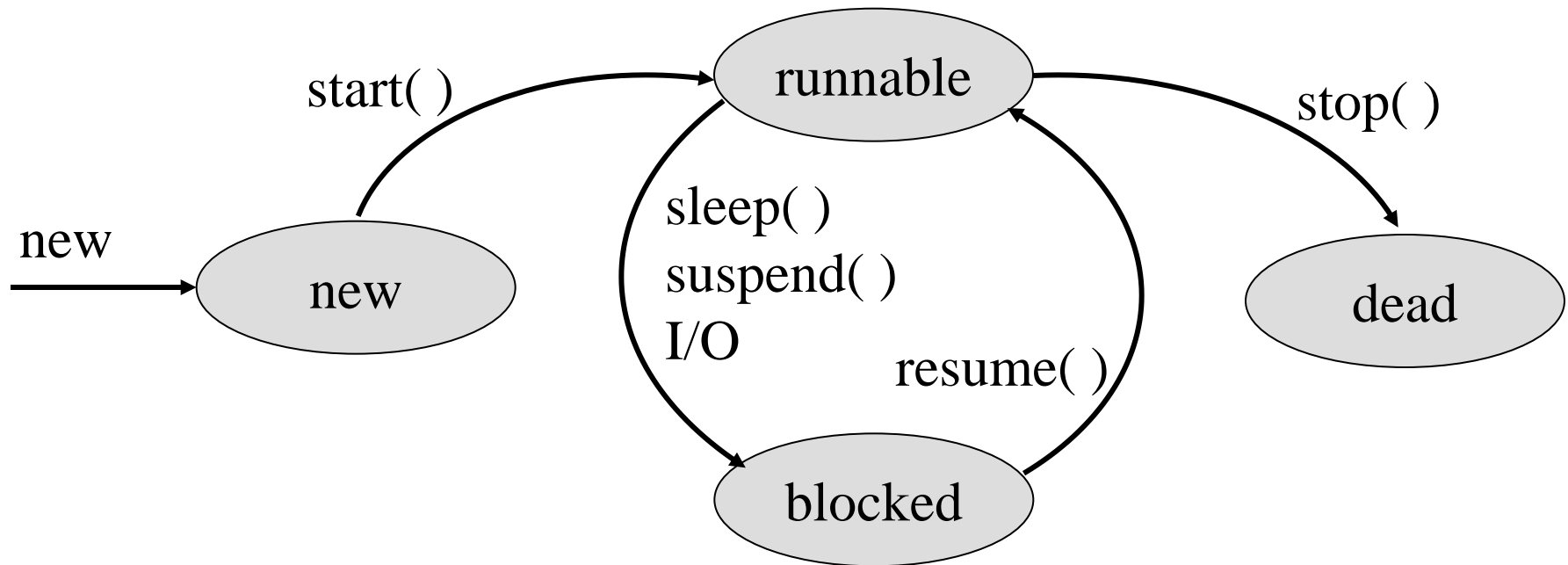
    // close the thread handle
    CloseHandle(ThreadHandle);

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


Java Threads

- Java provides support **at the language level** for the creation and management of threads.
- Java threads are managed by the JVM.
- All Java programs comprise at least a single thread of control.
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface
 - When a class implements runnable, it must define a run() method.
 - The code implementing the run() method is what runs as a separate thread.

Supplement: Java Thread States



4.4 Threading Issues

- **The fork() and exec() system calls**
- **cancellation**
- **Signal handling**
- **Thread pools**
- **Thread specific data**
- **Scheduler Activations**

The `fork()` and `exec()` system calls

- If one thread in a program calls `fork`, does the new process duplicate all threads or is the new process single-threaded?
 - Duplicates all threads
 - duplicates only the thread that invoked the `fork` system call.
- If a thread invokes the `exec` system call, the program specified in the parameter to `exec` will replace the entire process, including all threads and LWPs.
- If `exec` is called immediately after forking, duplicating only the calling thread is appropriate.
- If the separate process does not call `exec` after forking, the separate process should duplicate all threads.

Thread cancellation

- Terminates a thread before it has completed.
- Two different scenarios to cancel a target thread
 - **asynchronous cancellation**: one thread immediately terminates the target thread.
 - **Deferred cancellation**: the target thread can periodically check if it should terminate.
- The difficulty with cancellation occurs in situations:
 - resources have been allocated to a cancelled thread, or
 - if a thread was cancelled while in the middle of updating data it is sharing with other threads.
- Most operating systems allow a process or thread to be cancelled asynchronously.
- Pthread API provides deferred cancellation.

Signal handling

- A **signal** is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously, depending on the source and the reason.
- All signals follow the same pattern:
 - A signal is generated by the occurrence of a particular event.
 - A generated signal is delivered to a process.
 - Once delivered, the signal must be handled.
- Every signal may be handled by one of two possible handlers
 - a user-defined signal handler
 - a default signal handler, which is run by the kernel

Signal handling

- In multithreaded programs, a signal should be delivered
 - to the thread to which the signal applies
 - to every thread in the process
 - to certain threads in the process
 - to a specific thread which is assigned to receive all signals for the process.
- The method delivering a signal depends on signal type
 - **Synchronous signals** need to be delivered to the thread that generated the signal and not to other threads in the process
 - **Some asynchronous signals** should be sent to all threads
 - **Some asynchronous signals** may be delivered to only those threads that are not blocking the signal.
(Some multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.)

Thread pools

■ Idea:

- Create a number of threads at process startup and place them into a pool, where they sit and wait for work.
- When a server receives a request, it awakens a thread from this pool, passing it the request to service.
- Once the thread completes its service, it returns to the pool awaiting more work.
- If the pool contains no available thread, the server waits until one becomes free.

■ Benefits of thread pool

- usually faster to service a request with an existing thread than waiting to create a thread
- Allows the number of threads in the application to be bound to the size of the pool

Thread specific data

- **Allows each thread to have its own copy of data**
- **Useful when you do not have control over the thread creation process (i.e., when using a thread pool)**
- **example**
 - **In a transaction-processing system, we might service each transaction in a separate thread. Each transaction may be assigned a unique identifier. To associate each thread with its unique identifier we could use thread-specific data.**

Scheduler Activations

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

4.5 Operating-system examples

- **Windows XP Threads**
- **Linux Threads**
- **Solaris 2 Threads**

Windows XP Threads

- Implements the one-to-one mapping.
- Each thread contains
 - a **thread id**, identifying the thread
 - **register set**, representing the status of the processor
 - **separate user and kernel stacks**, used when the thread is running in user mode or kernel mode
 - **private data storage area**, used by various run-time libraries and dynamic link libraries.
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread
 - ETHREAD(executive thread block)
 - KTHREAD(kernel thread block)
 - TEB(thread environment block)

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)
- When `clone()` is invoked, it passed a set of flags, which determine how much sharing is to take place between the parent and child tasks
 - `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
- If none of these flags are set, no sharing, similar to `fork()`



OS example?

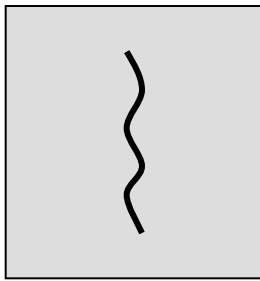
Homework (page 146)

4.2 thinking: 4.1 4.3

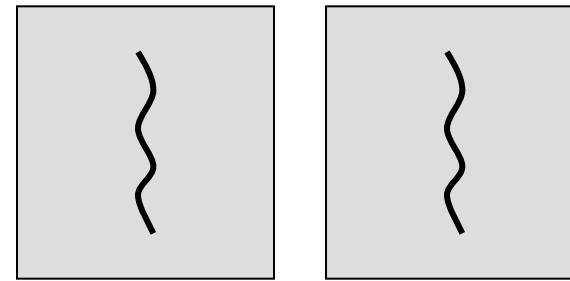
- 1. What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?**
- 2. Describe the actions taken by a kernel to context switch between kernel level threads.**
- 3. What resources are used when a thread is created? How do they differ from those used when a process is created?**

Supplement: Multithreading

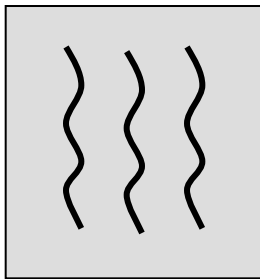
- **Operating system supports multiple threads of execution within a single process**
- **MS-DOS supports a single thread**
- **UNIX supports multiple user processes but only supports one thread per process**
- **Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads**



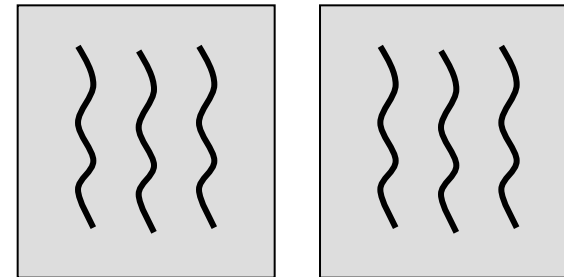
one process
one thread
MS DOS



multiple processes
one thread per process
UNIX



one process
multiple threads
JAVA



multiple processes
multiple threads per process
W2K, Linux, Solaris

Supplement: Process and Thread

Process:

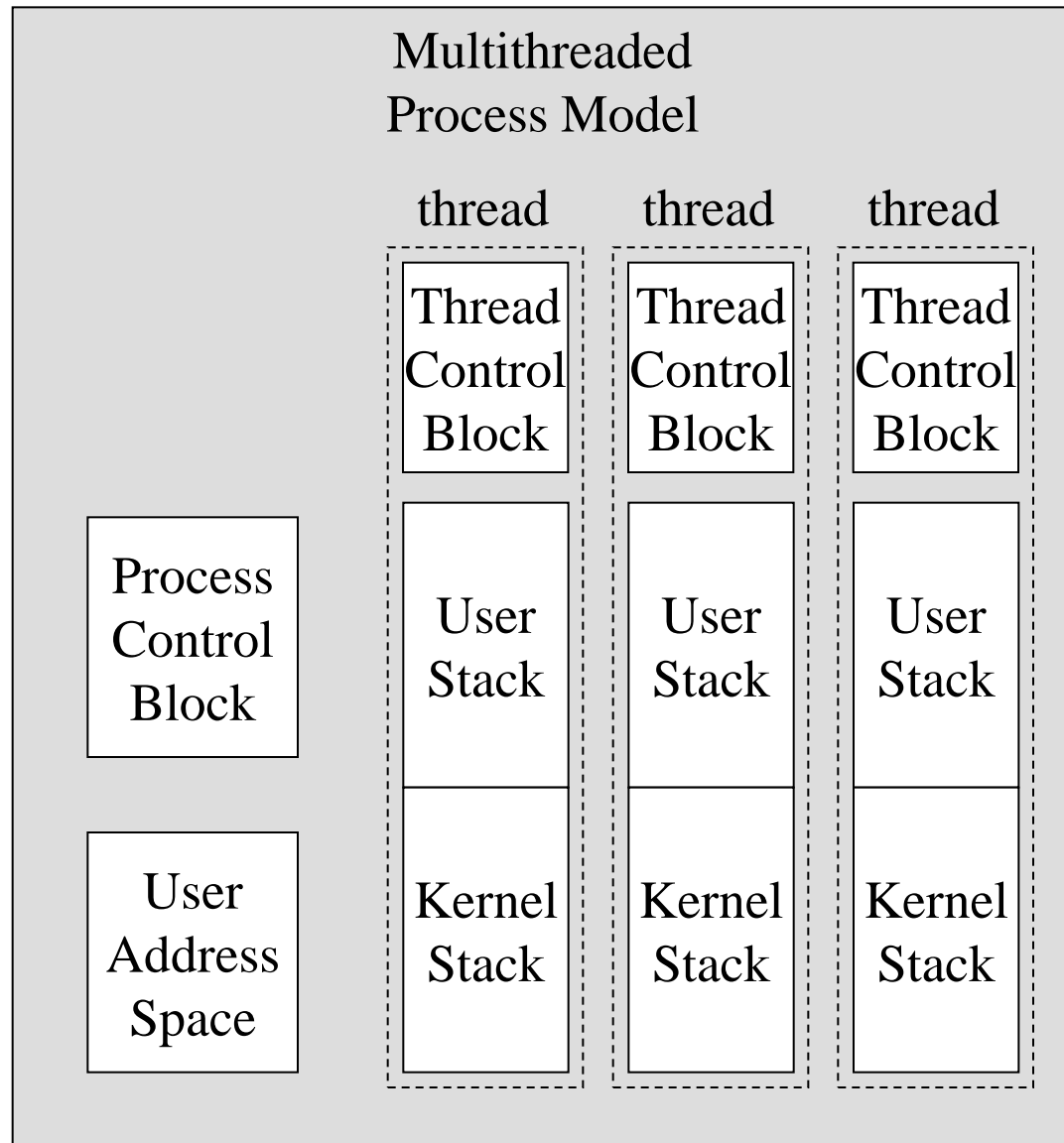
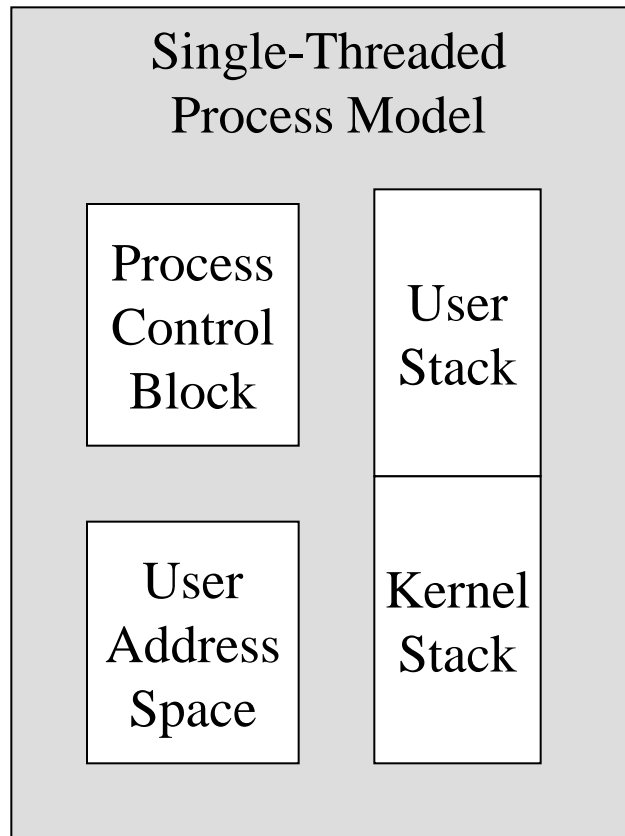
- Have a virtual address space which holds the process image
- Protected access to processors, other processes, files, and I/O resources

Thread:

- An execution state (running, ready, waiting, etc.)
- Saved thread context when not running
- Has an execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process
 - all threads of a process share these resources

Supplement:

Single Threaded and Multithreaded Process Model



Supplement: Benefits of Threads

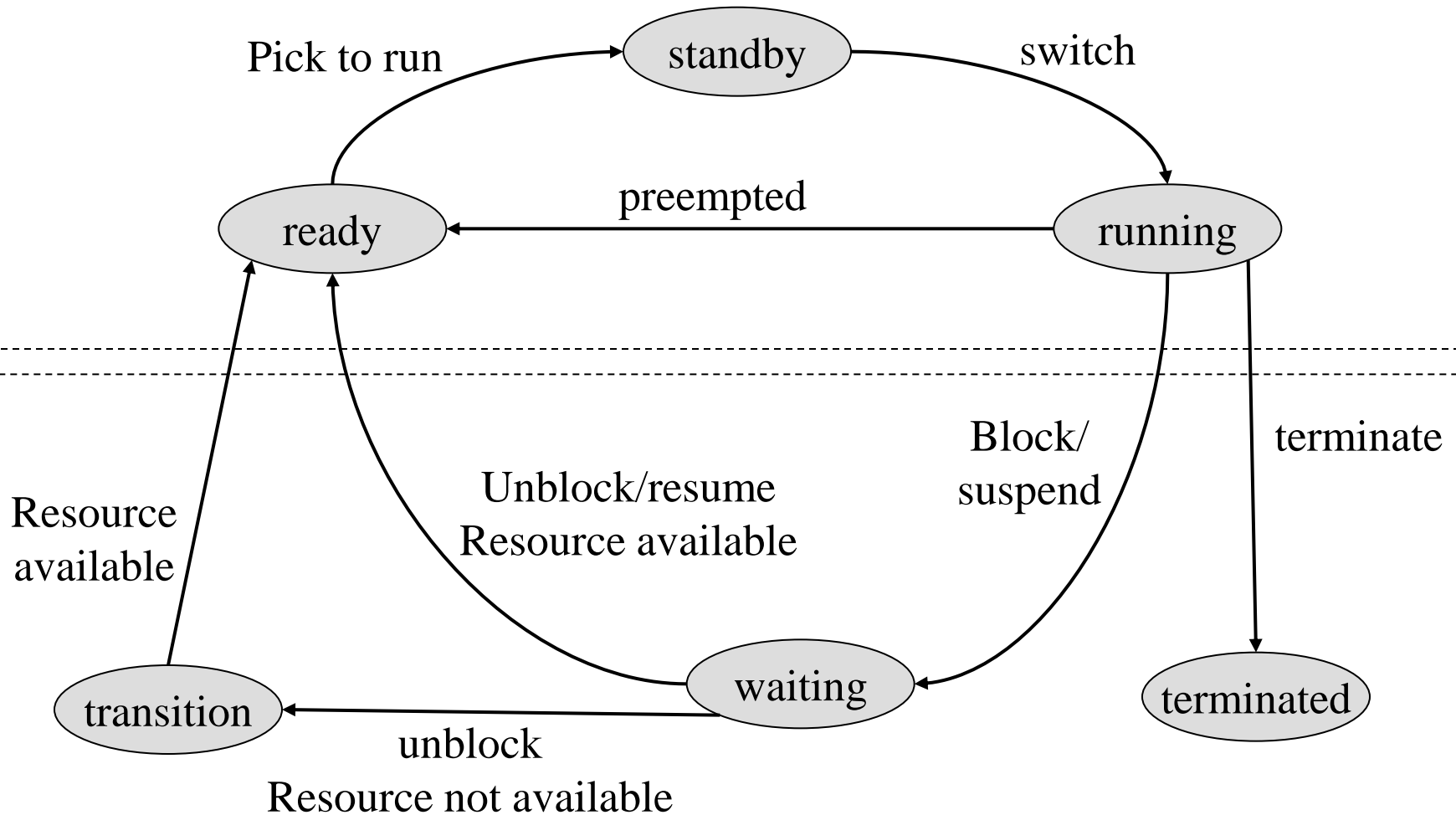
- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

Supplement: Windows XP Thread States

- **Ready**
- **Standby**
- **Running**
- **Waiting**
- **Transition**
- **Terminated**

Supplement: Windows XP Thread States

runnable

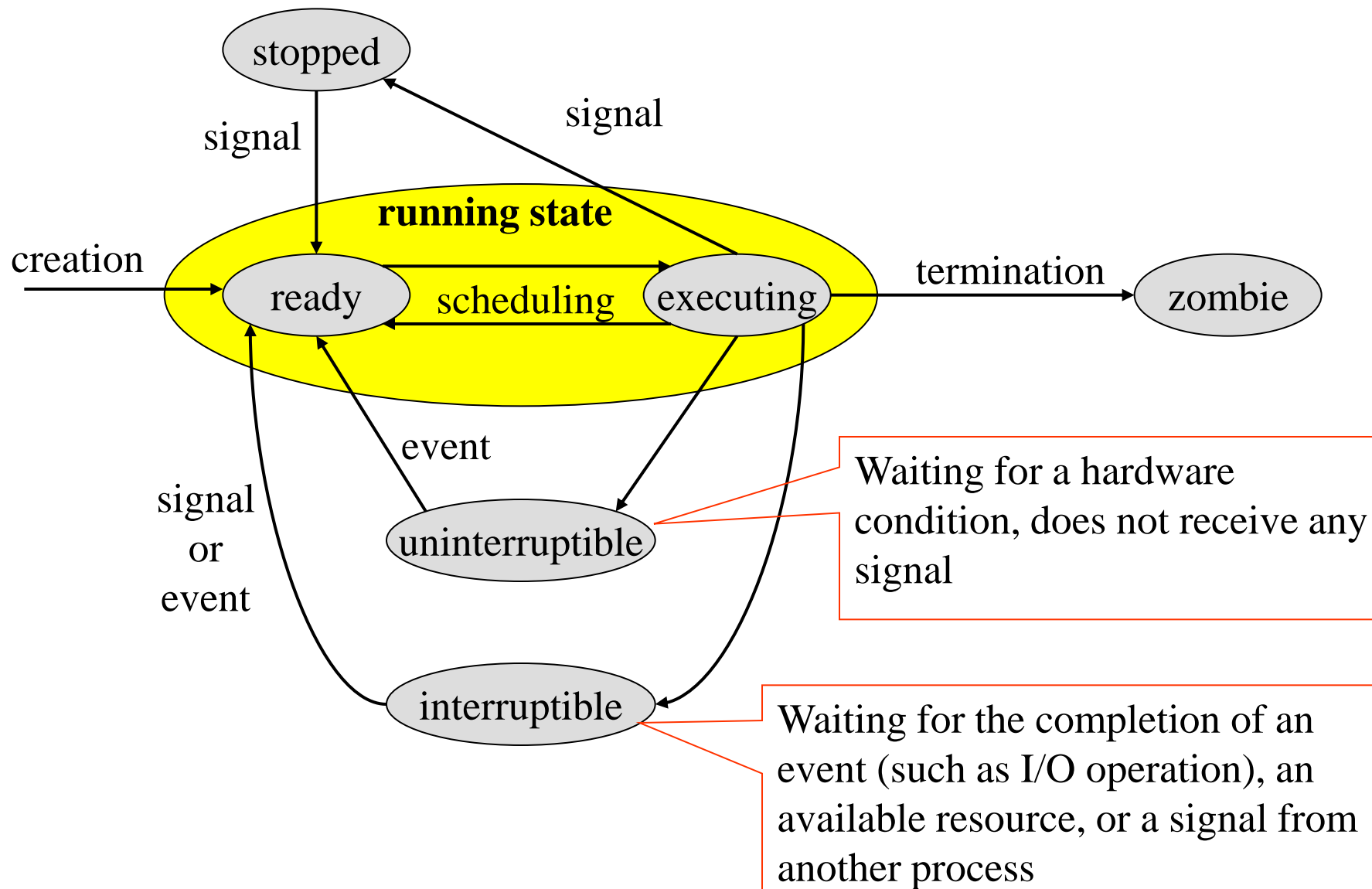


Not runnable

Supplement: Linux States of a Process

- Running
- Interruptable
- Uninterruptable
- Stopped
- Zombie (僵死)

Supplement: Linux Process / Thread Model



Supplement: Solaris 2 Threads

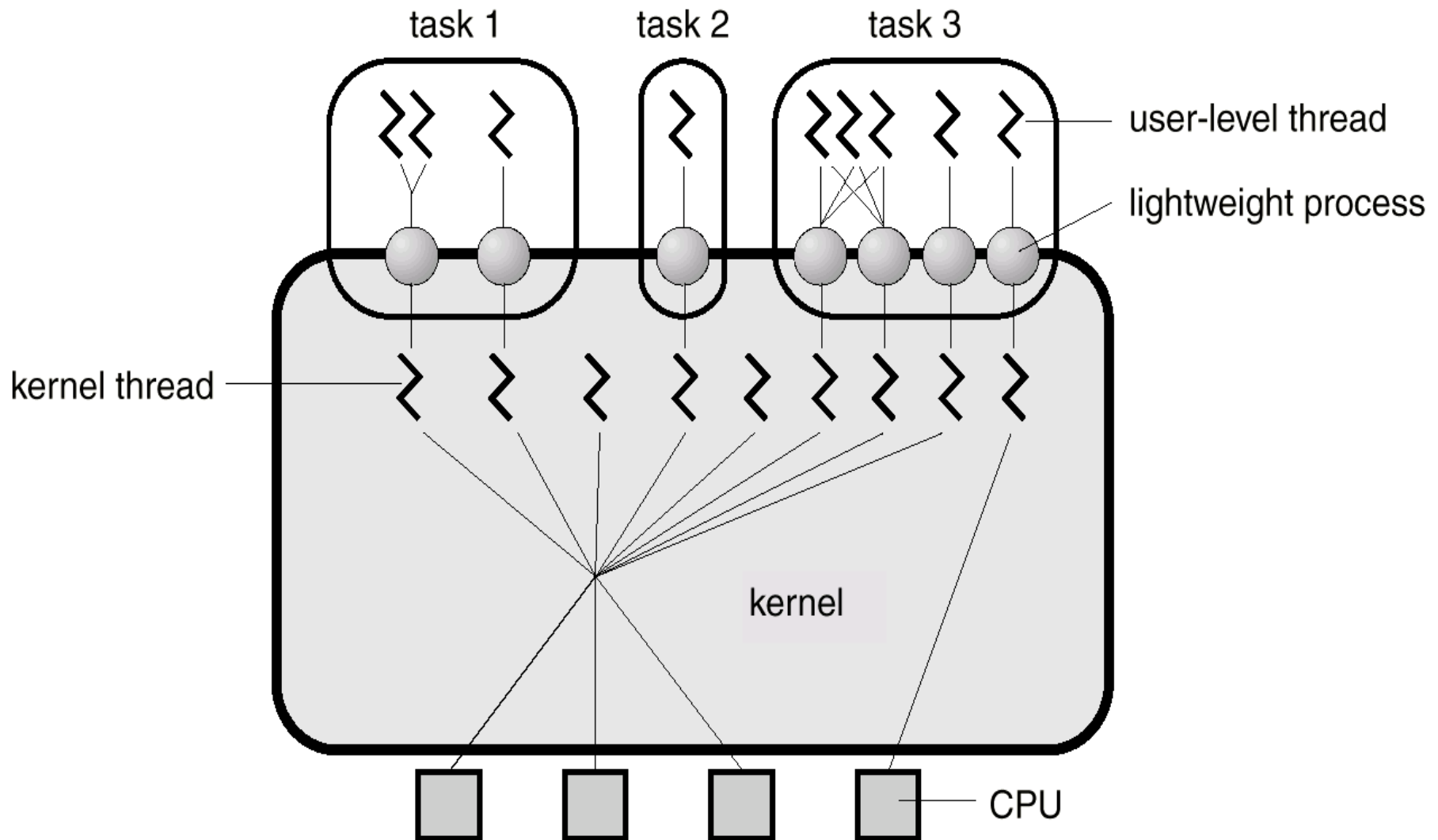
- Solaris 2 is a version of UNIX supporting for threads at kernel and user level, symmetric multiprocessing(SMP), and real-time scheduling.
- Solaris 2 supports user-level threads with a library containing APIs for thread creation and management. Known as UI threads.
- Solaris 2 defines an intermediate level of thread. Between user-level and kernel-level threads are **lightweight processes (LWPs)**. Each process contains at least one LWP.
- Standard kernel-level threads execute all operations within the kernel.

Supplement: Solaris threads

- **Process, including the user's address space, stack, and process control block**
- **User-level threads**
- **Lightweight processes**
- **Kernel threads**

Supplement: Solaris 2 Threads

■ Many-to-many model



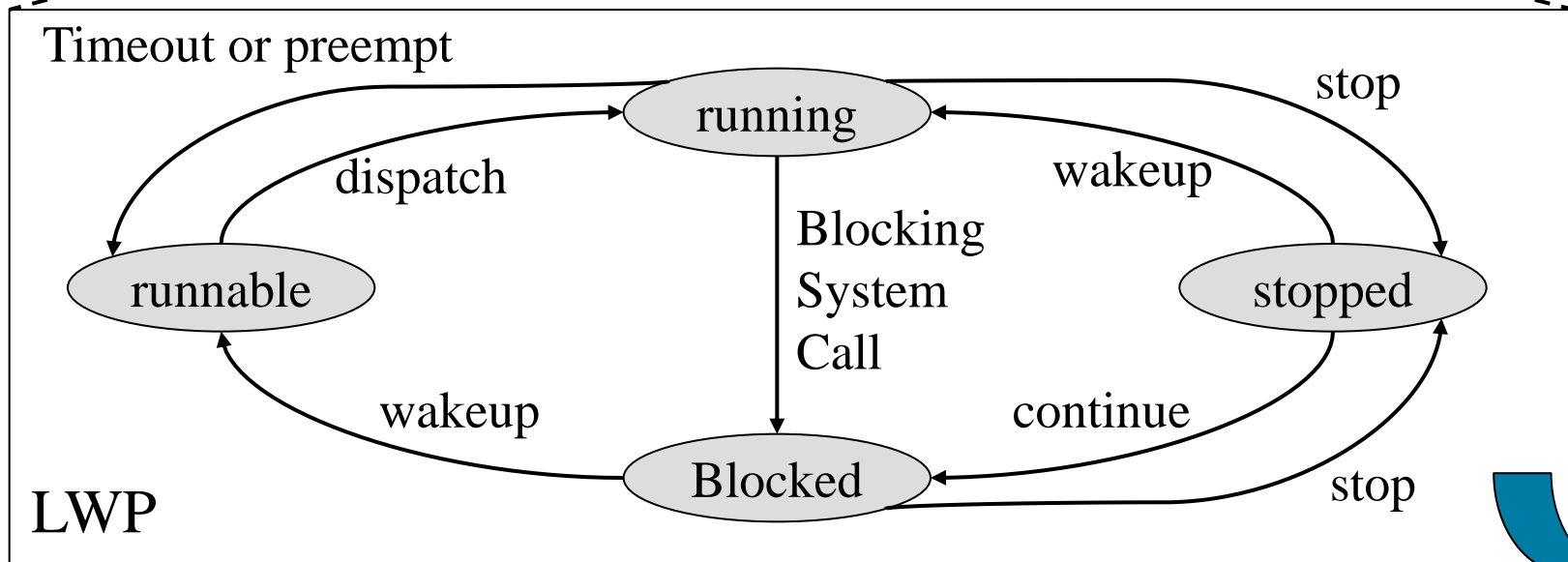
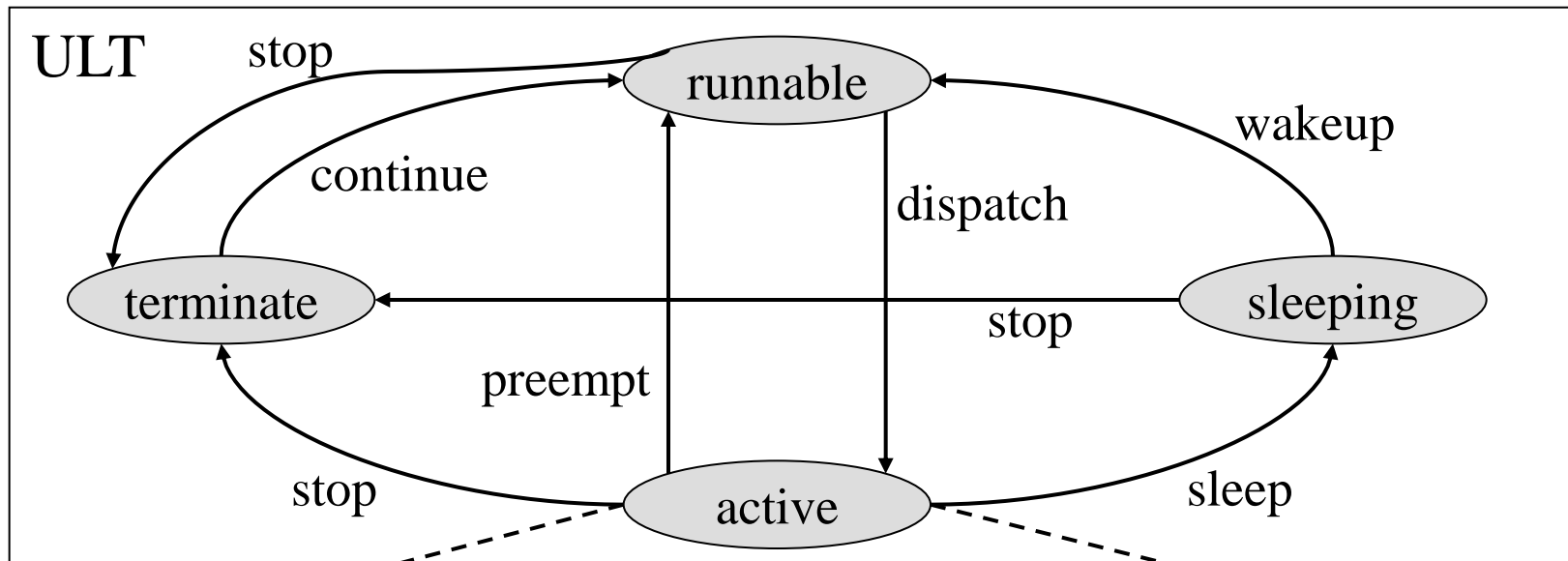
Supplement: Solaris 2 Threads

- **User-level threads may be either bound or unbound.**
 - A bound ULT is permanently attached to an LWP
 - An unbound thread is not permanently attached to any LWP. (default)
- **data structures used**
 - A ULT contains a thread ID, register set (including a program counter and stack pointer), stack, and priority.
 - A LWP has a register set for the user-level thread it is running, as well as memory and accounting information.
 - A kernel thread has only a small data structure and a stack. The data structure includes a copy of the kernel registers, a pointer to the LWP to which it is attached, and a priority and scheduling information.

Supplement: Solaris Thread Execution

- **Synchronization**
- **Suspension**
- **Preemption**
- **Yielding**

Supplement: Solaris ULT and LWP States



Chapter 5 CPU Scheduling



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

Scheduling Criteria

Scheduling Algorithms

Chapter Objectives

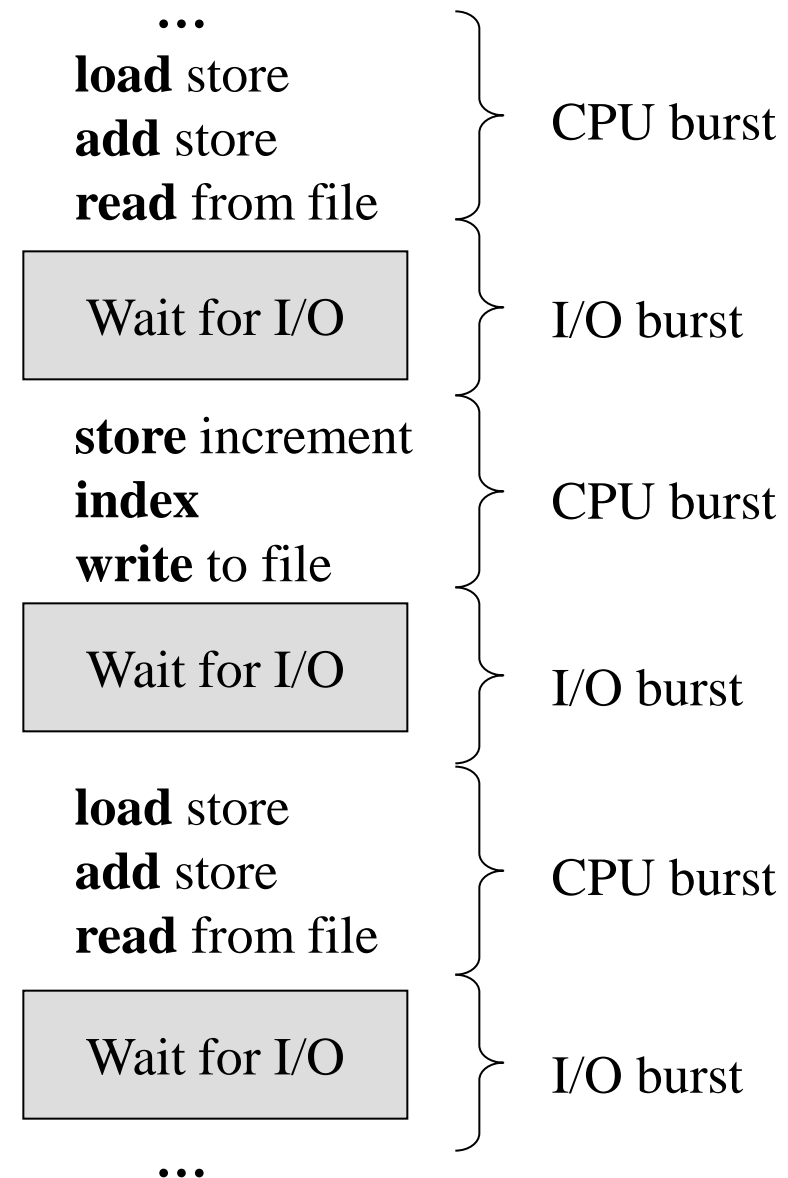
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

Contents

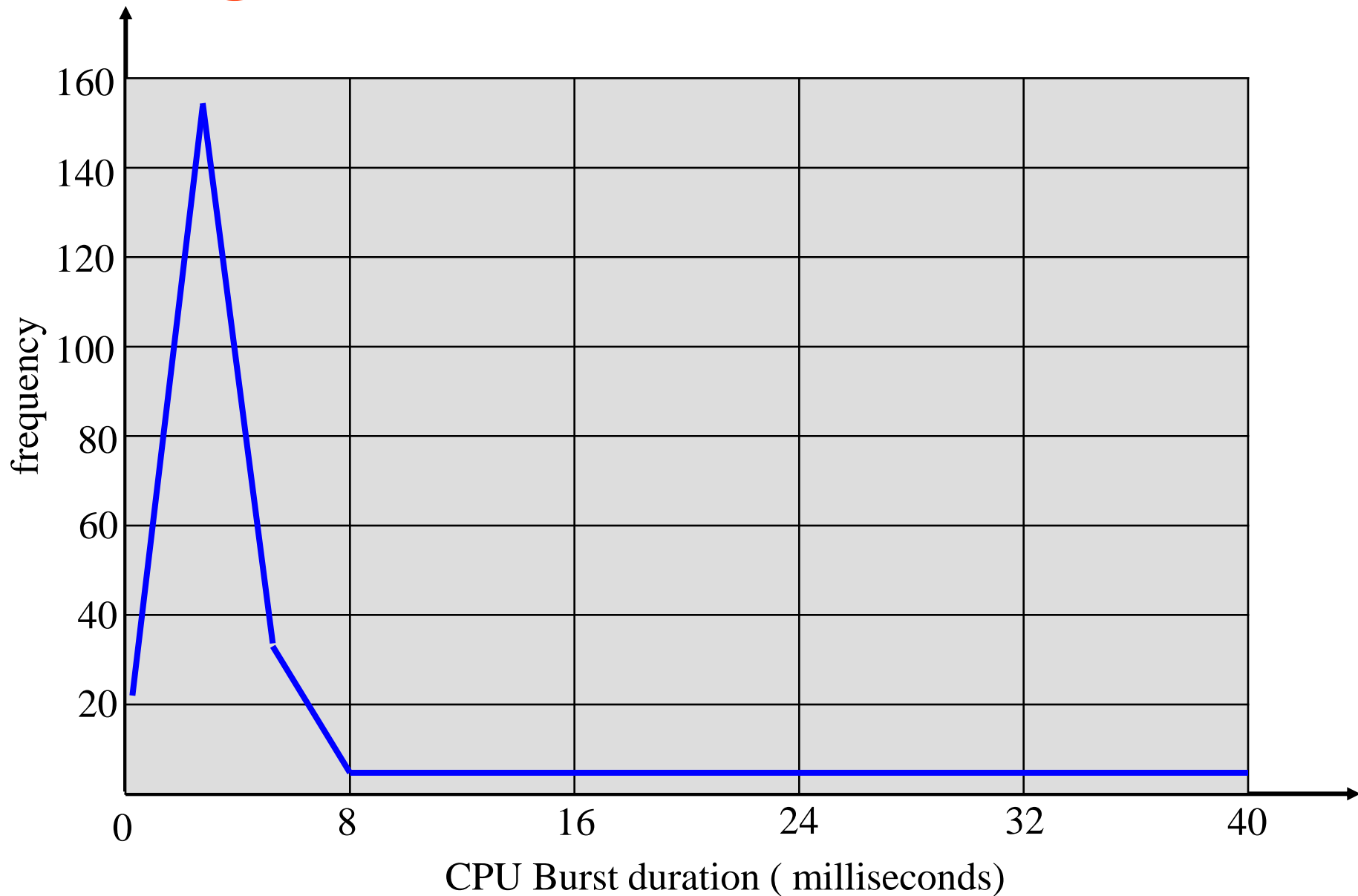
- 5.1 Basic Concepts**
- 5.2 Scheduling Criteria**
- 5.3 Scheduling Algorithms**
- 5.4 Multiple-Processor Scheduling (*)**
- 5.5 Thread scheduling (*)**
- 5.6 Operating Systems Examples (*)**
- 5.7 Algorithm Evaluation (*)**

5.1 Basic Concepts

- The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization
- CPU-I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution

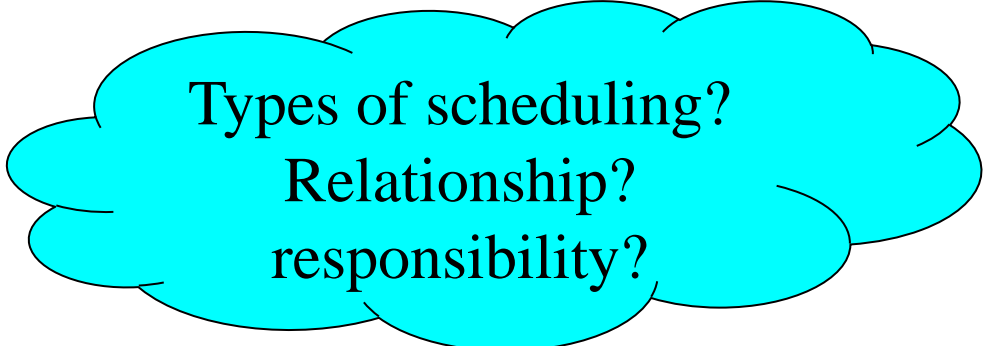


Histogram of CPU-burst Times



Supplement: Aim of Scheduling

- **Response time**
- **Waiting time**
- **Turnaround time**
- **Throughput**
- **Processor efficiency/utilization**



Types of scheduling?
Relationship?
responsibility?

CPU Scheduler

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

Decision Mode

■ Nonpreemptive

- Once a process is in the running state, it will continue until it terminates or blocks itself for I/O

■ Preemptive

- Currently running process may be interrupted and moved to the Ready state by the operating system
- Allows for better service since any one process cannot monopolize(独占) the processor for very long

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.**

5.2 Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput**(吞吐量) – the number 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)

Supplement:

Classification of Scheduling Criteria

■ User-oriented

– Response Time

- Elapsed time between the submission of a request until there is output.

■ System-oriented

– Effective and efficient utilization of the processor

■ Performance-related

– Quantitative(定量的)

– Measurable such as response time and throughput

■ Not performance related

– Qualitative(定性的)

– Predictability

Supplement:

Classification of Scheduling Criteria

	Performance related	not performance related
User oriented	Turnaround time Response time	Predictability Deadline
System oriented	Throughput CPU utilization	Fairness Forced priority Balance resources

Optimization Criteria

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

5.3 Scheduling Algorithms

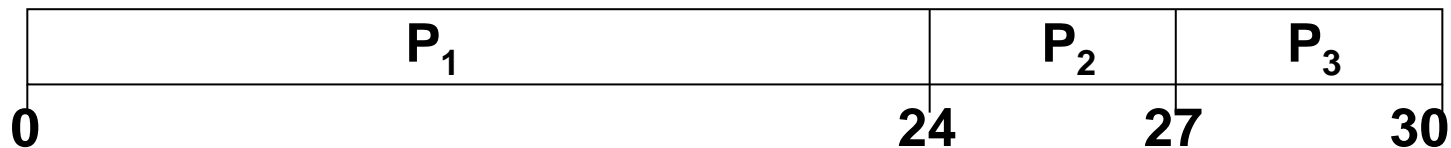
- **First-Come, First-Served (FCFS) Scheduling**
- **Shortest-Job-First (SJR) Scheduling**
- **Priority Scheduling**
- **Round Robin (RR) Scheduling**
- **Multilevel Queue Scheduling**
- **Multilevel Feedback Queue Scheduling**
- **Highest Response-Ratio Next (HRRN) scheduling**

5.3.1 First-Come First-Served (FCFS) Scheduling

- Each process joins the Ready queue
- When the current process ceases to execute, the oldest process in the Ready queue is selected
- the FCFS scheduling algorithm is nonpreemptive.

<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:



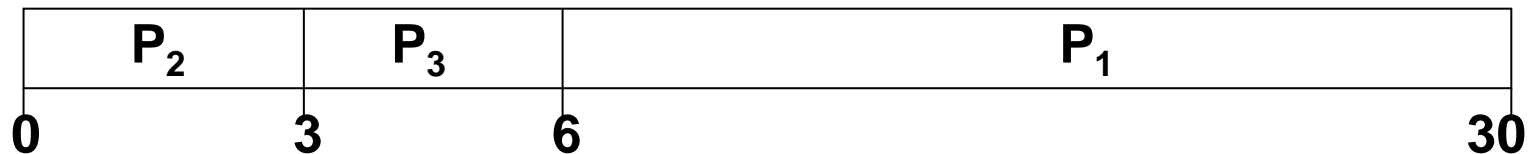
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

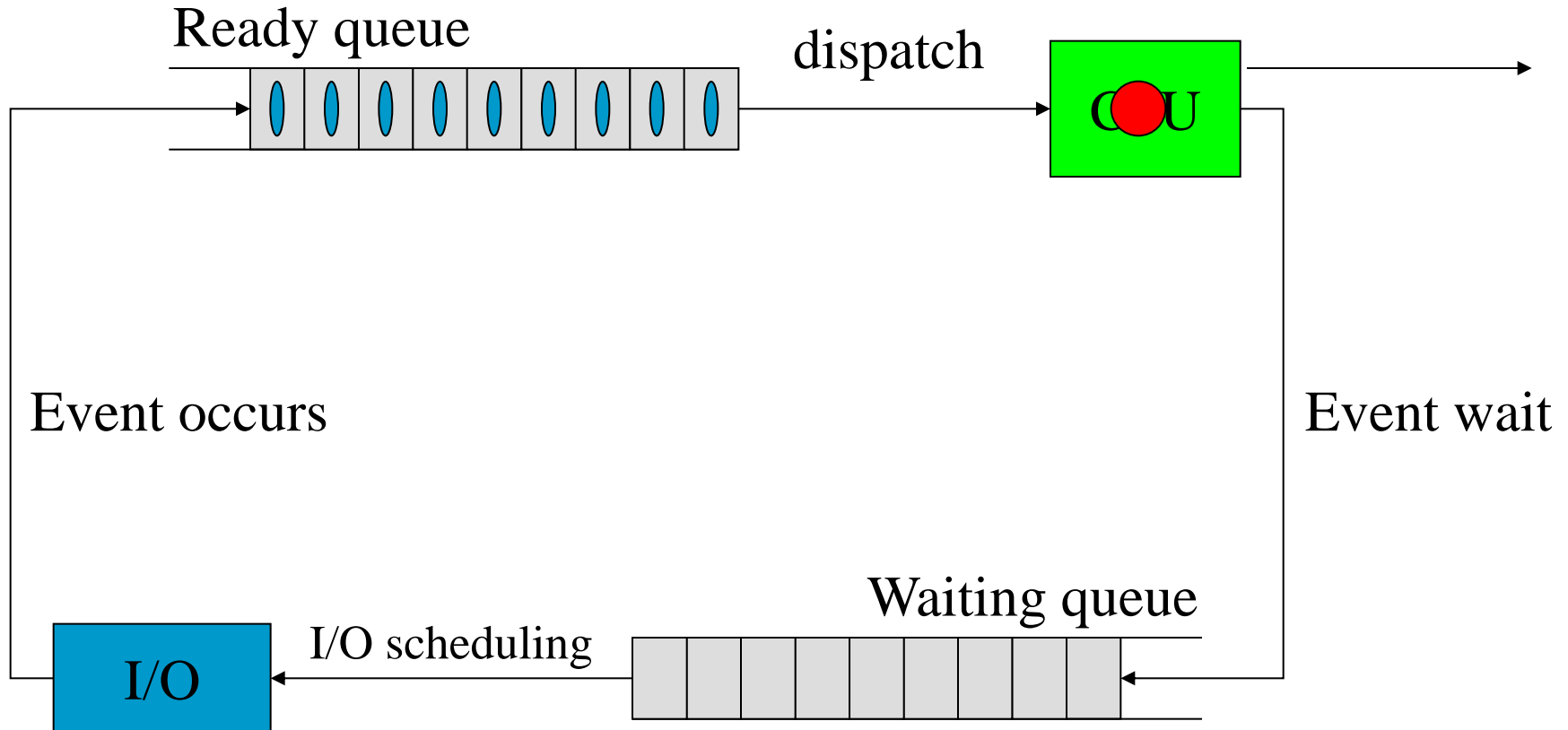
$$P_2, P_3, P_1.$$

- 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.
- A short process may have to wait a very long time before it can execute
- Favors CPU-bound processes
 - I/O processes have to wait until CPU-bound process completes
- *Convoy(护航) effect*: short process behind long process

Convoy(护航) effect



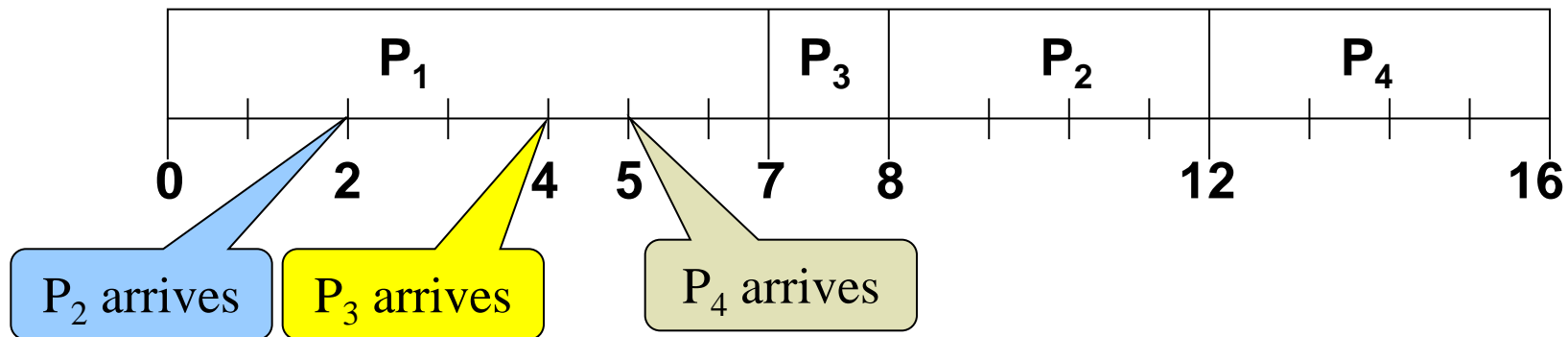
5.3.2 Shortest-Job-First (SJF) Scheduling

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

Example of Non-Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

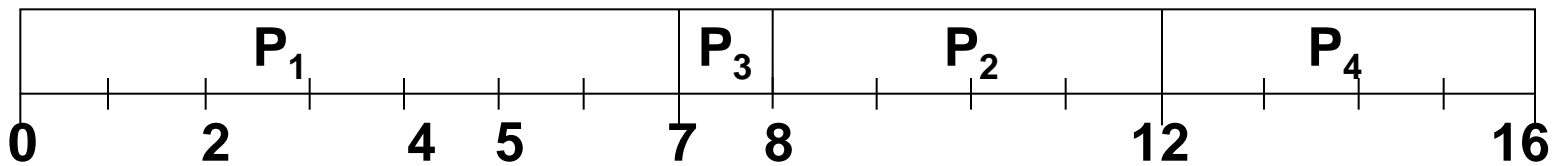
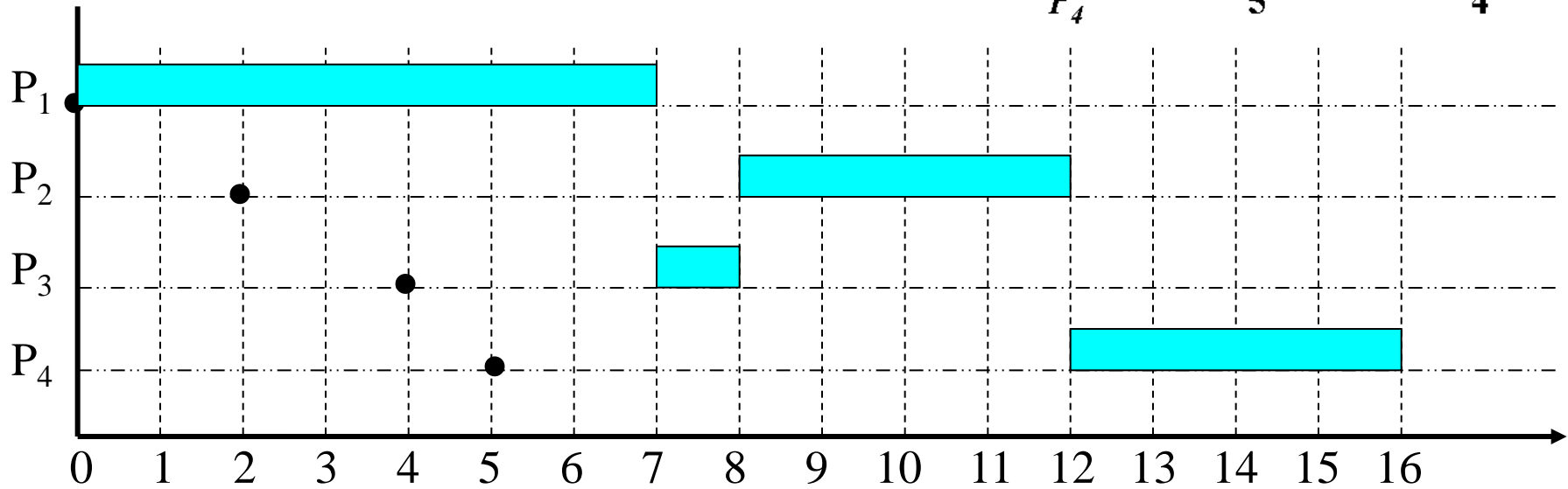
■ SJF (non-preemptive)



- Waiting time for $P_1=0$; $P_2=6$; $P_3=3$; $P_4=7$
- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

SJF

P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

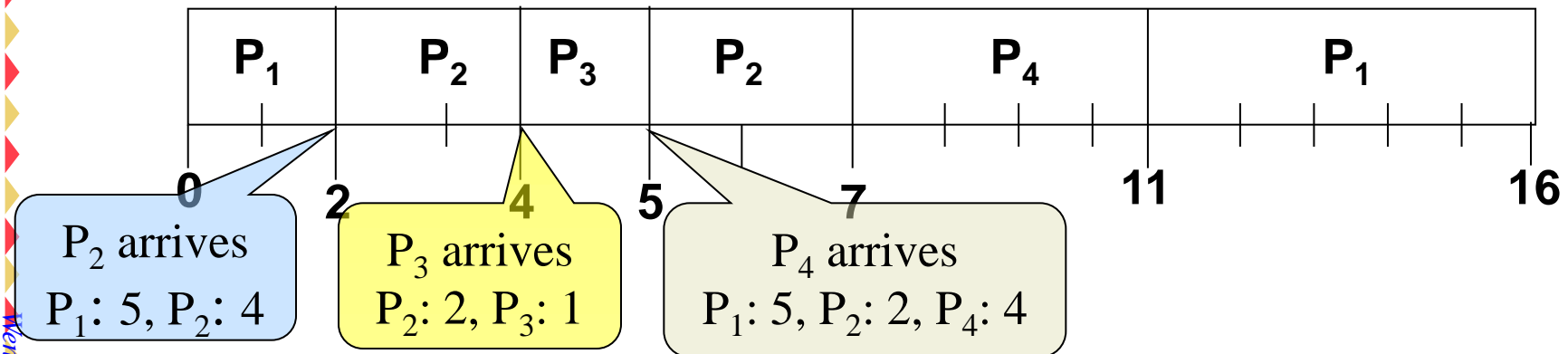


- Waiting time for $P_1=0$; $P_2=6$; $P_3=3$; $P_4=7$
- turnaround time $P_1=7$; $P_2=10$; $P_3=4$; $P_4=11$

Example of Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

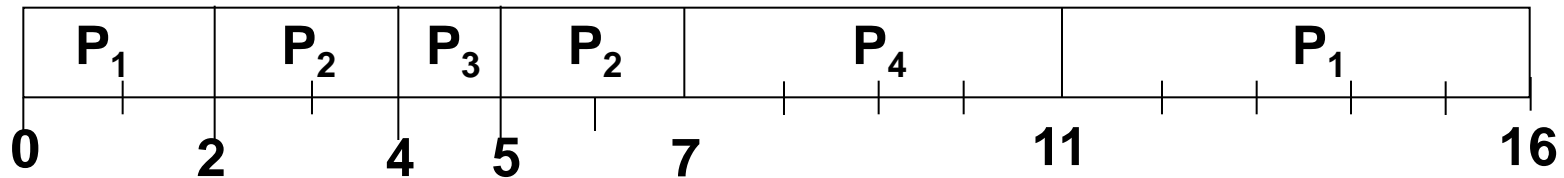
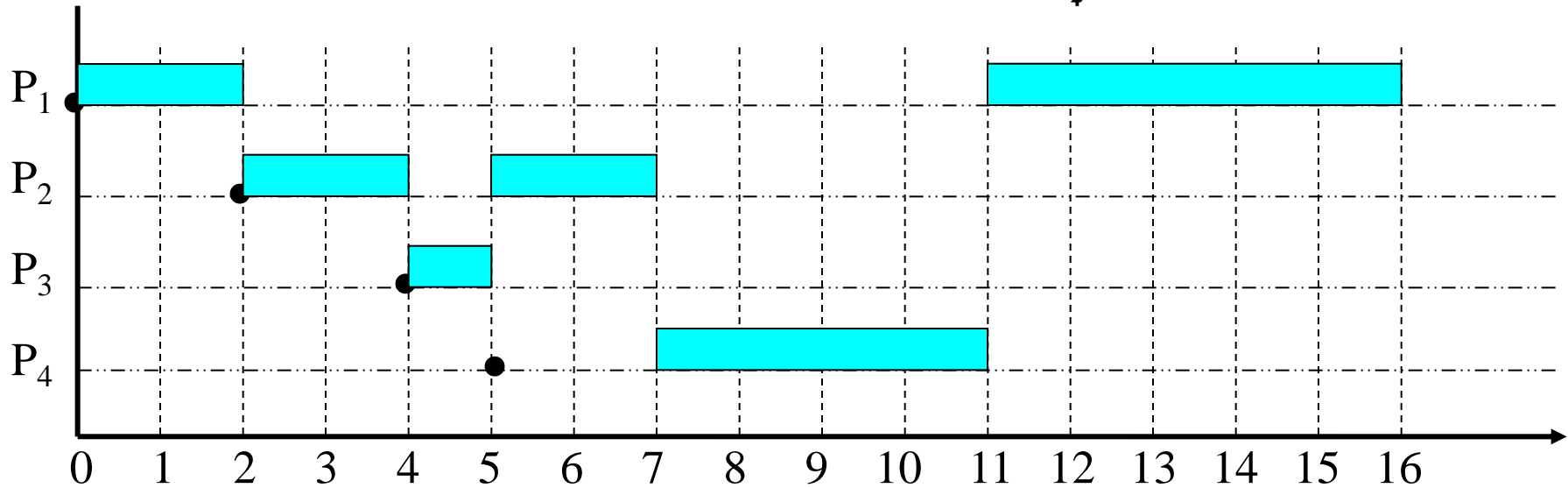
■ SRTF (preemptive SJF)



- Wait time for $P_1=9$; $P_2=1$; $P_3=0$; $P_4=2$
- Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

SRTF

P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Waiting time for $P_1=9$; $P_2=1$; $P_3=0$; $P_4=2$
- turnaround time $P_1=16$; $P_2=5$; $P_3=1$; $P_4=6$

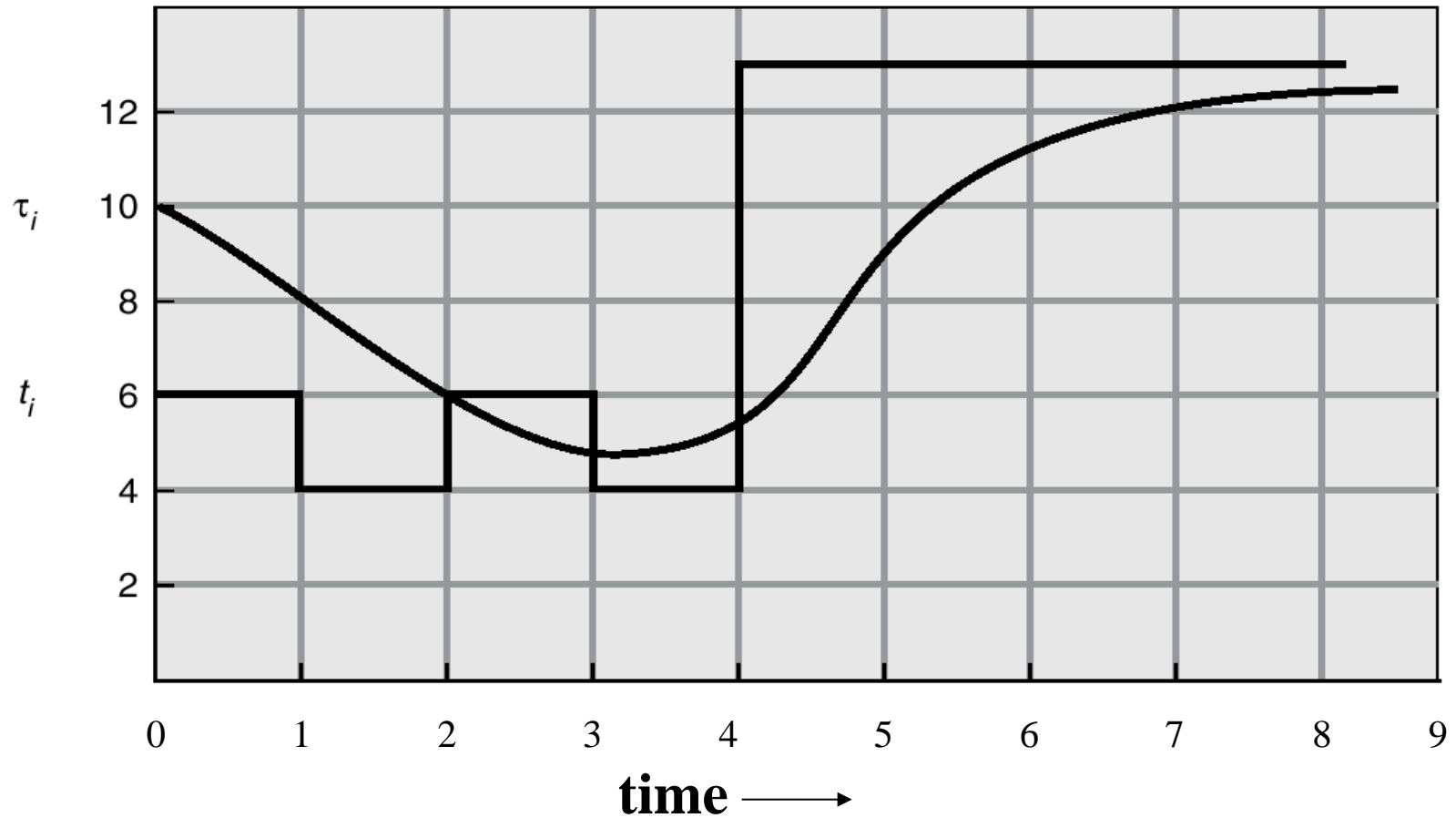
Prediction of the Length of the Next CPU Burst (*)

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.
 - t_n = actual length of n th CPU burst
 - τ_{n+1} = predicted value for the next CPU burst
 - α . $0 \leq \alpha \leq 1$
 - define:

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

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

Prediction of the Length of the Next CPU Burst (*)



$$\tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \quad \alpha = 1/2 \quad \tau_0 = 10$$

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

Examples of Exponential Averaging

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

■ $\alpha = 0$

– $\tau_{n+1} = \tau_n$ Recent history does not count.

■ $\alpha = 1$

– $\tau_{n+1} = t_n$ Only the actual last CPU burst counts.

■ If we expand the formula, we get:

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

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

5.3.3 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).
- Equal-priority processes are scheduled in FCFS order.
- Two schemes:
 - **Preemptive**: preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
 - **Nonpreemptive**: simply put the new process at the head of the ready queue.
- Priority can be defined either internally or externally.

Priority Scheduling(Cont.)

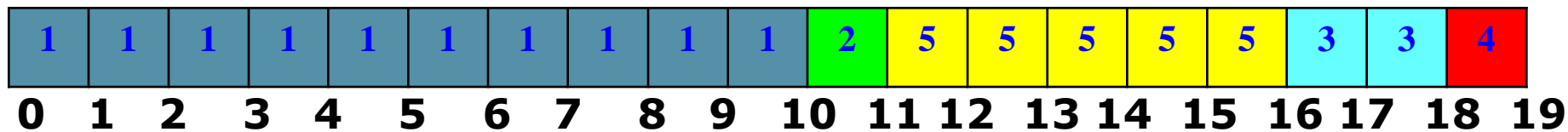
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem \equiv Starvation (or **indefinite blocking**)– low priority processes may never execute.
- Solution \equiv **Aging** – as time progresses increase the priority of the process.
- For example
 - If priorities range from 127(low) to 0(high)
 - We could decrease the priority number of a waiting process by 1 every 15 minutes.
 - Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

举例——静态优先级、非剥夺式调度

- 某系统采用非剥夺式优先级调度算法，5个进程P₁、P₂、P₃、P₄、P₅依次进入就绪队列，其需要的CPU时间、到达时刻、及优先数如右：

进程	CPU 时间	到达 时刻	优先数
P1	10	0	3
P2	1	2	1
P3	2	3	3
P4	1	5	4
p5	5	7	2

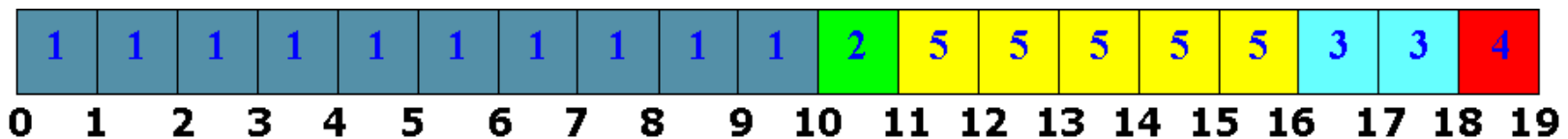
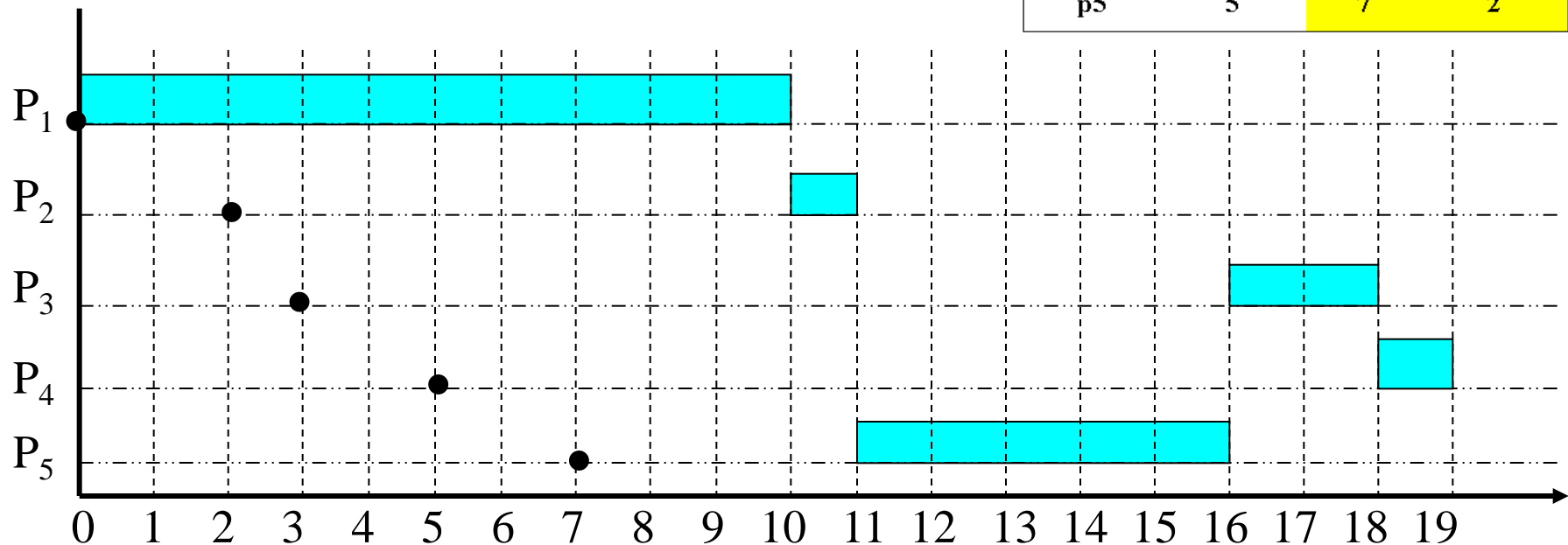
- 进程调度的甘特图如下：



- Waiting time(等待时间):
 - T1=0、T2=8、T3=13、T4=13、T5=4; average: TA=7.6
- Turnaround time(周转时间):
 - T1=10、T2=9、T3=15、T4=14、T5=9; average: TA=11.4

Nonpreemptive priority

进程	CPU 时间	到达 时刻	优先 级
P1	10	0	3
P2	1	2	1
P3	2	3	3
P4	1	5	4
P5	5	7	2



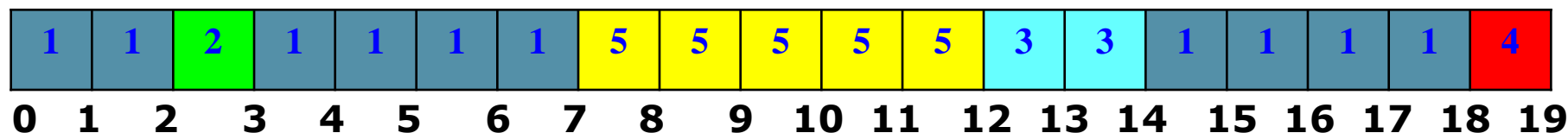
- Waiting time for $P_1=0$; $P_2=8$; $P_3=13$; $P_4=13$; $P_5=4$
- turnaround time $P_1=10$; $P_2=9$; $P_3=15$; $P_4=14$; $P_5=9$

举例——静态优先级、剥夺式调度

- 某系统采用剥夺式优先级调度算法，5个进程P₁、P₂、P₃、P₄、P₅依次进入就绪队列，其需要的CPU时间、到达时刻、及优先数如右：

进程	CPU 时间	到达 时刻	优先数
P1	10	0	3
P2	1	2	1
P3	2	3	3
P4	1	5	4
p5	5	7	2

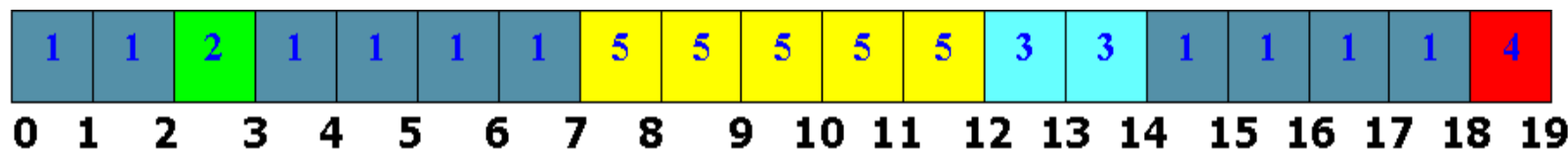
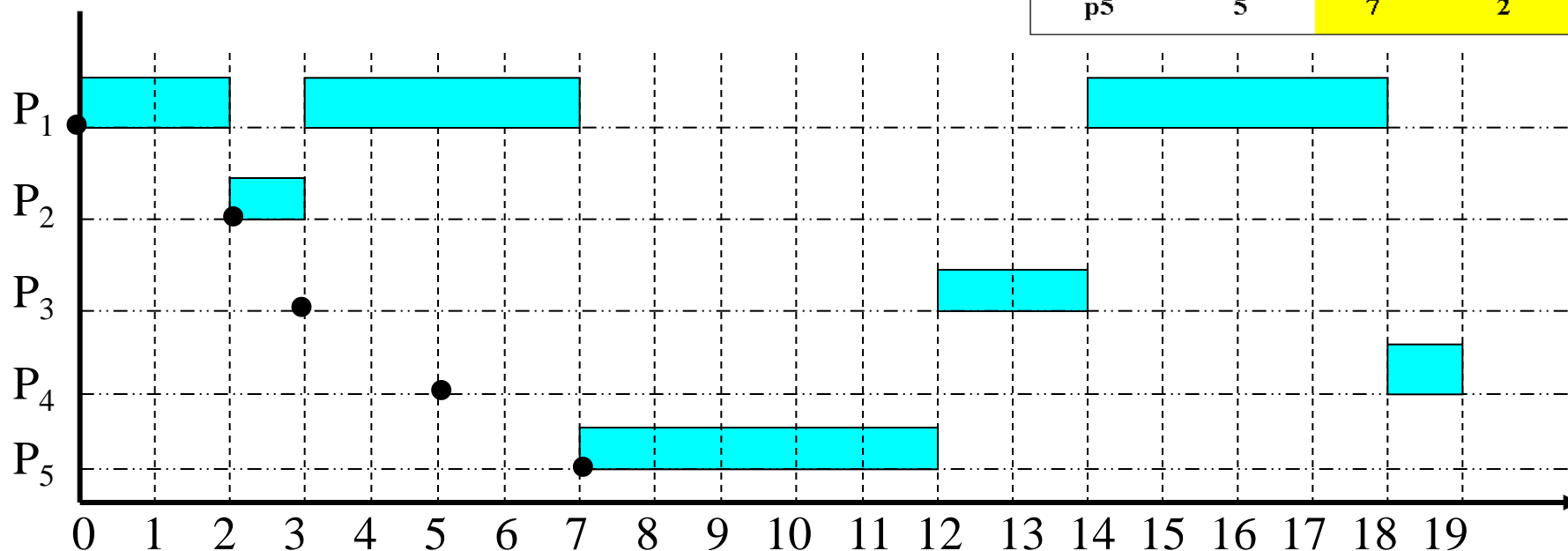
- 进程调度的甘特图如下：



- Waiting time(等待时间):
 - T₁=8、T₂=0、T₃=9、T₄=13、T₅=0; average: TA=6
- Turnaround time(周转时间):
 - T₁=18、T₂=1、T₃=11、T₄=14、T₅=5; average: TA=9.8

preemptive priority

进程	CPU 时间	到达 时刻	优先 级
P1	10	0	3
P2	1	2	1
P3	2	3	3
P4	1	5	4
P5	5	7	2



- Waiting time for $P_1=8$; $P_2=0$; $P_3=9$; $P_4=13$; $P_5=0$
- turnaround time $P_1=18$; $P_2=1$; $P_3=11$; $P_4=14$; $P_5=5$

举例——动态优先级、剥夺式调度

- 某系统采用基于动态优先级的剥夺式调度算法，并且**优先数越大的进程其优先级越高**。

系统为所有新建进程赋予优先数0，当一个进程在就绪队列中等待CPU时，其优先数的变化速率为 α ；进程获得CPU后开始执行，执行过程中，其优先数的变化速率为 β 。

为参数 α 和 β 设置不同的值，则导致不同的调度算法。

问题：

- a. 如果 $\beta > \alpha > 0$ ，则调度原则是什么？
- b. 如果 $\alpha < \beta < 0$ ，则调度原则是什么？

解答a: 如果 $\beta > \alpha > 0$, 调度原则是FCFS

已知: 优先数越大、优先级越高。

分析:

- 由于新建进程的优先数是0, 具有最低优先级。
- 当它在就绪队列中等待时, 进程的优先数不断增加, 即优先级不断提高, 所以最早进入就绪队列的进程, 其优先级也最高。
- 优先级最高的进程获得调度。
- 当进程在CPU上执行时, 它的优先数以更快的速度增加, 故就绪队列中的任何进程不可能具有比执行中进程更高的优先级, 执行进程将一直占有CPU, 直到它执行结束。
- 然后, 在就绪队列中等待时间最长的进程被调度到CPU上执行。

所以, 调度原则是 first-come first-served.

解答b: 如果 $\alpha < \beta < 0$, 调度原则是LCFS

已知: 优先数越大、优先级越高。

分析:

- 由于新建进程的优先数是0, 一旦它进入就绪队列, 它的优先数就开始减少, 就绪时间越长, 其优先数越小, 即优先级越低, 所以新建进程具有最高优先级。
- 由于新建进程优先级最高, 所以被调度到CPU上执行。
- 当进程在CPU上执行时, 它的优先数也不断减少, 但减少的速度要比就绪进程的慢, 故就绪队列中的任何进程不可能具有比执行中进程更高的优先级。
- 如果进程执行过程中, 有新建进程进入就绪队列, 则由于新进程的优先级最高, 将抢占CPU而执行, 原来的执行进程进入就绪队列, 它的优先级以更快的速度降低。
- 如果进程执行过程中, 没有新建进程, 则执行进程将一直占有CPU, 直到它执行结束。
- 然后, 在就绪队列中等待时间最短的进程被调度到CPU上执行。

所以, 调度原则是 Last-Come First-Served.

5.3.4 Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (*time quantum, or time slice*), usually 10-100 milliseconds. After this time has elapsed, the process is **preempted** and added to the end of the ready queue.
- Keep the ready queue as a FIFO queue of processes.
- Set a timer to interrupt after 1 time quantum.
- Two cases: CPU burst of the currently running process
 - less than 1 time quantum. The process release the CPU voluntarily.
 - longer than 1 time quantum. The timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue

Example of RR with Time Quantum=20

<u>Process</u>	<u>Burst Time</u>	<u>arrival time</u>
P_1	53	0
P_2	17	0.01
P_3	68	0.02
P_4	24	0.03

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3	
0	20	37	57	77	97	117	121	134	154	162

- Waiting time for: $P_1=81$; $P_2=20$; $P_3=94$; $P_4=97$
- Average waiting time = $(81 + 20 + 94 + 97)/4 = 73$

RR Scheduling(Cont.)

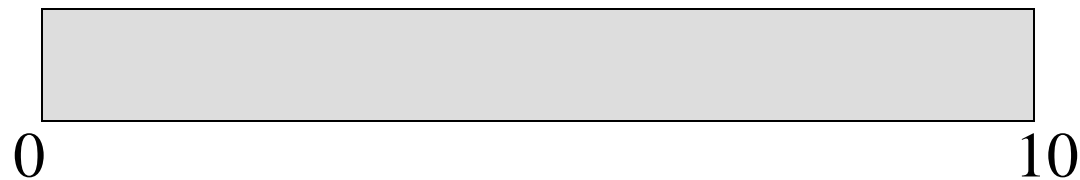
- The RR scheduling algorithm is preemptive.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units. No process waits more than $(n-1)q$ time units.
- Performance depends on the size of the time quantum
 - q very large (infinite) \Rightarrow FCFS
 - q very small \Rightarrow **processor sharing**, q must be large with respect to context switch, otherwise overhead is too high.

Time Quantum and Context Switch Time

Process time=10

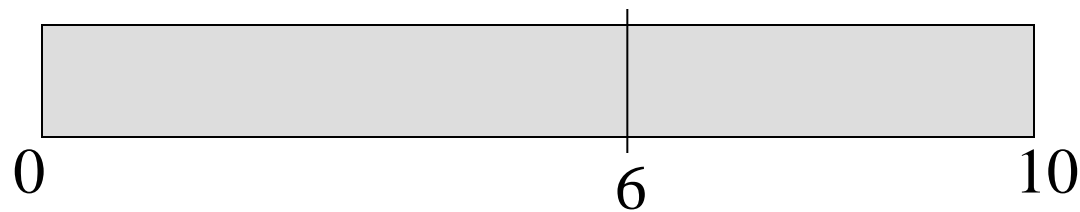
quantum

context
switches



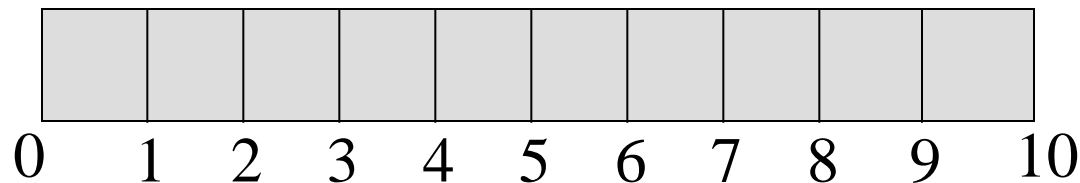
12

0



6

1



1

9

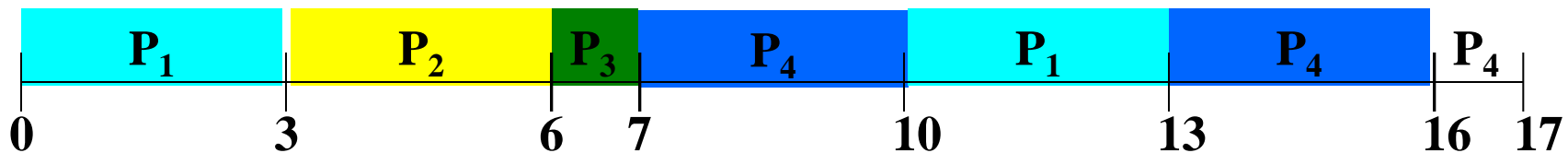
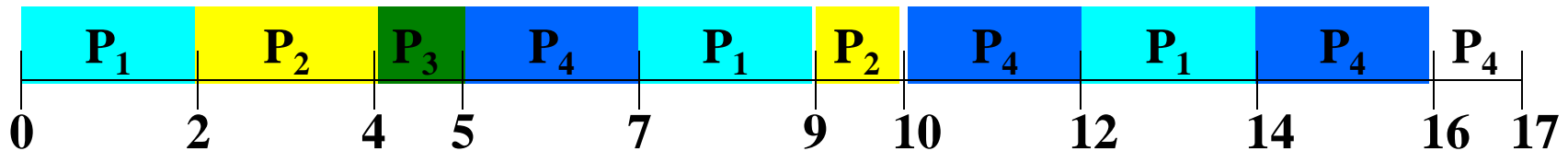
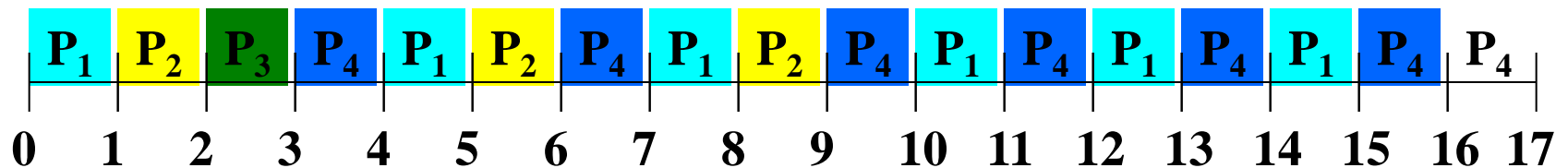
Turnaround Time Varies With The Time Quantum

Process	time
P ₁	6
P ₂	3
P ₃	1
P ₄	7

q=1
 $T_1 = 15;$
 $T_2 = 9;$
 $T_3 = 3;$
 $T_4 = 17;$
 $a_T = 11$

q=2
 $T_1 = 14;$
 $T_2 = 10;$
 $T_3 = 5;$
 $T_4 = 17;$
 $a_T = 11.5$

q=3
 $T_1 = 13;$
 $T_2 = 6;$
 $T_3 = 7;$
 $T_4 = 17;$
 $a_T = 10.75$



Turnaround Time Varies With The Time Quantum

Process	time
P ₁	6
P ₂	3
P ₃	1
P ₄	7

q=4

T₁ = 14;

T₂ = 7;

T₃ = 8;

T₄ = 17;

a_T = 11.5

q=5

T₁ = 15;

T₂ = 8;

T₃ = 9;

T₄ = 17;

a_T = 12.25

q=6

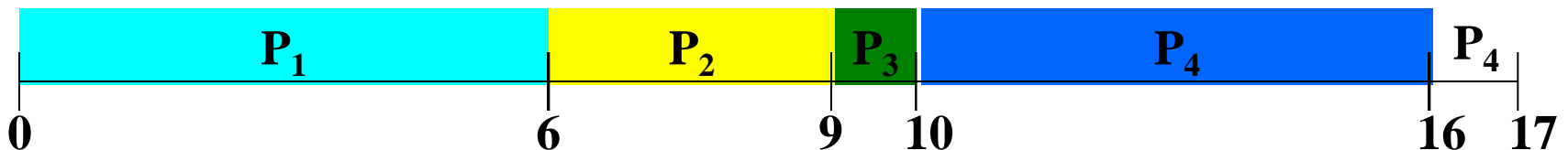
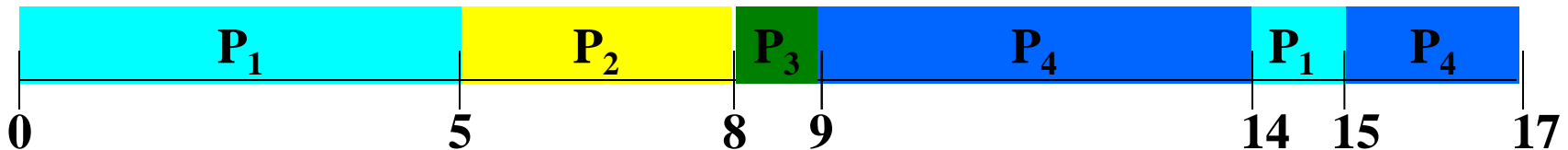
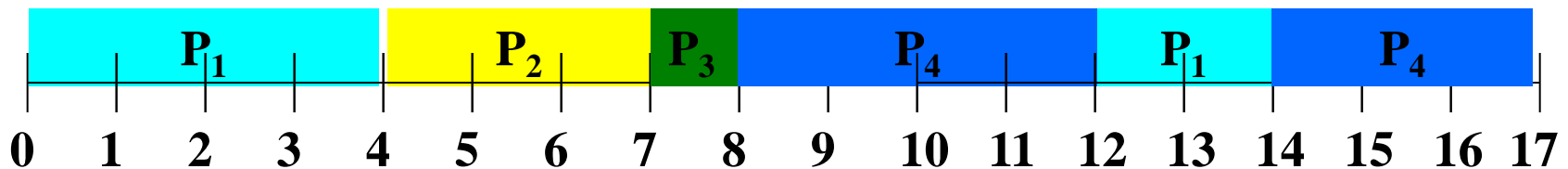
T₁ = 6;

T₂ = 9;

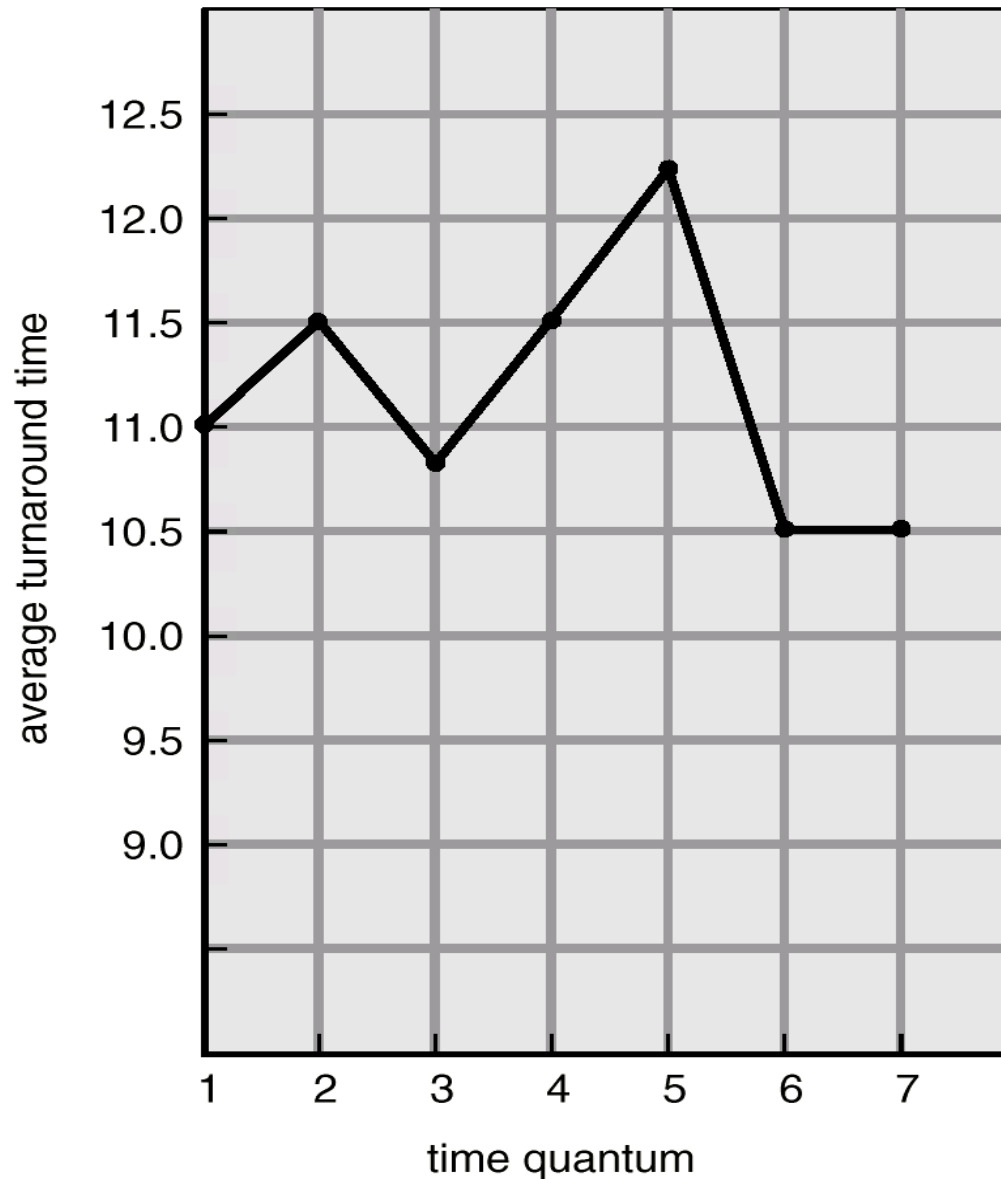
T₃ = 10;

T₄ = 17;

a_T = 10.5

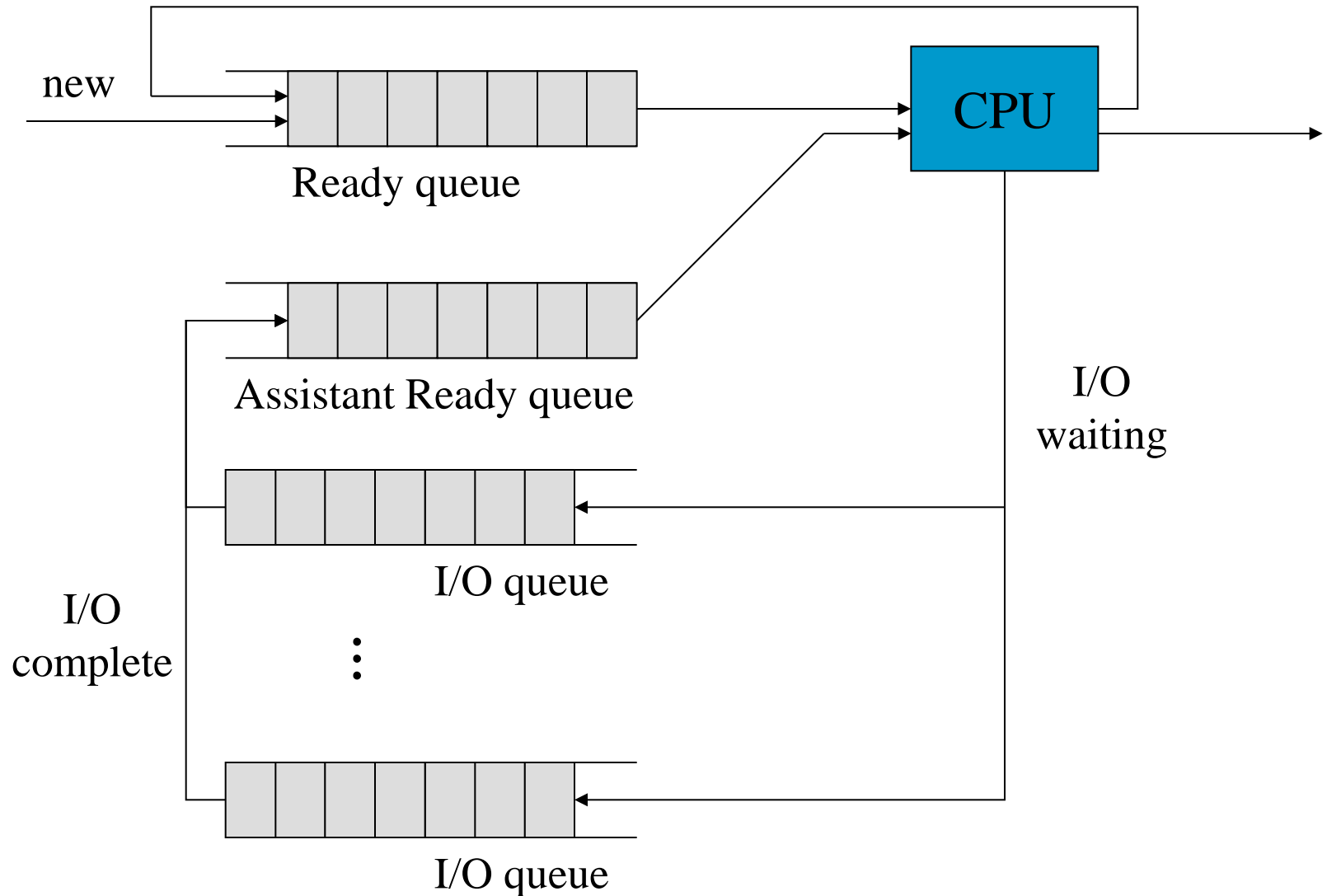


Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Supplement: Virtual Round Robin Scheduling

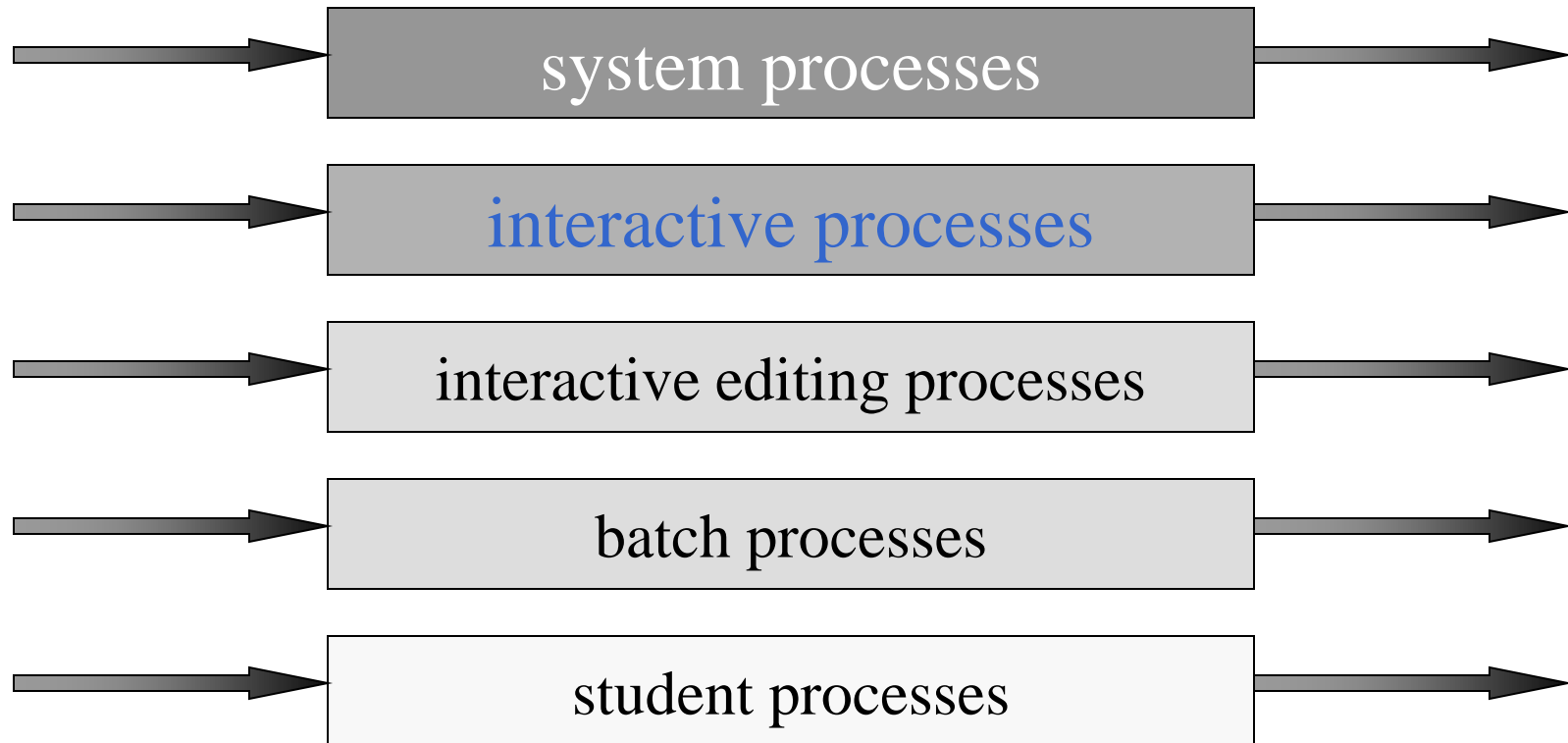


5.3.5 Multilevel Queue Scheduling

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

Multilevel Queue Scheduling

highest priority



lowest priority

5.3.6 Multilevel Feedback Queue Scheduling

- A process can move between the various queues.
- To separate processes with different CPU-burst characteristics.
 - If a process uses too much CPU time, it will be moved to a lower-priority queue.
 - If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue.
- Leaves I/O-bound and interactive processes in the higher-priority queues.
- This form of aging prevents starvation.

Multilevel Feedback Queue Scheduling

- **Multilevel-feedback-queue scheduler is defined by the following parameters:**
 - the number of queues
 - the scheduling algorithms for each queue
 - the method used to determine when to upgrade a process to a higher-priority queue
 - the method used to determine when to demote a process to a lower-priority queue
 - the method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

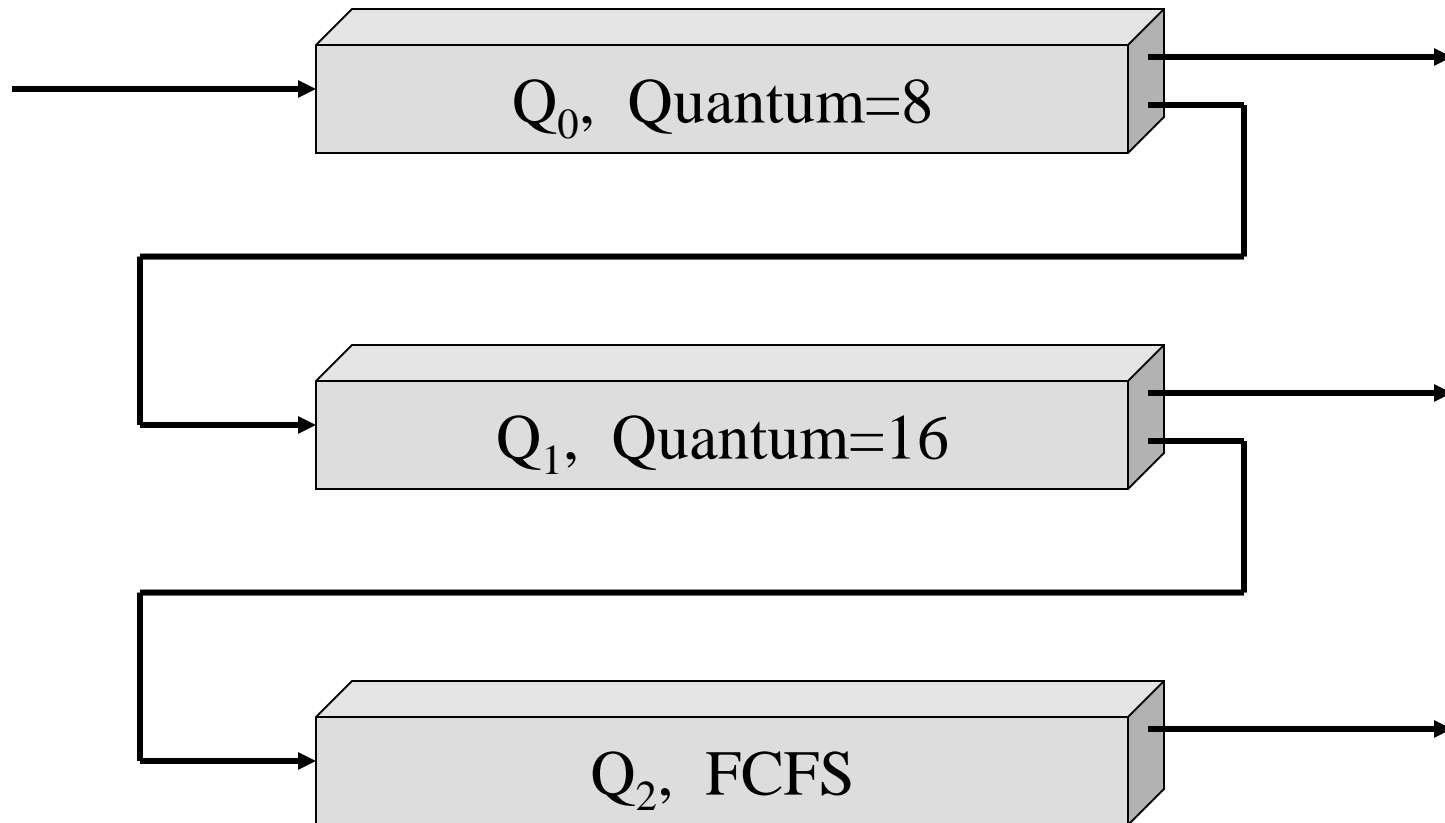
■ Three queues:

- Q_0 – time quantum 8 milliseconds --highest priority
- Q_1 – time quantum 16 milliseconds --lower priority
- Q_2 – FCFS -- lowest priority

■ Scheduling

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

Multilevel Feedback Queues



Supplement: 高响应比优先(HRRN)算法

Highest Response-Ratio Next Scheduling

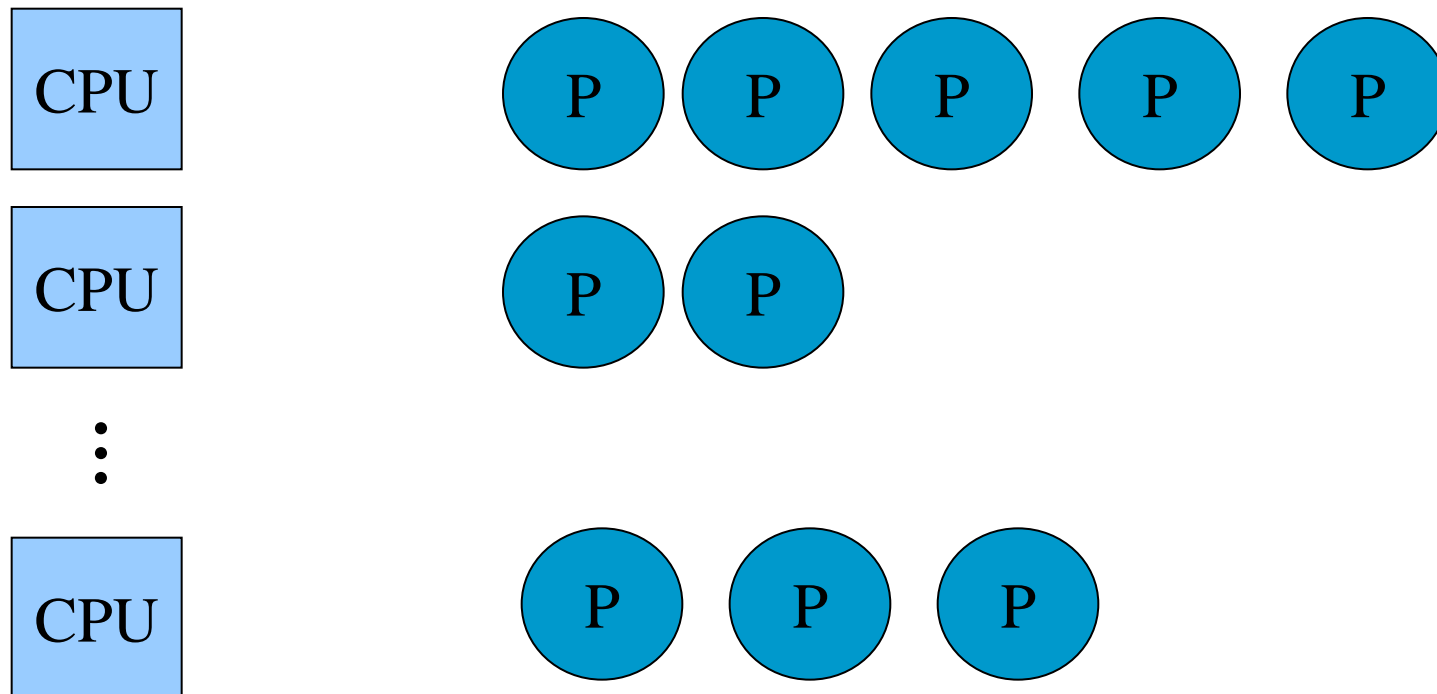
- **Non-preemptive scheduling algorithm.**
- **Response-Ratio :** $R=(W+T)/T=1+W/T$
 - W: waiting time in ready queue
 - T: CPU burst time
- **The process with the highest response-ratio will be scheduled next.**
- **HRRN considers the waiting time and CPU-burst time simultaneously.**
- **Time consuming, system overhead to calculate the response-ratio for each processes.**

5.4 Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- Classifications of Multiprocessor Systems
 - Loosely coupled multiprocessor
 - Each processor has its own memory and I/O channels
 - Tightly coupled multiprocessing
 - Processors share main memory
 - Controlled by operating system
 - Functionally specialized processors
 - Such as I/O processor
 - Controlled by a master processor
- *Homogeneous processors*: within a multiprocessor, any available processor can be used to run any processes in the queue.
- *Heterogeneous system*: only programs compiled for a given processor's instruction set could be run on that processor.

Process Scheduling

- Assignment of processes to processors
- Use of multiprogramming on individual processors
- Actual dispatching of a process



Assignment of Processes to Processors

- *Asymmetric multiprocessing* – only one processor accesses the system data structures, reducing the need for data sharing.
- **Master/slave architecture**
 - Key kernel functions always run on a particular processor
 - Master is responsible for scheduling
 - Slave sends service request to the master
 - Disadvantages
 - Failure of master brings down whole system
 - Master can become a performance bottleneck

Assignment of Processes to Processors

- **Peer architecture (symmetric multiprocessing)**
 - Operating system can execute on any processor
 - Each processor is self-scheduling
- **Ready queue**
 - Each processor has its own private ready queue
 - A common ready queue
- **Using private queue, permanently assign process to a processor**
 - Possible to provide a dedicate short-term queue for each processor
 - Problem: one processor could be idle while another one was very busy
- **Using common queue, must ensure:**
 - two processors do not choose the same process
 - processes are not lost from the queue

Processor Affinity

- **Processor Affinity(亲和力)**
 - A process has an affinity for the processor on which it is currently running
 - keep a process running on the same processor
- **Example: high cost of invalidating and re-populating cache memory**
- **Take forms**
 - **Soft affinity**
 - The OS has a policy of attempting to keep a process running on the same processor, but doesn't guarantee that it will do so.
 - **Hard affinity**
 - The OS provides system calls, allowing a process to specify that it is not to migrate to other processors

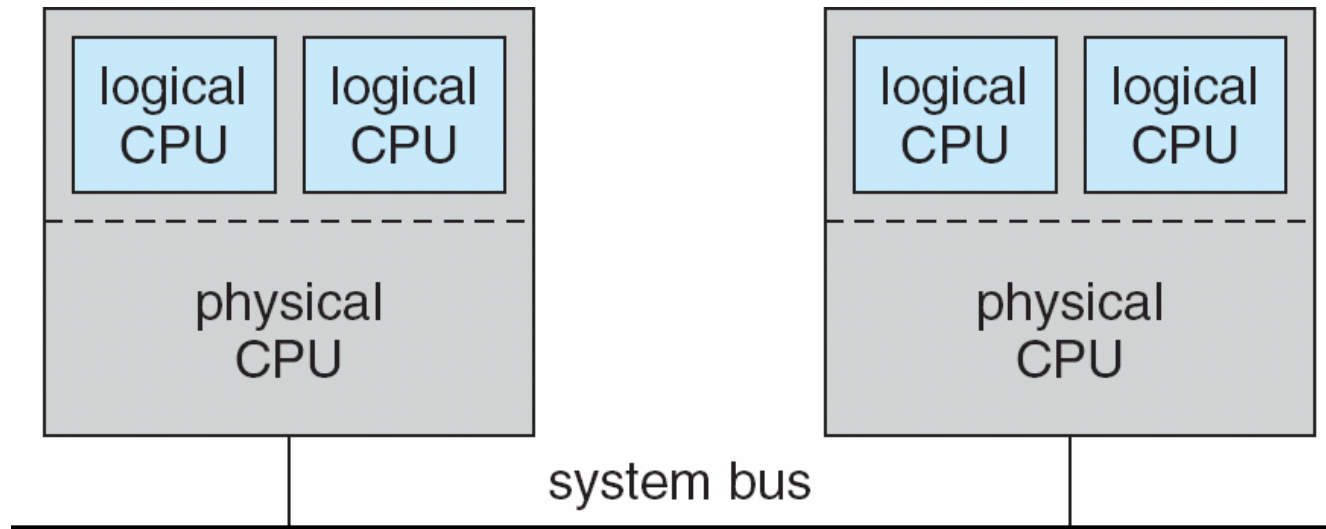
Load balancing

- **Load balancing: keep the workload evenly distributed across all processors in an SMP system**
- **Load balancing is *only necessary* on systems where each processor has its own private ready queue**
- **Two approaches**
 - **Push migration: a *specific task* periodically checks the load on each processor, if imbalance, pushing processes from overloaded processor to idle or less-busy ones.**
 - **Pull migration: an idle processor pulls a waiting task from a busy processor**
- **Often counteracts the benefits of processor affinity**

Symmetric multithreading (SMT)

- **SMP systems allow several threads to run concurrently by providing multiple processors**
- **SMT: provide multiple logical processors**
Hyperthreading technology on Intel processors
- **Create multiple logical processors on the same physical processor**
 - **presenting a view of several logical processors to the OS, even on a system with only a single physical processor.**
 - **Each logical processor has its own architecture state (including general-purpose and machine-state registers)**
 - **Each logical processor is responsible for its own interrupt handling**
 - **Each logical processor shares the resources of its physical processor, such as cache memory and buses**

A typical SMT architecture



- **SMT is a feature provided in hardware**
 - Hardware must provide the representation of the architecture state for each logical processor and interrupt handling

5.5 Thread Scheduling

■ Threads

- Executes separate from the rest of the process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- Threads running on separate processors yields a dramatic gain in performance

■ Thread Scheduling

- Local Scheduling – How the threads library decides which user-level thread to put onto an available LWP.
m:1 and m:n model, process-contention scope (PCS)
- Global Scheduling – How the kernel decides which kernel thread to run next.
System-contention scope (SCS)
1:1 model uses only SCS.

Supplement:

Multiprocessor Thread Scheduling

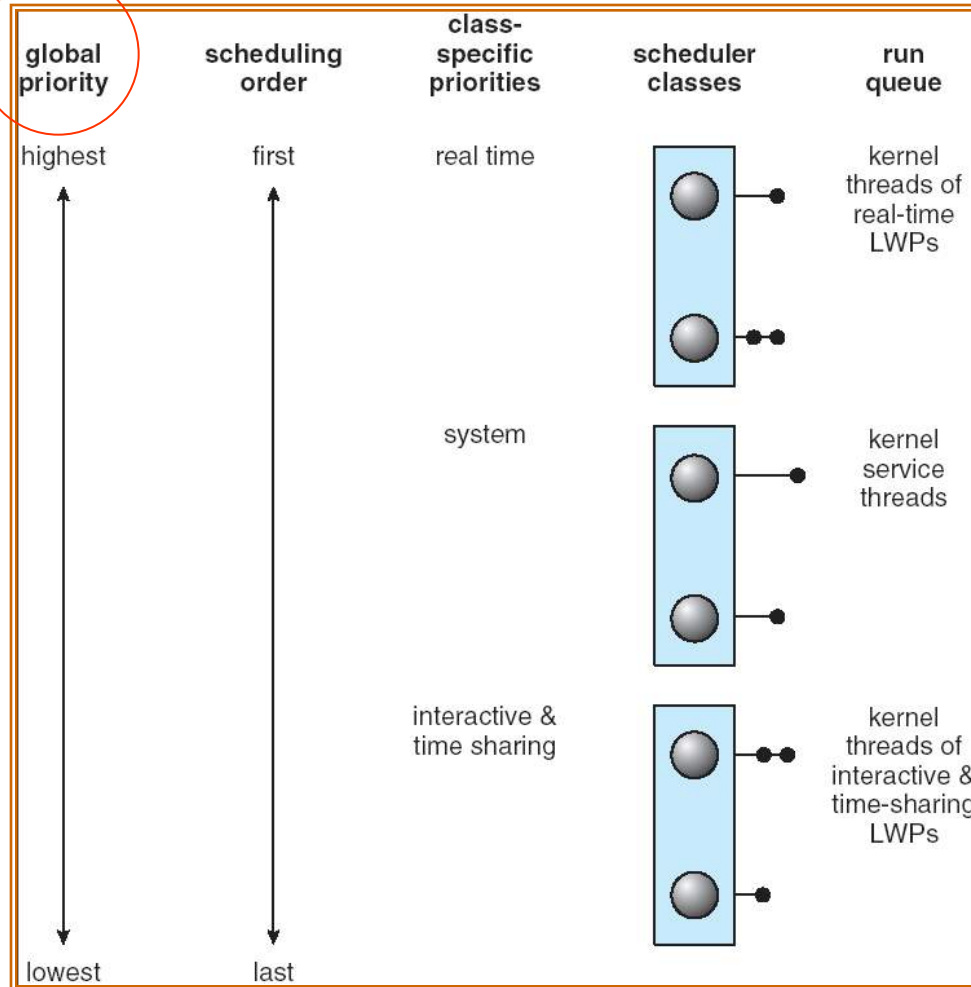
- **Load sharing**
 - Processes are not assigned to a particular processor
- **Gang scheduling**
 - A set of related threads is scheduled to run on a set of processors at the same time
- **Dedicated processor assignment**
 - Threads are assigned to a specific processor
- **Dynamic scheduling**
 - Number of threads can be altered during course of execution

5.6 Operating System Examples

- **Solaris scheduling**
 - Priority-based thread scheduling
- **Windows XP scheduling**
 - Priority-based, preemptive scheduling (multi-queue)
- **Linux scheduling**
 - Preemptive, priority-based scheduling

Solaris Scheduling

A thread with the highest global priority is selected to run.
Threads with the same priority are scheduled by using RR.



Threads in the real-time class are given the highest priority

Use the system class to run kernel processes, such as scheduler and paging daemon

Default scheduling class. Dynamically alter priorities and assigns time slices of different lengths using a multilevel feedback queue

Solaris Dispatch Table for interactive and time-sharing threads

new priority

low

high

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

Windows XP Priorities

- The priority of each thread is based on the priority class and its relative priority
- Processes are typically members of the `NORMAL_PRIORITY_CLASS`
- The initial priority of a thread is typically the base priority of the process the thread belongs to.
- When time slice runs out, priority lowered, but never lowered below the base priority; when resumed from a wait operation, priority is boosted.

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

Real-time class

Variable class

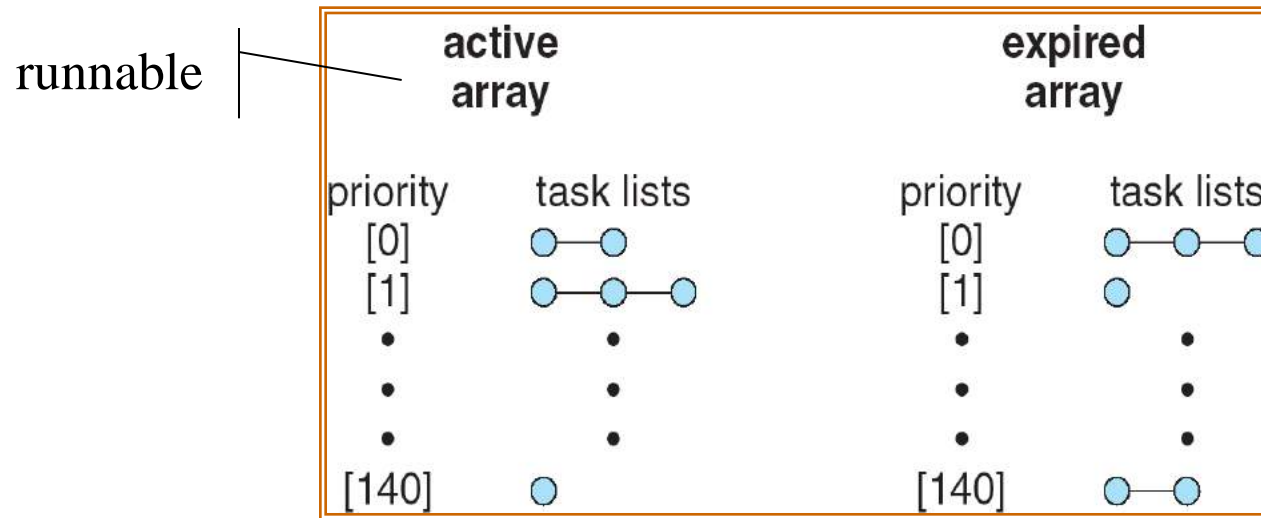
Linux Scheduling

- **Two algorithms: time-sharing and real-time**
- **Time-sharing**
 - **Prioritized credit-based – process with most credits is scheduled next**
 - **Credit subtracted when timer interrupt occurs**
 - **When credit = 0, another process chosen**
 - **When all processes have credit = 0, recrediting occurs**
 - **Based on factors including priority and history**
- **Real-time**
 - **Soft real-time**
 - **Posix.1b compliant – two classes**
 - **FCFS and RR**
 - **Highest priority process always runs first**

The Relationship Between Priorities and Time-slice length

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99		other tasks	10 ms
100			
•			
•			
•			
140	lowest		

List of Tasks Indexed According to Priorities



- A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice.
- The kernel maintains a list of all runnable task in a runqueue data structure, each runqueue contains two priority arrays (see above fig.)
- Each processor maintains its own runqueue, schedules itself.
- When all task have exhausted their time slice, the two priority arrays are exchanged.
- The recalculation of a task's dynamic priority occurs when the task has exhausted its time slice and is to be moved to the expired array.

5.7 Algorithm Evaluation

- **Defining the criteria to be used in selecting an algorithm.**
 - CPU utilization, response time, or throughput
- **Define the relative importance of these measures**
 - Maximum CPU utilization under the constraint that the maximum response time is 1 second.
 - Maximum throughput such that turnaround time is linearly proportional to total execution time.
- **Deterministic modeling**
- **Queuing models**
- **Simulations**
- **Implementation**

Deterministic modeling

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

- **Example**

- assume we have the workload:
- consider the FCFS, SJF, RR(q=10)
- compare their average waiting time

<u>Process</u>	<u>Burst time</u>
P ₁	10
P ₂	29
P ₃	3
P ₄	7
P ₅	12

- **Average waiting time**

- FCFS: $(0+10+39+42+49)/5=28$
- SJF: $(10+32+0+3+20)/5=13$
- RR: $(0+32+20+23+40)/5=23$

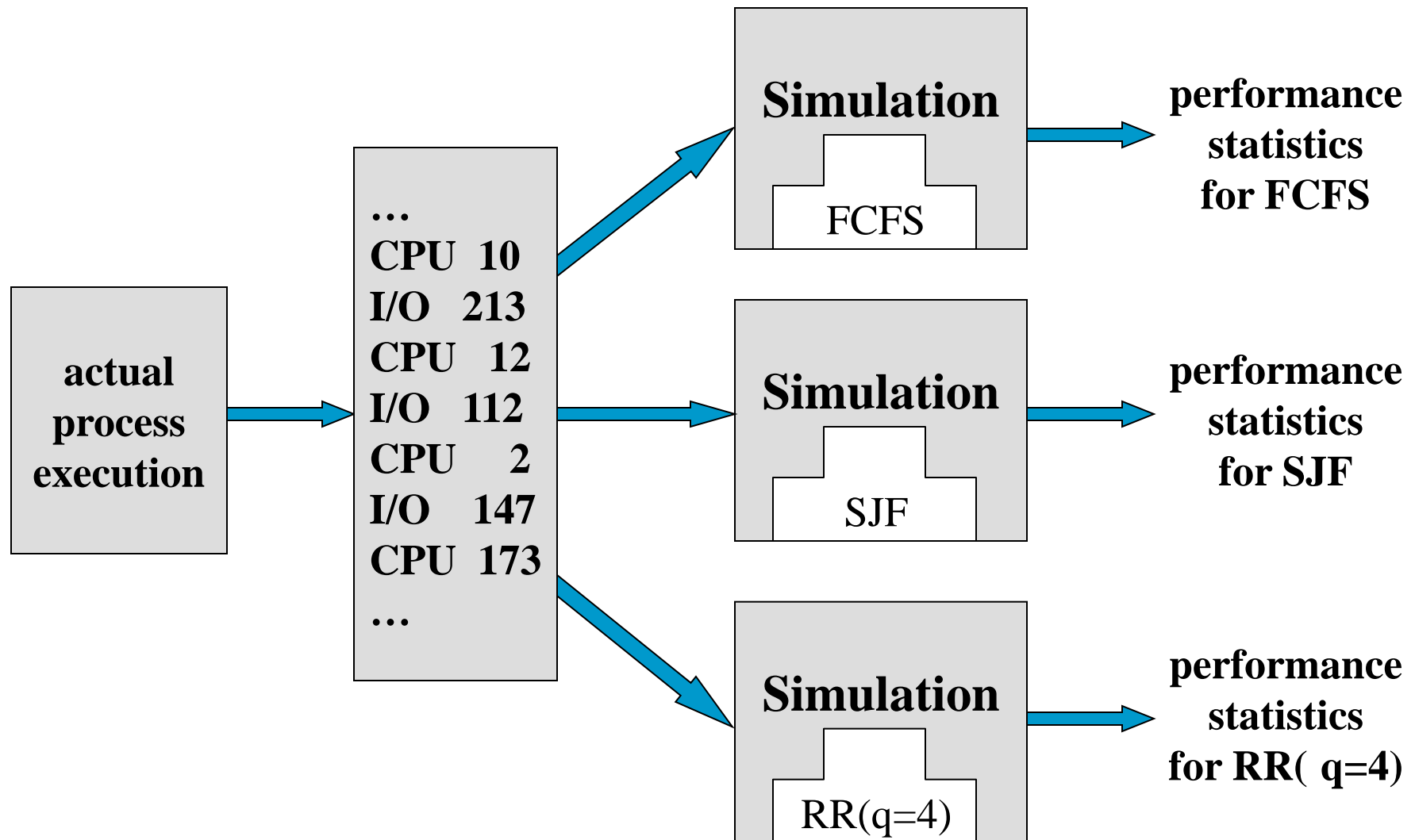
Queuing models

- The distribution of times when processes arrive in the system, the *arrival-time distribution*, must be given.
- Computer system is described as a network of servers.
 - Each server has a queue of waiting processes.
 - The CPU is a server with its ready queue, as is the I/O system with its device queues.
- Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time.
- **Little's formula:** $n = \lambda \times W$
 - n : average queue length excluding the process being serviced
 - W : average waiting time in the queue
 - λ : average arrival rate for new processes in the queue

Simulations

- Simulations involve programming a model of the computer system
 - software data structures--the major components of the system
 - a variable--clock
 - as the simulation executes, statistics that indicate algorithm performance are gathered and printed.
- The data to drive the simulation can be generated by using a *random-number generator*.
- The distributions may be defined mathematically or *empirically* (根据经验).
- A distribution-driven simulation may be inaccurate, due to relationships between successive events in the real system.

Evaluation of CPU Schedulers by Simulation



Implementation

- Puts the actual algorithm in the real system for evaluation under real operating system.
- Difficulties
 - *The cost*: the expense is incurred not only in coding the algorithm and modifying the operating system to support it as well as its required data structure, but also in the reaction of the users to a constantly changing operating system.
 - *The environment will change*: the environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler.

Homework (page 186)

5.4 5.5 5.9

- 1. Define the difference between preemptive and nonpreemptive scheduling.**
- 2. Is a nonpreemptive scheduling algorithm a good choice for an interactive system? Briefly, why?**
- 3. Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?**

4. Suppose that the following processes arrive for execution at the times indicated. Each process will run the listed amount of time. In answering the questions, use nonpreemptive scheduling and base all decisions on the information you have at the time the decision must be made.

process	Arrival time	Burst time
P1	0.0	8
P2	0.4	4
P3	1.0	1

- What is the average turnaround time for these processes with the FCFS scheduling algorithm?
- What is the average turnaround time for these processes with the SJF scheduling algorithm?
- The SJF algorithm is supposed to improve performance, but notice that we chose to run process *P1* at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes *P1* and *P2* are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.

5. Considering a real-time system, in which there are 4 real-time processes P_1 , P_2 , P_3 and P_4 that are aimed to react to 4 critical environmental events e_1 , e_2 , e_3 and e_4 in time respectively.

The arrival time of each event e_i , $1 \leq i \leq 4$, (that is, the arrival time of the process P_i), the length of the CPU burst time of each process P_i , and the deadline for each event e_i are given below. Here, the deadline for e_i is defined as the absolute time point before which the process P_i must be completed.

The priority for each event e_i (also for P_i) is also given, and a smaller priority number implies a higher priority.

<u>Events</u>	<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Priorities</u>	<u>Deadline</u>
e1	P1	0.00	4.00	3	7.00
e2	P2	3.00	2.00	1	5.50
e3	P3	4.00	2.00	4	12.01
e4	P4	6.00	4.00	2	11.00

- (1) Suppose that priority-based preemptive scheduling is employed
 Draw a Gantt chart illustrating the execution of these processes.
 What are the average waiting time and the average turnaround time.
 Which event will be treated with in time, that is, the process reacting to this event will be completed before its deadline?
- (2) Suppose that FCFS scheduling is employed
 Draw a Gantt chart illustrating the execution of these processes.
 What are the average waiting time and the average turnaround time.
 Which event will be treated with in time?

6. Consider the following set of processes P1, P2, P3 and P4. For $1 \leq i \leq 4$, the arrival time of each P_i , the length of the CPU burst time of each process P_i , and the priority number for each P_i are given as below, and a smaller priority number implies a higher priority.

Process	Arrival Time	Burst Time	Priority Number
P1	0.00	4.00	1
P2	3.00	3.00	1
P3	4.00	3.00	4
P4	5.00	5.00	3

- (1) For a process having been in the system, its *response-ratio* is defined as its waiting time divided by its CPU burst time (i.e. waiting time/CPU burst time). When the CPU is idle, Highest Response-Ratio Next(HRRN) scheduling selects the process with the highest response-ratio to allocate CPU, and a process in the waiting state is not allowed to preempt the CPU allocated to the other process in the running state. Suppose that HRRN scheduling is employed.

Draw a Gantt chart illustrating the execution of these processes.

What are the average waiting time and the average turnaround time.

- (2) Suppose that nonpreemptive priority scheduling is employed,

Draw a Gantt chart illustrating the execution of these processes.

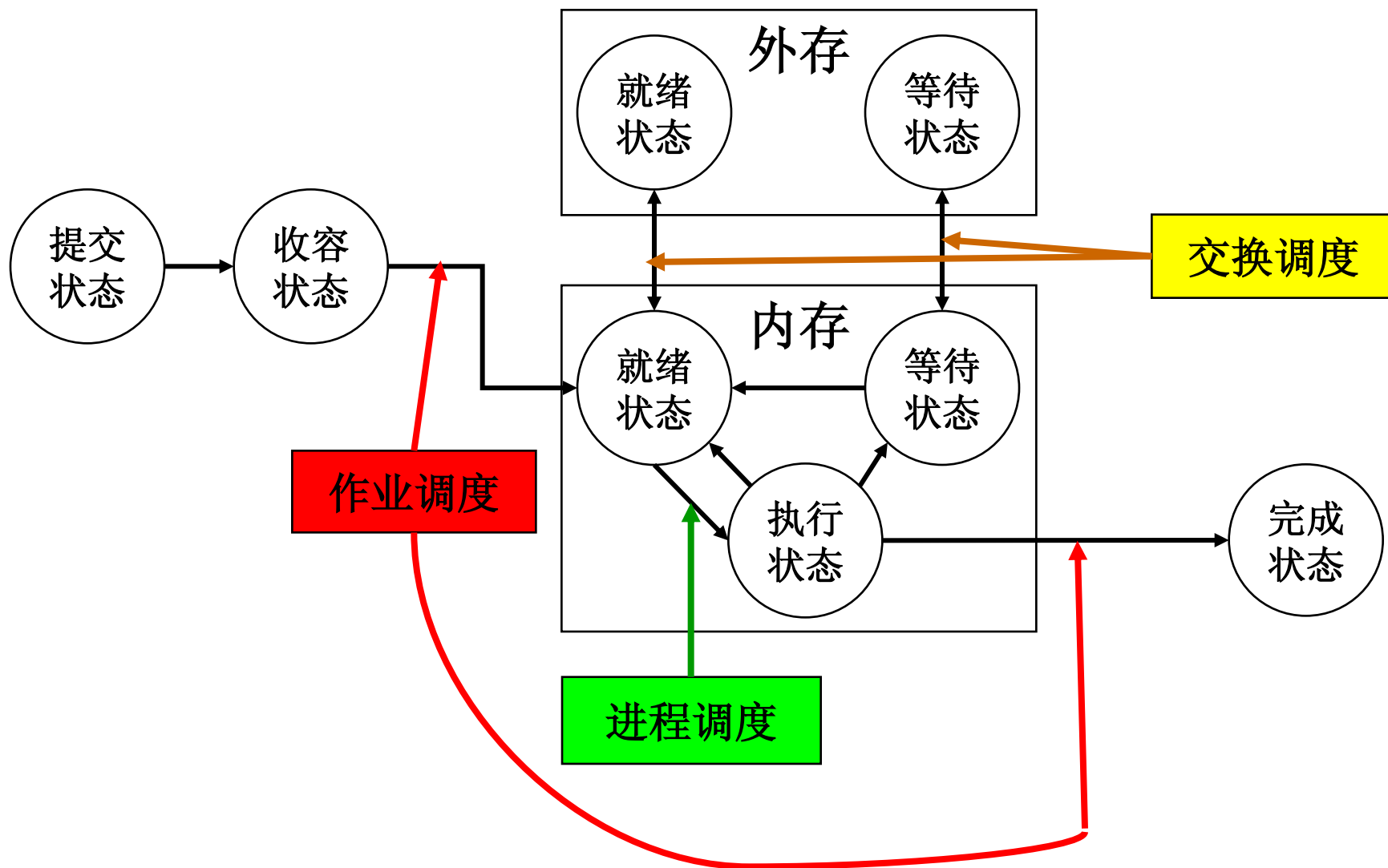
What are the average waiting time and the average turnaround time.



Supplement: Types of Scheduling

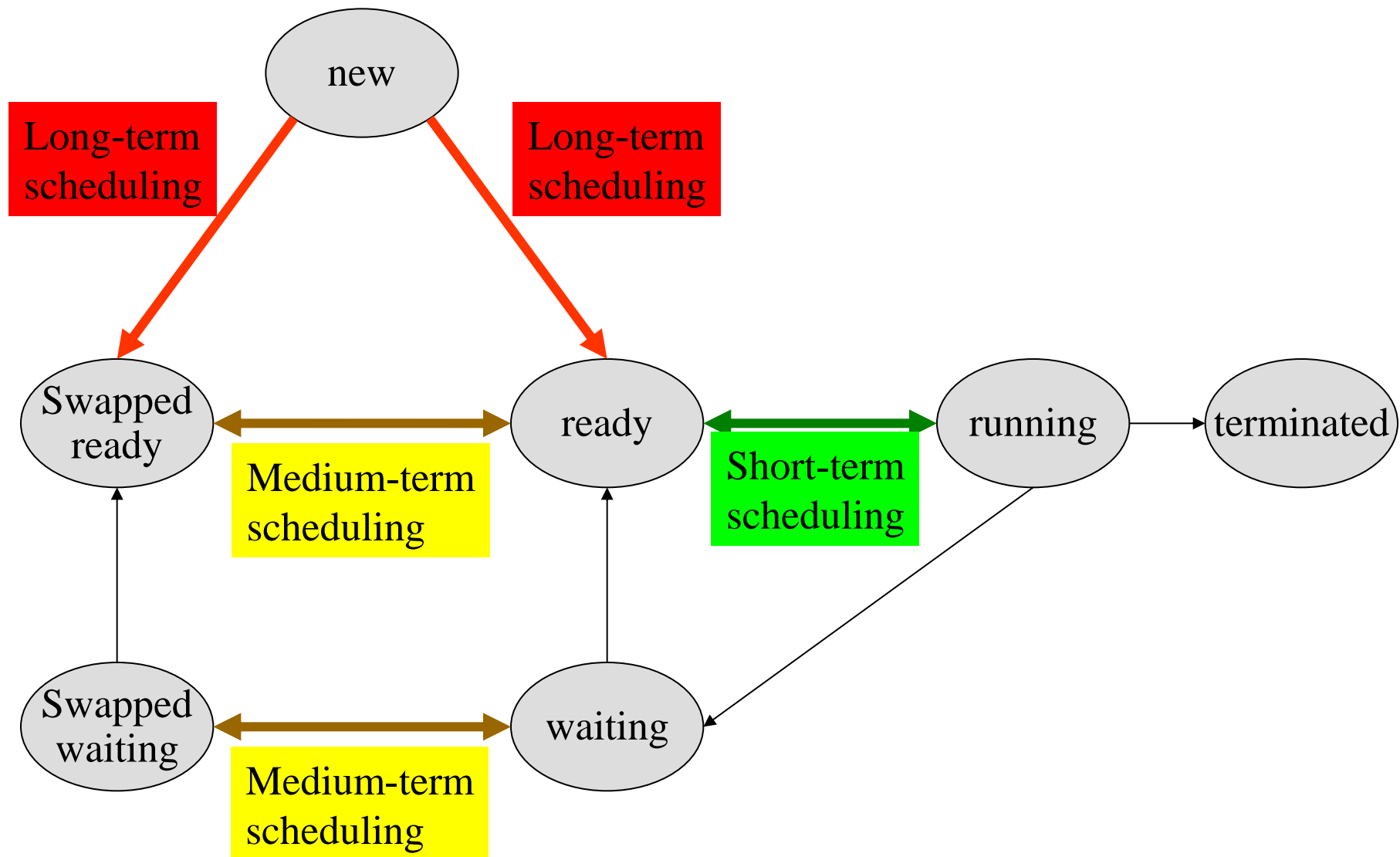
long-term scheduling	The decision to add to the pool of processes to be executed.
medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory.
short-term Scheduling	The decision as to which available process will be executed by the processor.
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device.

Supplement: 各级调度之间的关系



Supplement:

Scheduling and Process State Transitions

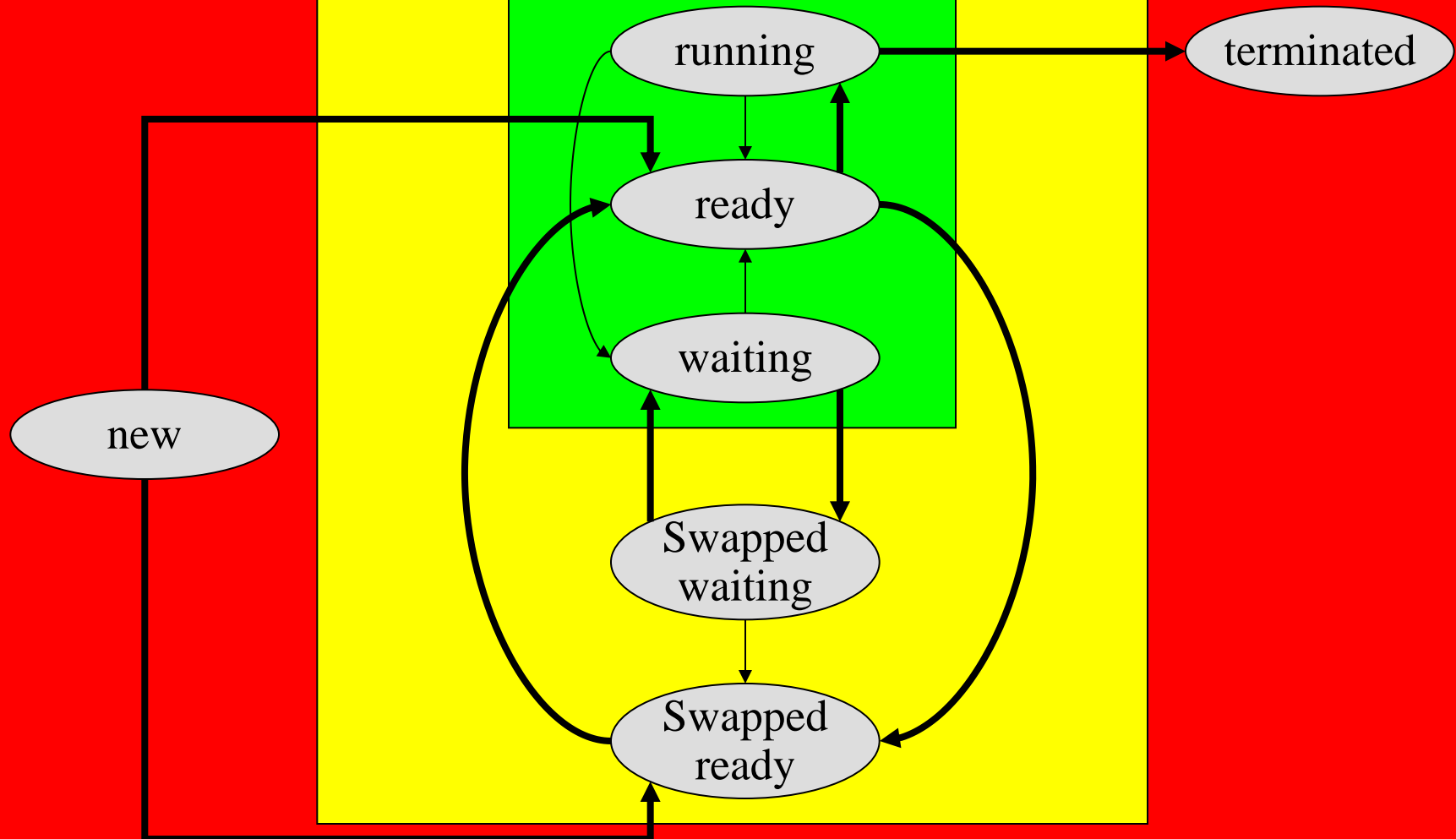


Supplement: Levels of Scheduling

Job scheduling

swapping

CPU scheduling



Supplement: Long-Term Scheduling

- **Determines which programs are admitted to the system for processing**
- **Controls the degree of multiprogramming**
- **More processes, smaller percentage of time each process is executed**

Supplement: Medium-Term Scheduling

- **Part of the swapping function**
- **Based on the need to manage the degree of multiprogramming**

Supplement: Short-Term Scheduling

- **Known as the dispatcher**
- **Executes most frequently**
- **Invoked when an event occurs**
 - **Clock interrupts**
 - **I/O interrupts**
 - **Operating system calls**
 - **Signals**

Chapter 6 Process Synchronization



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

Critical-Section Problem, Semaphores

Classical Problems of Synchronization

Monitors

Chapter Objectives

- **To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.**
- **To present both software and hardware solutions of the critical-section problem**
- **To introduce the concept of atomic transaction and describe mechanisms to ensure atomicity.**

Contents

- 6.1 Background**
- 6.2 The Critical-Section Problem**
- 6.3 Peterson's Solution**
- 6.4 Synchronization Hardware**
- 6.5 Semaphores**
- 6.6 Classic Problems of Synchronization**
- 6.7 Monitors**
- 6.8 Synchronization Examples**

6.1 Background

■ Concurrency

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time

■ Context

- Multiple applications
 - Multiprogramming
- Structured application
 - Application can be a set of concurrent processes
- Operating-system structure
 - Operating system is a set of processes or threads

Difficulties with Concurrency

- **Sharing global resources**
- **Management of allocation of resources**
- **Programming errors difficult to locate**

Example

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 3) allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.
 - Suppose that we modify the **producer-consumer** code by adding a integer variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer, and decremented each time an item is removed from the buffer.

Bounded-Buffer procedure-consumer problem

■ Shared data

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

Bounded-Buffer procedure-consumer problem

■ Producer process

item nextProduced;

```
while (1) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

■ Consumer process

item nextConsumed;

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


Bounded-Buffer procedure-consumer problem

- The statements
`counter++;`
`counter--;`

must be performed *atomically*.
- **Atomic operation** means an operation that completes in its entirety without interruption.

Bounded-Buffer procedure-consumer problem

- The statement “ **count++** ” may be implemented in machine language as:

$\text{register}_1 = \text{counter}$

$\text{register}_1 = \text{register}_1 + 1$

$\text{counter} = \text{register}_1$



- The statement “ **count--** ” may be implemented as:

$\text{register}_2 = \text{counter}$

$\text{register}_2 = \text{register}_2 - 1$

$\text{counter} = \text{register}_2$



Race Condition

- Assume **counter** is initially 5. One interleaving of statements is:

producer: $\text{register}_1 = \text{counter}$ (*register₁ = 5*)

producer: $\text{register}_1 = \text{register}_1 + 1$ (*register₁ = 6*)

consumer: $\text{register}_2 = \text{counter}$ (*register₂ = 5*)

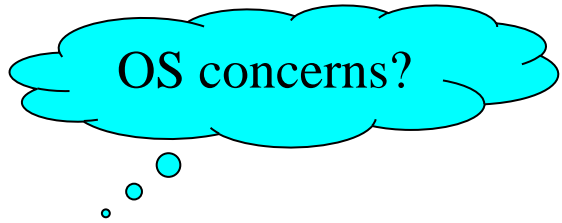
consumer: $\text{register}_2 = \text{register}_2 - 1$ (*register₂ = 4*)

producer: $\text{counter} = \text{register}_1$ (*counter = 6*)

consumer: $\text{counter} = \text{register}_2$ (*counter = 4*)

- The value of counter may be either 4 or 6, where the correct result should be 5.

Race Condition (Cont.)



OS concerns?

- **Race condition:** The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon the particular order in which the access takes place.
- **Race conditions in operating systems**
 - The code implementing an OS is subject to several possible race conditions.
 - System open-file-table
 - Structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling
- To prevent race conditions, concurrent processes must be **synchronized**.

6.2 The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- Need to design a protocol that the processes can use to cooperate.
 - Each process must *request* permission to enter its critical section.
 - The section of code implementing this request is the *entry section*.
 - The critical section may be followed by an *exit section*.
 - The remaining code is the remainder section.

Solution to Critical-Section Problem

must satisfy the following three requirements

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then 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.

Types of solution for critical section

- **Software: programming**
- **Hardware instructions**
 - TestAndSet
 - Swap
- **Semaphores**

6.3 Peterson's Solution

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)
 - do {
 - entry section*
 - critical section
 - exit section*
 - reminder section
 - } while (1);
- Processes may share some common variables to synchronize their actions.

Supplement: Algorithm 1

Mutual exclusion?	✓
Progress?	✗
Bounded-waiting?	✗

■ Shared variables:

- int turn;
initially turn = 0
- $\text{turn} = i \Rightarrow P_i$ can enter its critical section

■ Process P_0

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

■ Process P_1

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

Supplement: Algorithm 2

■ Shared variables

- boolean flag[2];
initially flag [0] = flag [1] = false.
- flag [i] = true $\Rightarrow P_i$ ready to enter its critical section

Mutual exclusion? \checkmark
Progress? **X**
Bounded-waiting? **X**

■ Process P_0

do {

while (flag[1]) ;

flag[0] := true;

critical section

flag [0] = false;

remainder section

} while (1);

■ Process P_1

do {

while (flag[0]) ;

flag[1] := true;

critical section

flag [1] = false;

remainder section

} while (1);

Algorithm 3 (Peterson Algorithm)

- Combined shared variables of algorithms 1 and 2.

- Process P_i

do {

 flag [i] := true;

 turn = j;

 while (flag [j] and turn = j) ;

 critical section

 flag [i] = false;

 remainder section

} while (1);

Mutual exclusion? ✓

Progress? ✓

Bounded-waiting? ✓

Solution for multiprocesses?

6.4 Synchronization Hardware

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

Test and Set Instruction

- Test and modify the content of a word atomically

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

Mutual Exclusion with Test-and-Set

- Shared data:
 boolean lock = false;
- Process P_i
 do {
 while (TestAndSet(&lock)) ;
 critical section
 lock = FALSE;
 remainder section
 } while(TRUE)

Mutual exclusion?	✓
Progress?	✓
Bounded-waiting?	×

Swap Instruction

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to false):
boolean lock;

- Process P_i
do {
 key = true;
 while (key == TRUE)
 Swap (&lock, &key);
 critical section
 lock = FALSE;
 remainder section
} while(TRUE)

Mutual exclusion?

✓

Progress?

✓

Bounded-waiting?

×

- Shared data (initialized to false):
boolean lock;
boolean waiting[n];
- Process P_i

do {

```

waiting[i] = TRUE;    key = TRUE;
while (waiting[i] && key)  key = TestAndSet(&lock);
waiting[i] = FALSE;

```

critical section

```

j = (i+1) % n;
while ((j != i) && !waiting[j])  j = (j+1) % n;
if (j == i)  lock = FALSE;
else  waiting[j] = FALSE;

```

remainder section

} while(TRUE);

Mutual exclusion? ✓

Progress? ✓

Bounded-waiting? ✓

Mutual Exclusion Machine Instructions

■ Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections

■ Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
- Deadlock
 - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

6.5 Semaphores

- Synchronization tool that does not require busy waiting.
- Special variable, Semaphore S – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (S):

```
while  $S \leq 0$  ;    //do no-op  
     $S--$ ;
```

signal (S):

```
 $S++$ ;
```

- Wait and signal operations cannot be interrupted.
- Semaphore is a variable that has an integer value
 - May be initialized to a nonnegative number
 - Wait operation decrements the semaphore value
 - Signal operation increments semaphore value

Usage

Critical Section of n Processes

■ Shared data:

semaphore mutex; // initially *mutex* = 1

```
wait (S):
    while  $S \leq 0$  ;
    S--;
signal (S):
    S++;
```

■ Process P_i :

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```

Shared data:

Semaphore synch; // 0

P1:

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

P2:

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

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {  
    int value;  
    struct process *List;  
} semaphore;
```
- Assume two simple operations:
 - Block() suspends the process that invokes it.
 - wakeup(*P*) resumes the execution of a blocked process *P*.

Implementation (Cont.)

- Semaphore operations now defined as *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); }**

Critical-Section Problem of Semaphore

■ Uniprocessor environment

- Inhibit interrupts during the time that wait and signal operations are executing.

■ multiprocessor environment

- inhibiting interrupts does not work.
- Special hardware instructions
- Software solution for the critical-section problem, where the critical section consists of the wait and signal procedures.

■ Busy waiting problem

- have removed busy waiting from the entry to the critical sections of application programs.
- have limited busy waiting to only the critical sections of the wait and signal operations, these sections are short.

Problems: Deadlock and Starvation

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

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

P_0 : wait (S);
 P_1 : wait (Q);
 P_0 : wait (Q);
 P_1 : wait (S);

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

Supplement: Two Types of Semaphores

- *Binary* semaphore – integer value can range only between 0 and 1; can be simple to implement.
- *Counting* semaphore – integer value can range over an unrestricted domain.
- A counting semaphore S Can be implemented by using binary semaphores.

– Data structures:

binary-semaphore S1, S2;

int C;

– Initialization:

S1 = 1

S2 = 0

C = initial value of semaphore S

Supplement:

Implementing S as a Binary semaphore

■ *Wait(S)* operation

```
wait (S1);  
C--;  
if (C < 0) {  
    signal (S1);  
    wait (S2)  
}  
else  
    signal (S1);
```

■ *Signal(S)* operation

```
wait (S1);  
C ++;  
if (C <= 0)  
    signal(S2);  
signal(S1);
```

6.6 Classical Problems of Synchronization

- **The Bounded-Buffer Problem**
- **The Readers-Writers Problem**
- **The Dining-Philosophers Problem**

The Bounded-Buffer Problem

- Assume the pool consists of n buffers, each capable of holding one item.
- Shared data
semaphore **full, empty, mutex;**
- Initially:
full = 0
empty = n
mutex = 1

Bounded-Buffer Problem

Producer Process

```
do {  
    ...  
    produce an item in nextp  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    add nextp to buffer  
    ...  
    signal(mutex);  
    signal(full);  
} while (1);
```

Consumer Process

```
do {  
    wait(full);  
    wait(mutex);  
    ...  
    remove an item from  
    buffer to nextc  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    consume the item in nextc  
    ...  
} while (1);
```

The Readers-Writers Problem

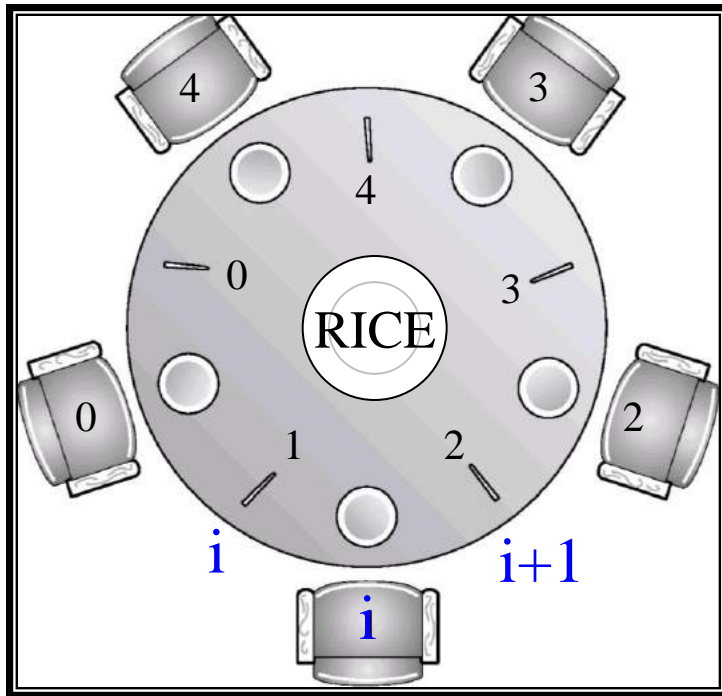
- **Readers:** only read the content of the shared object.
- **Writers:** may update the content of the shared object.
- **The writers have exclusive access to the shared object.**
- **the first readers-writers problem: reader first**
 - no reader will be kept waiting unless a writer has already obtained permission to use the shared object.
 - Writers may starve.
- **the second readers-writers problem: writer first**
 - once a writer is ready, that writer performs its write as soon as possible.
 - Readers may starve.

The Readers-Writers Problem(Cont.)

- Shared data
semaphore **mutex, wrt**;
- Initially
mutex = 1
wrt = 1
readcount = 0
- Writer process
wait(wrt);
...
writing is performed
...
signal(wrt);

- Reader process
wait(mutex);
readcount++;
if (readcount == 1)
 wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
 signal(wrt);
signal(mutex);

Dining-Philosophers Problem



```

■ Philosopher  $i$ :
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think
    ...
} while (1);
  
```

- Shared data
semaphore **chopstick[5];**
Initially all values are 1

Solution 1: allow at most 4 philosophers to be setting simultaneously at the table

```
semaphore chopstick[5]={1,1,1,1,1};  
semaphore room=4;  
int i;  
void philosopher(int i);  
{  
  while(1){  
    thinking;  
    wait (room);  
    wait (chopstick[i]);  
    wait (chopstick[(i+1) % 5]);  
    eating;  
    signal (chopstick[(i+1) % 5]);  
    signal (chopstick[i]);  
    signal (room);  
  }  
}
```

Solution 2: an odd philosopher picks up first her left and then right chopstick, whereas an even philosopher picks up her right and then left chopsticks

```
semaphore chopstick[5]={1,1,1,1,1};
int i;
void philosopher(int i);
{
    while(1){
        thinking;
        if(i%2!=0){
            wait (chopstick[i]);
            wait (chopstick[(i+1) % 5]);
        }
        else{
            wait (chopstick[(i+1) % 5]);
            wait (chopstick[i]);
        }
        eating;
        signal (chopstick[(i+1) % 5]);
        signal (chopstick[i]);
    }
}
```

Problems using semaphore

- All processes share a semaphore variable mutex (initialized 1)

```
wait (mutex);
...
critical section
...
signal (mutex);
```



```
signal (mutex);
```

```
...
```

```
critical section
```

```
...
```

```
wait (mutex);
```

X

```
wait (mutex);
```

```
...
```

```
critical section
```

```
...
```

```
wait (mutex);
```

X

```
critical section
```

```
...
```

```
signal (mutex);
```

X

```
wait (mutex);
```

```
...
```

```
critical section
```

```
...
```

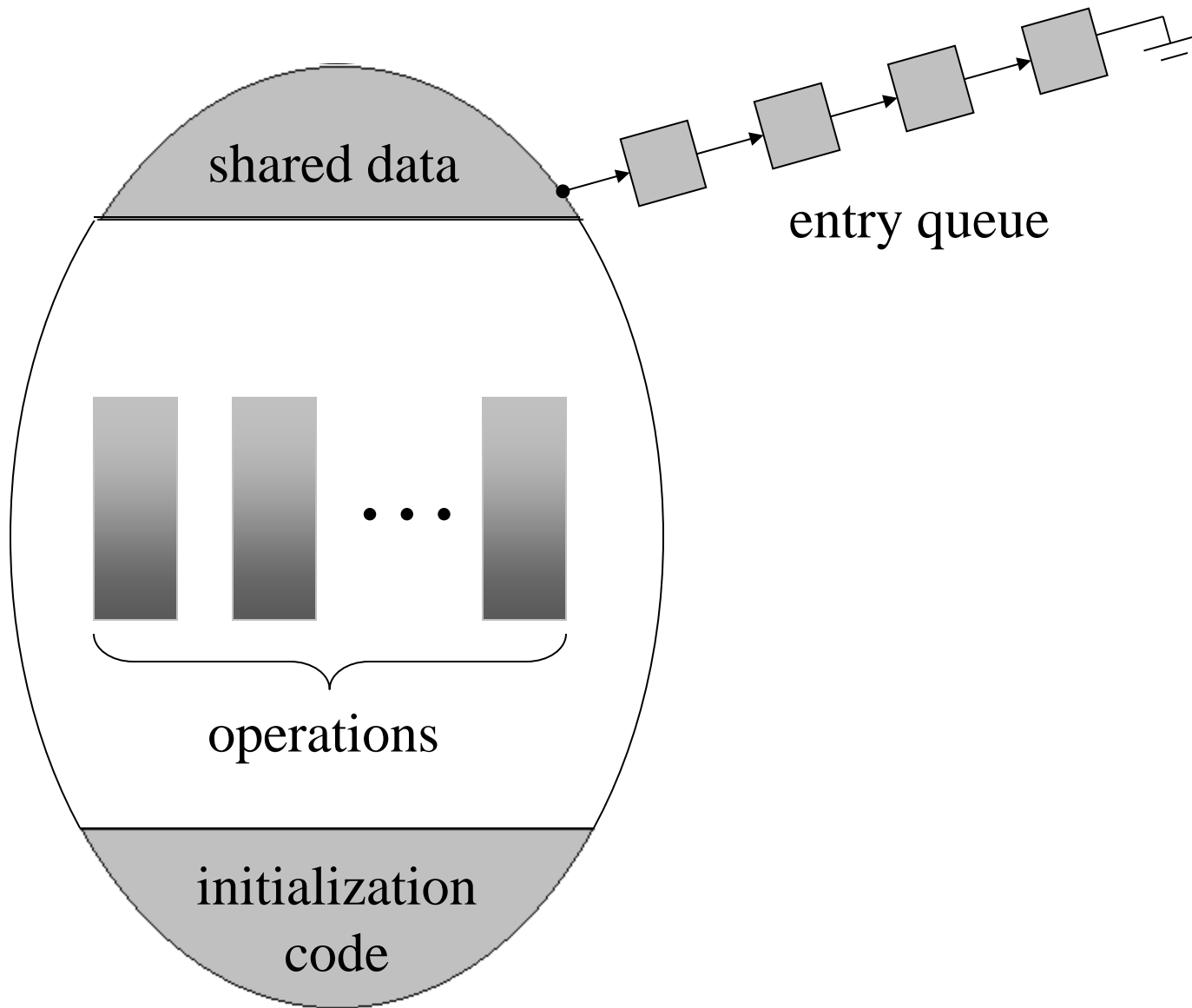
X

6.7 Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- Monitor is a software module
- Chief characteristics
 - Local data variables are accessible only by the monitor
 - Process enters monitor by invoking one of its procedures
 - Only one process may be executing in the monitor at a time

```
monitor monitor-name
{
    shared variable
    declarations
    procedure body P1 (...)
        { ... }
    procedure body P2 (...)
        { ... }
    procedure body Pn (...)
        { ... }
    {
        initialization code
    }
}
```

Schematic View of a Monitor

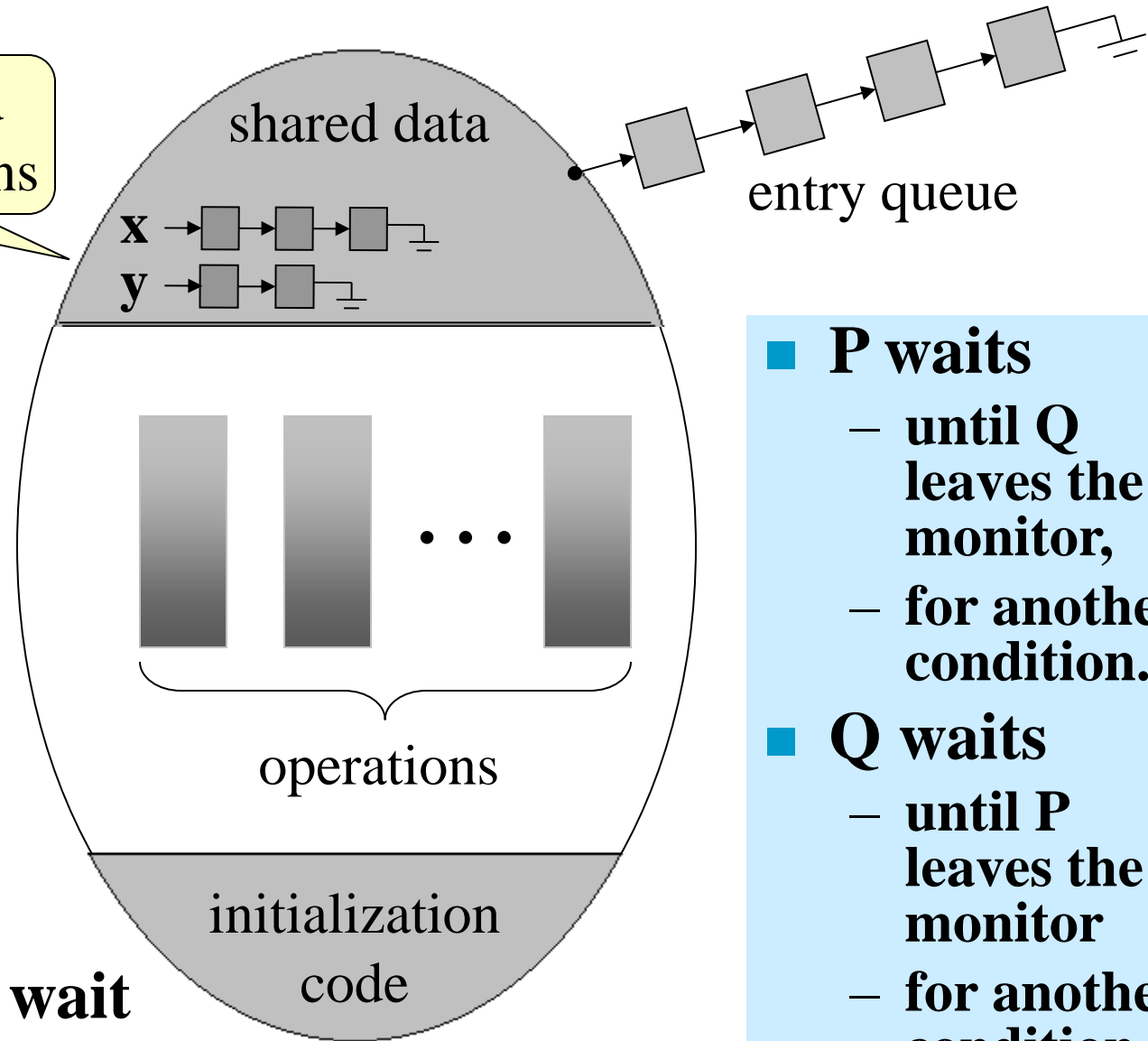


Condition variable and operations on it

- To allow a process to wait within the monitor, a *condition* variable must be declared, as
condition x, y;
- Condition variable can only be used with the operations *wait* and *signal*.
 - The operation
x.wait();
means that the process invoking this operation is suspended until another process invokes
x.signal();
 - The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Monitor With Condition Variables

queue associated
with x, y conditions



Q:
suspended
associated
with x.
P: x.signal()
--- ???

Solution:

- Signal and wait
- Signal and continue

- **P waits**
 - until Q leaves the monitor,
 - for another condition.
- **Q waits**
 - until P leaves the monitor
 - for another condition

Dining-Philosophers Solution Using Monitor

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


Dining Philosophers (Cont.)

```
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}
```

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

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

```
dp.pickup(i);
...
eat
...
dp.putdown;
```

Bounded Producer-Consumer Problem

Monitor boundedbuffer

```
char buffer[N];
int nextin, nextout;
int count;
condition notfull, notempty;
void append (char x)
{
    if (count == N)
        notfull.wait();
    buffer[nextin]=x;
    nextin=(nextin+1)%N;
    count++;
    notempty.signal()
}
```

void take (char x)

```
{
    if (count == 0)
        notempty.wait();
    x=buffer[nextout];
    nextout=(nextout+1)%N;
    count--;
    notfull.signal()
}

{
    nextin=0;
    nextout=0;
    count=0;
}
```

Bounded Producer-Consumer Problem

```
void producer()  
    char x;  
    {  
        while (true)  
        {  
            produce(x);  
            append(x);  
        }  
    }
```

```
void consumer()  
    char x;  
    {  
        while (true)  
        {  
            take(x);  
            consume(x);  
        }  
    }  
Void main()  
    {  
        Parbegin(producer,consumer);  
    }
```

6.8 Synchronization Examples

- **Solaris**
- **Windows XP**
- **Linux**
- **Pthreads**

Homework(P.231)

6.1 6.11 6.16

补充作业

1. 图书馆阅览室管理问题：共有100个座位，读者进入之前要登记，退出之前要注销登记，登记本要互斥使用。用信号量机制解决这个问题。
2. 用信号量机制描述4*100米接力赛。

3. 考虑下面各图中的3个进程 P1, P2, 和 P3.

(1) 图(a)中, 只有 P1 结束后, P2 才可以开始执行, 并且 P2 结束后, P3 才可以开始执行;

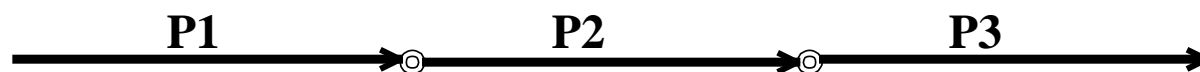


图 (a)

(2) 图(b)中, 只有当 P1 和 P2 都结束后, P3 才可以开始执行;

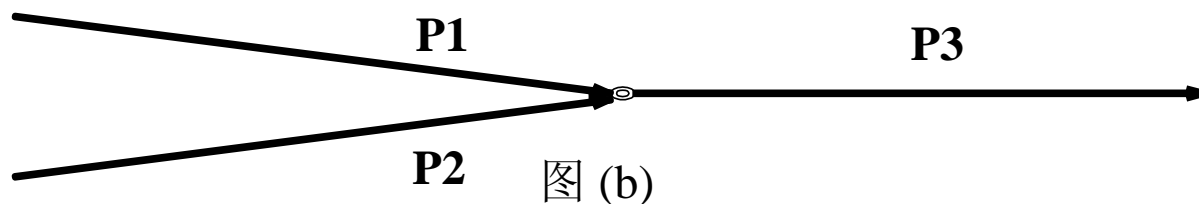


图 (b)

(3) 图(c)中, 只有当 P1结束后, P2 和 P3才可以开始执行.

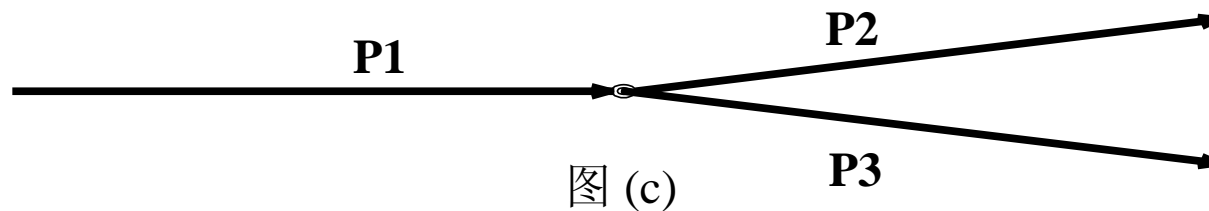


图 (c)

试定义信号量，通过对这些信号量的wait/signal操作实现进程 P1, P2, 和P3 的同步，假设P1、P2、和P3各自的动作如下：

P1:
begin
 S1;
 S2;
end

P2:
begin
 S3;
 S4;
end

P3:
begin
 S5;
 S6;
end

4. 有一个可以存放3个水果的盘子，父亲向盘子中放苹果，母亲向盘子中放桔子，女儿从盘子中取苹果吃，儿子从盘子中取桔子吃。他们只能以互斥的方式操作盘子，并且每次只能 向/从 盘子中 放/取 一个水果。

请设计父亲、母亲、儿子、女儿4个进程，用信号量机制实现他们对盘子的正确操作。

要求：

1. 定义信号量，为信号量赋初值、并说明每个信号量的作用
2. 描述父亲、母亲、儿子、女儿4个进程代码结构

5. 某应用有三个进程，数据采集进程**C**、数据处理进程**M**、和数据输出进程**P**。

- **C**把采集到的数据放到**buf1**中
- **M**从**buf1**中取数据进行处理，并把结果放入**buf2**中
- **P**从**buf2**中取出结果打印输出

考虑以下两种情况，用信号量机制实现进程**C**、**M**、和**P**之间的同步，分别给出进程**C**、**M**、和**P**的代码结构（假设各进程对缓冲区的操作需要互斥）

a) Buf1、buf2都只能保存一个数据

b) Buf1可以保存m个数据、buf2可以保存n个数据



Supplement: Operating System Concerns

- **Keep track of active processes**
- **Allocate and deallocate resources**
 - **Processor time**
 - **Memory**
 - **Files**
 - **I/O devices**
- **Protect data and resources**
- **Result of process must be independent of the speed of execution of other concurrent processes**

Supplement: Process Interaction

- **Processes unaware of each other**
 - **Competition Among Processes for Resources**

- **Processes indirectly aware of each other**
 - **Cooperation Among Processes by Sharing**

- **Process directly aware of each other**
 - **Cooperation Among Processes by Communication**

Supplement:

Competition Among Processes for Resources

■ **Mutual Exclusion**

– **Critical sections**

- **Only one program at a time is allowed in its critical section**
- **Example only one process at a time is allowed to send command to the printer**

■ **Deadlock**

■ **Starvation**

Supplement:

Cooperation Among Processes by Sharing

- **Writing must be mutually exclusive**
- **Critical sections are used to provide data integrity**

Supplement:

Cooperation Among Processes by Communication

- **Messages are passed**
 - **Mutual exclusion is not a control requirement**
- **Possible to have deadlock**
 - **Each process waiting for a message from the other process**
- **Possible to have starvation**
 - **Two processes sending message to each other while another process waits for a message**

Supplement: Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,4,5...

Supplement: Bakery Algorithm (Cont.)

- Notation \leq lexicographical order (ticket #, pid #)
 - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
 - $\max(a_0, \dots, a_{n-1})$ is a number, k , such that $k \geq a_i$ for $i = 0, \dots, n - 1$

- Shared data

boolean choosing[n];

int number[n];

Data structures are initialized to false and 0 respectively

Supplement: Bakery Algorithm (Cont.)

```

do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number [n-1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((number[j] != 0) && ((number[j],j) < (number[i],i)) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);

```

Chapter 7 Deadlocks



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

Deadlock Characterization

Methods for Handling Deadlocks

Deadlock Prevention, Avoidance , Detection and Recovery

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.**

Contents

Deadlock model?

- 7.1 System Model
- 7.2 Deadlock Characterization
- 7.3 Methods for Handling Deadlocks
- 7.4 Deadlock Prevention
- 7.5 Deadlock Avoidance
- 7.6 Deadlock Detection
- 7.7 Recovery from Deadlock

7.1 System Model

- **Resource types R_1, R_2, \dots, R_m**
CPUs, CPU cycles, memory space, files, I/O devices (printers)
- **The resources may be either physical resources or logical resources.**
- **Each resource type R_i has W_i instances.**
- **Each process utilizes a resource as follows:**
 - request
 - use
 - Release
- **A system table records whether each resource is free or allocated, and if a resource is allocated, to which process.**

7.2 Deadlock Characterization

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .
- A deadlock situation can arise if the above four conditions hold simultaneously.

Resource-Allocation Graph

- **Deadlock can be described in terms of a System resource-allocation graph**
 - Directed graph
 - A set of vertices V and a set of edges E .
- **V is partitioned into two different 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.
- **E is classed into two different types:**
 - request edge – directed edge $P_i \rightarrow R_j$, Process P_i requested an instance of resource type R_j , and is currently waiting for that resource.
 - assignment edge – directed edge $R_j \rightarrow P_i$, An instance of resource type R_j has been allocated to process P_i .

Resource-Allocation Graph (Cont.)

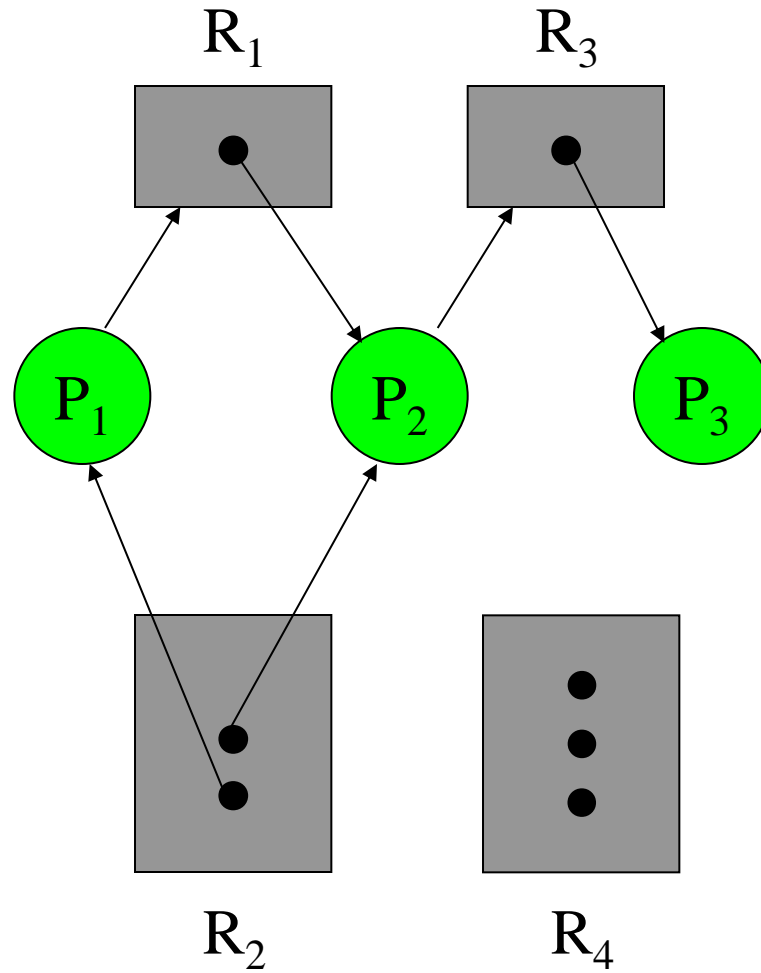
■ Process 

■ Resource Type with 4 instances 

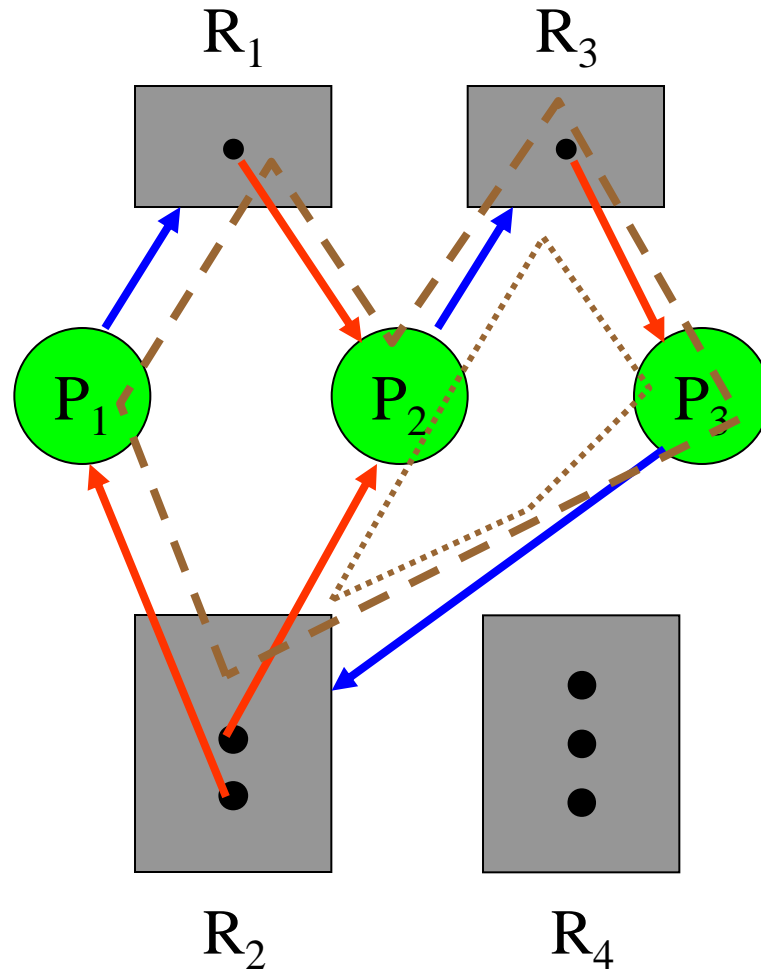
■ P_i requests instance of R_j 

■ P_i is holding an instance of R_j 

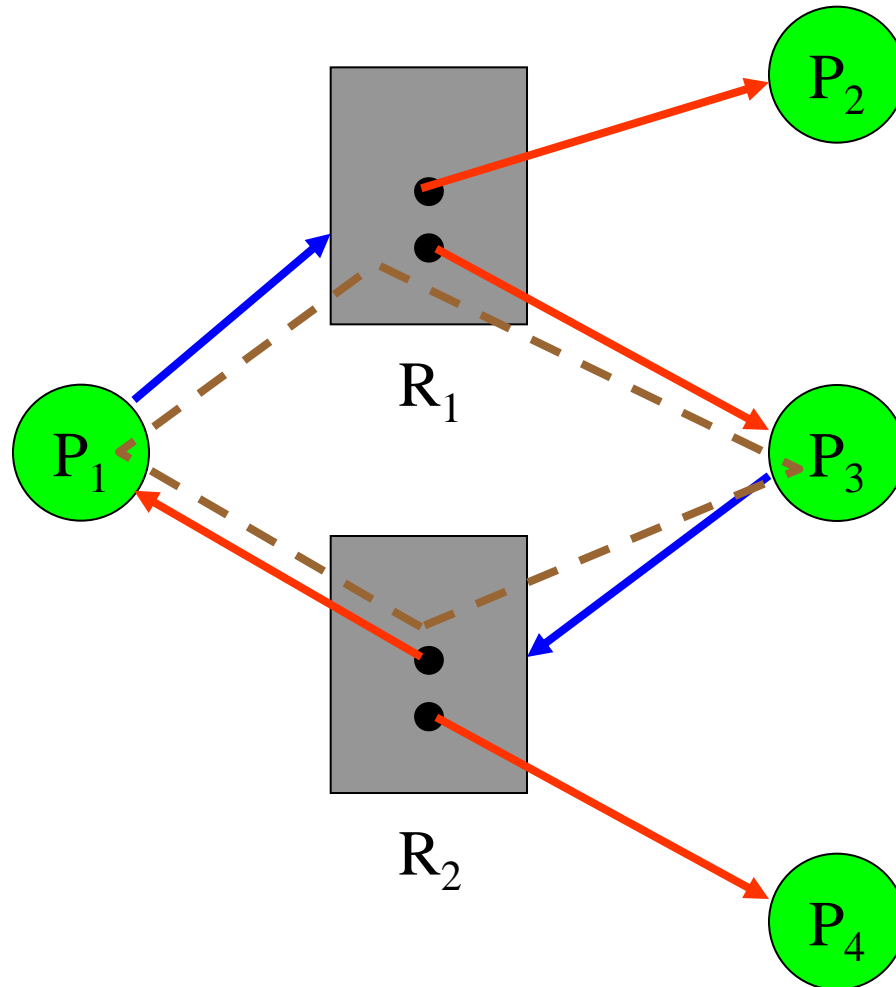
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

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

7.3 Methods for Handling Deadlocks

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

7.4 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. Two alternations:
 - Require process to request and be allocated all its resources before it begins execution
 - allow process to request resources only when the process has none (release resources first).
- Disadvantages:** Low resource utilization; Starvation possible.

Deadlock Prevention (Cont.)

■ No Preemption – two alternations

- **Waiting with no resources**
 - 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.
- **Waiting with holding resources, but resources may be preempted by other process requesting them.**

Often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. Two alternations:
 - The process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. If several instances of the same resource type are needed, a single request for all of them must be issued.
 - Whenever a process requests an instance of resource type R_j , it has released any resources R_i such that $F(R_i) \geq F(R_j)$

7.5 Deadlock Avoidance

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

Processes under consideration must be independent;
No synchronization requirements.

- 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.

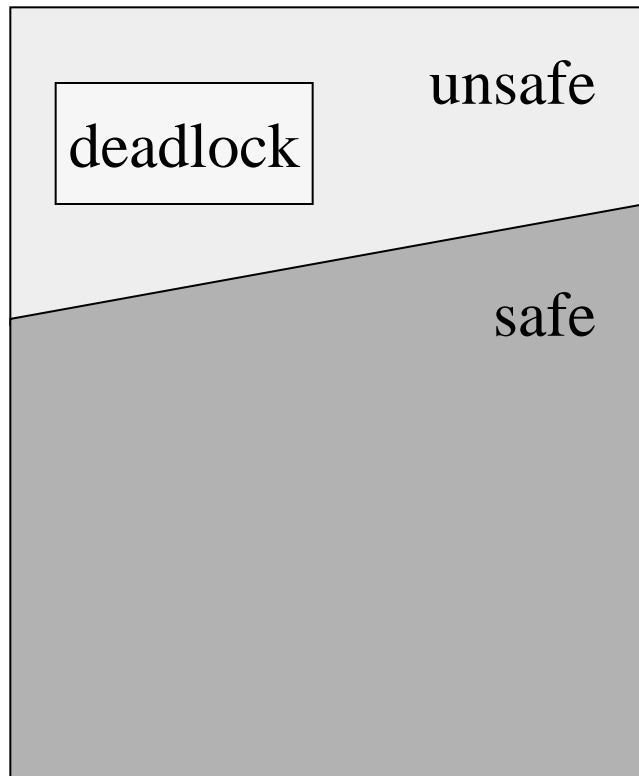
Two Approaches to Deadlock Avoidance

- **Do not start a process if its demands might lead to deadlock**
- **Do not grant an incremental resource request to a process if this allocation might lead to deadlock**

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 **safe sequence** of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if 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 $j < i$.
 - If the resources that P_i 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 its 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.

Total: 12

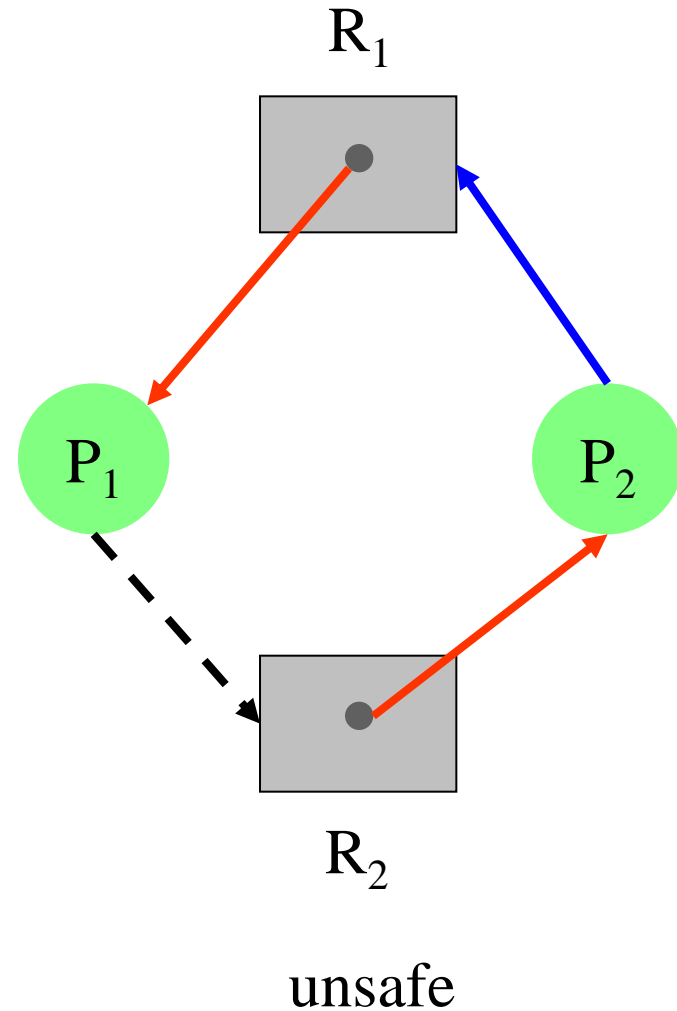
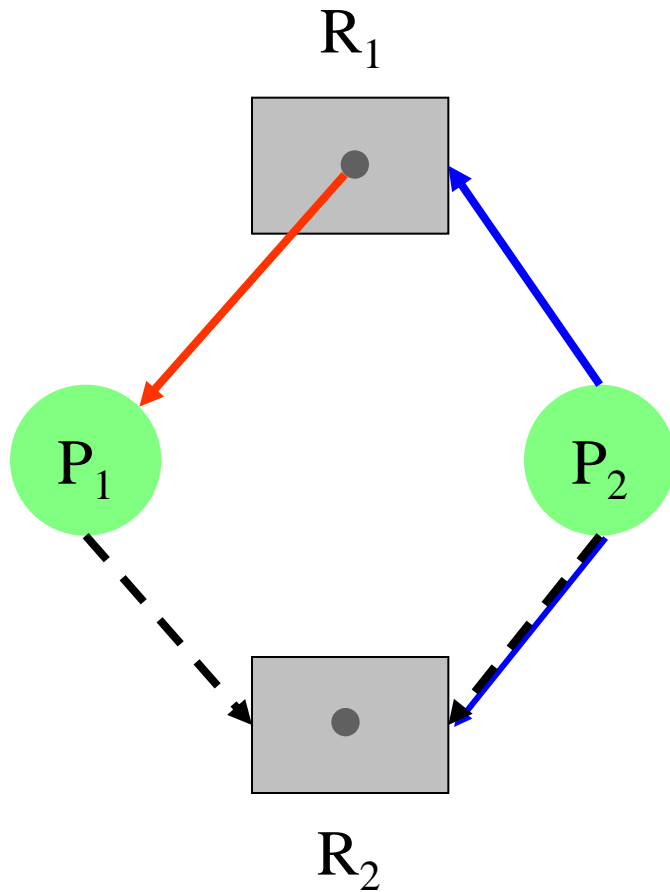
	Max needs	current holding
P_0	10	5
P_1	4	2
P_2	9	3

- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.

Resource-Allocation Graph Algorithm

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

Resource-Allocation Graph Algorithm (Cont.)



Banker's Algorithm

- **Applicable to a resource-allocation system with multiple instances of each resource type.**
- **Each process must a priori claim maximum use of instances of each resource type.**
- **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

n = number of processes

m = number of resources types

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

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

notation

- Let X and Y be vectors of length n
 $X \leq Y \iff X[i] \leq Y[i] \text{ for all } i=1,2,\dots,n$
- Each row in the matrices *allocation* and *need* can be treated as vectors and referred to as *allocation_i* and *need_i*
 - *Allocation_i* specifies the resources currently allocated to process P_i
 - *Need_i* specifies the additional resources that process P_i may still request to complete its task.

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize:
Work = *Available*
Finish [i] = *false* for $i = 1, 2, \dots, n$.
2. Find an i such that both:
(a) *Finish* [i] = *false*
(b) $Need_i \leq Work$
If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
Finish [i] = *true*
go to step 2.
4. If *Finish* [i] == *true* for all i , then the system is in a safe state.

Resource-Request Algorithm for Process P_i

- **$Request_i$** = request vector for process P_i .
 - If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j .
- When a request for resource is made by process P_i ,
 1. If **$Request_i \leq Need_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
 2. If **$Request_i \leq Available$** , go to step 3. Otherwise P_i must wait, since resources are not available.
 3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request_i;$ $Allocation_i = Allocation_i + Request_i;$ $Need_i = Need_i - Request_i;$
 4. 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

- A system with
 - 5 processes P_0 through P_4 , and
 - 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>		<u>Need</u>	
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$		$A \ B \ C$	
P_0	0 1 0	7 5 3	3 3 2	7 5 5	P_0 7 4 3	P_1
P_1	2 0 0	3 2 2	+	5 3 2	P_1 1 2 2	P_3
P_2	3 0 2	9 0 2		10 5 7	P_2 6 0 0	P_4
P_3	2 1 1	2 2 2		7 4 3	P_3 0 1 1	P_0
P_4	0 0 2	4 3 3		7 4 5	P_4 4 3 1	P_2

Example P_1 Request (1,0,2) (Cont.)

- Suppose $\text{request}_1 = (1,0,2)$.
- Check that $\text{Request}_1 \leq \text{Need}_1$ (that is, $(1,0,2) \leq (1,2,2) \Rightarrow \text{true}$.
Check that $\text{Request}_1 \leq \text{Available}$ (that is, $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$.
- Pretend this request has been fulfilled. New state:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>		
	A B C	A B C	A B C		
P_0	0 1 0	7 4 3	2 3 0	P_1	5 3 2
P_1	3 0 2	0 2 0		P_3	7 4 3
P_2	3 0 2	6 0 0		P_4	7 4 5
P_3	2 1 1	0 1 1		P_0	7 5 5
P_4	0 0 2	4 3 1		P_2	10 5 7

- Determine whether this new state is safe.

Example P_1 Request (1,0,2) (Cont.)

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.

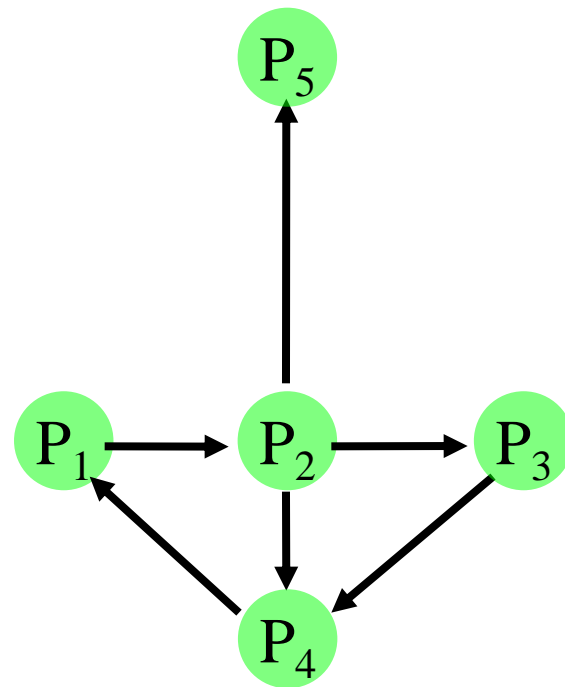
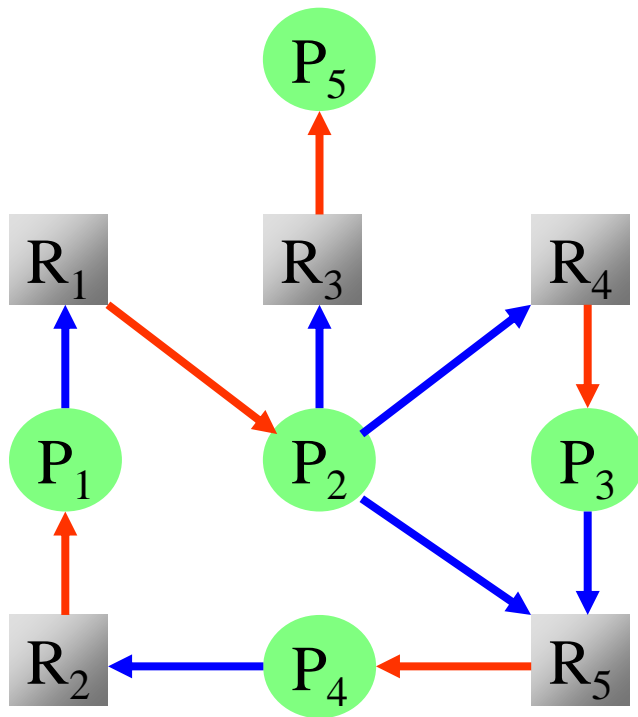
	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Can request for (3,3,0) by P_4 be granted?
 - Available=(2,3,0) < request (3,3,0)
- Can request for (0,2,0) by P_0 be granted?

7.6 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 to release a resource that P_i needs.
- $P_i \rightarrow P_j$ exists in wait-for graph $\iff \exists q, P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ exist in resource-allocation graph.

Single Instance of Each Resource Type

- a deadlock exists in the system \leftrightarrow the wait-for graph contains a cycle
- To detect deadlocks, the system needs
 - maintain the wait-for graph
 - Periodically invoke an algorithm that searches for a cycle in the graph.
- 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.

Several Instances of a Resource Type

Several time-varying data structures used

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

Rows in **Allocation** and **Request** can be treated as vectors, and referred to as **Allocation_i** and **Request_i** respectively

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 $Allocation_i \neq 0$, then $Finish[i] = \text{false}$; otherwise, $Finish[i] = \text{true}$.
2. Find an index i such that both:
 - (a) $Finish[i] == \text{false}$
 - (b) $Request_i \leq Work$
 If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$ $Finish[i] = \text{true}$
go to step 2.
4. If $Finish[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system is in a deadlock state.
Moreover, if $Finish[i] == \text{false}$, then P_i is deadlocked.

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

Example of Detection Algorithm

- A system with 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>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Seq.	Work
	0 0 0
P_0	0 1 0
P_2	3 1 3
P_3	5 2 4
P_4	5 2 6
P_1	7 2 6

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

Example (Cont.)

- Suppose P_2 requests an additional instance of type C .

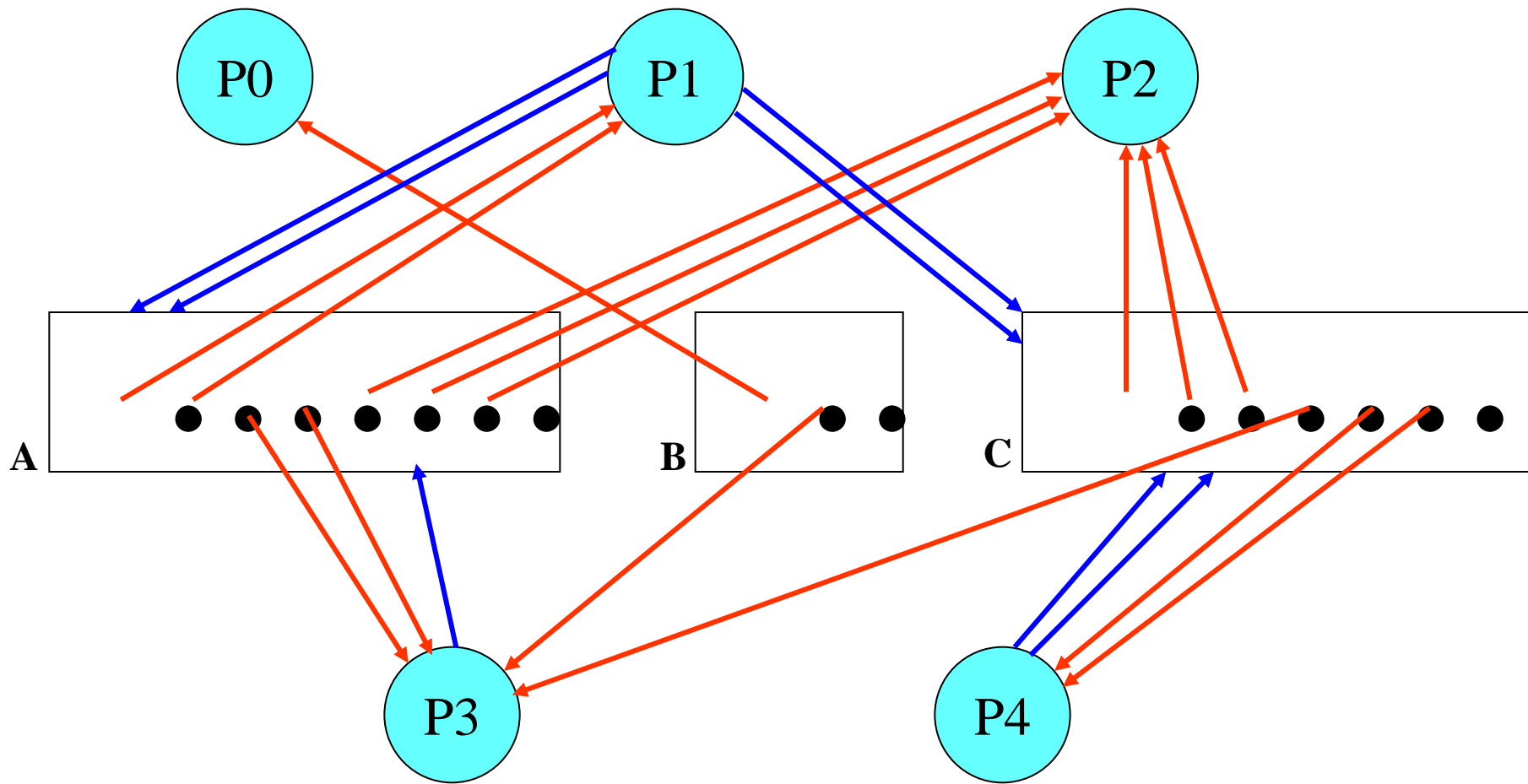
	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- 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

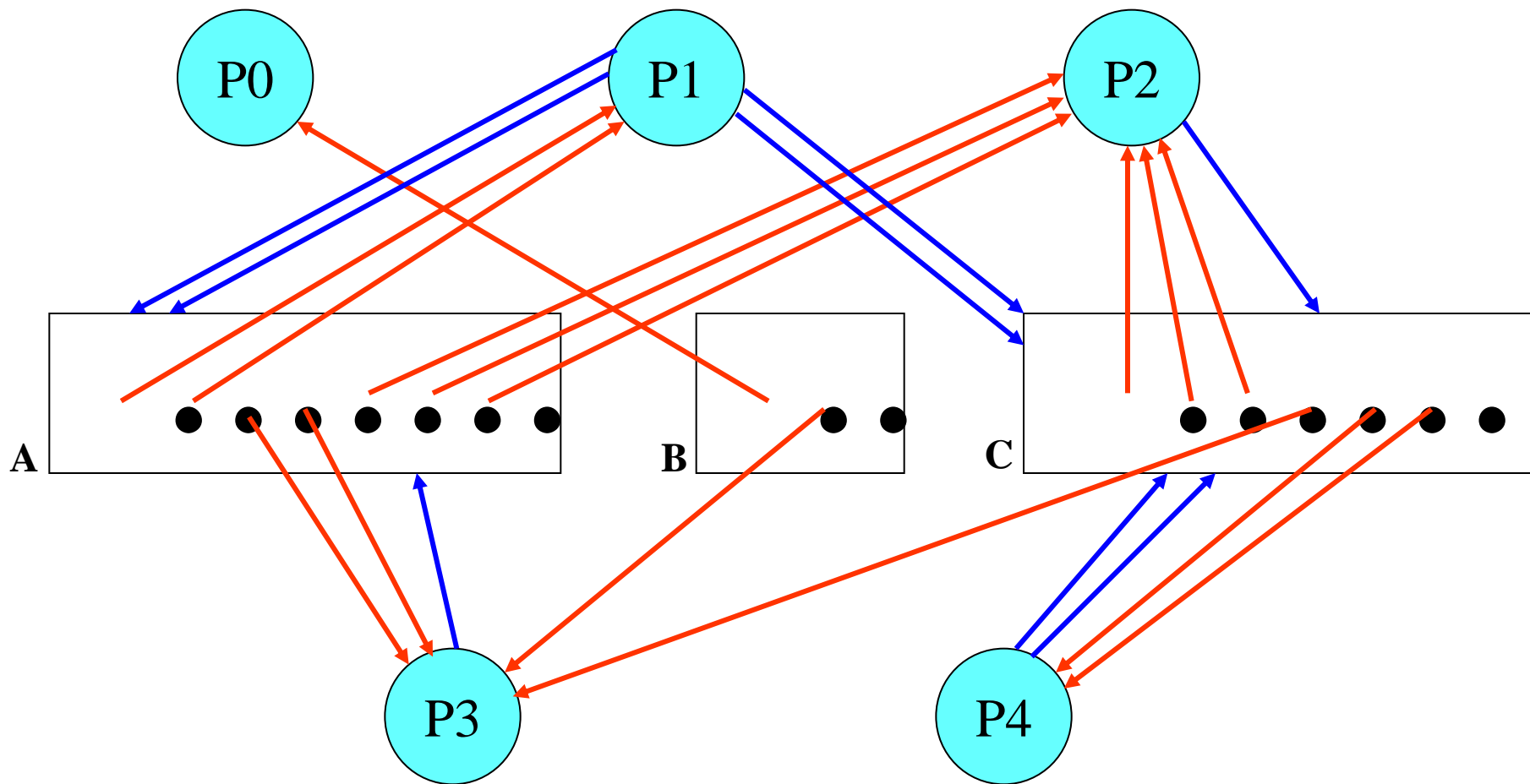
- **When, and how often, to invoke depends on:**
 - **How often a deadlock is likely to occur?**
 - **How many processes will need to be rolled back?**
 - **one for each disjoint cycle**
- **If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would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

Suspended: 资源图消边法检测死锁



 request edge
 assignment edge

Suspended: 资源图消边法检测死锁



 request edge
 assignment edge

7.7 Recovery from Deadlock

----- Process Termination

- **To eliminate deadlock by aborting a process**
 - Abort all deadlocked processes.
 - Abort one process at a time until the deadlock cycle is eliminated.

- **In which order should we choose to abort?**
 - Priority of the process.
 - How long the process has computed, and how much longer to completion.
 - How much and what type of resources the process has used.
 - How much more resources the process needs to complete.
 - How many processes will need to be terminated.
 - Is the process interactive or batch?

Recovery from Deadlock (Cont.)

----- Resource Preemption

- **Selecting a victim** – which resources and which processes are to be preempted?
Minimize cost.
- **Rollback** – return the process to some safe state, and restart it from that state.
Abort the process and then restart it.
- **Starvation** – the same process may always be picked as victim.
Include the number of rollbacks in cost factor.

Combined Approach to Deadlock Handling

- **Combine the three basic approaches**

- prevention
- avoidance
- detection

allowing the use of the optimal approach for each of resources in the system.

- **Partition resources into hierarchically ordered classes.**

- **Use most appropriate technique for handling deadlocks within each class.**

Homework (page 268)

7.5

7.6

7.11

Thinking:

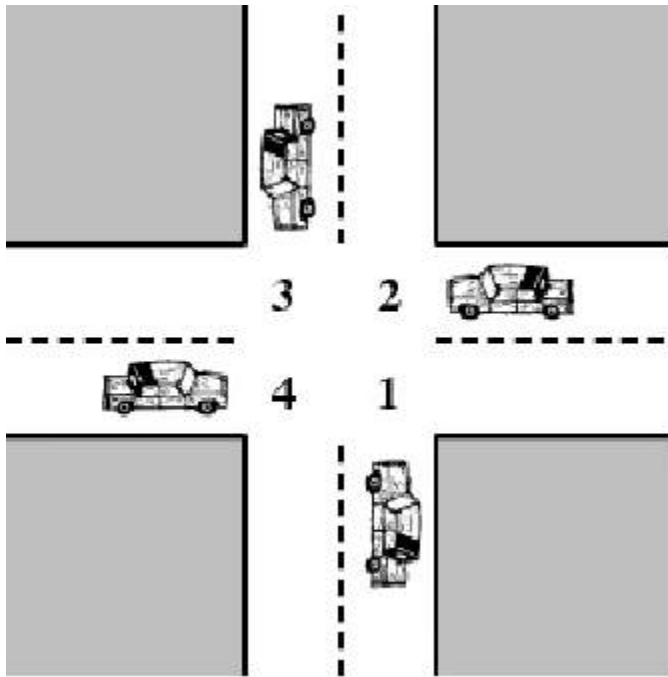
7.1

7.7

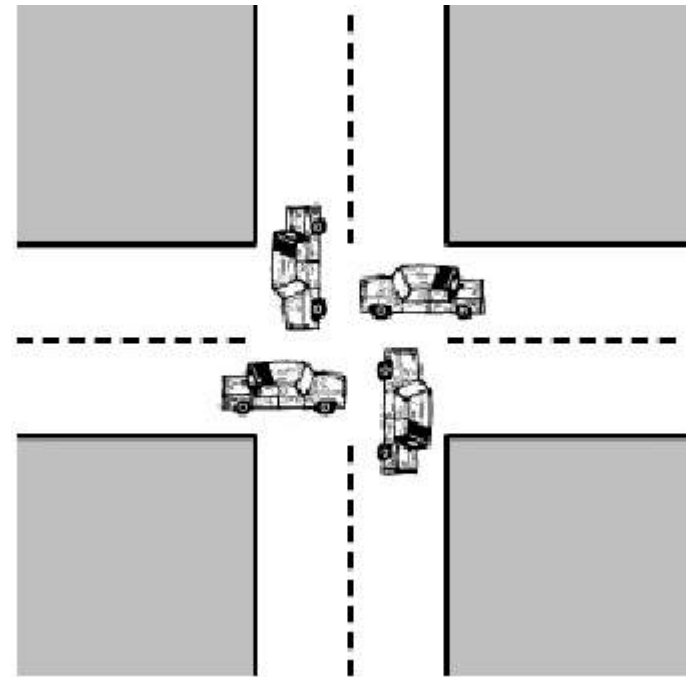


Suspended: Deadlock Model

Illustration of Deadlock (intersection)

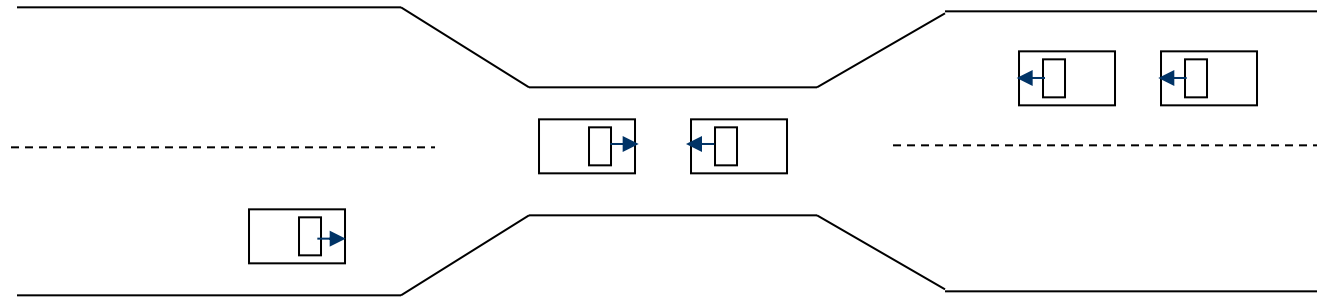


Deadlock possible



Deadlock

Suspended: Bridge Crossing Example



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

Suspended: 产生死锁的原因

- 竞争资源
- 进程间推进顺序

Suspended: 系统资源与死锁

■ 系统中的永久性资源

– 可剥夺资源（可重用资源）

- 当进程所占有的并且正在使用的资源被剥夺走时，对进程不产生破坏性的影响，如：内存、CPU
- 对于可剥夺资源，通过资源的重新分配，很容易恢复进程的执行

– 不可剥夺资源（非剥夺性资源）

- 当进程所占有的并且正在使用的资源被剥夺走时，可能引起进程执行失败，如打印机。
- 死锁主要是在共享这类资源时产生。

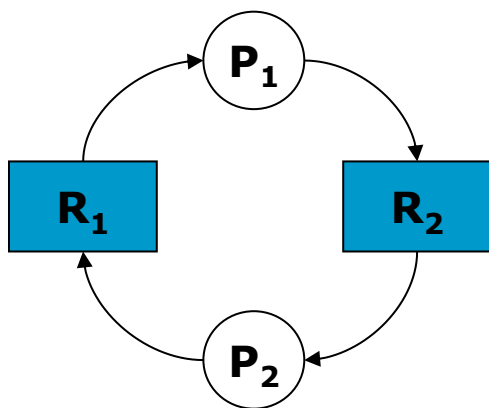
■ 临时性资源

- 由一个进程产生，被另一个进程使用一短暂时间后便无用的资源，也称为消耗性资源。
- 竞争临时性资源也可能引起死锁。

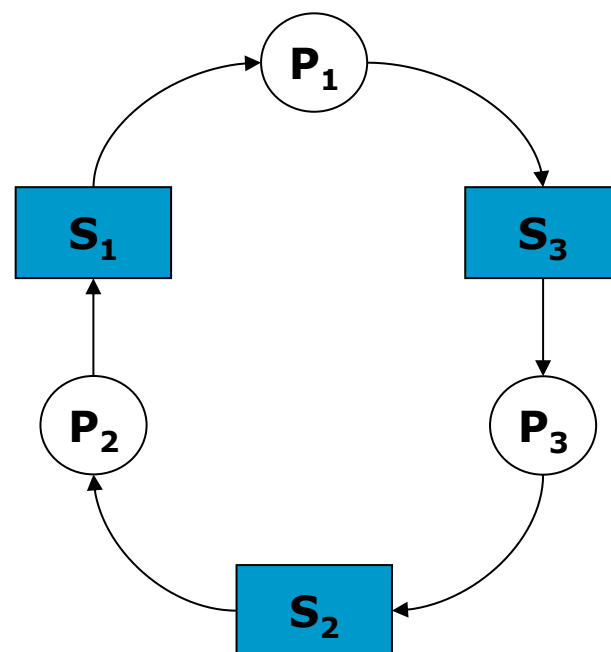
Suspended: 举例

■ 争用I/O设备时的死锁

- 系统有2台磁带机
- P_1 和 P_2 都需要同时使用这两台磁带机进行数据处理
- P_1 和 P_2 分别占有其中的一台, 并且又都需要另一台



■ 进程之间通信时的死锁



Suspended:

The Deadlock Problem in Computer System

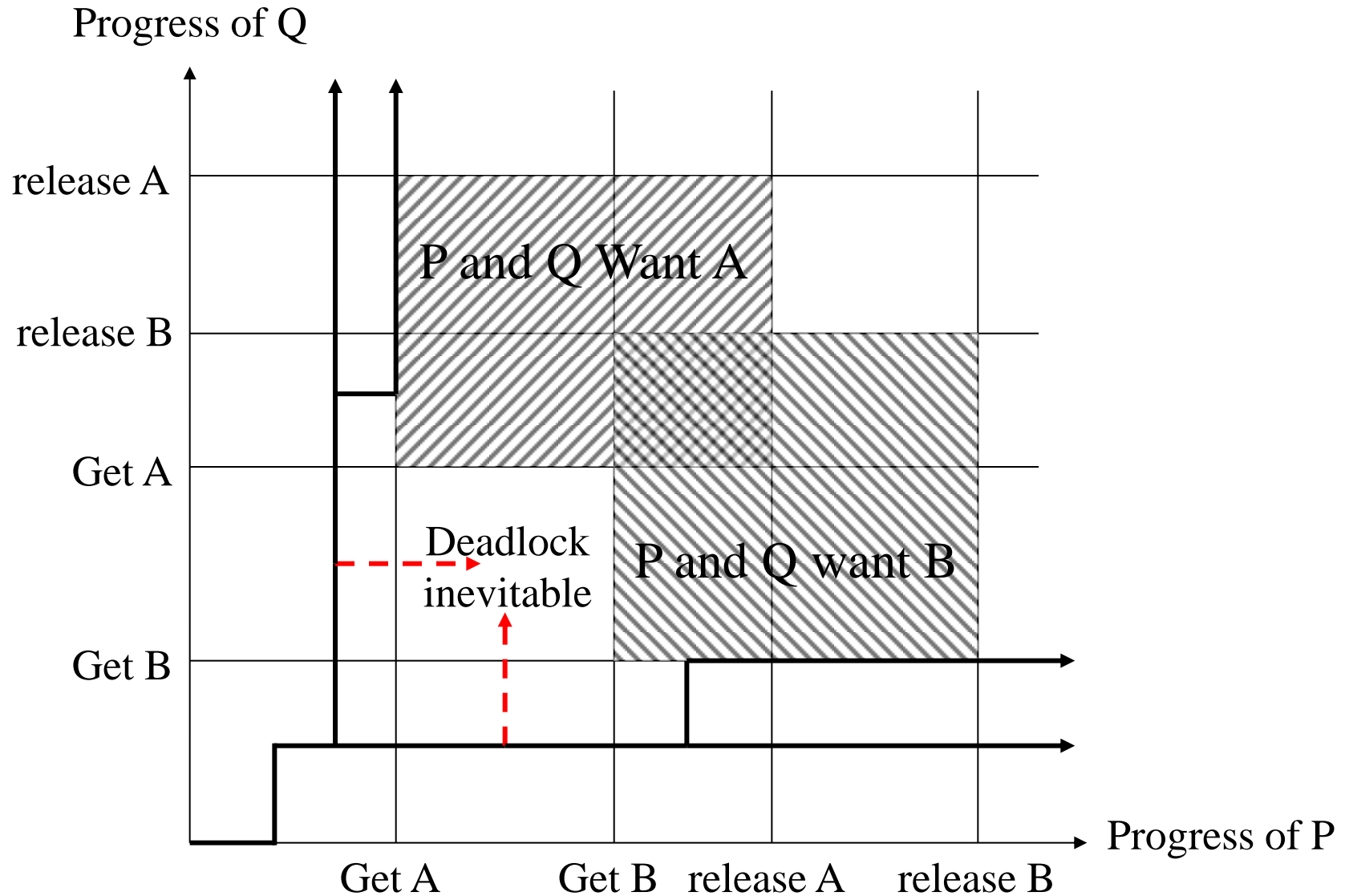
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example 1
 - System has 2 tape drives.
 - P_1 and P_2 each hold one tape drive and each needs another one.
- Example 2
 - semaphores A and B , initialized to 1

P_0	P_1
<i>wait (A);</i>	<i>wait(B)</i>
<i>wait (B);</i>	<i>wait(A)</i>

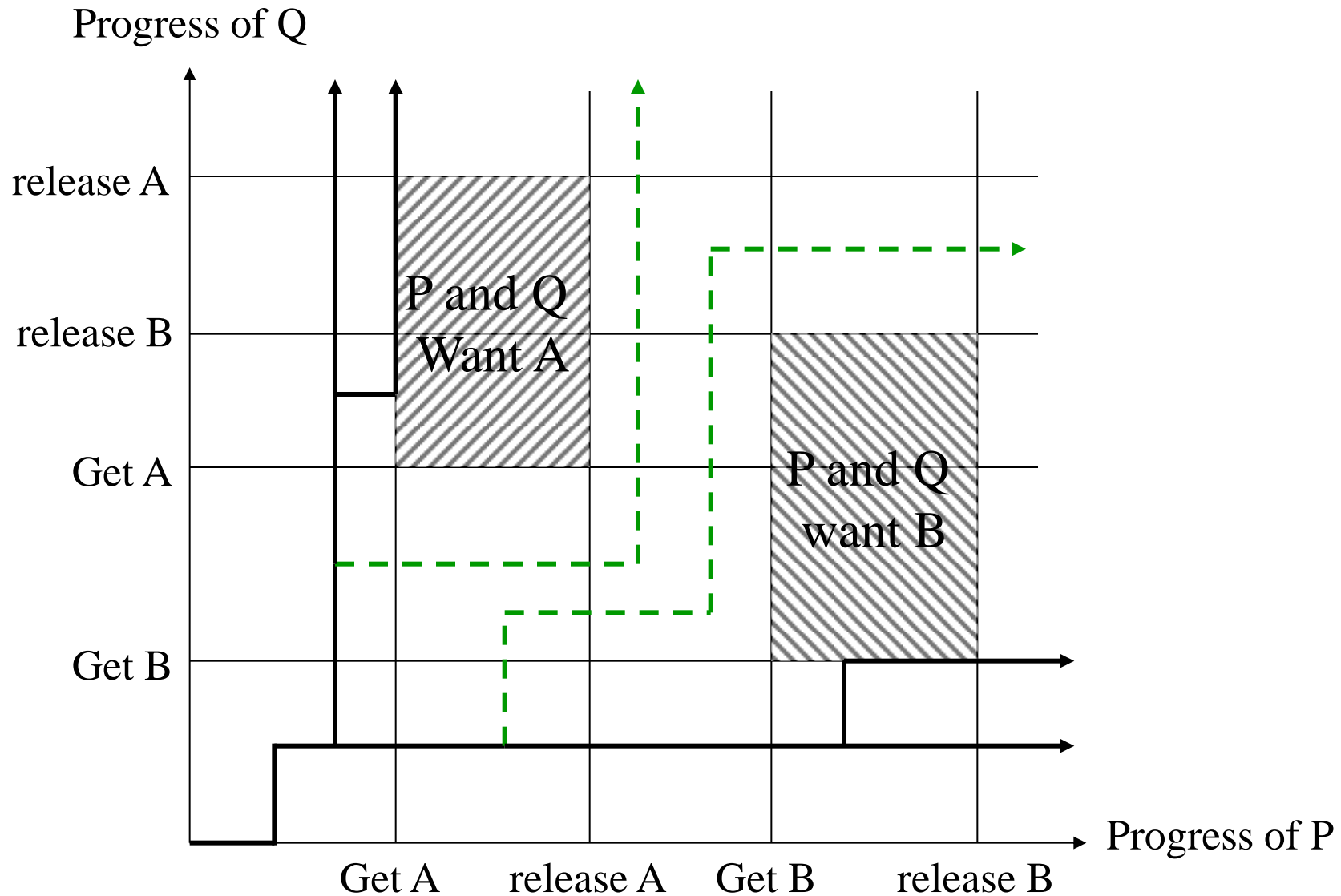
P_0 : wait (A);
 P_1 : wait (B);

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

Suspended: Example of Deadlock



Suspended: Example of No Deadlock



Suspended: **Deadlock**

- **Permanent blocking of a set of processes that either compete for system resources or communicate with each other**
- **Involve conflicting needs for resources by two or more processes**
- **No efficient solution**

Chapter 8 Main Memory



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

Paging

Segmentation

Chapter Objectives

- To provides 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.

Contents



**About Memory
Management?**

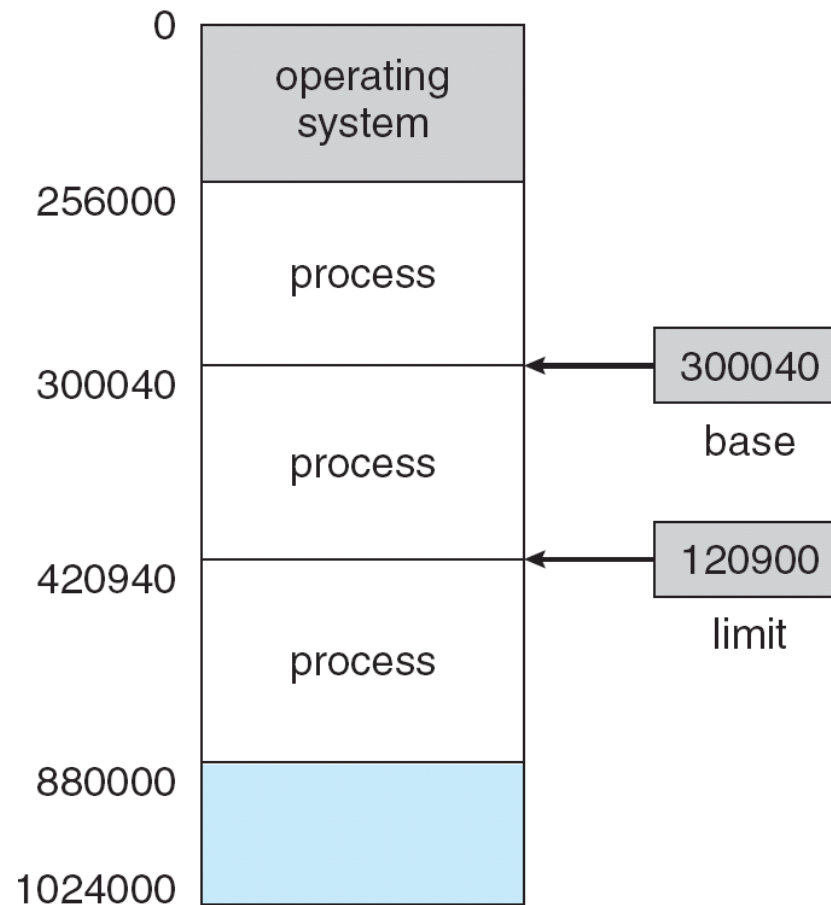
- 8.1 Background**
- 8.2 Swapping**
- 8.3 Contiguous Memory Allocation**
- 8.4 Paging**
- 8.5 Structure of the Page Table**
- 8.6 Segmentation**
- 8.7 Example: The Intel pentium**

8.1 Background

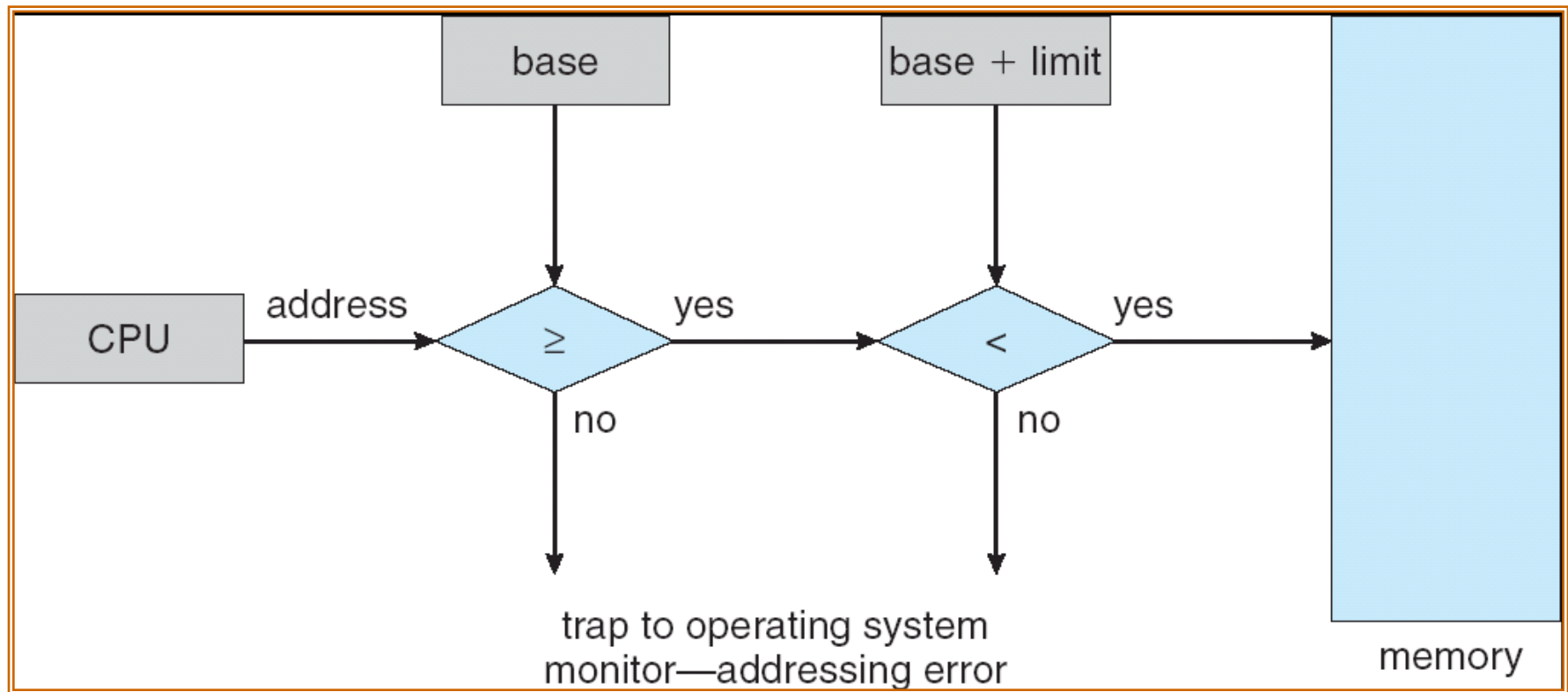
- **Memory consists of a large array of words or bytes, each with its own address.**
- **The program must be brought into memory and placed within a process for it to be run.**
- **Main memory and registers are only storage CPU can access directly**
- **Register access in one CPU clock (or less)**
- **Main memory can take many cycles of the CPU clock**
- **Cache sits between main memory and CPU registers**
- **Protection of memory required to ensure correct operation**

Protection: Base and Limit Registers

- A pair of **base** and **limit** registers define the physical address space

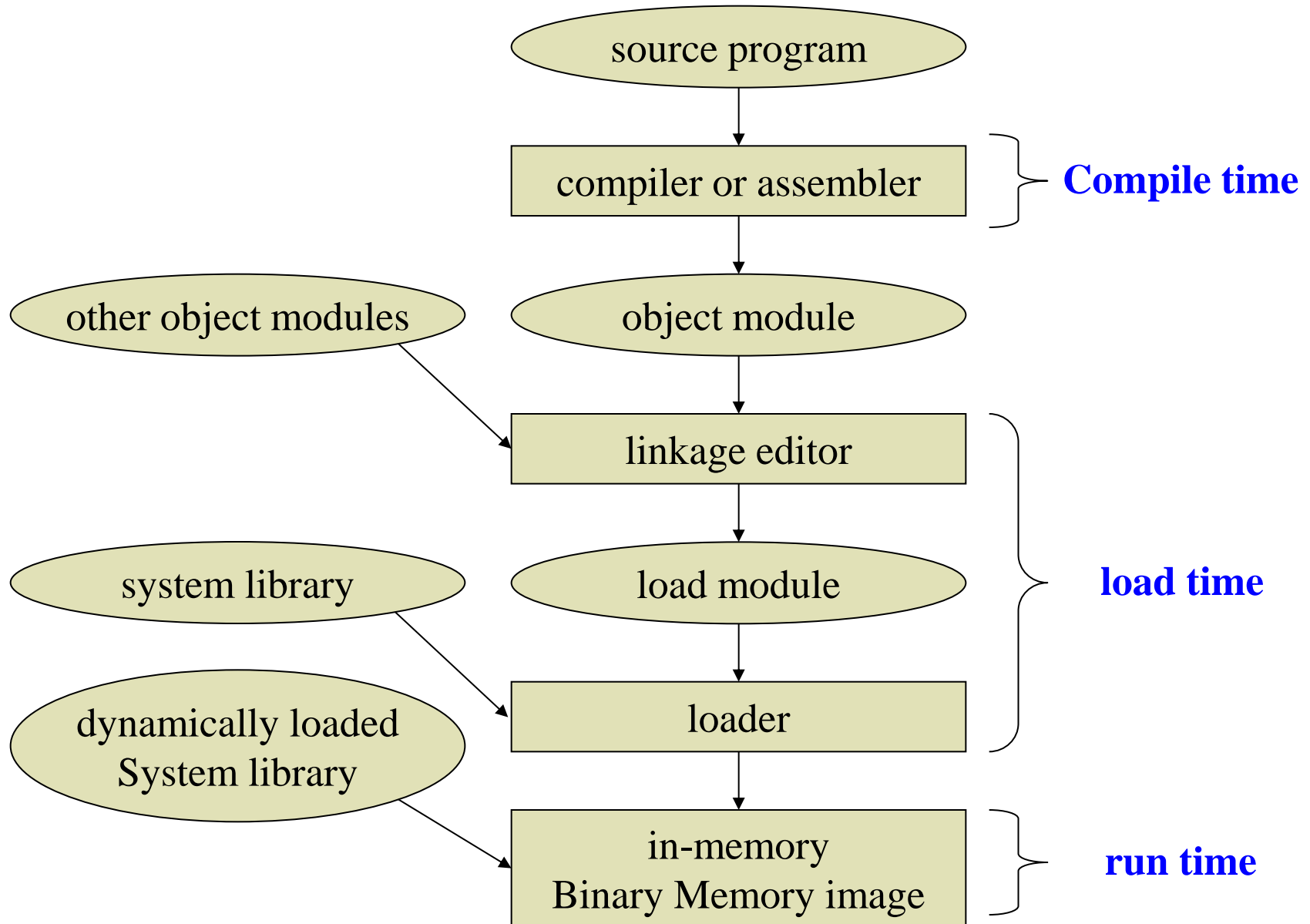


Protection: Base and Limit Registers

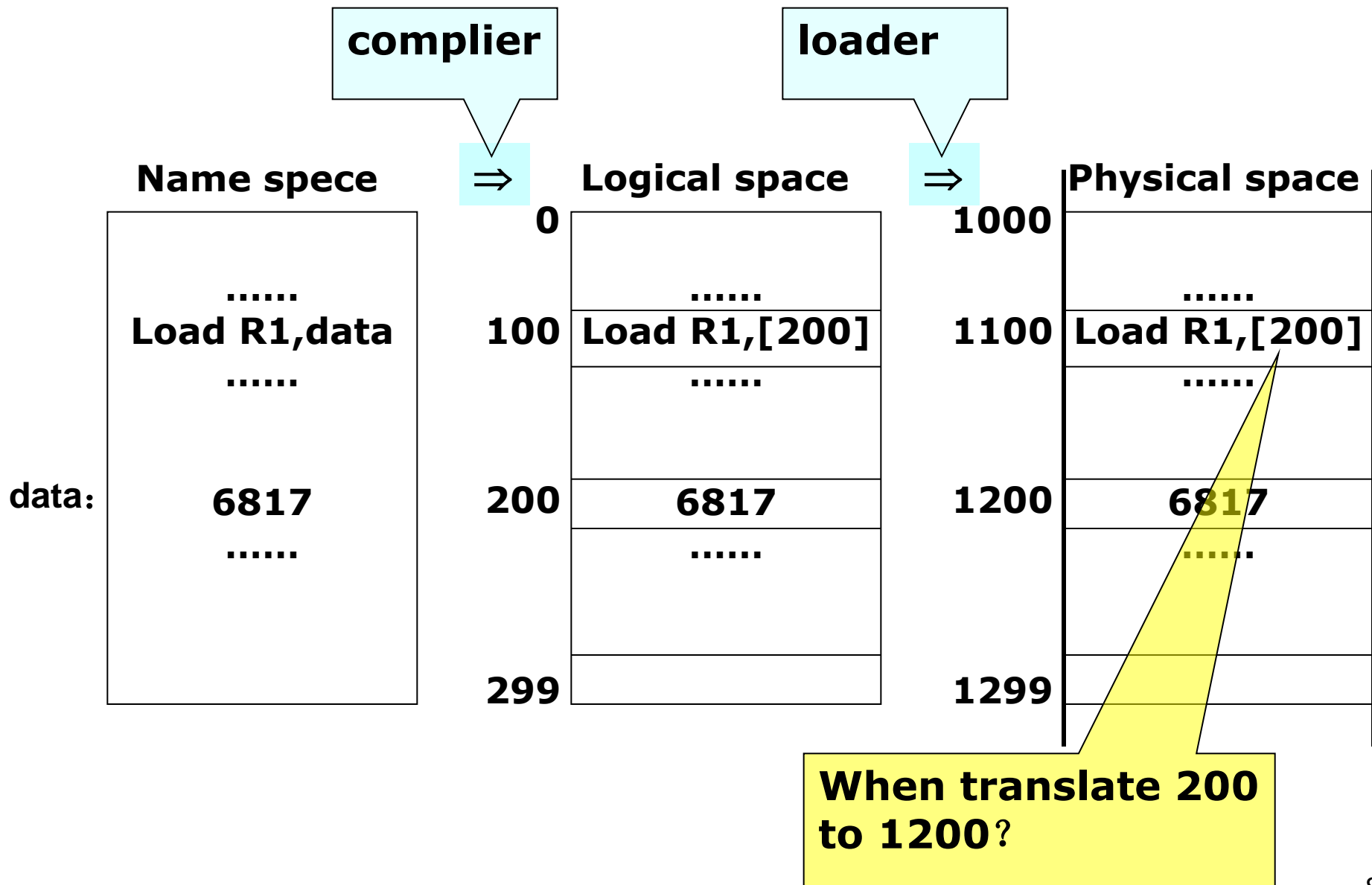


- The base and limit registers can be loaded only by the operating system, which uses a special privileged instructions.

Multistep Processing of a User Program



Address Binding



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 must be 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*).

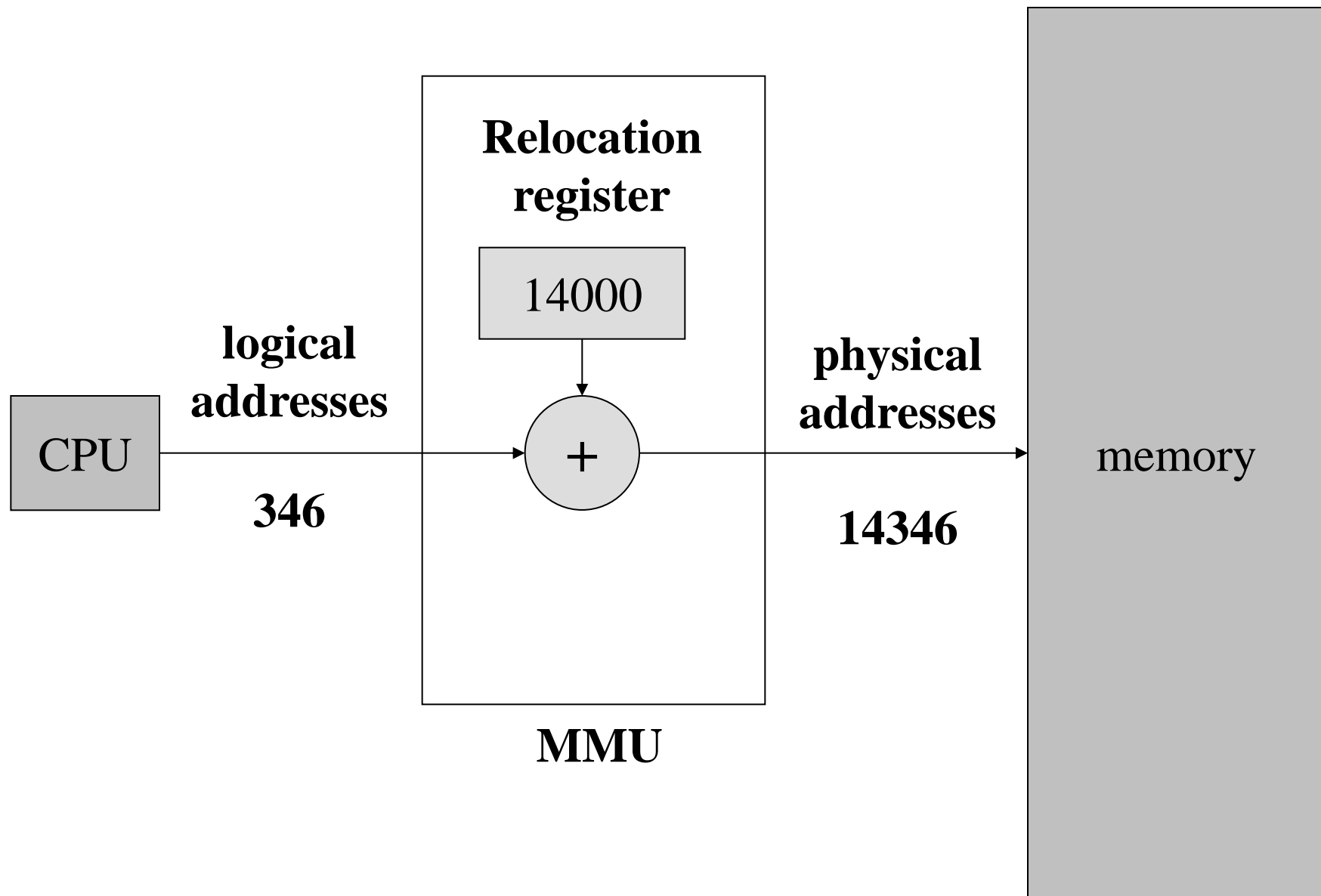
Logical vs. Physical Address Space

- 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; loaded into the memory-address register.
 - **Logical address space**: The set of all logical addresses generated by a program
 - **Physical address space**: the set of all physical addresses corresponding to these logical addresses.
- 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.

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address in run-time.
- In MMU scheme, the value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with **logical** addresses; it never sees the **real** physical addresses.

Dynamic relocation using a relocation register



* Dynamic Loading

- Routine is not loaded until it is called.
- Main program is loaded into memory and is executed, all routines are kept on disk in a relocatable load format.
- When needed, the **relocatable linking loader** is called.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
- No special support from the operating system is required, implemented through program design.

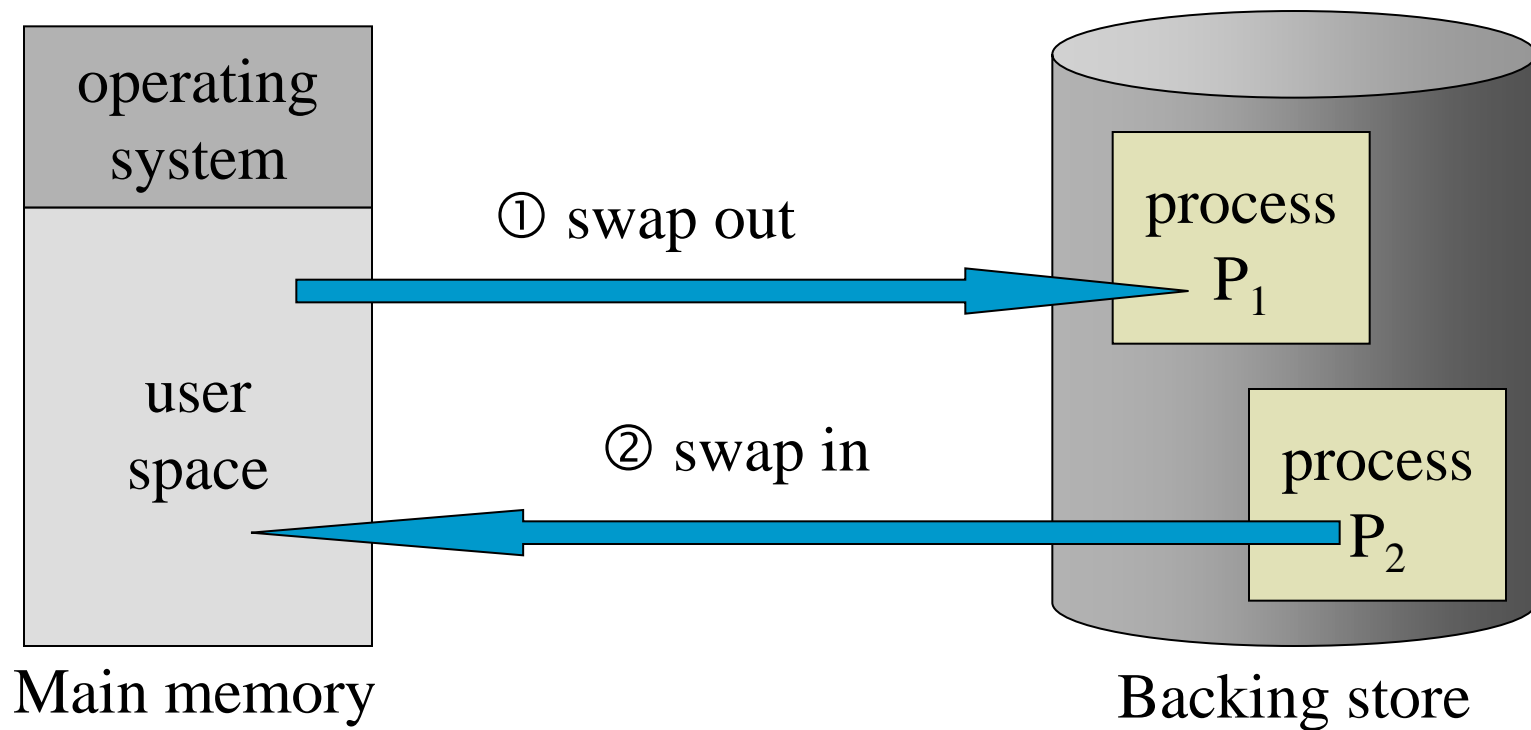
* Dynamic Linking

- Linking postponed until execution time.
- A *stub* is included in the image for each library-routine reference.
- *stub*, Small piece of code, used to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
- When executed, the *stub* replaces itself with the address of the routine, and executes the routine.
- Dynamic linking generally requires help from the operating system (Operating system needed to check if routine is in processes' memory address).
- Dynamic linking is particularly useful for libraries.
 - A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
- *shared libraries*

8.2 Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- 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.
- *ready queue* — Consists of all processes whose memory images are on the backing store or in memory and are ready to run.
- *Dispatcher* is called whenever the CPU scheduler decides to execute a process.

Swapping using a backing store



- a process that is swapped out will be swapped back into the same memory space that it occupied previously?
- The context-switch time is fairly high.

Swapping (Cont.)

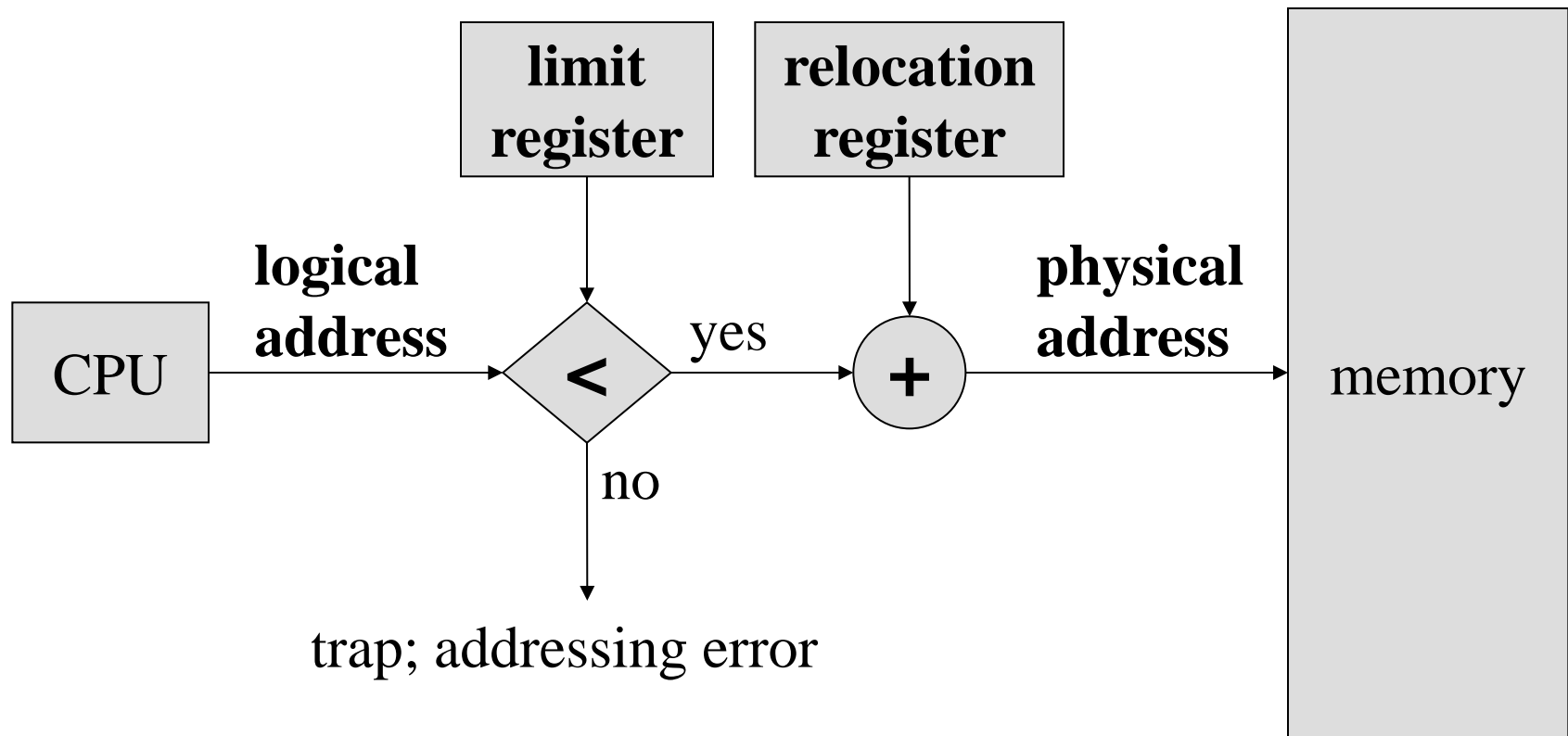
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
 - be useful to know exactly how much memory a user process is using, not simply how much it might be using.
 - The user must keep the system informed of any changes in memory requirements.
- Ensure the process we want to swap is **completely idle**.
- **Solution to pending I/O**
 - Never swap a process with pending I/O.
 - Execute I/O operations only into Operating-System buffers.
- Standard swapping is used in few systems. Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

Overlay?

8.3 Contiguous Memory Allocation

- Main memory is usually divided into two partitions:
 - Resident operating system, usually held in low memory with interrupt vector.
 - User processes, held in high memory.
- User space is divided into Multiple partitions
- Single-partition allocation for each process
- **Relocation-register scheme** used to protect user processes from each other, and from changing operating-system code and data.
 - **Relocation register** contains the value of the smallest physical address;
 - **limit register** contains the range of logical addresses – each logical address must be less than the limit register.

Hardware Support for Relocation and Limit Registers



Memory allocation

■ Fixed Partitioning

– Equal-size partitions

- any process whose size is less than or equal to the partition size can be loaded into an available partition
- if all partitions are full, the operating system can swap a process out of a partition
- a program may not fit in a partition. The programmer must design the program with **overlays**

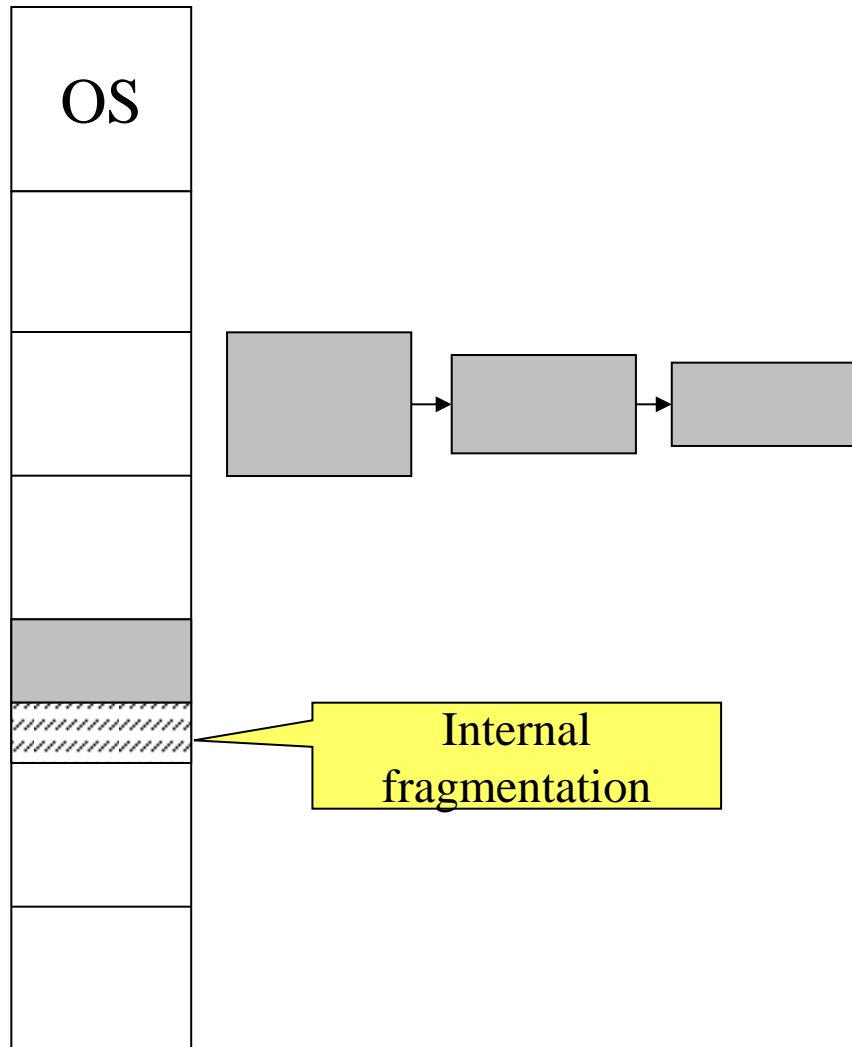
– Unequal-size partitions

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This results in **internal fragmentation**.

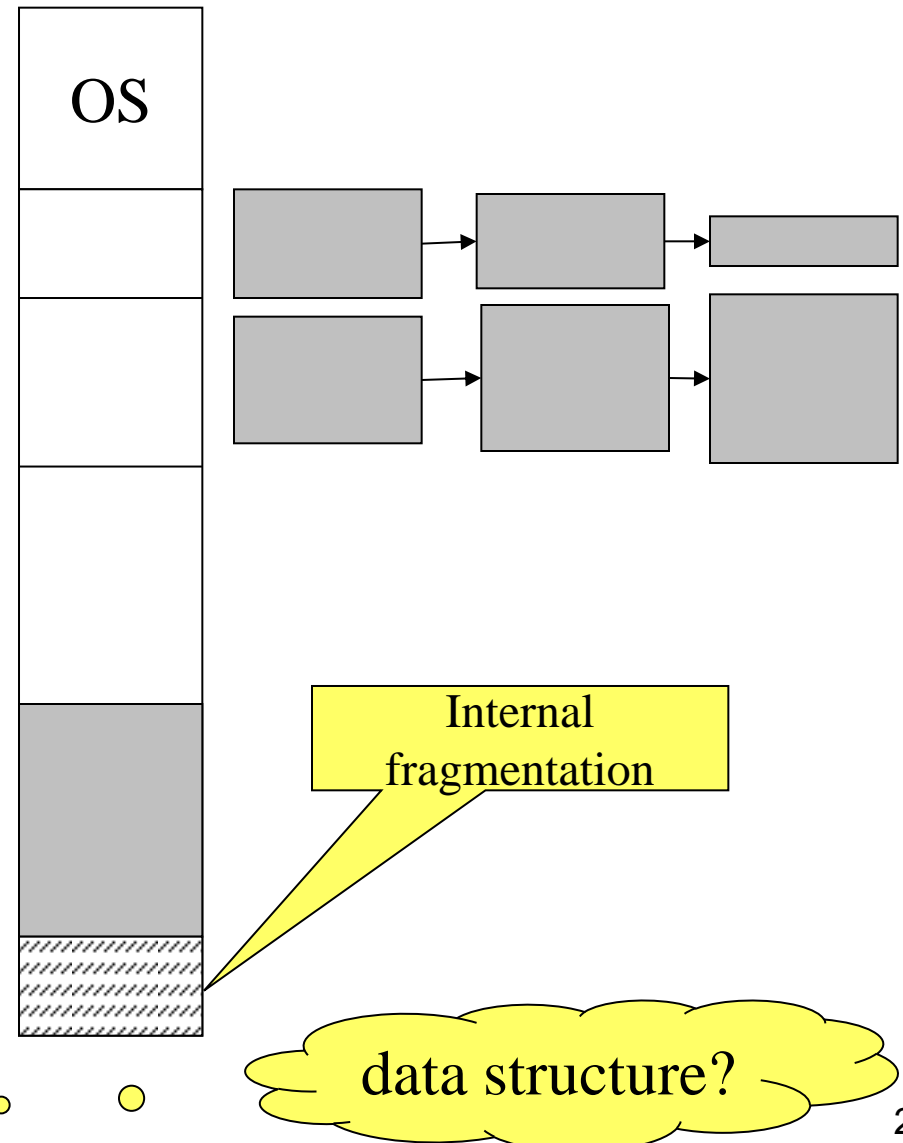
Placement Algorithm with Partitions

- **Equal-size partitions**
 - because all partitions are of equal size, it does not matter which partition is used
- **Unequal-size partitions**
 - can assign each process to the smallest partition within which it will fit
 - queue for each partition
 - processes are assigned in such a way as to minimize wasted memory within a partition

Equal-size partitions



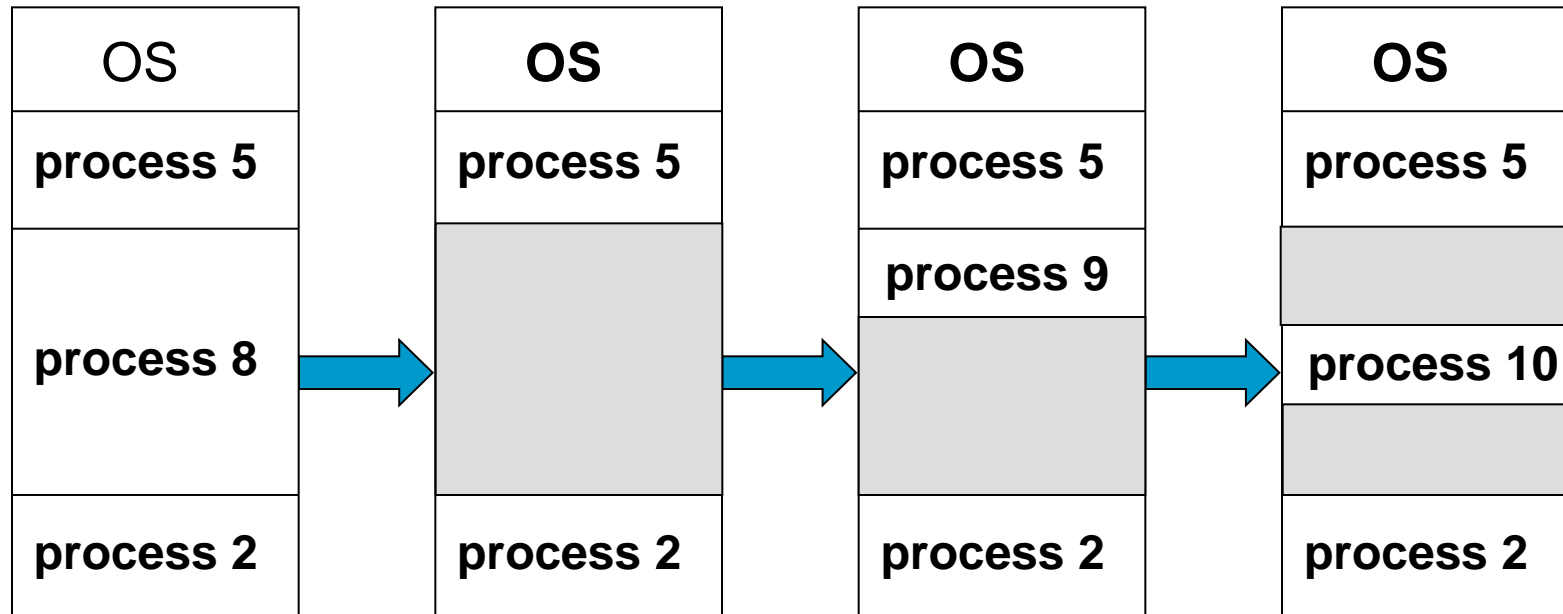
Unequal-size partitions



Variable-partition scheme

- **Dynamic Partitioning (MVT)**
 - Partitions are of variable length and number
 - When a process arrives, it is allocated memory from a **hole** large enough to accommodate it. Process is allocated exactly as much memory as required
 - Eventually get small holes in the memory. This is called **external fragmentation**
 - Must use compaction to shift processes so they are contiguous and all free memory is in one block
- **Hole** – block of available memory
 - Initially, all memory is available for user processes, a hole;
 - Holes of various size are scattered throughout memory.
- The operating system maintains a **table** containing the information about:
 - Which partitions are allocated, and to which process
 - Which partitions are free (hole)

Variable-partition scheme



- At any given time, we have a list of available block sizes and the input queue.
- A large hole is split into two parts: one is allocated to the process, the other is returned to the set of holes.

data structure?

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 the entire list, unless ordered by size. Produces the smallest leftover hole.
- ***Worst-fit***: Allocate the *largest* hole; must also search the entire list. Produces the largest leftover hole.

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

- These algorithms suffer from external fragmentation

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.
- **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.



MVT

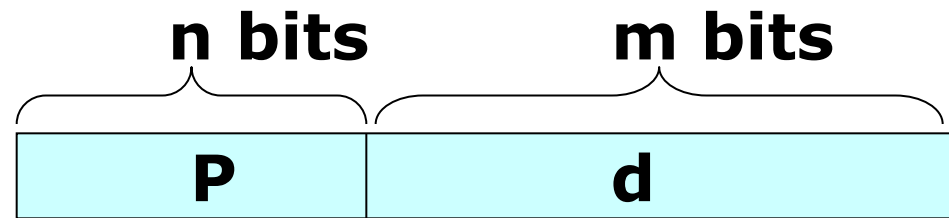
8.4 Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Physical memory is divided into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Logical memory is divided into blocks of same size called **pages**.
- Backing store is divided into fixed-sized **blocks** that are of the same size as the memory frames.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

Paging (Cont.)

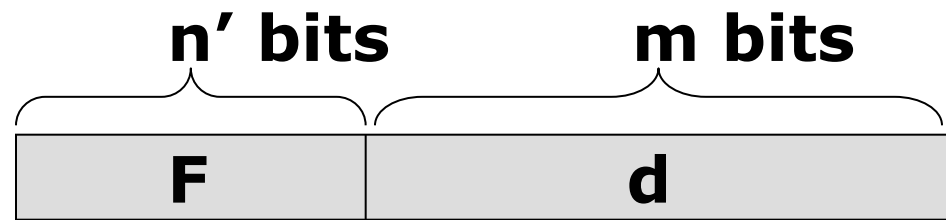
- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Operating system maintains a **page table** for each process
 - contains the frame location for each page in the process
 - page table used to translate logical to physical addresses.
- Internal fragmentation.
- Every address generated by the 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.

Address structure



■ Logical address

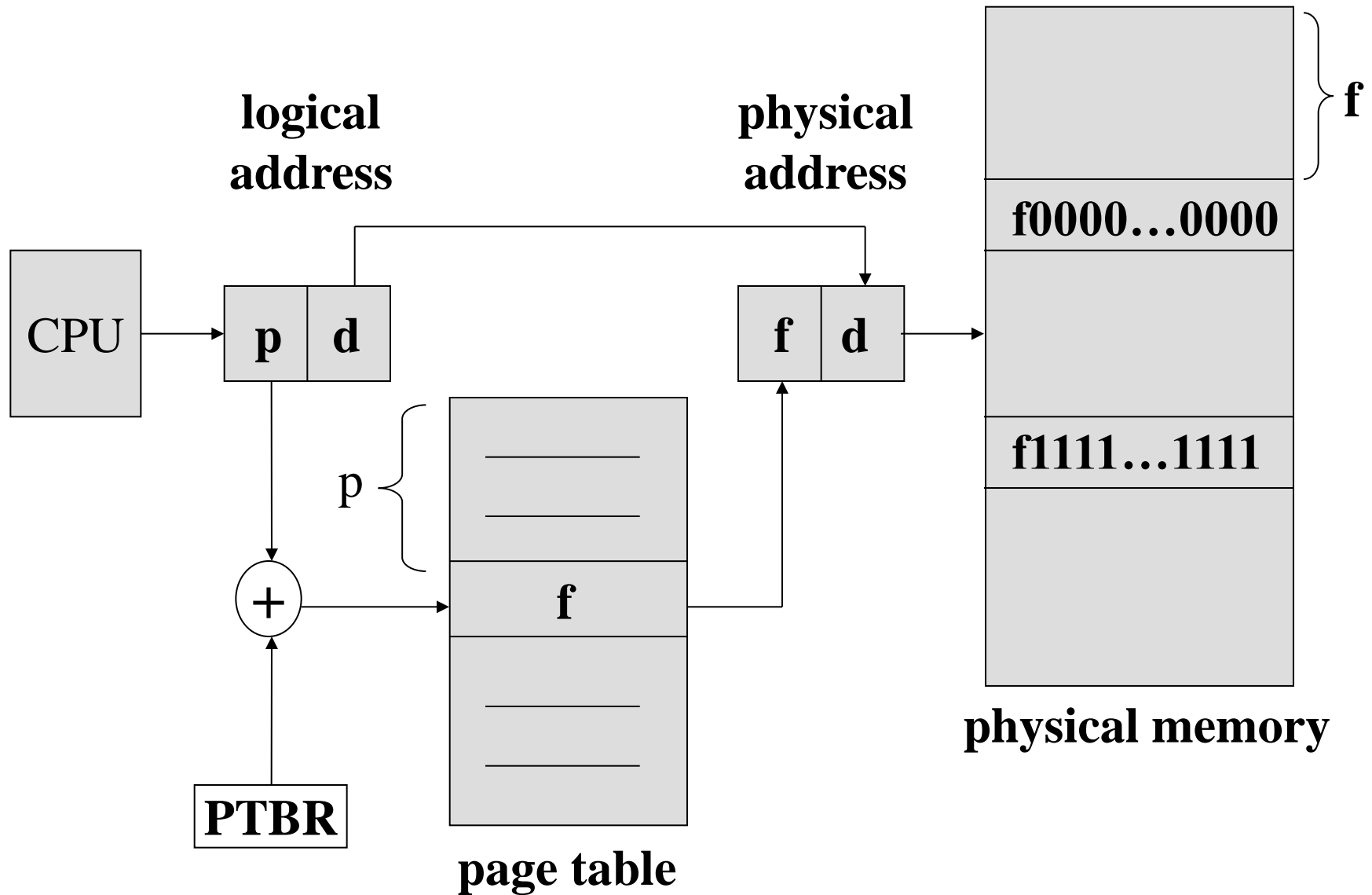
- two part: page number(P)、offset(d)
- bits of P: decided by the number of Page
- bits of d: decided by the size of page
- bits of logical address: $n+m$



■ Physical memory address

- two part: frame number(F)、offset(d)
- bits of F: decided by the number of memory frame
- bits of d: decided by the size of frame/page
- bits of physical address: $n'+m$

Address Translation Architecture



Paging model

page 0
page 1
page 2
page 3

logical memory

0	1
1	4
2	3
3	7

Page table

frame
number

0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

physical memory

The size of a page is typically a power of 2, varying between 512 bytes and 16MB per page

page number

page offset

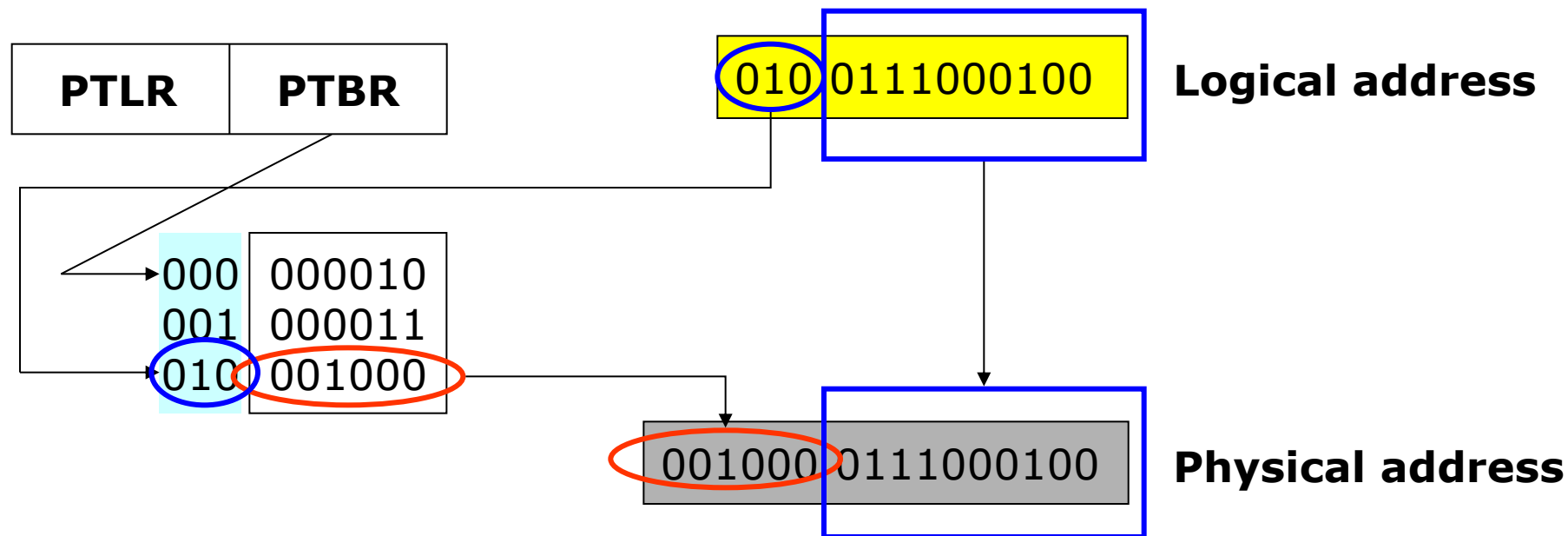
p	d
---	---

m-n bits

n bits

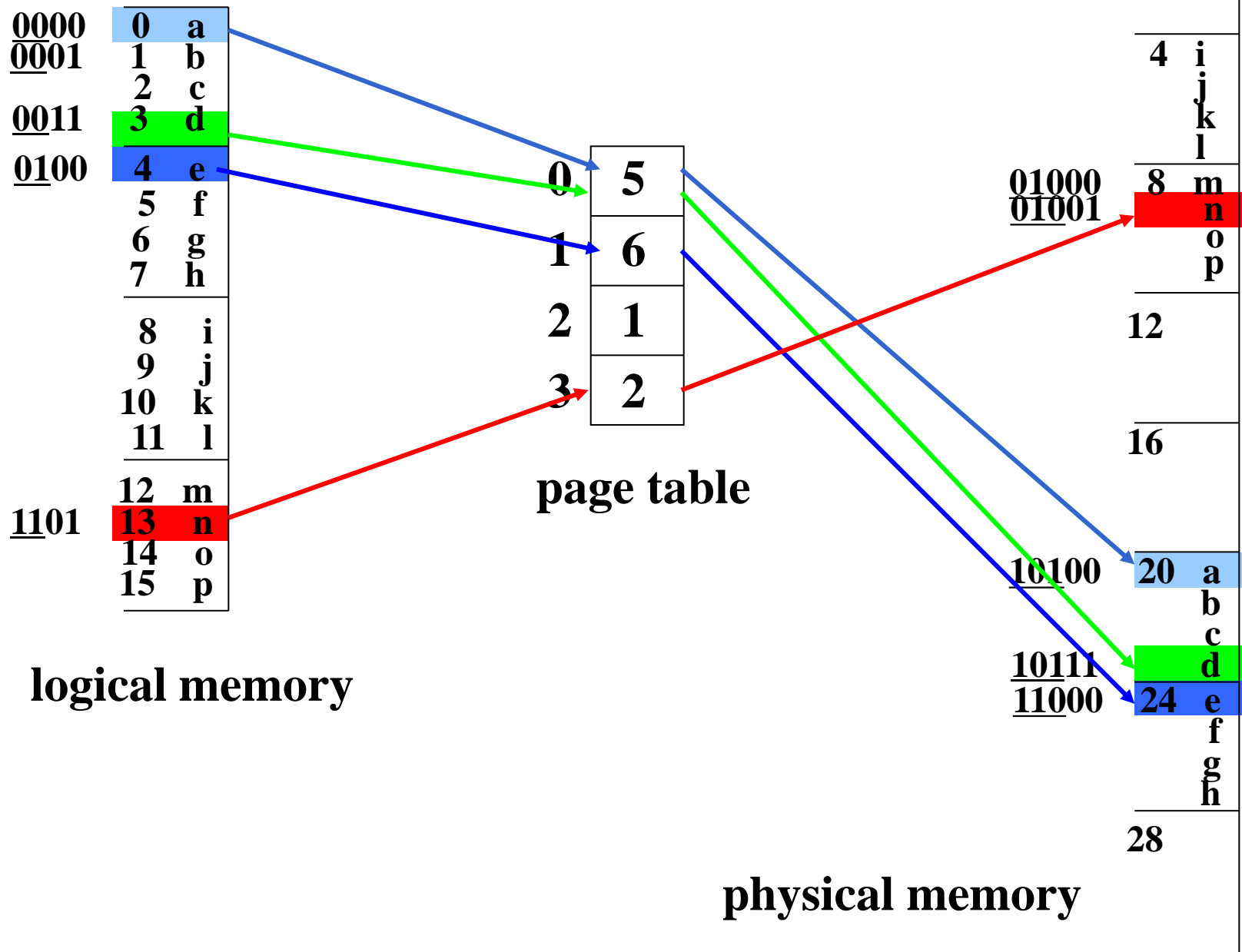
example of address mapping

- Page saize: 1KB, a process can have 8 pages, there are 64 memory frames.
- Length of LA is 13bits (3+10)
- length of PA is 16bits (6+10)
- LA: 2500
 $= 2 * 1024 + 452 \rightarrow 010\ 0111000100$



- PA: 001000 0111000100 $\rightarrow 8 * 1024 + 452 = 8644$

Paging Example



Free Frames

free-frame list

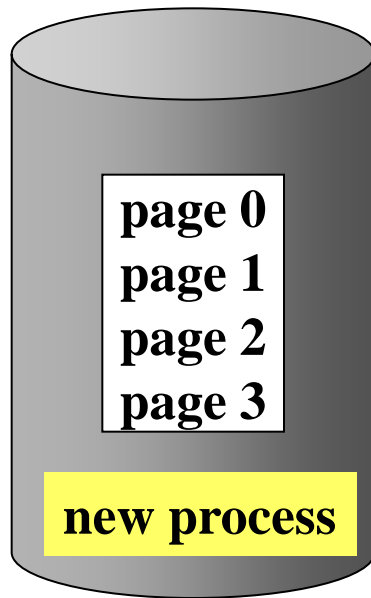
14

13

18

20

15



13

14

15

16

17

18

19

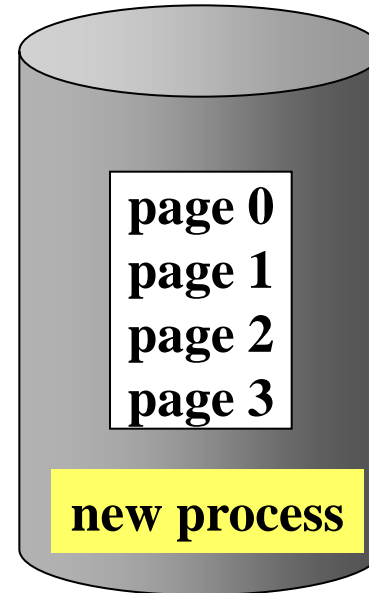
20

21

before allocation

free-frame list

15



0

14

1

13

2

18

3

20

New process page table

after allocation

13

page 1

14

page 0

15

16

17

18

page 2

19

20

page 3

21

Hardware support

- 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 entry and one for the data/instruction.
- The two memory access problem can be solved by the use of a special, small, fast-lookup hardware cache, called *translation look-aside buffers* (转换后备缓冲区 **TLBs**), or *associative memory* (联想存储器).

Translation Look-aside Buffer (转换后备缓冲区)

- Contains page table entries that have been most recently used
- Functions same way as a memory cache
- Given a logical address, processor examines the TLB
- If page table entry is found (a **hit**), the frame number is retrieved and the physical address is formed
- If page table entry is not found in the TLB (a **miss**), the page number is used to index the process page table
- First checks if page is already in main memory
 - if not in main memory a page fault is issued
- The TLB is updated to include the new page entry

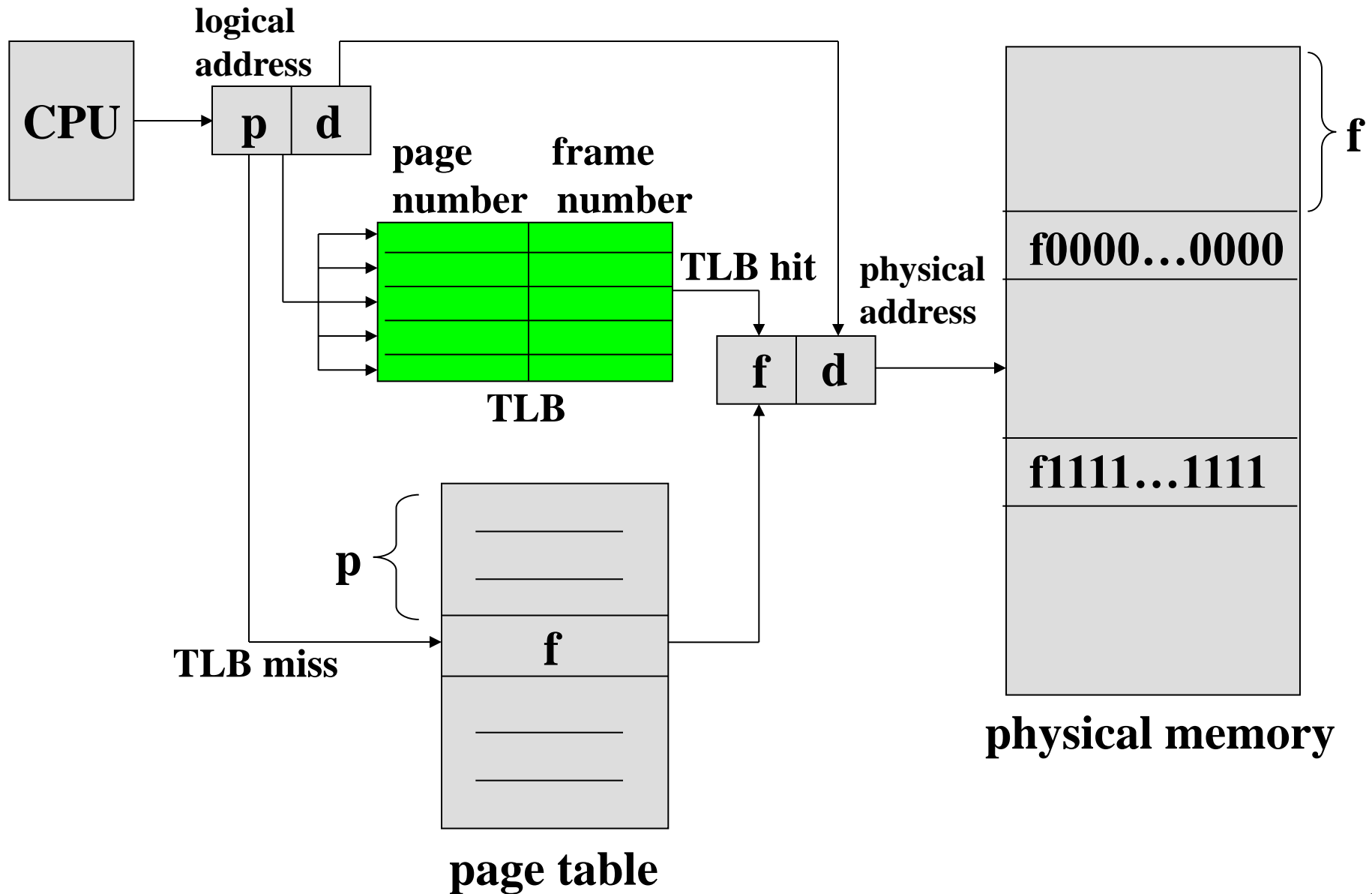
Translation Look-aside Buffer (Cont.)

- The TLB is associative, high-speed memory.
- Associative memory – parallel search

Page #	Frame #

- Address translation (A' , A'')
 - If A' 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 microsecond
- Assume memory cycle time is τ microsecond
- **Hit ratio** – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio = α
- Effective Access Time (EAT)

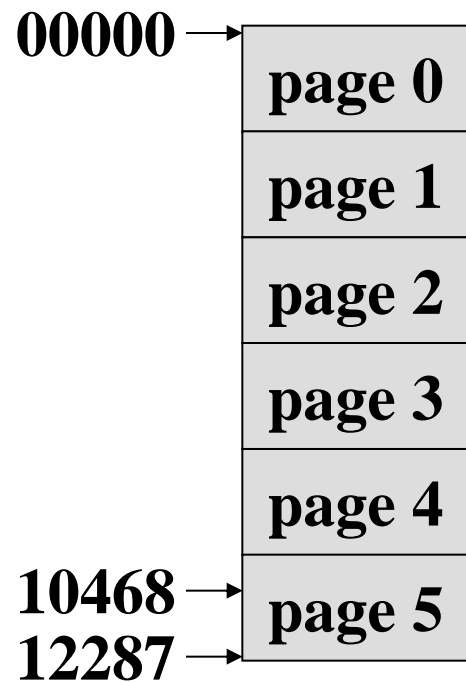
$$\begin{aligned} \text{EAT} &= (\varepsilon + \tau) \alpha + (\varepsilon + 2\tau)(1 - \alpha) \\ &= 2\tau + \varepsilon - \tau\alpha \end{aligned}$$

- $\varepsilon=20\text{ns}$, $\alpha=0.8$, $\tau=100\text{ns}$ $\text{EAT}=140\text{ns}$
- $\varepsilon=20\text{ns}$, $\alpha=0.98$, $\tau=100\text{ns}$ $\text{EAT}=122\text{ns}$

Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- Normally these bits are kept in the page table.
- **RW** bit, can define a page to be read-write or read-only.
- *Valid-invalid* bit attached to each entry in the page table:
 - “**valid**” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
 - “**invalid**” indicates that the page is not in the process’ logical address space.

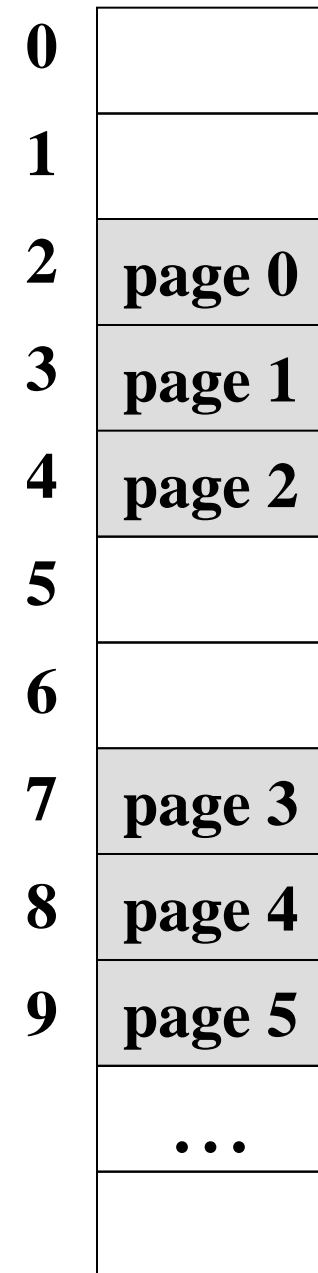
Valid (v) or Invalid (i) Bit



frame number valid-invalid bit

0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Page table



physical memory

Shared Pages

■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes.

■ Private code and data

- Each process keeps a separate copy of the code and data.
- The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example

ed1
ed2
ed3
data1

process P_1

ed1
ed2
ed3
data2

process P_2

ed1
ed2
ed3
data3

process P_3

logical memory

3
4
6
1

page table
for P_1

3
4
6
7

page table
for P_2

3
4
6
2

page table
for P_3

page table

0	
1	data1
2	data3
3	ed1
4	ed2
5	
6	ed3
7	data2
8	
9	
10	

physical memory

8.5 Structure of Page Table

- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

Hierarchical Page Tables

- A logical address (on 32-bit machine with 4KB page size) is divided into:
 - a page number consisting of 20 bits.
 - a page offset consisting of 12 bits.
- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.
 - the page table itself is also paged
- Since the page table is paged, the **page number** is further divided into:
 - a 10-bit page number.
 - a 10-bit page offset.

Two-Level Paging Example

- Thus, a logical address is as follows:

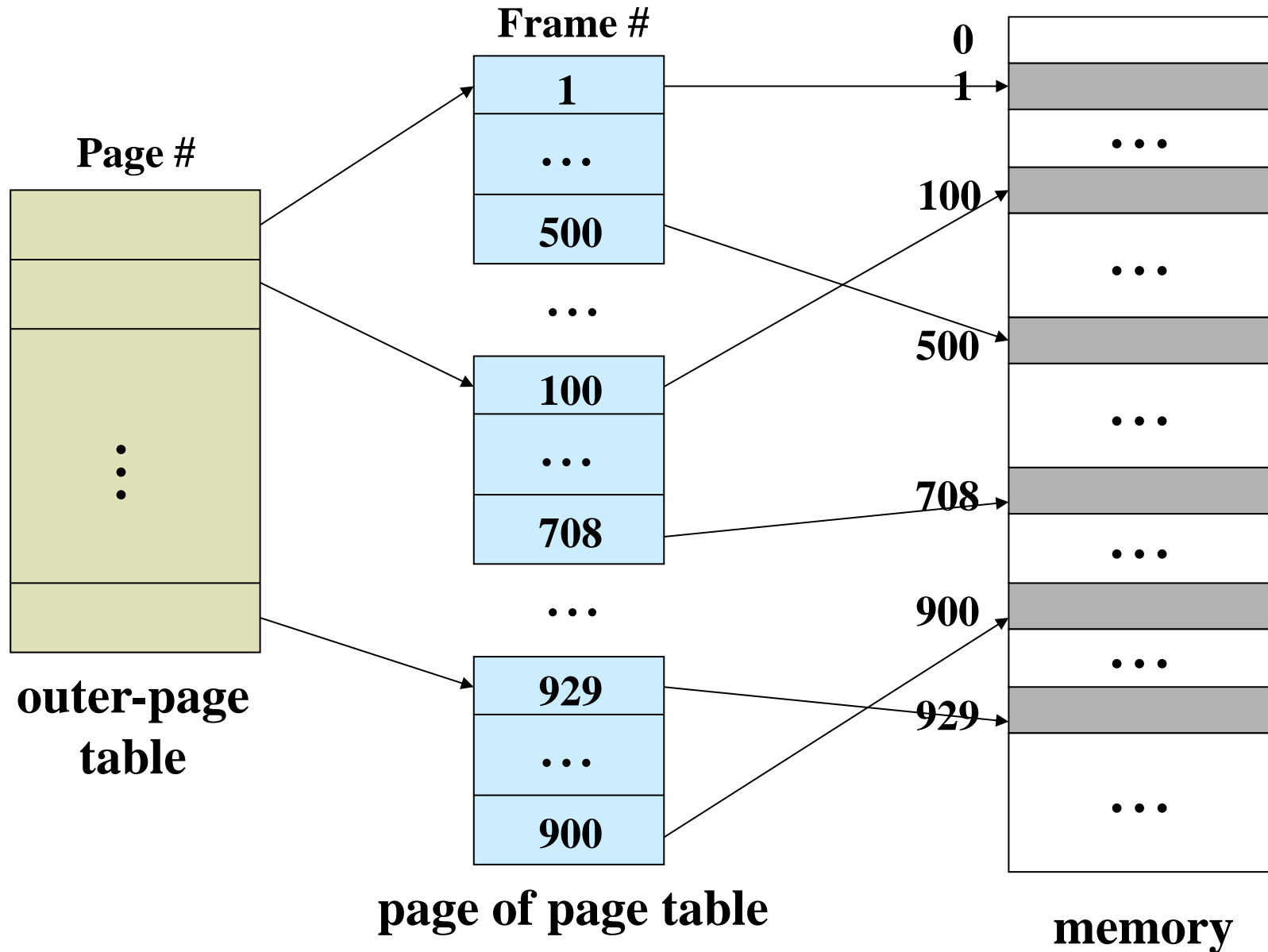
page number		page offset
p_1	p_2	d
10	10	12

where

p_1 is an index into the outer page table, and

p_2 is the displacement within the page of the outer page table.

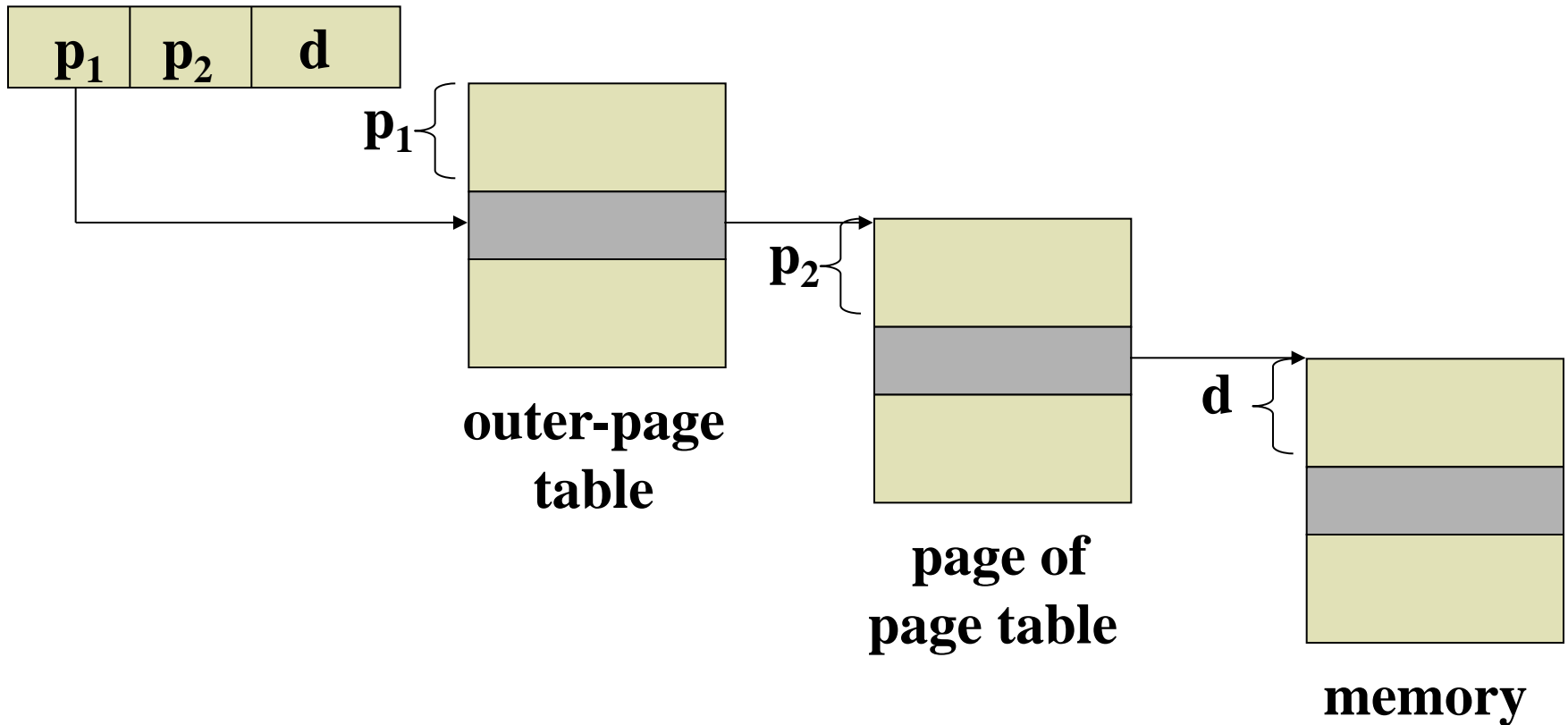
Two-Level Page-Table Scheme



Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

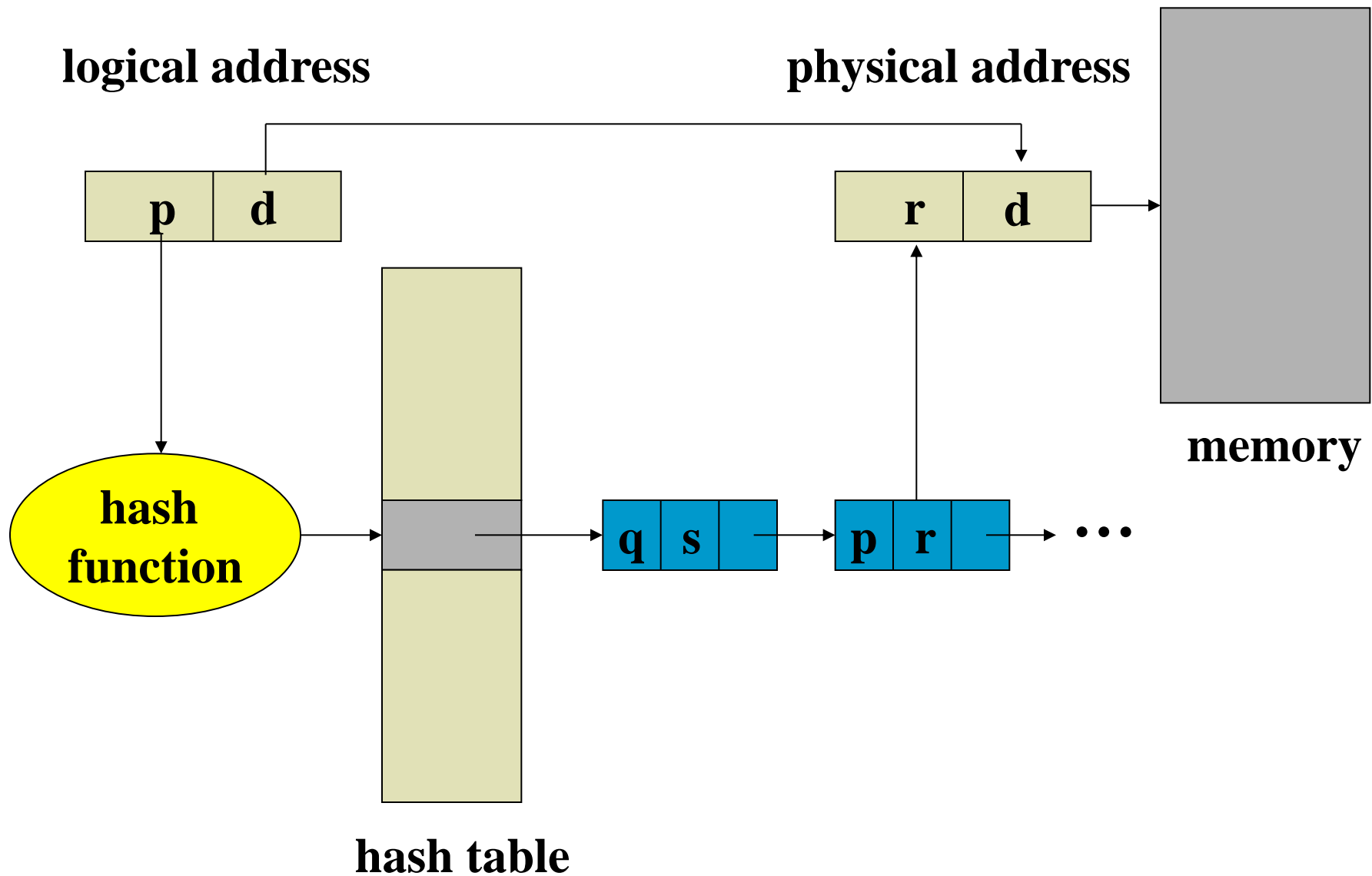
Logical address



Hashed Page Tables

- Common in address spaces > 32 bits.
- Each entry in the hash table contains a linked list of elements that hashed to the same location.
- Each element consists of three fields:
 - (a) The virtual page number
 - (b) The value of the mapped page frame (frame number)
 - (c) A pointer to the next element in the linked list
- The virtual page number is hashed into the hash table.
- The virtual page number is compared to field (a) in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Hashed Page Table

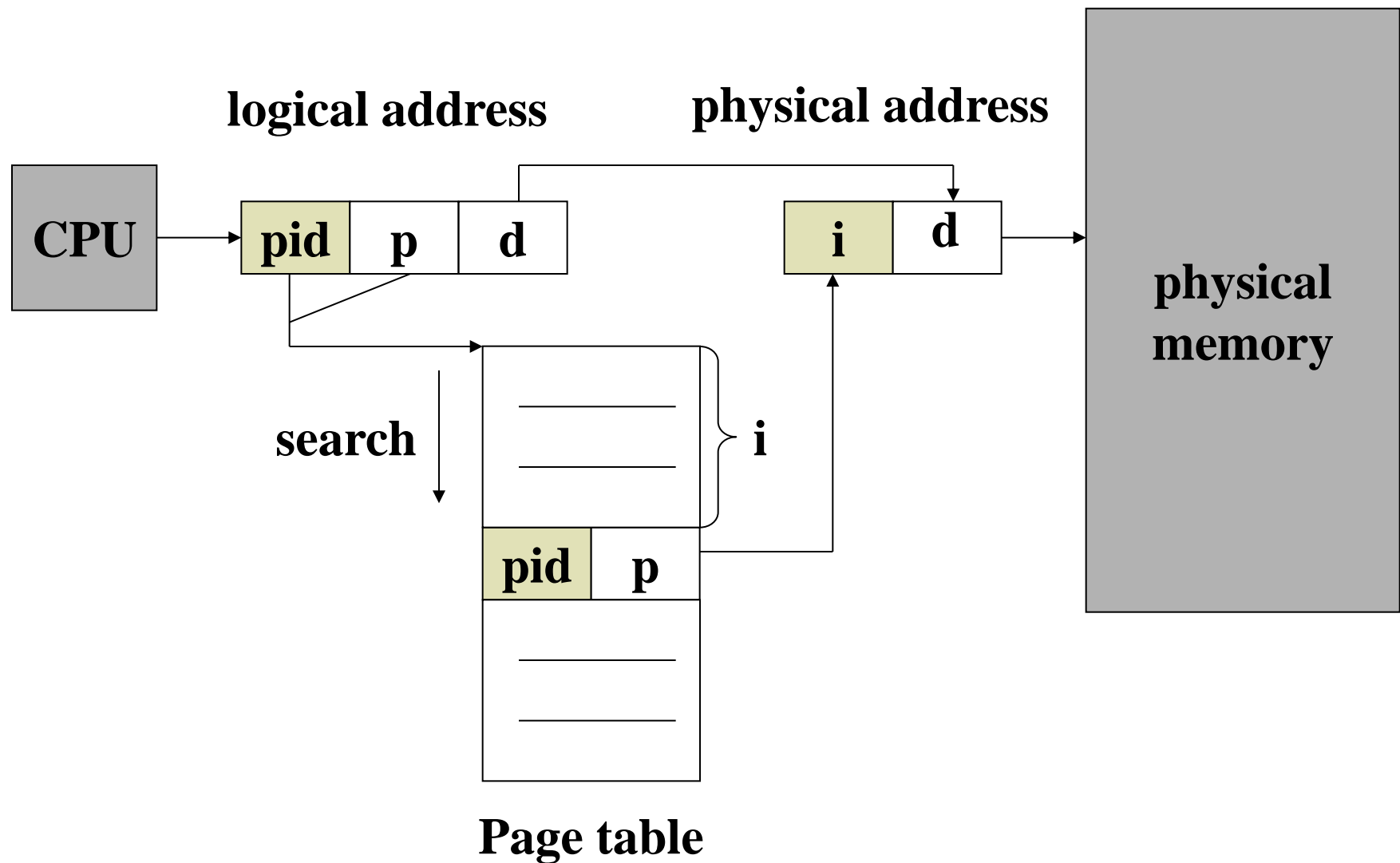


Inverted Page Table

- Has one entry for each real frame of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Often requires an address-space identifier (ASID) stored in each entry of the page table. Ensures the mapping of a logical page for a particular process to the corresponding physical page frame.
- a simplified version of the implementation used in the IBM RT:

< process-id, page-number, offset >

Inverted Page Table Architecture



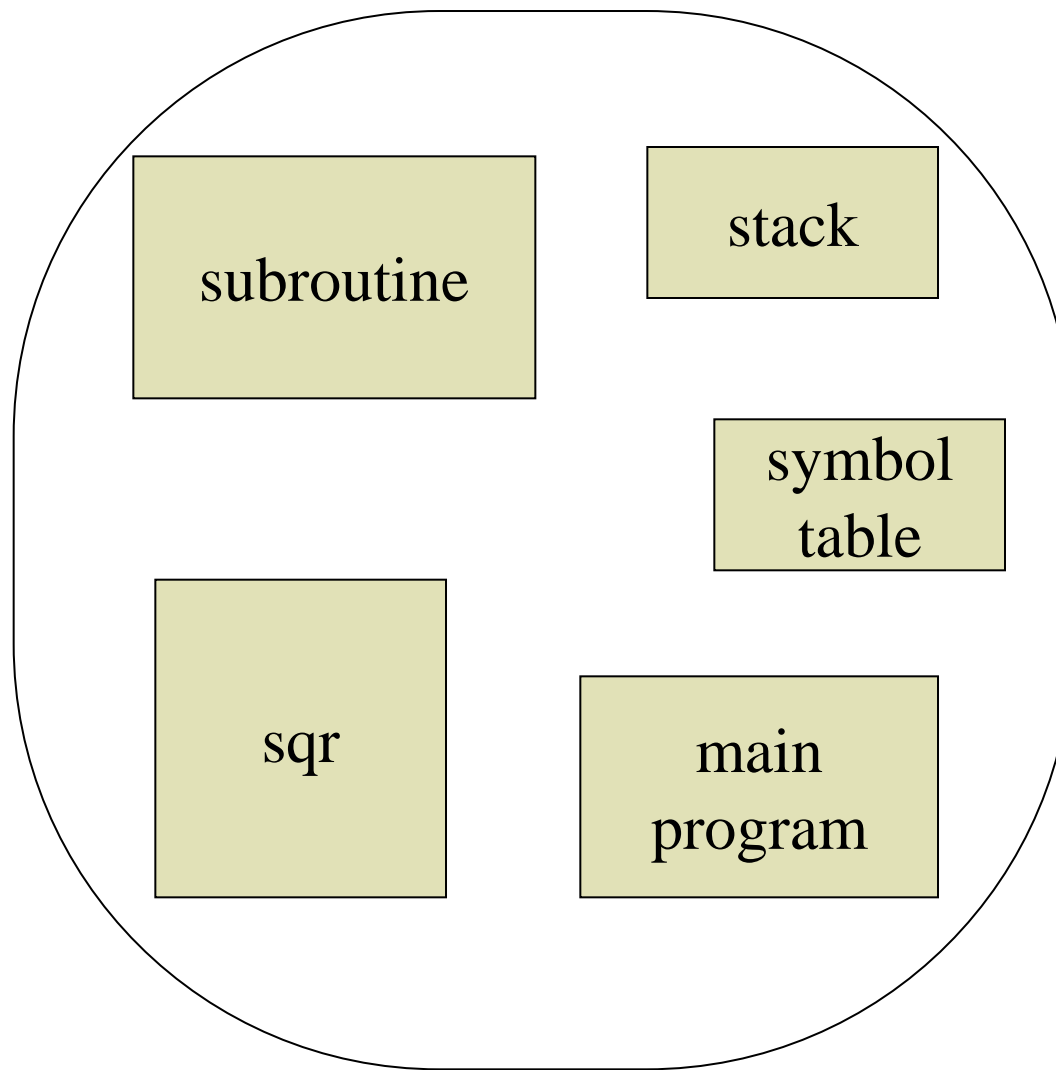
Inverted Page Table (Cont.)

- **Decreases the amount of memory needed to store each page table, but increases the amount of 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.**

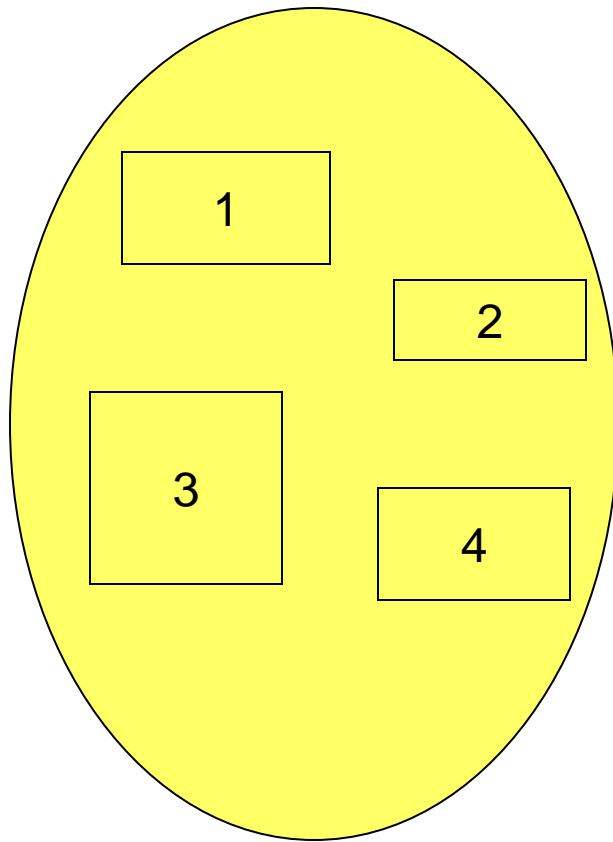
8.6 Segmentation

- **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**

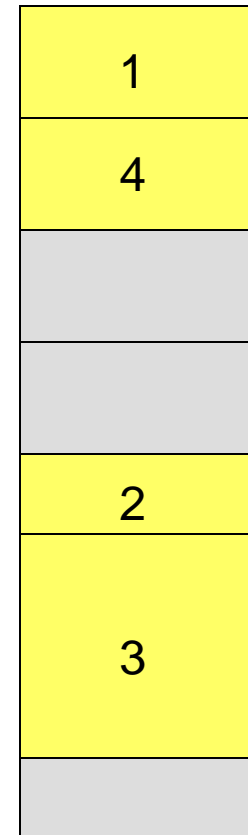
User's View of a Program



Logical View of Segmentation



user space



physical memory space

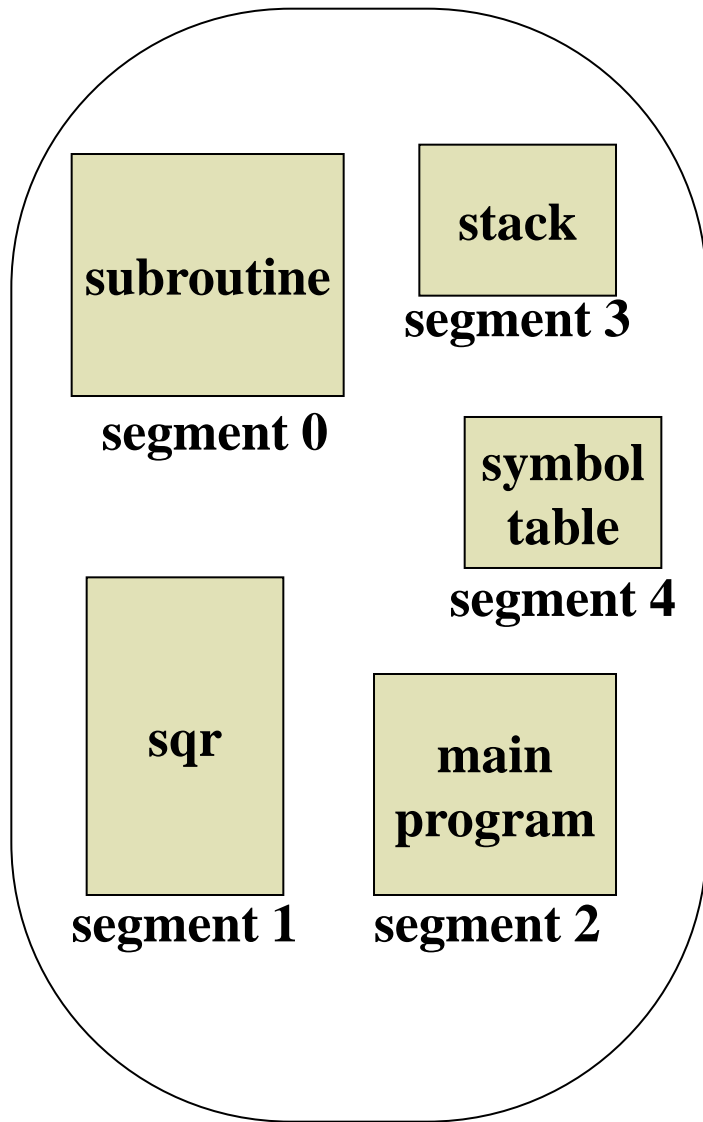
Segmentation Architecture

- All segments of all programs do not have to be of the same length
- There is a maximum segment length
- Logical address consist of two parts - a segment number and an offset

<segment-number, offset>
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
 - *base* – contains the starting physical address where the segments reside in memory.
 - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

segment number s is legal if $s < \text{STLR}$.

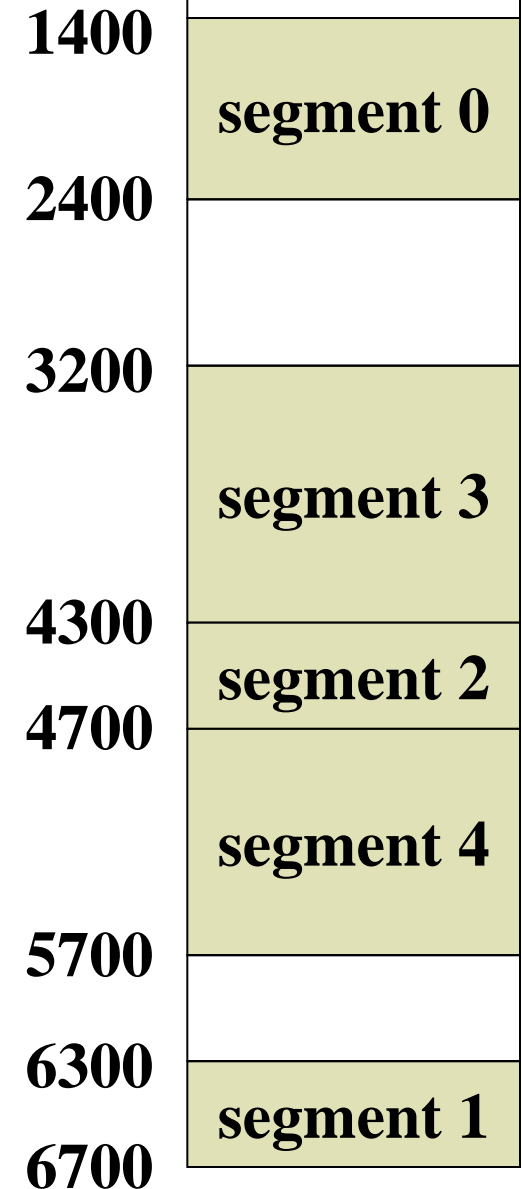
Example of Segmentation



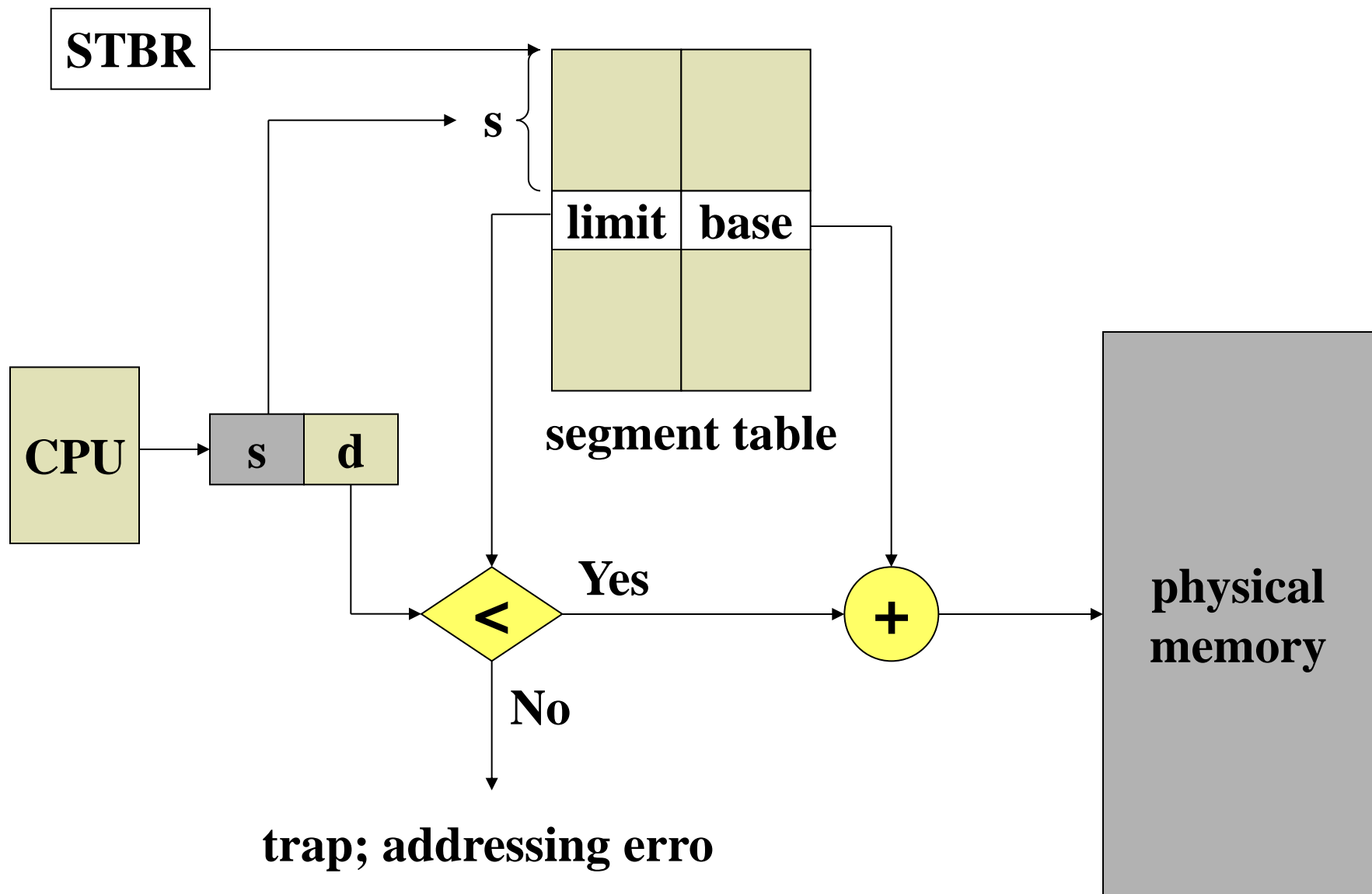
Logical address space

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

segment table



Segmentation Hardware



Supplement: example

■ Segment table:

Segment no.	Limit	base
0	660	2219
1	140	3300
2	100	90
3	580	1237
4	960	1959

■ mapping logical address to physical address:

[0, 432], [1, 10], [2, 500], [3, 400]

■ Summarize the procedure that mapping a logical address to physical address.

Supplement:

Segmentation Architecture (Cont.)

■ Relocation

- dynamic
- by segment table

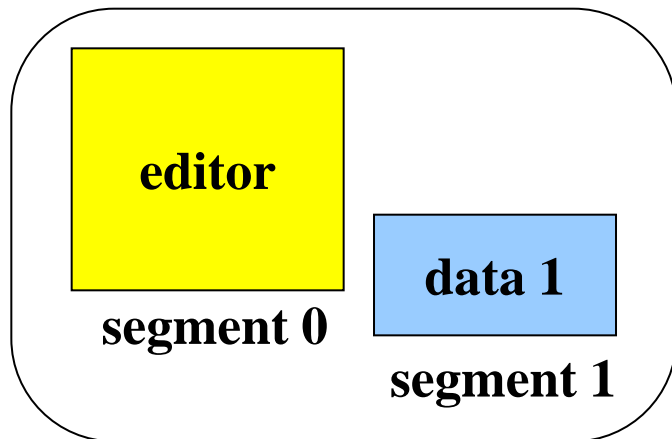
■ Allocation

- first fit/best fit
- external fragmentation

■ Sharing

- shared segments
- same segment number

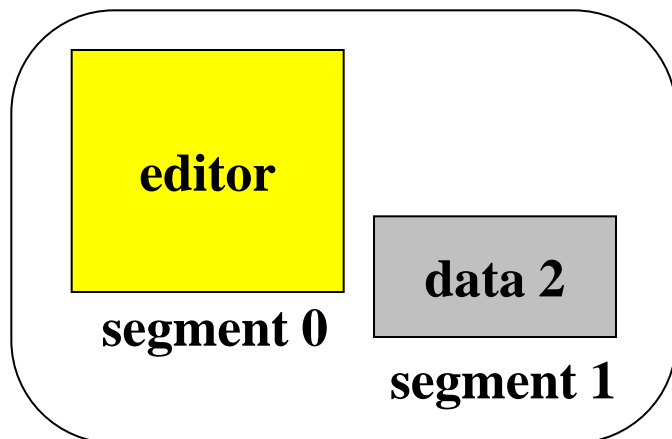
Supplement: Sharing of Segments



logical address space
process P_1

	limit	base
0	25286	43062
1	4425	68348

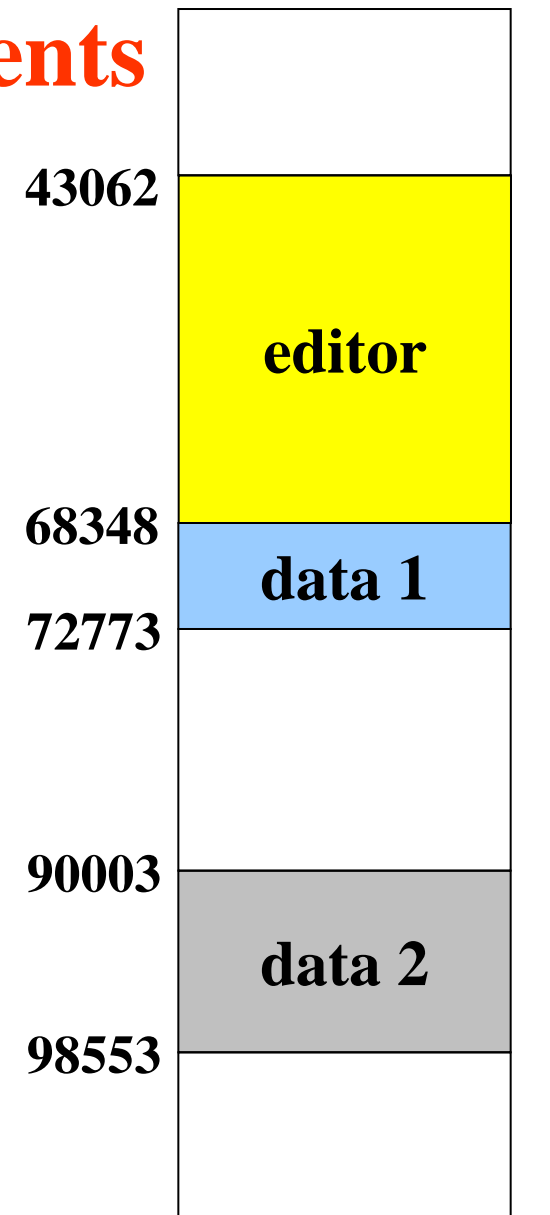
segment table
process P_1



logical address space
process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2



physical memory

Supplement:

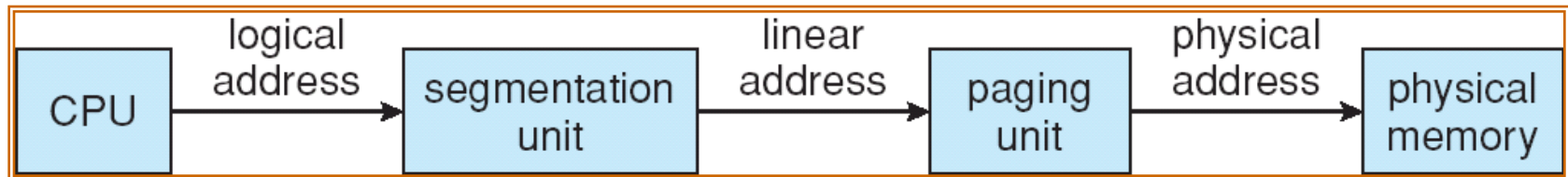
Segmentation Architecture (Cont.)

- **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.**

How to avoid external fragmentation?

8.7 Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
 - Given to segmentation unit, Which produces linear addresses
 - Linear address given to paging unit, Which generates physical address in main memory
- The segmentation and Paging units form the equivalent of MMU



Pentium segmentation

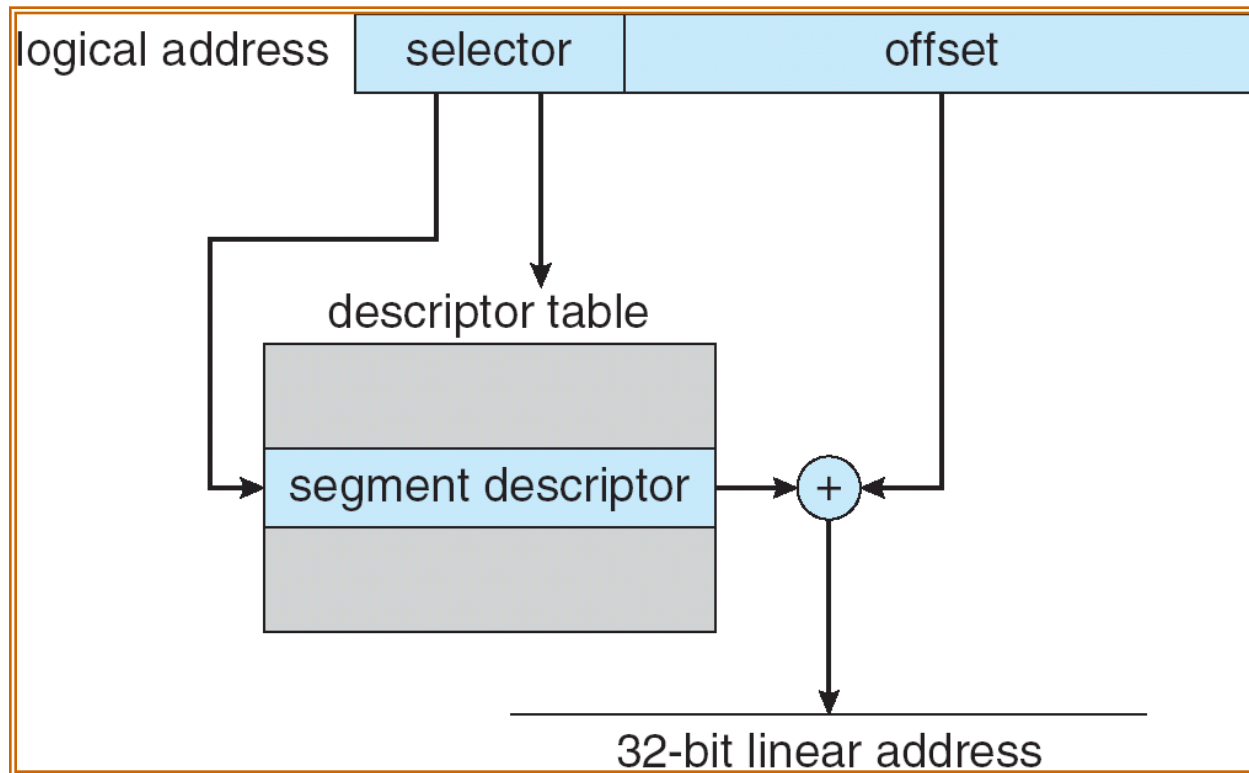
- Allows a segment to be as large as 4GB, maximum number of segments per process is 16K(=16384).
- The logical-address space of a process is divided into two partition
 - 1st, up to 8K segments, private to that process
 - 2nd, up to 8K segments, shared among all the processes
- Information about the 1st partition is kept in the local descriptor table, LDT; information about the 2nd partition is kept in the global descriptor table, GDT.
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.

Pentium segmentation

- **Logical address: (selector, offset)**
 - Selector is a 16-bit number, **13-bit s**, **1-bit g**, and **2-bit p**
 - **s**, the segment number
 - **g**, whether the segment is in the GDT or LDT
 - **p**, deals with protection
 - offset, 32-bit number, location of byte within the segment
- **The machine has six segment registers, allowing 6 segments to be addressed at any time by a process.**
- **It has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT. (cache)**

Pentium segmentation

- The **segment register** points to the appropriate entry in the LDT or GDT.
- The **base and limit** information is used to generate a linear address.



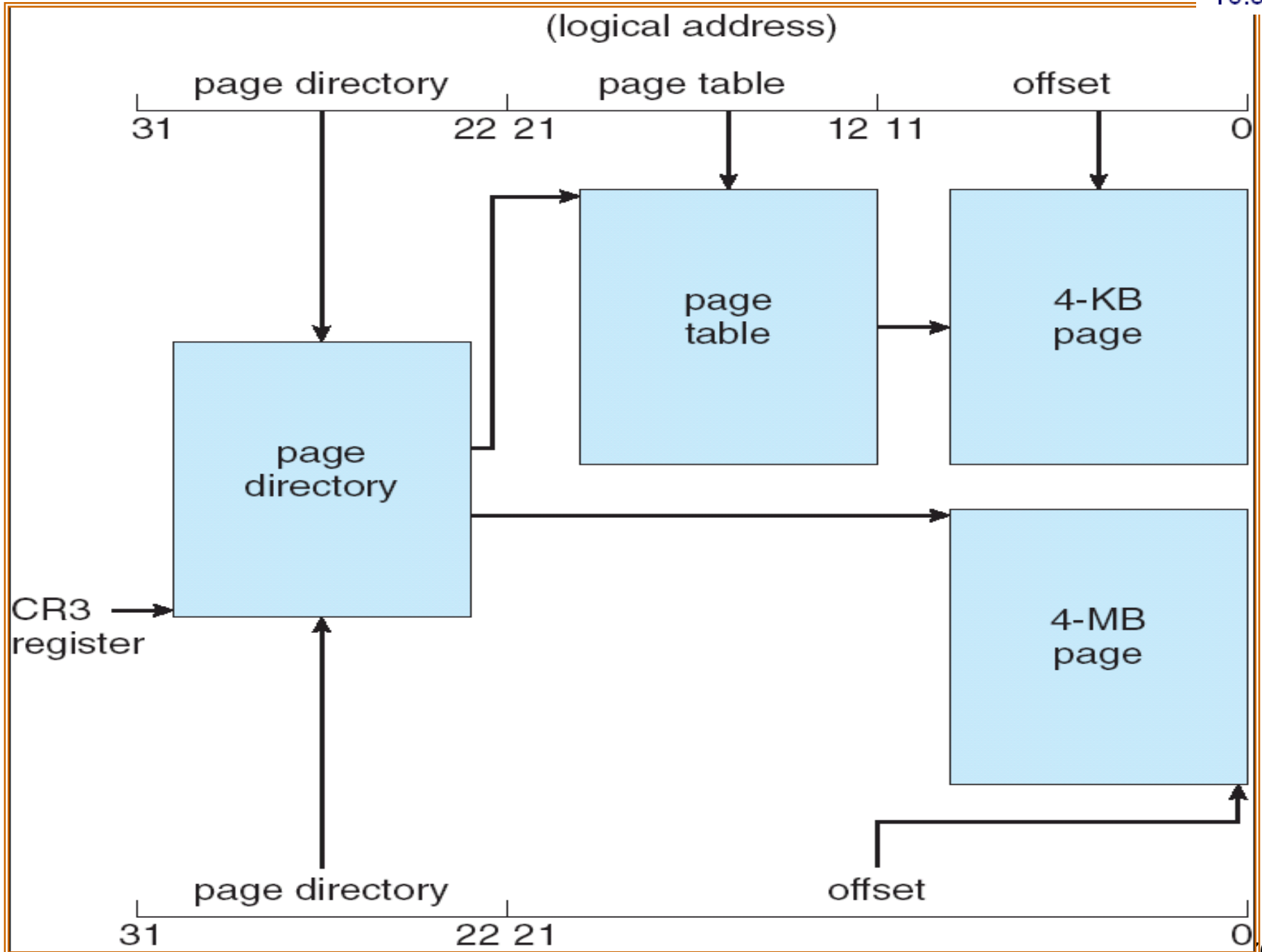
Pentium paging

- Allow a page size of either 4KB or 4MB
- For 4KB pages, the Pentium uses a two-level paging scheme.
- The linear address: 32-bit long
 - 20-bit page number
 - 10-bit page directory pointer (P1)
 - 10-bit page table pointer (P2)
 - 12-bit offset (d)

page number		page offset
P ₁	P ₂	d

Pentium paging

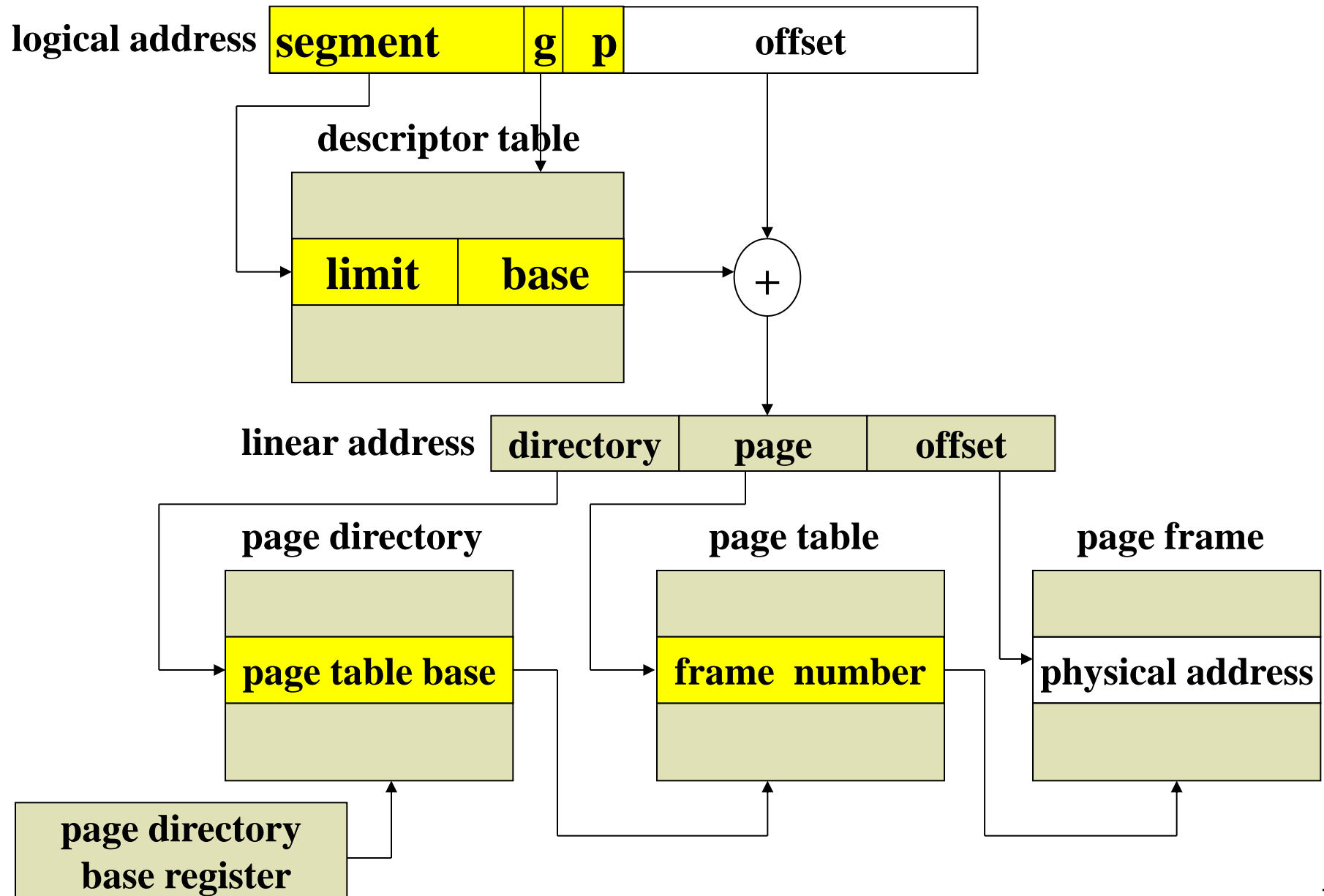
- **One entry in the page directory is the Page Size flag,**
 - If set, page size is 4MB
 - If not set, page size is 4KB
- **For 4MB page size**
 - the page directory points directly to the 4-MB page frame
 - and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.



Pentium paging

- **Page table can be swapped to disk**
 - An invalid bit is used
 - If swapped, the OS uses the other 31 bits to specify the disk location of the table

Address Translation



Linux on Pentium systems

- **Provide only limited support for segmentation**
 - **Linux does not rely on segmentation and uses it minimally.**
 - **On the Pentium, Linux uses only six segment:**
 - **A segment for kernel code**
 - **A segment for kernel data**
 - **A segment for user code**
 - **A segment for user data**
 - **A task-state segment (TSS)**
 - **A default LDT segment**
- } shared by all processes
running in user mode
- Used to store the hardware context of
each process during context switches.
- Is normally shared by all processes and is usually not used

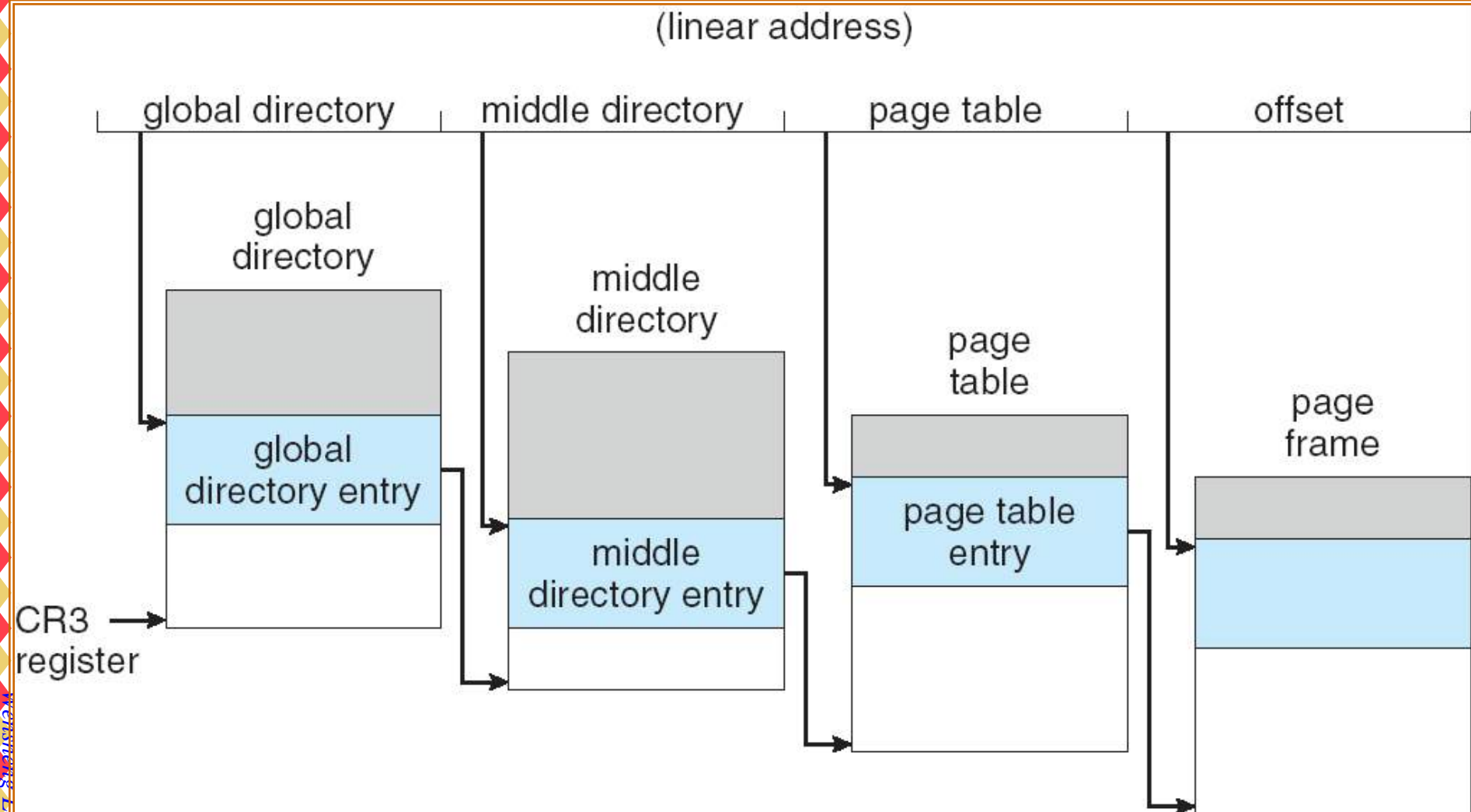
Linux on Pentium systems

- Each segment selector includes a **2-bit field for protection**, Linux only recognizes two: user mode and kernel mode
- Linux adopts a **three-level paging strategy** that works well for both 32-bit and 64-bit architectures.
- The linear address in Linux:



- Each task in Linux has its own set of page tables
- The CR3 register points to the global directory for the task currently executing.

Three-level Paging in Linux



Homework (page 310)

8.3

8.5

8.9

8.12

Thinking about:

8.1

8.10

8.11

8.13



Supplement:

About Main Memory Management

■ Goals of Memory Management

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated efficiently to pack as many processes into memory as possible

Supplement:

Memory Management Requirements (1/5)

■ Relocation

- Programmer does not know where the program will be placed in memory when it is executed
- While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
- Memory references must be translated in the code to actual physical memory address

Supplement:

Memory Management Requirements (2/5)

■ Protection

- Processes should not be able to reference memory locations in another process without permission
- Must be checked during execution
 - Impossible to check absolute addresses in programs since the program could be relocated
 - Operating system cannot anticipate(预测) all of the memory references a program will make

Supplement:

Memory Management Requirements (3/5)

■ Sharing

- Allow several processes to access the same portion of memory
- Better to allow each process (person) access to the same copy of the program rather than have their own separate copy

Supplement:

Memory Management Requirements (4/5)

■ Logical Organization

- Programs are written in modules
- Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules

Supplement:

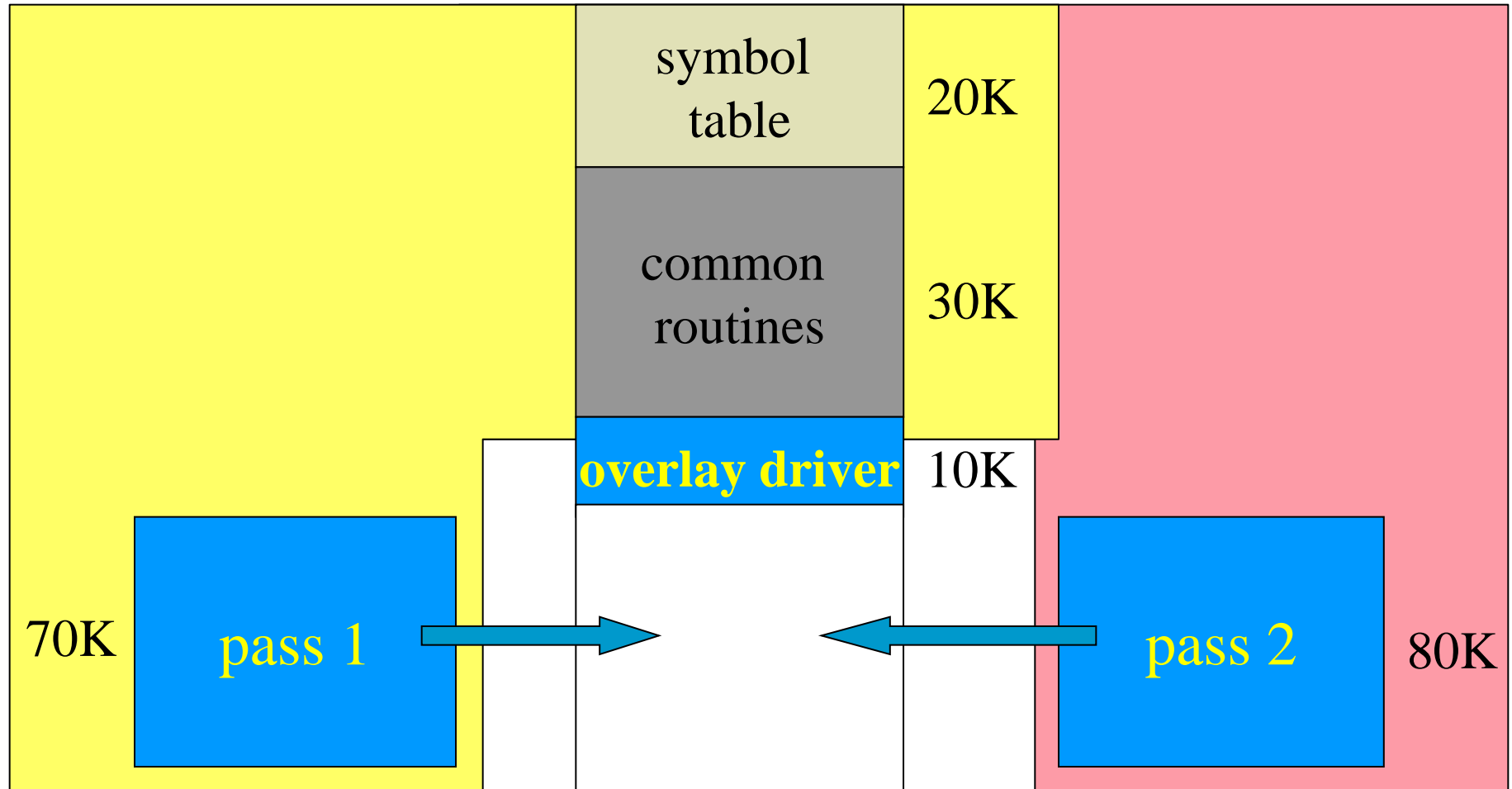
Memory Management Requirements (5/5)

■ Physical Organization

- Memory available for a program plus its data may be insufficient
 - Overlaying allows various modules to be assigned the same region of memory
- Programmer does not know how much space will be available

Supplement: Overlays

- Keep in memory only those instructions and data that are needed at any given time.



Supplement: Overlays (Cont.)

- **Needed when process is larger than amount of memory allocated to it.**
- **Implemented by user, no special support needed from operating system, programming design of overlay structure is complex**
 - **This task can be a major undertaking, requiring complete knowledge of the structure of the program, its code, and its data structures.**
- **Currently limited to microcomputer and other systems that have limited amounts of physical memory and that lack hardware support for more advanced techniques.**

Supplement: 覆盖与交换技术

- 多道程序设计环境下用来扩充内存的两种方法
- 覆盖技术
 - 当用户程序所需要的内存空间比分配给它使用的内存空间大时，可以使用覆盖技术。
 - 由程序员实现，不需要操作系统特别支持
 - 设计思想：把程序划分为若干个功能上相对独立的程序段，按照程序的逻辑结构让那些不会同时执行的程序段共享同一块内存区域。
 - 对程序员要求很高
 - 必须十分清楚程序的逻辑结构
 - 明确规定各程序段的执行和覆盖顺序。
 - 设计实现覆盖驱动模块。

Supplement: 覆盖与交换技术

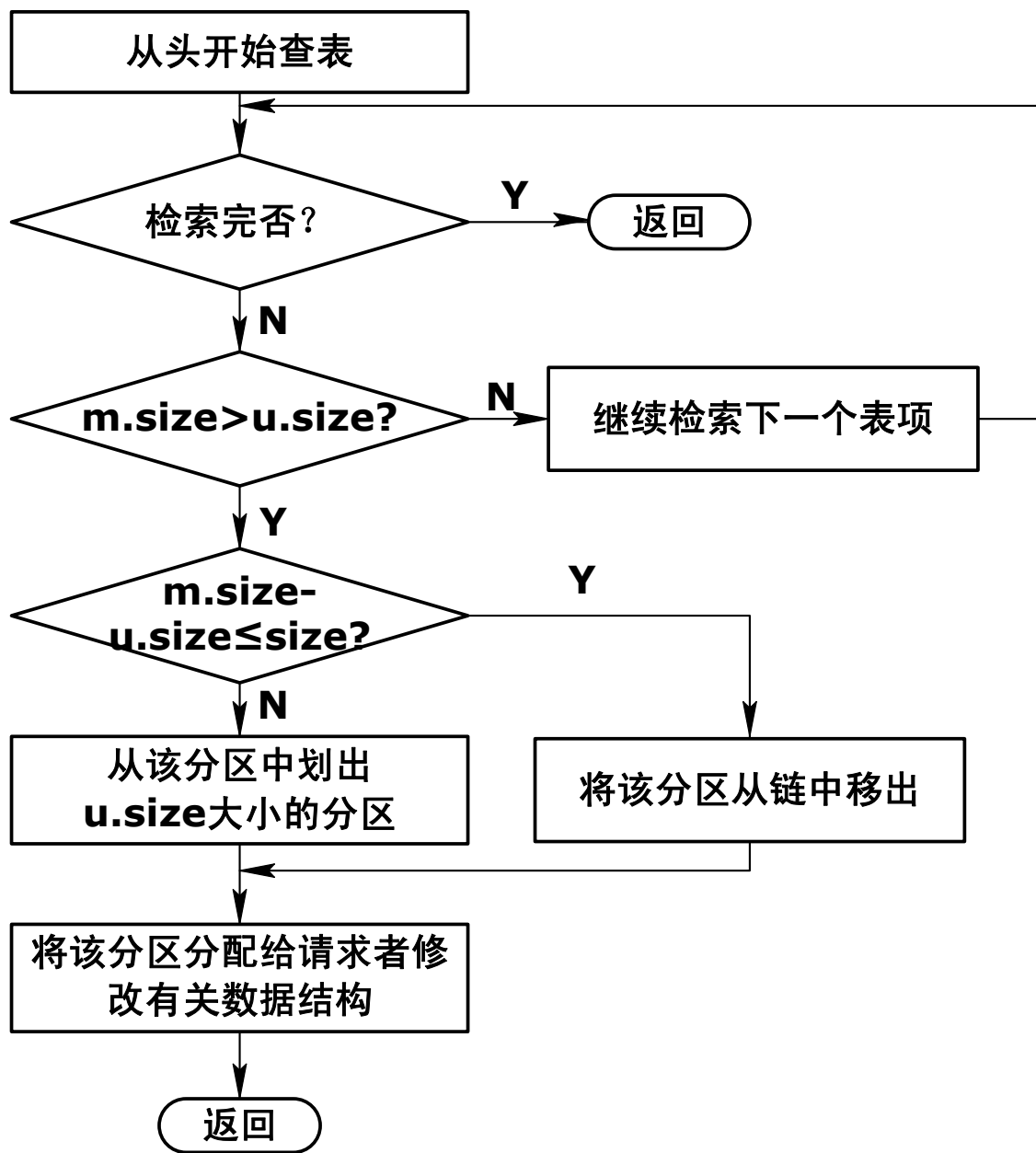
■ 交换技术

- 由操作系统中的交换程序实现，对用户透明
- 将内存某区域中的进程调出内存写入外存交换区（swap out），再从外存交换区中把指定的进程调入内存（swap in）并让它执行的一种技术。达到扩充内存的效果。

■ 交换与覆盖的区别

- 覆盖技术要求程序员自己设计覆盖结构，而交换对用户是透明的。
- 交换主要是在进程和进程之间进行，而覆盖则是在同一作业/进程内进行。

分配流程



动态分区的回收

- 用户进程执行结束后，存储管理程序要收回已经使用完毕的存储空间，并将其插入空闲分区表/链中。
- 对于所释放的空闲区，在将它插入空闲分区表/链中之前，要考虑内存中与其上下相邻的存储空间的使用情况，如果有空闲分区，则要进行拼接，使之成为较大的空闲区。
- 上下相邻分区的状态
 - 上下相邻分区均为空闲分区
 - 更新上邻空闲区记录的长度=上邻区长度+释放区长度+下邻区长度
 - 从表/链中取消下邻区的记录
 - 上邻分区为空闲分区：
 - 更新上邻区记录的长度=上邻区长度+释放区长度
 - 下邻分区为空闲分区
 - 更新下邻区记录的长度=释放区长度+下邻区长度
 - 更新下邻区记录的起始地址=释放区的起始地址
 - 上下邻分区均非空闲区
 - 为释放区在表/链中适当的位置建立记录

补充：首次适应法

- 要求空闲区按**首址递增**的次序组织空闲区表(队列)
- 分配：当进程申请大小为 S 的内存时，系统从空闲区表的第一个表目开始查询，直到首次找到等于或大于 S 的空闲区。从该区中划出大小为 S 的分区分配给进程，余下的部分仍作为一个空闲区留在空闲区表中，但要修改其首址和大小。
- 回收：按释放区的首址，查询空闲区表，若有与释放区相邻的空闲区，则合并到相邻的空闲区中，并修改该区的大小和首址，否则，把释放区作为一个空闲区，将其大小和首址**按照首地址大小递增**的顺序插入到空闲区表的适当位置。

补充：分析

- 注意：每次分配和回收后空闲区表或空闲区队列都要按首址递增的次序排序。
- 首次适应法的优点：
 - 释放某一存储区时，若与空闲区相邻则合并到相邻空闲分区中去，这种情况并不改变该区在表中的位置，只要修改其大小或首址。
 - 这种算法是尽可能地利用低地址空间，从而保证高地址空间有较大的空闲区。

补充：最佳适应法

- 要求按空闲区大小从小到大的次序组成空闲区表/队列
- 分配：当进程申请一个存储区时，系统从表头开始查找，当找到第一个满足要求的空闲区时，停止查找，并且这个空闲区是最佳的空闲区。
- 所谓最佳即选中的空闲区是满足要求的最小空闲区。
- 回收：按释放区的首址，查询空闲区表/队列，若有与释放区相邻的空闲区，则合并到相邻的空闲区中，并修改该区的大小和首址，否则，把释放区作为一个空闲区插入空闲区表/队列。
- 分配和回收后要对空闲区表/队列重新排序。

补充：分析

■ 优点：

- 在系统中若存在一个与申请分区大小相等的空闲区，必定会被选中，而首次适应法则不一定。
- 若系统中不存在与申请分区大小相等的空闲区，则选中的空闲区是满足要求的最小空闲区，而不致于毁掉较大的空闲区。

■ 缺点：

- 空闲区的大小一般与申请分区大小不相等，因此将其一分为二，留下来的空闲区一般情况下是很小的，以致无法使用。随着时间的推移，系统中的小空闲区会越来越多，从而造成存储区的大量浪费。

补充：最坏适应法

- 要求空闲区按**大小递减**的顺序组织空闲区表/队列
- 分配：进程申请一个大小为 S 的存储区时，总是检查空闲区表的第一个空闲区的大小是否大于或等于 S 。若空闲区小于 S ，则分配失败；否则从空闲区中分配 S 的存储区给用户，然后修改和调整空闲区表。
- 回收：按释放区的首址，查询空闲区表/队列，若有与释放区相邻的空闲区，则合并到相邻的空闲区中，并修改该区的大小和首址，否则，把释放区作为一个空闲区插入空闲区表/队列。
- 分配和回收后要对空闲区表/队列重新排序。

补充：分析

■ 优点：

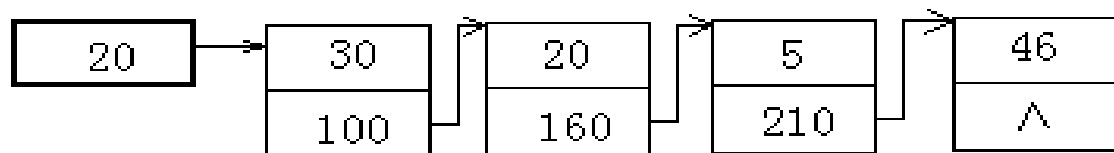
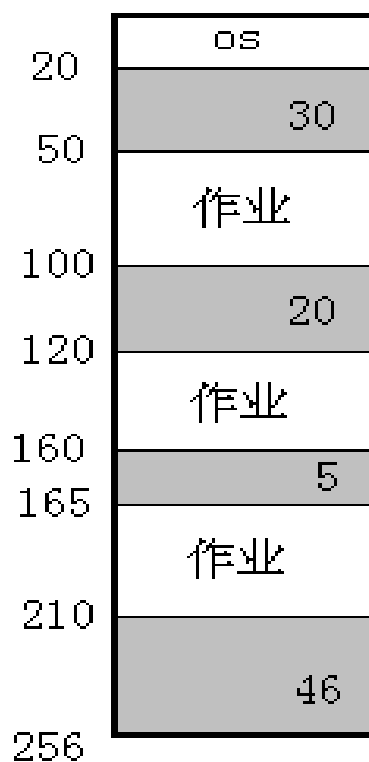
- 当程序装入内存中最大的空闲区后，剩下的空闲区还可能相当大，还能装下较大的程序。
- 另一方面每次仅作一次查询工作。

补充：三种策略比较

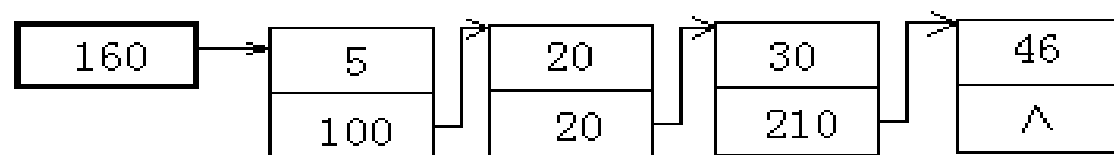
- 上述三种放置策略各有利弊，到底哪种最好不能一概而论，而应针对具体作业序列来分析。
- 对某一作业序列来说，某种算法能将该作业序列中所有作业安置完毕，那么我们说**该算法对这一作业序列是合适的**。
- 对某一算法而言，如它不能立即满足某一要求，而其它算法却可以满足此要求，则**这一算法对该作业序列是不合适的**。

补充：举例

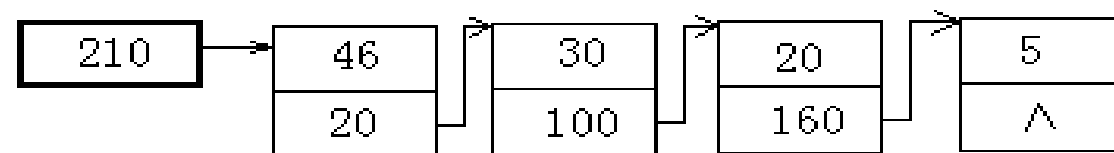
- 例1：有作业序列：A要求18K；B要求25K，C要求30K。系统中空闲区按三种算法组成的空闲区队列
- 经分析可知：最佳适应法对这个作业序列是合适的，而其它两种对该作业序列是不合适的。



首次适应法



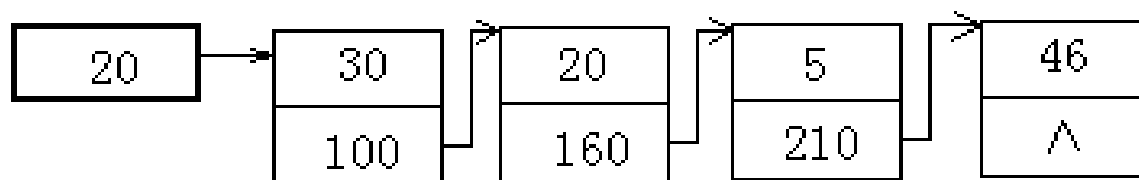
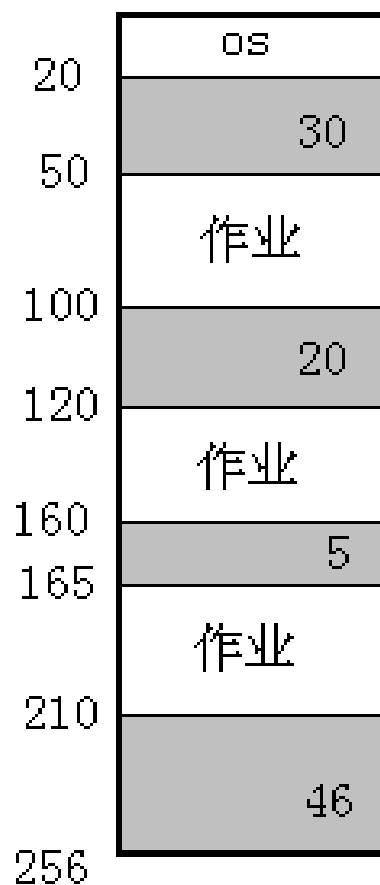
最佳适应法



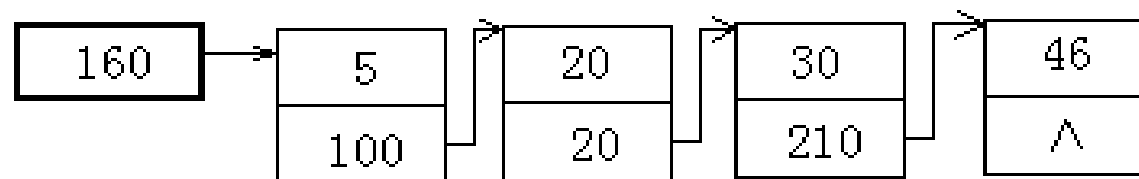
最坏适应法

补充：练习

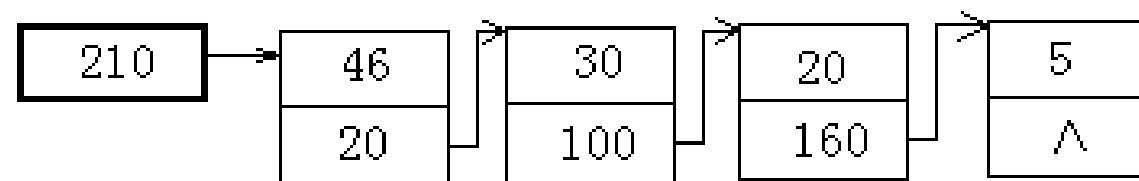
- 有作业序列：A要求21K；B要求30K，C要求25K。



首次适应法



最佳适应法



最坏适应法

补充：思考

- 用可变分区(动态重定位)方式管理主存时，假定主存中按地址顺序依次有5个空闲区，空闲区的大小依次为32K、10K、5K、228K和100K。

现在有5个作业A、B、C、D、E。它们各需主存大小为：1K、10K、108K、28K和115K。

请问：

- (1) 若采用首次适应算法，能把这5个作业按顺序全部装入主存吗？
- (2) 按怎样的次序装入这5个作业可使主存的利用率最高？

Supplement: 段页式存储管理方式

■ 页式存储管理的特点

- 以页面为单位进行内存的分配和管理，使得内存中不存在外部碎片、并且产生的内部碎片也很小，有利于提高内存的利用率。
- 不能兼顾作业本身的逻辑结构，难于实现代码段/数据段的共享。

■ 段式存储管理的特点

- 为用户程序提供一个二维的地址空间，能够反映程序的逻辑结构，便于实现存储保护和共享。
- 以段为单位进行内存分配和管理，内存中不存在内部碎片，但常常会出现较大的外部碎片，不利于提高内存的利用率。

■ 解决方法——段页式存储管理

- 把段式管理和页式管理结合起来实现对内存的管理

Supplement:

Segmentation with Paging

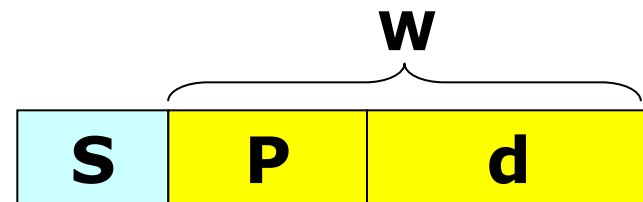
- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

Supplement: 段页式存储管理的基本思想

- 内存空间被划分成与页大小相同的页面。
- 对程序既分段又分页
 - 进程中具有独立逻辑功能的程序或数据仍被划分成段
 - 各段再分页，每个段的各页可以装入内存中不连续的页面中。

Supplement: 段页式管理的实现原理

■ 虚拟地址的构成



- 进程的二维逻辑地址空间仍被划分成段，每个段有自己的段号S。
- 段内的程序或数据，按照内存页面的大小划分为若干页。
- 进程的虚拟地址
 - 三部分：段号S、页号P、以及页内地址d
 - 程序员可见：段号S、段内相对地址W
 - 页号P和页内地址d是由地址变换机构根据W生成的

Supplement:

主要数据结构——段表和页表

- 为每个进程建立一张段表
 - 管理内存的分配与释放、存储保护和地址变换等
 - 包含每段所对应页表的起始地址和页表长度的表项
- 为每个段建立一张页表
 - 保存段中的每个逻辑页在内存中的实际内存块号

Supplement:

段表、页表与内存之间的关系示例

段表长度	段表始址
------	------

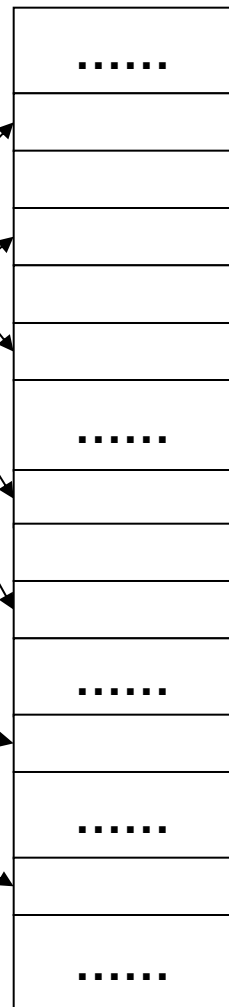
段号	其他	页表长度	页表始址
0		5	1024
1		7	1029
2		9	1036

页号	其他	页面
0		12
1		19
2		21
3		8
4		10

第**0**段的页表

页号	其他	页面
0		29
1		50
.....	
8		

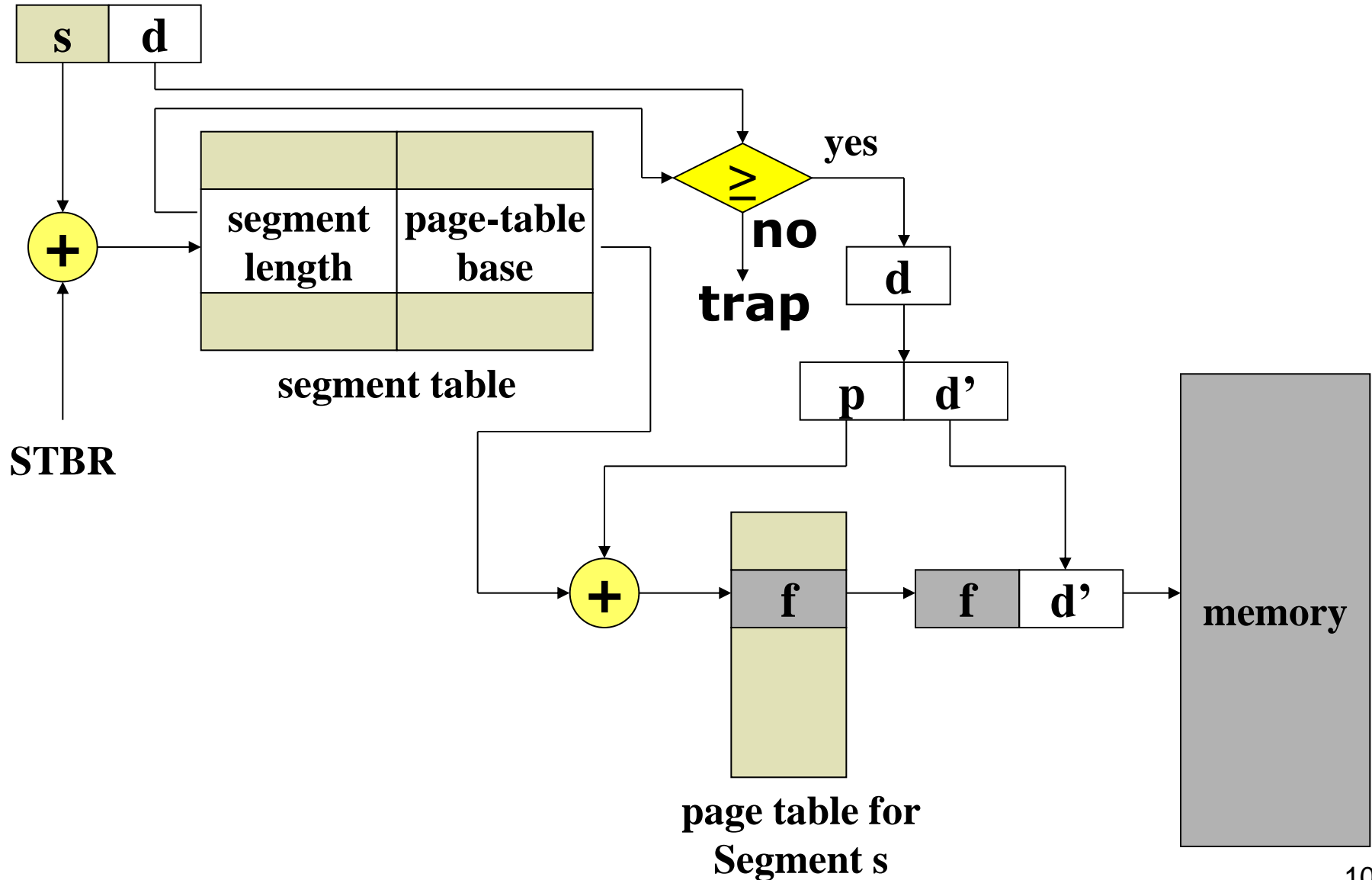
第**2**段的页表



内存

Supplement: Address Translation Scheme

logical address



Supplement: 快表 (TLB)

- 对内存中指令或数据进行一次存取的话，至少需要访问三次以上的内存。
- 设置快速联想寄存器，存放当前最常用的段号S、页号p和对应的内存块号。
- 当要访问内存空间某一单元时，首先在快速联想寄存器中查找其段号和页号。
 - 快表命中：取出对应的内存块号F、拼接物理地址、访问内存；
 - 快表失败：查找内存中的段表、页表、取得页面号，拼接物理地址，访问内存。并将查到的段号、页号、页面号等存入快表。

Chapter 9 Virtual Memory



LI Wensheng, SCS, BUPT

Teaching hours: 4h

Strong point:

Demand Paging

Page Replacement

Thrashing

contents

- 9.1 Background**
- 9.2 Demand Paging**
- 9.3 Copy-on-write**
- 9.4 Page Replacement**
- 9.5 Allocation of Frames**
- 9.6 Thrashing**
- 9.7 Memory-mapped file***
- 9.8 Allocating Kernel Memory***
- 9.9 Other considerations***
- 9.10 Operating-System Examples***

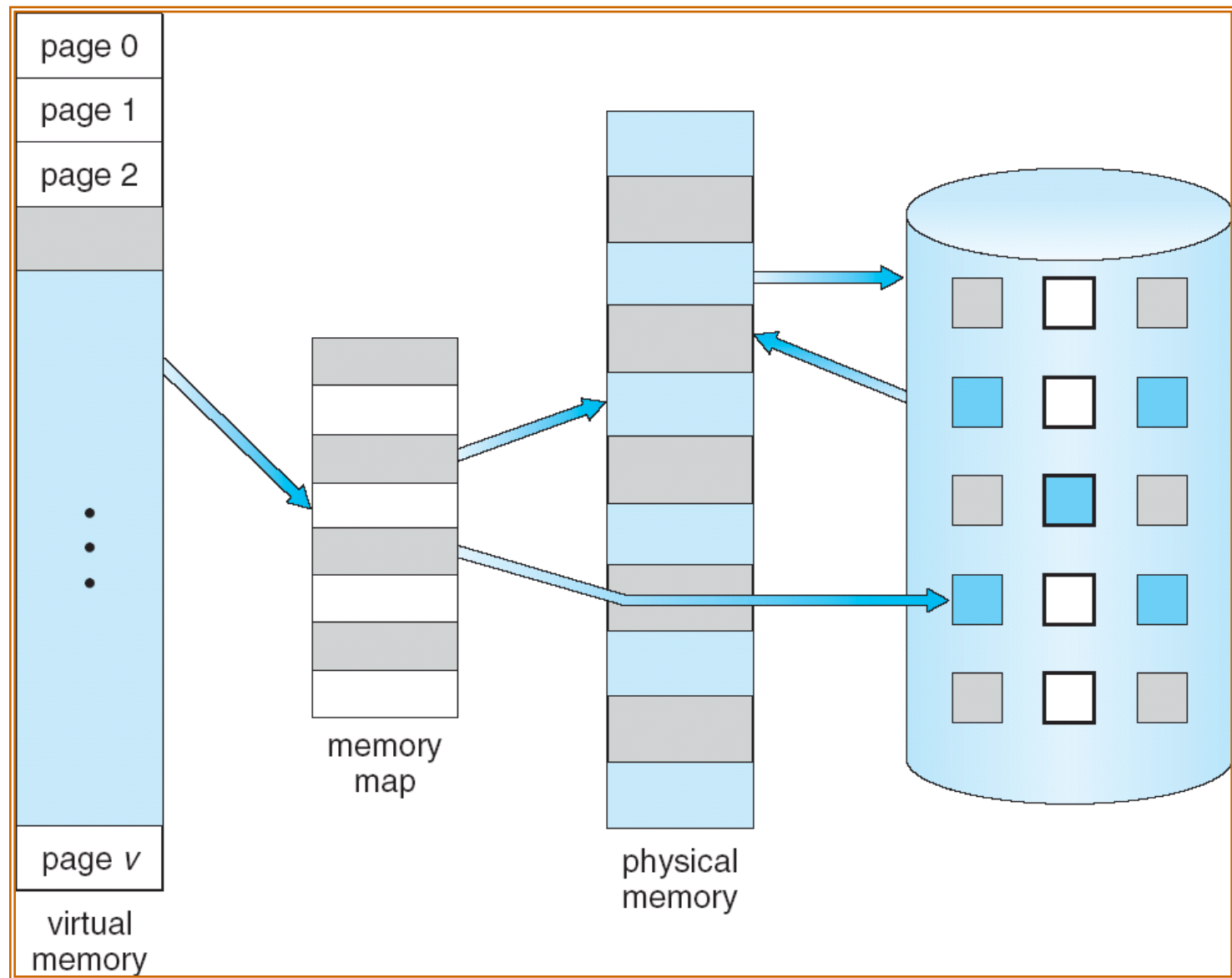
Objectives

- 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

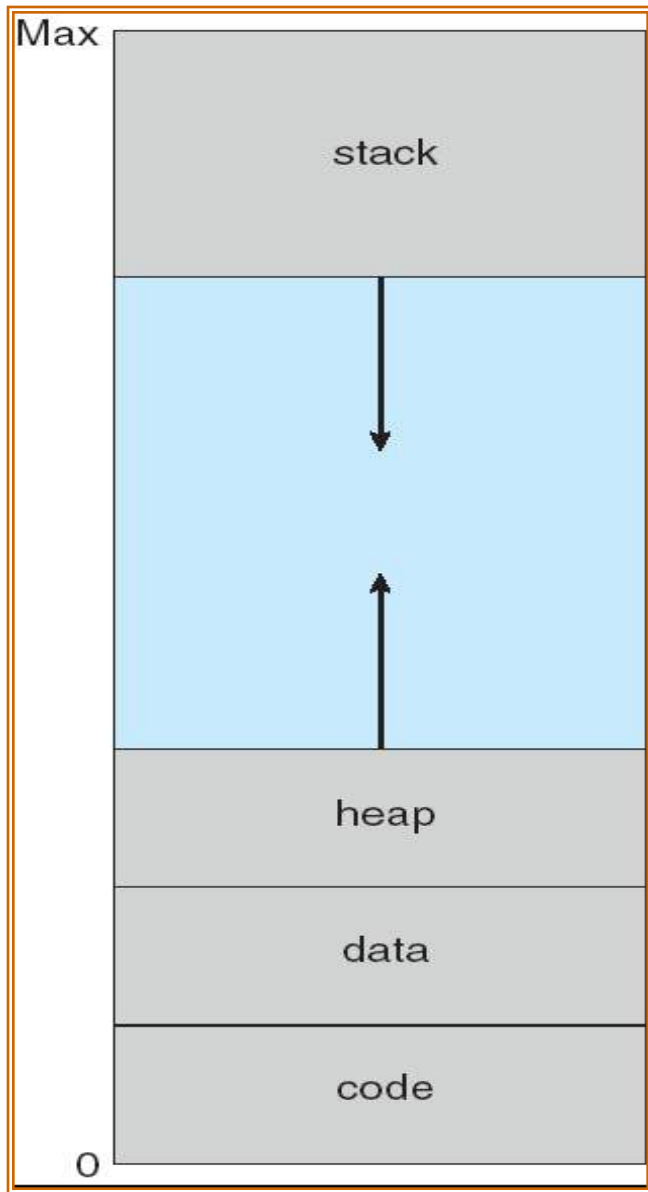
9.1 Background

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

Virtual Memory is Larger Than Physical Memory



Virtual-address Space

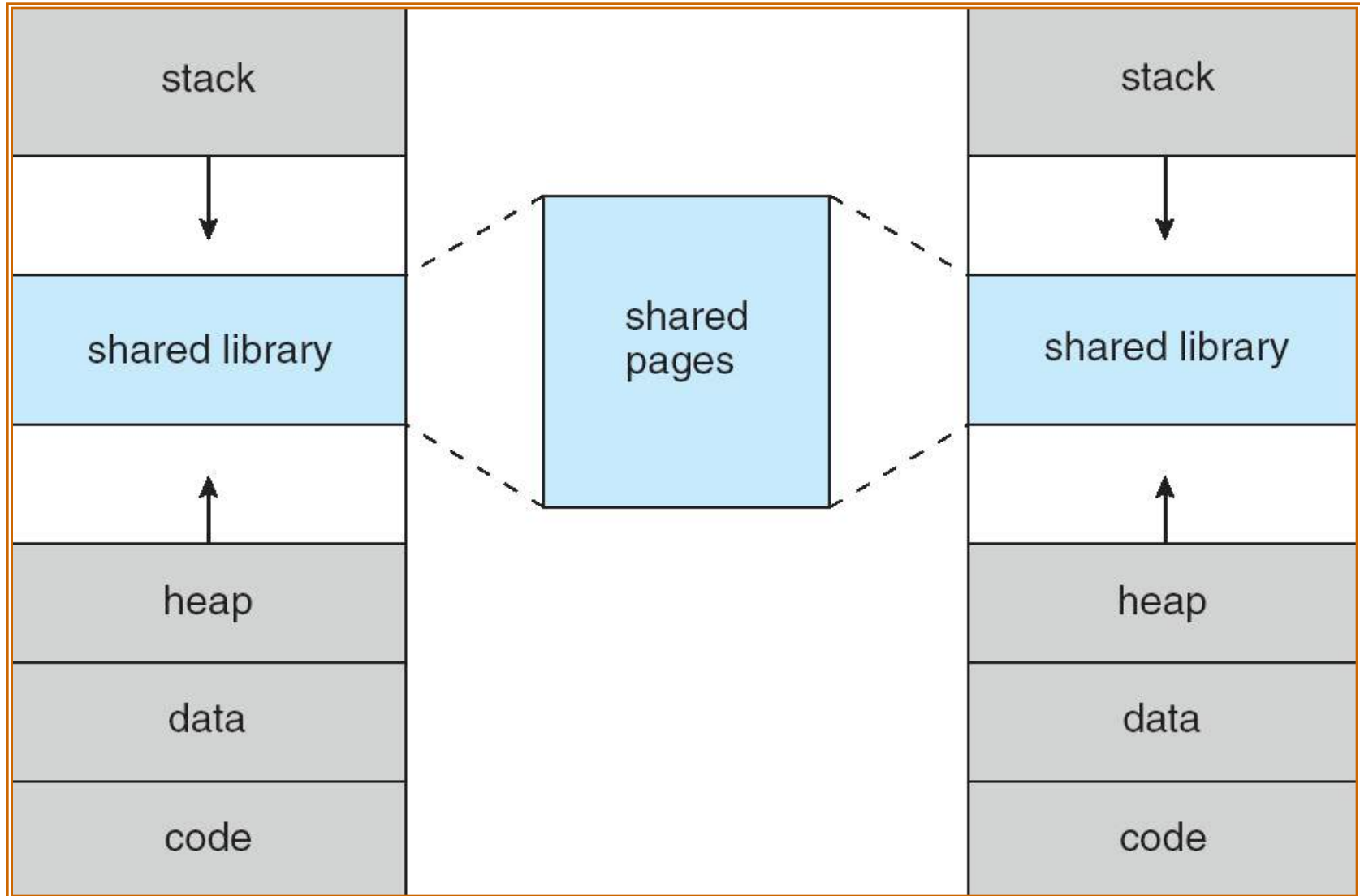


Physical memory may be organized in page frames, and the physical page frames assigned to a process may not be contiguous.

It is up to the MMU to map logical pages to physical page frames in memory.

The large blank space between the heap and the stack is part of the virtual address space, but will require actual physical pages only if the heap or stack grows.

Shared Library Using Virtual Memory

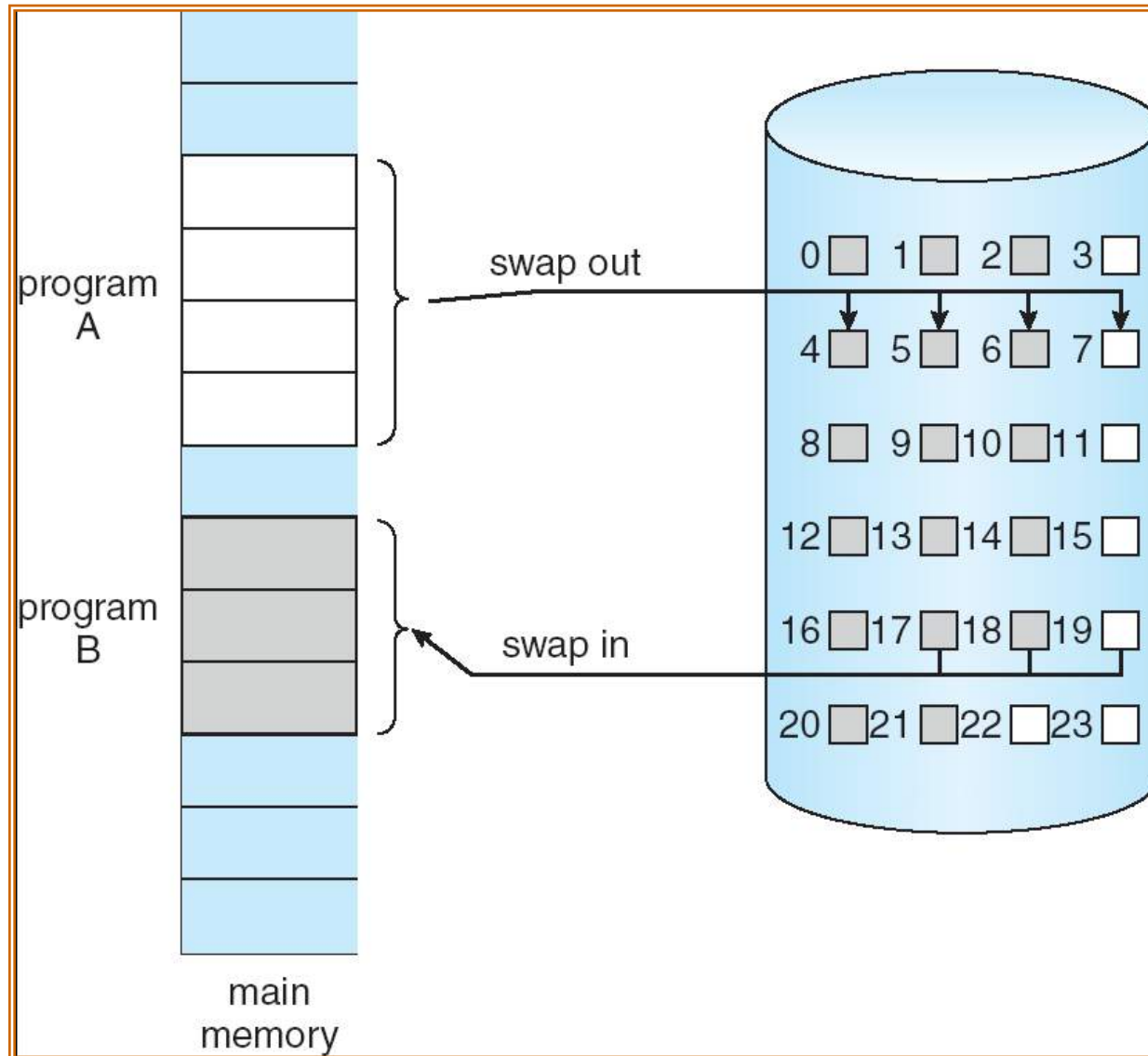


9.2 Demand Paging

- **Bring a page into memory only when it is needed.**
- **Benefits of demand paging**
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users
- **Page is needed \Rightarrow reference to it**
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring it into memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**

Swapper:

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (1 \Rightarrow in-memory, 0 \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to 0 on all entries.
- Example of a page table snapshot.

	Frame#	valid-invalid bit
0		1
1		1
2		0
3		1
4		0
5		1
6		0
7		0

Page table

- During address translation, if valid–invalid bit in page table entry is 0 \Rightarrow page fault (缺页)

Pager: lazy swapper

Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

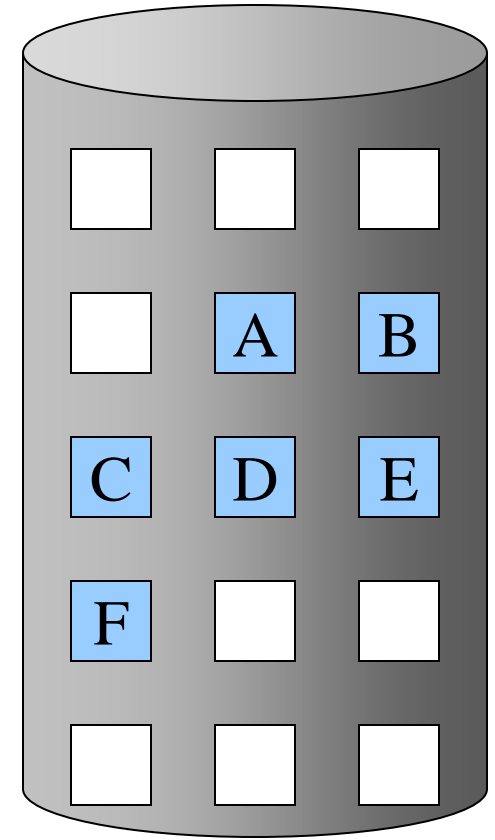
logical
memory

	valid-invalid bit
Frame#	
0	4 1
1	0
2	6 1
3	0
4	0
5	9 1
6	0
7	0

page table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

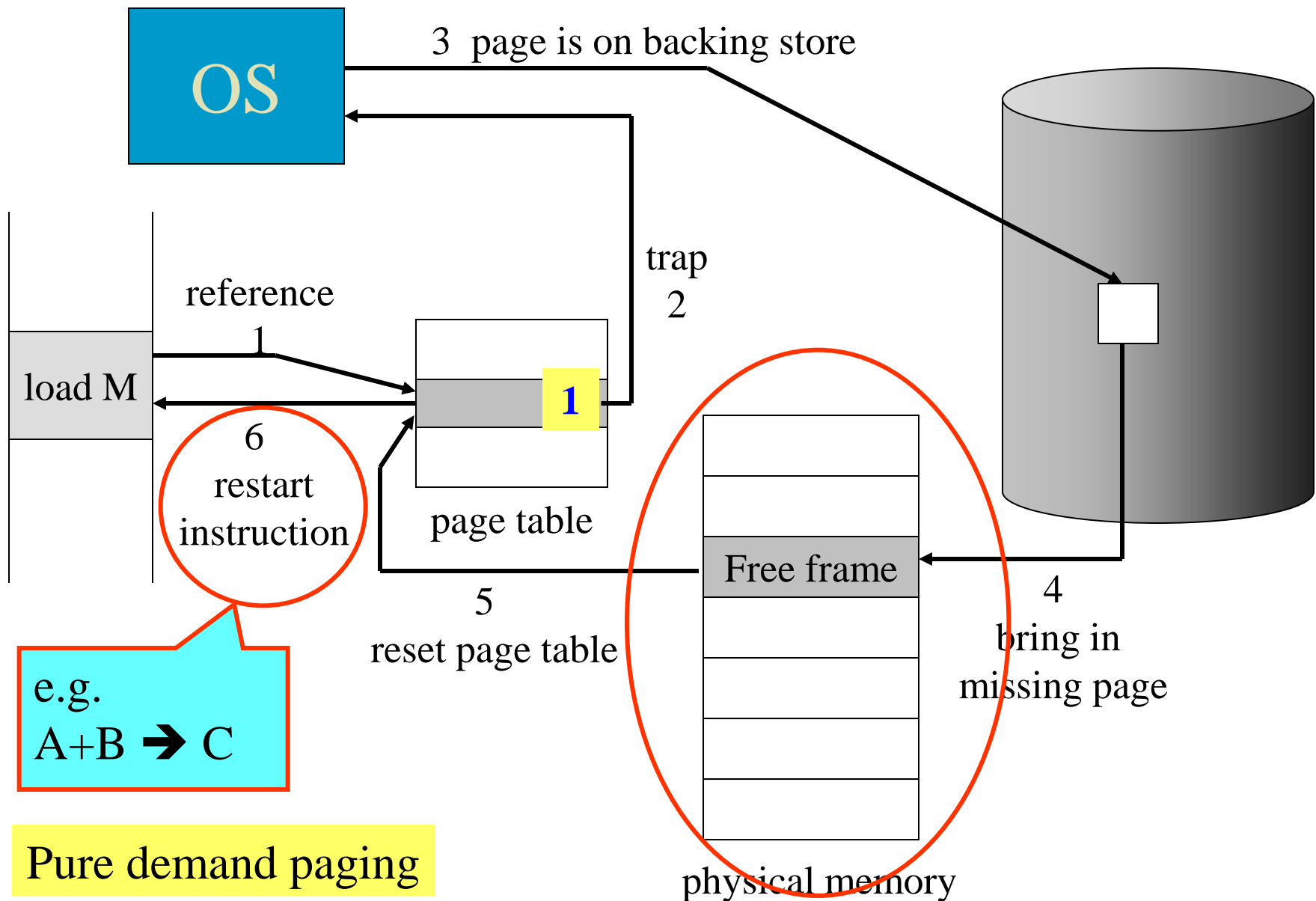
physical memory



Page Fault

- If there is ever a reference to a page, first reference will trap to OS \Rightarrow page fault
- OS looks at another internal table to decide:
 - Invalid reference \Rightarrow abort.
 - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction

Steps in Handling a Page Fault



What happens if there is no free frame?

- **Page replacement – find some page in memory, but not really in use, swap it out.**
 - algorithm
 - performance – want an algorithm which will result in minimum number of page faults.
- **Same pages may be brought into memory several times.**

Performance of Demand Paging

- **Page Fault Rate $0 \leq p \leq 1.0$**
 - if $p = 0$, no page faults
 - if $p = 1$, every reference is a fault

- **Effective Access Time (EAT)**

$$\text{EAT} = (1-p) \times \text{memory access time} \\ + \underline{p \times \text{page fault service time}}$$

$$\begin{aligned} \text{page fault service time} = & \text{page fault overhead} \\ & + [\text{swap page out}] \\ & + \text{swap page in} \\ & + \text{restart overhead} \end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- average page-fault service time ≈ 8 milliseconds
 - servicing the page-fault interrupt and restarting the process may be reduced to several hundred instructions, and each may take from 1 to 100 microseconds.
 - The page-switch time will probably be close to 8 milliseconds
 - Hard disk has an average latency of 3 milliseconds;
 - hard disk has a seek of 5 milliseconds;
 - transfer time of 0.05 milliseconds.

■ Effective Access Time

$$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p \times (8\,000\,000) \\ &= 200 + 7\,999\,800 \times p \end{aligned}$$

9.3 Copy-on-Write—process creation

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

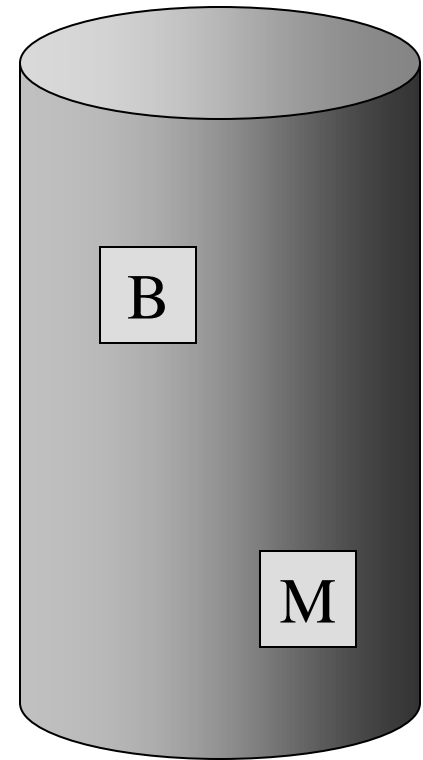
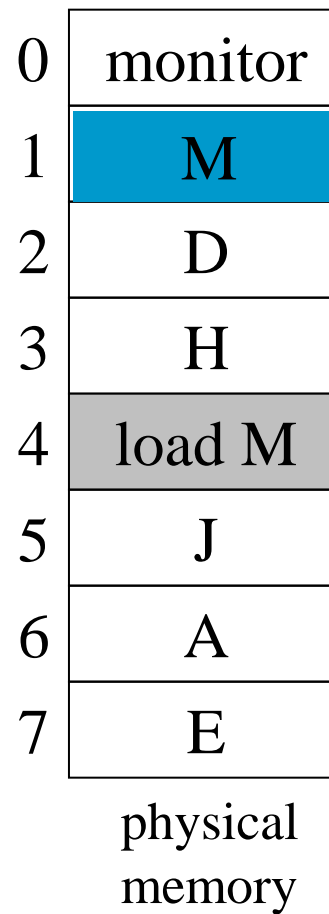
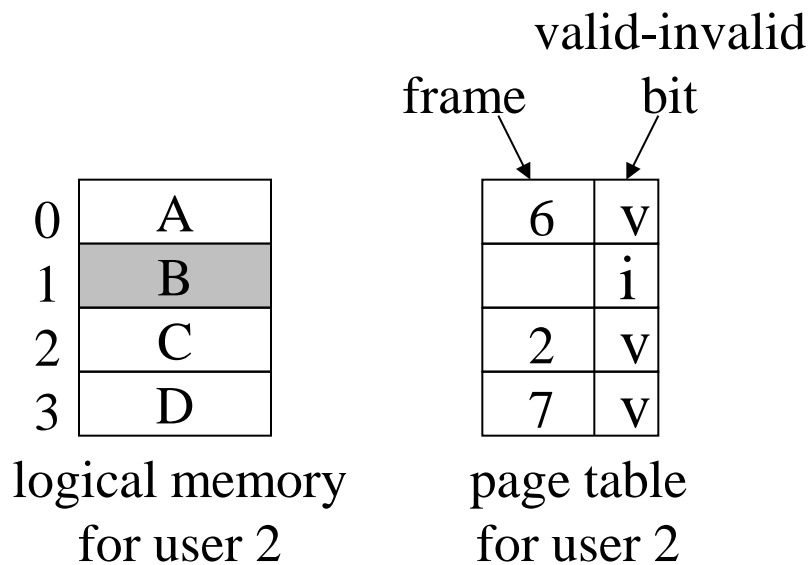
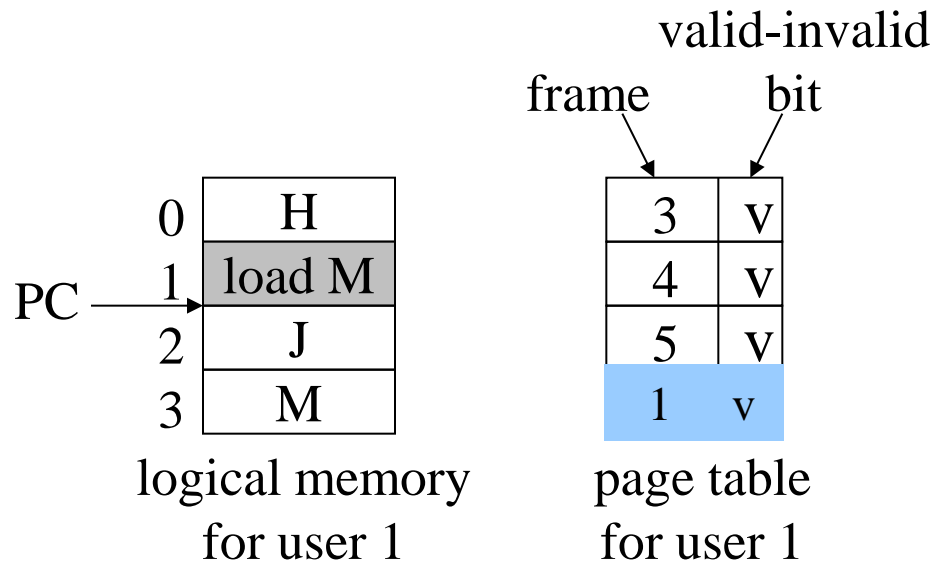
If either process modifies a shared page, only then is the page copied.

- COW allows more efficient process creation as only modified pages are copied. (see P.326 Figure 9.7-9.8)
- Free pages are allocated from a *pool* of zeroed-out pages.

9.4 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
 - modify bit is needed to indicate if the page has been altered since it was last loaded into main memory
 - If no change has been made, the page does not have to be written to the disk when it needs to be swapped out .
 - 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.

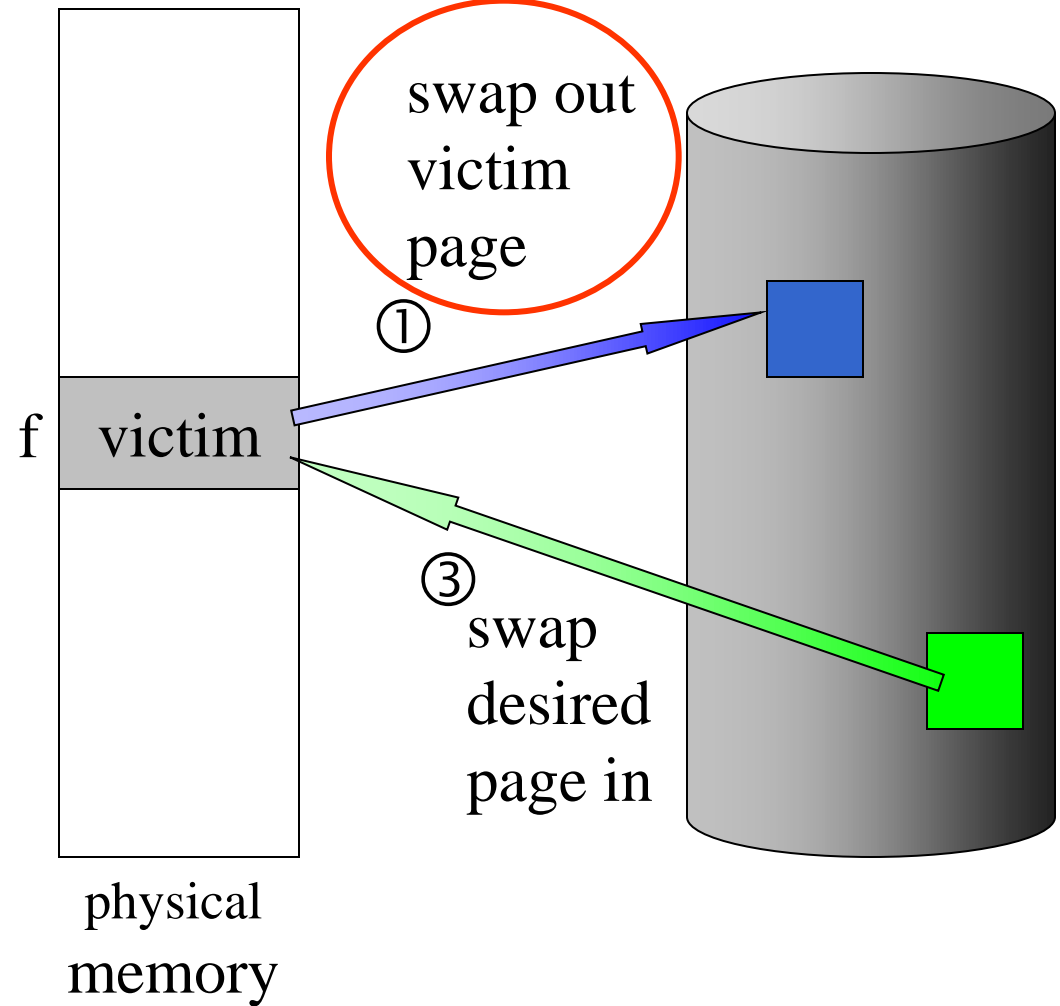
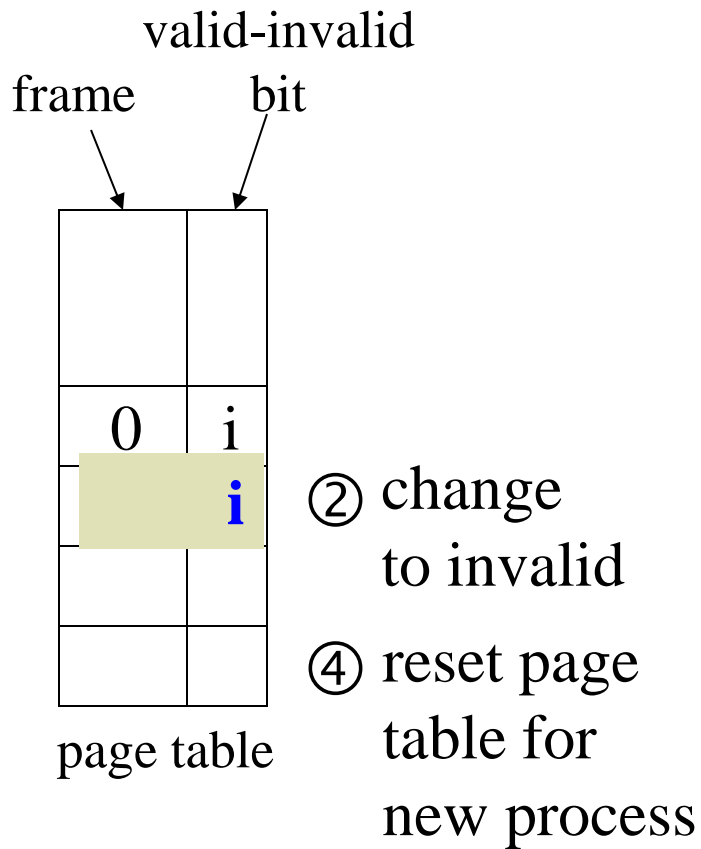
Need For Page Replacement



Basic Page Replacement

- Find the location of the desired page on disk.
- 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 the victim frame to the disk; change the page and frame tables accordingly.
- Read the desired page into the (newly) free frame. Update the page and frame tables.
- Restart the user process.

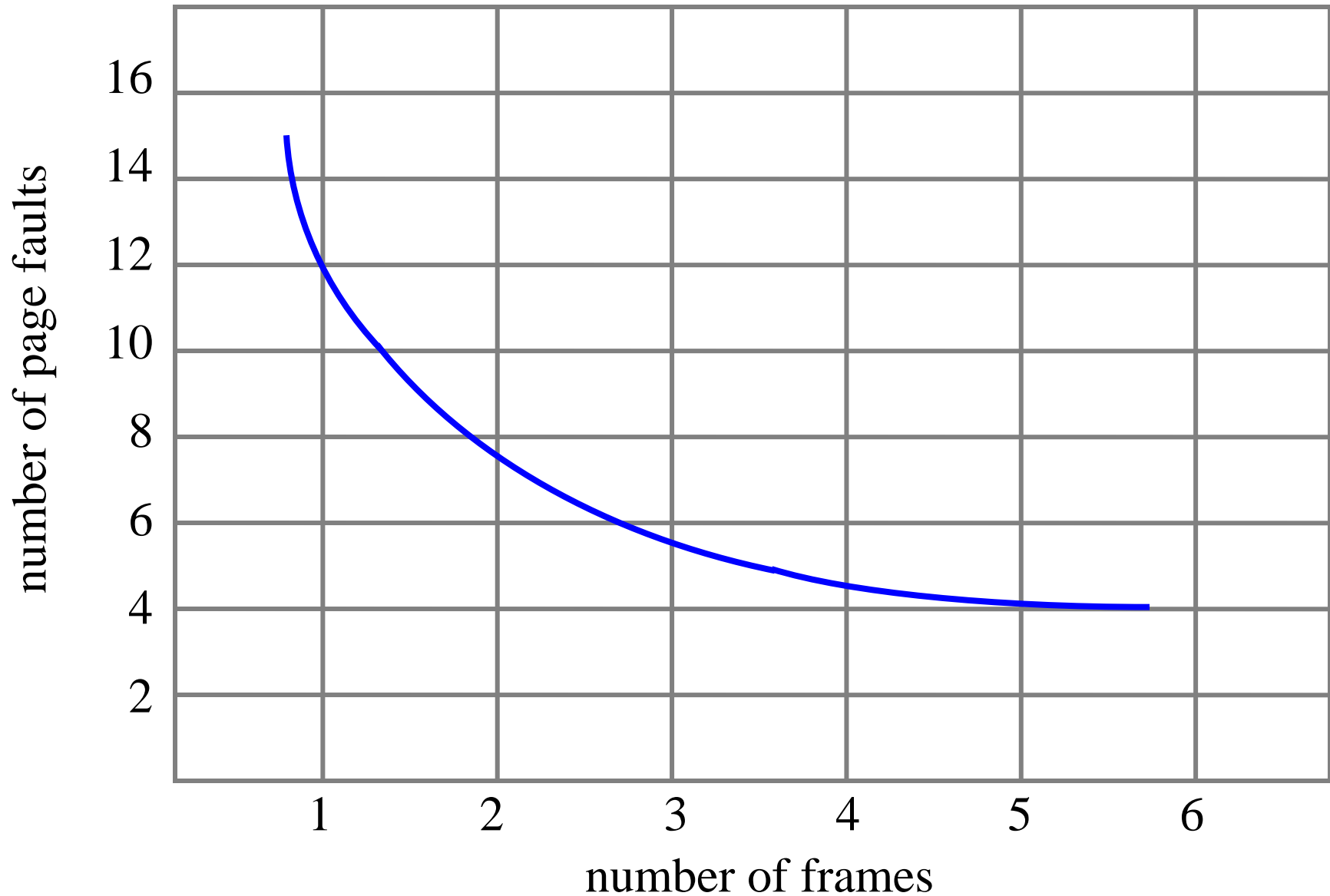
Page Replacement



Page Replacement Algorithms

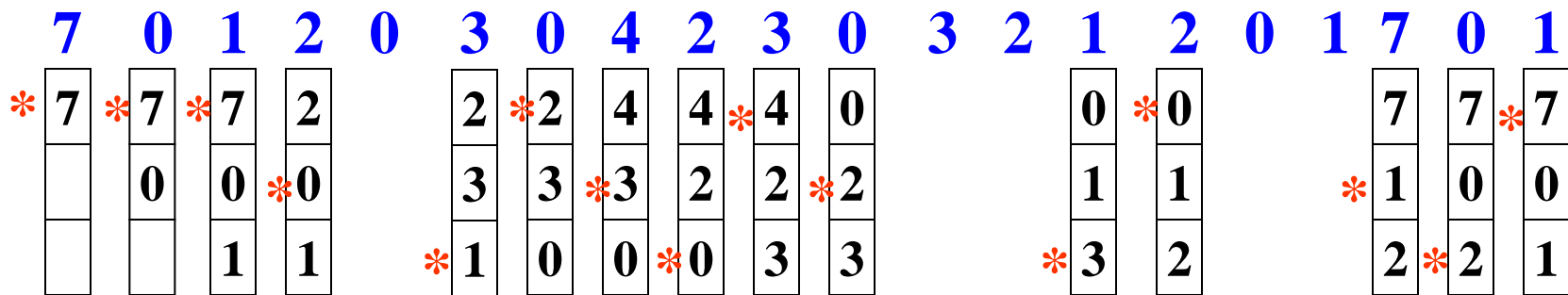
- **Placement Policy**
 - Which page is replaced?
 - Page removed should be the page least likely to be referenced in the near future
 - Most policies predict the future behavior on the basis of past behavior
- **Want lowest page-fault rate**
- **Evaluate algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string.**
- **We can generate reference strings artificially (by a random-number generator) or we can trace a given system and record the address of each memory reference.**
- **In all our examples, the reference strings are**
 - **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1**
 - **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

Graph of Page Faults Versus The Number of Frames



FIFO (First-In-First-Out) Algorithm

- Associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- Example
 - 3 frames (3 pages can be in memory at a time per process)



- Treats frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
- These pages may be needed again very soon

Optimal Algorithm

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

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

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

- Only 9 page faults
- OPT is much better than FIFO algorithm.

Least Recently Used (LRU) Algorithm

- replaces the page that has not been used for the longest period of time.
- By the principle of locality, this should be the page least likely to be referenced in the near future
- Associates with each page the time of that page's last use. This would require a great deal of overhead.
- Example

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

Belady's Anomaly more frames \Rightarrow more page faults

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames

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

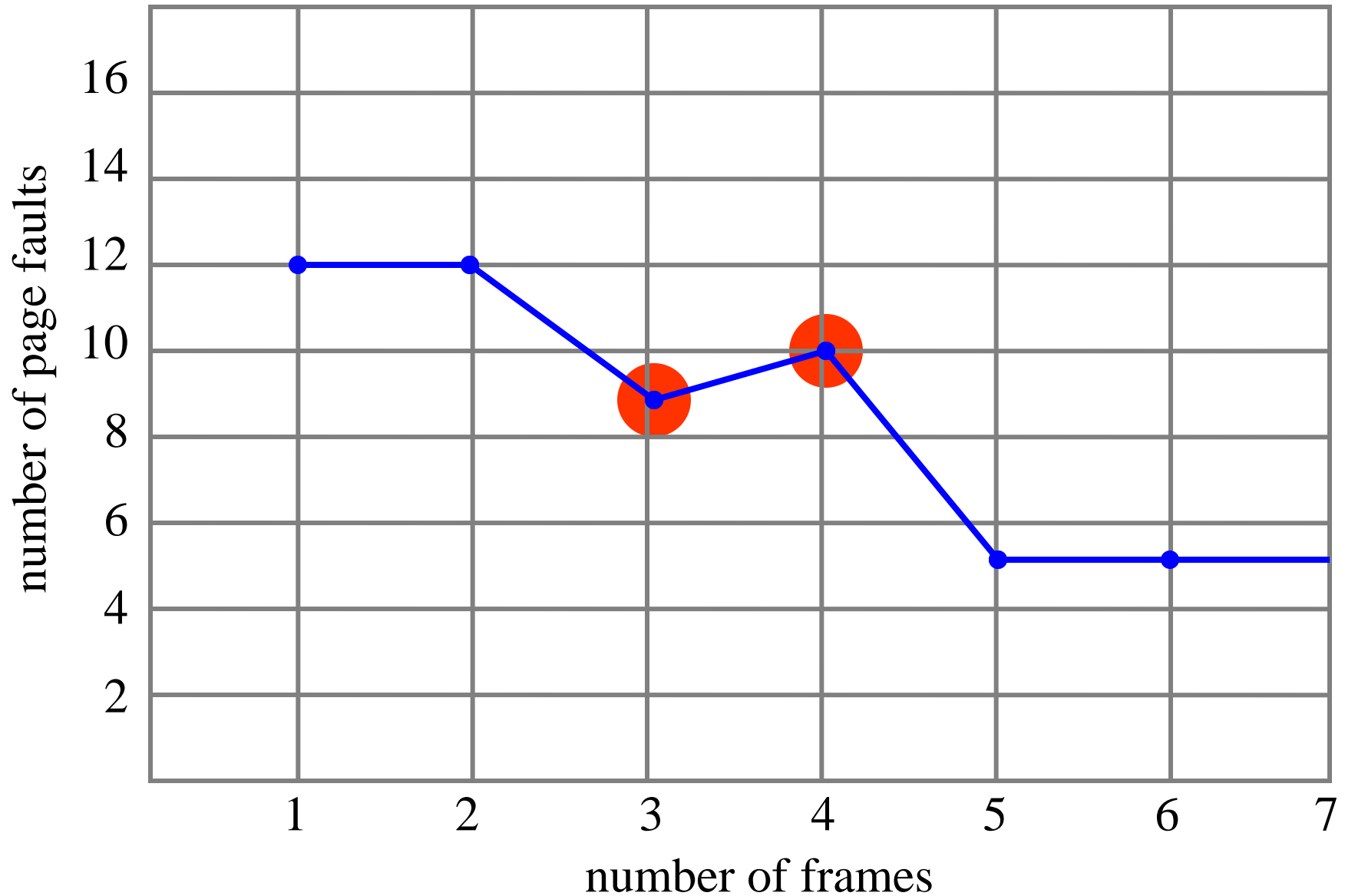
9 times page fault
Page fault rate:
 $9/12=75\%$

- 4 frames

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

10 times page fault
Page fault rate:
 $10/12=83.3\%$

FIFO Illustrating Belady's Anomaly



Optimal Algorithm example

- Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
- 3 frames

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

7 times page fault
Page fault rate:
 $7/12=58.3\%$

- 4 frames

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

6 times page fault
Page fault rate:
 $6/12=50\%$

Optimal Algorithm example

- **Thinking about:**
 - How do you know the reference string?
- **Impossible to have perfect knowledge of future events.**
- **Used for measuring how well your algorithm performs.**

LRU Algorithm example

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames

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

10 times page fault
Page fault rate:
 $10/12=83.3\%$

- 4 frames

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

8 times page fault
Page fault rate :
 $8/12=66.7\%$

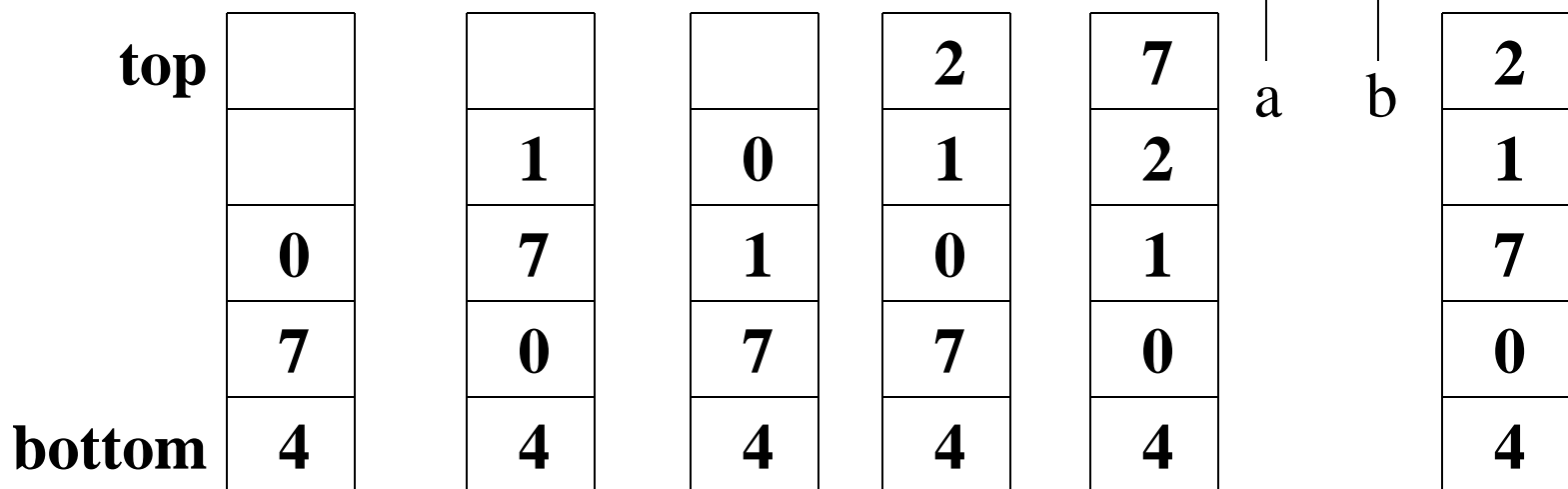
LRU implementation

■ Counters

- 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 replaced, look at the counters to determine which are to be replaced.

LRU implementation

- **Stack** – keep a stack of page numbers in a double link form:
 - **Page referenced:**
 - move it to the top
 - requires 6 pointers to be changed
 - **No search for replacement**
- **example:** 4 7 0 7 1 0 1 2 1 2 7 1 2



LRU Approximation Algorithms

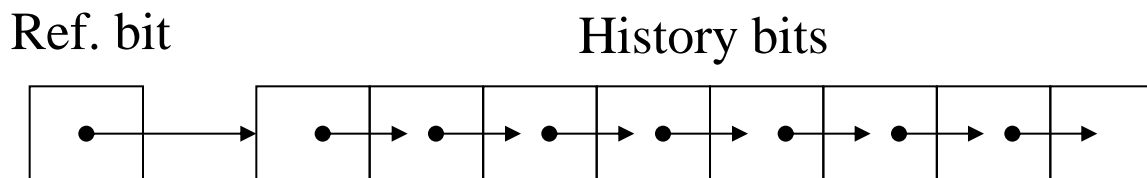
■ Reference bit algorithm

- With each page associate a bit, initially = 0.
- When page is referenced, the bit is set to 1.
- Replace the one which is 0 (if one exists).
- Problem: We do not know the order.

LRU Approximation Algorithms

■ Additional-reference-bits algorithm

- Keep an 8-bit byte for each page in a table in memory.
- At regular intervals, a timer interrupts, OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit
- These 8-bit byte contains the history of page use for the last 8 time periods.
- The page with the lowest number is the LRU page, can be replaced.

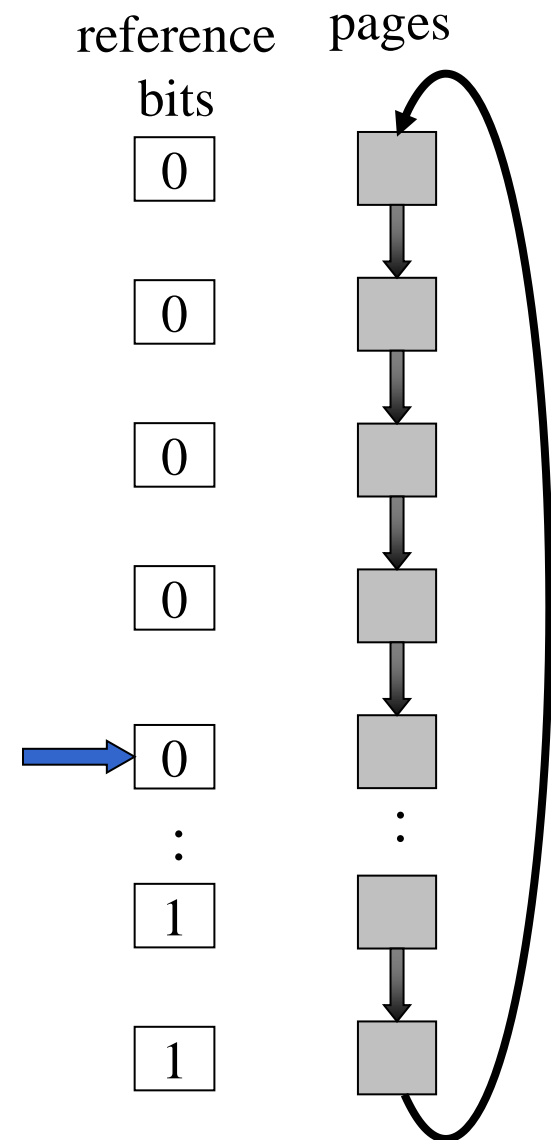
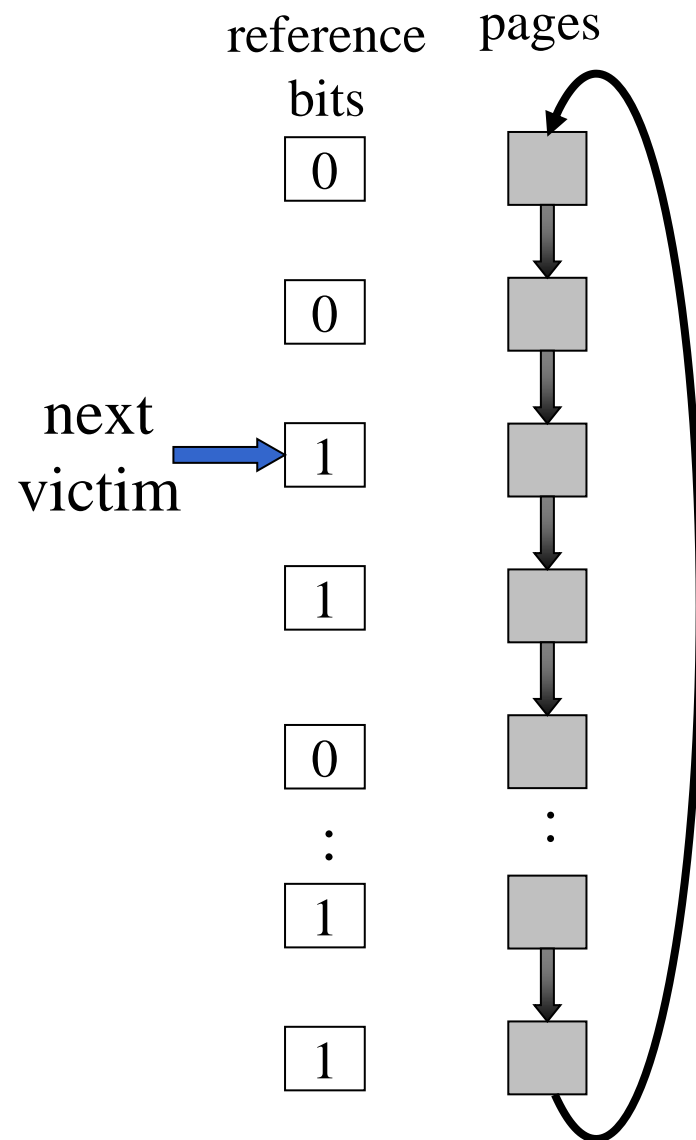


LRU Approximation Algorithms

■ Second chance

- Basic algorithm: FIFO replacement algorithm
- Need reference bit
- Clock replacement
- If page to be replaced (in clock order) has reference bit = 1.
then:
 - set reference bit 0.
 - leave page in memory.
 - replace next page (in clock order), subject to same rules.

Second-Chance (clock) Page-Replacement Algorithm



LRU Approximation Algorithms

Enhanced Second-Chance Algorithm

- Considering both the reference bit and the modify bit as an ordered pair.
- Each page is in one of four classes.
 - **(0,0)** neither recently used nor modified -- best page to replace
 - **(0,1)** not recently used but modified -- not quite good, the page will need to be written out before replacement.
 - **(1,0)** recently used but clean -- probably will be used again soon
 - **(1,1)** recently used and modified -- probably will be used again soon, and the page will need to be written out to disk before it can be replaced.

LRU Approximation Algorithms

Counting-based page replacement

- **Keep a counter of the number of references that have been made to each page.**
- **LFU Algorithm: Least Frequently Used**
 - replaces page with the smallest count.
- **MFU Algorithm: Most Frequently Used**
 - based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

* Page-Buffering Algorithm

- **Keep a pool of free frames**
 - When page fault occurs, the desired page is read into a free frame from the pool before the victim is written out.
 - When the victim is later written out, its frame is added to the free-frame pool.
- **Maintain a list of modified pages**
 - Whenever the paging device is idle, a modified page is selected and is written to the disk.
- **Keep a pool of free frames, but to remember which page was in each frame**
 - The old page can be reused directly from the free-frame pool if it is needed before that frame is reused.

9.5 Allocation of Frames

- Each process needs **minimum** number of pages.
- 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.
- Example:
 - All memory reference instructions have only one memory address
 - One-level indirect addressing is allowed
 - Requires at least 3 frames per process
- Two major allocation schemes
 - fixed allocation
 - priority allocation

Fixed Allocation

- **Equal allocation** – e.g., if 100 frames and 5 processes, give each 20 pages.
- **Proportional allocation** – Allocate according to the size of process.
 - s_i size of virtual memory for process p_i
 - $S = \sum s_i$
 - m total number of frames
 - a_i allocation for process p_i
$$a_i = s_i / S \times m$$
- when a page fault occurs, one of the pages of that process must be replaced.
- **Local replacement**

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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.
Local replacement
 - select for replacement a frame from a process with lower priority number.
Global replacement

Global vs. Local replacement

■ Global replacement

- process selects a replacement frame from the set of all frames; one process can take a frame from another.
- Easiest to implement, Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs

■ Local replacement

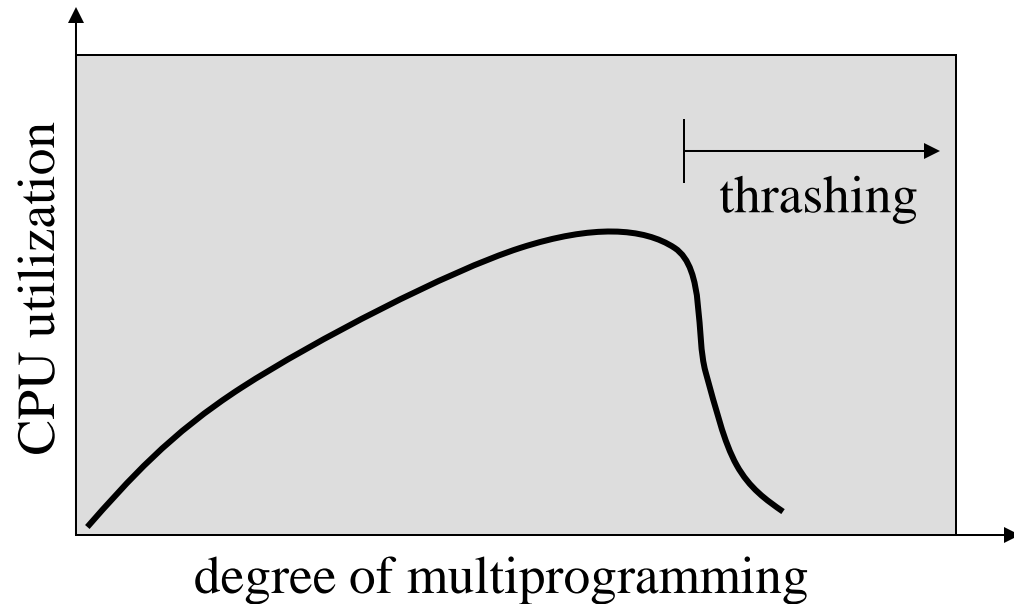
- When page fault occurs, select page from among the resident set of the process that suffers the fault.
- When new process added, allocate number of page frames based on application type, program request, or other criteria

- One problem with a global replacement algorithm is that a process cannot control its own page-fault rate.

9.6 Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high.
- The process spends most of its time swapping pages rather than executing user instructions
- Swapping out a page of a process just before that page is needed
- **Thrashing** \equiv a process is busy swapping pages in and out.
- Thrashing leads to:
 - low CPU utilization.
 - operating system thinks that it needs to increase the degree of multiprogramming.
 - another process added to the system.

Thrashing



■ How does paging work?

Locality model

- Process migrates from one locality to another.
- Locality: a set of pages that are actively used together
- Localities may overlap.

■ Why does thrashing occur?

Σ size of locality > total memory size

Principle of Locality

- **Program and data references within a process tend to cluster**
- **Only a few pieces of a process will be needed over a short period of time**
- **Possible to make intelligent guesses about which pieces will be needed in the future**
- **This suggests that virtual memory may work efficiently**

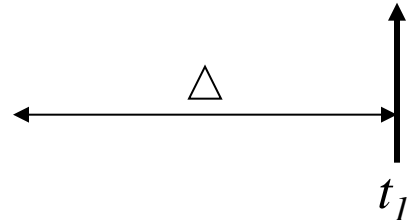
Working-Set Model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instruction
- WSS_i (working set size 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
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes.

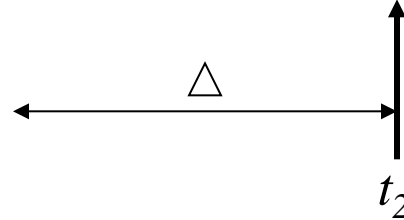
Example of Working-set

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$

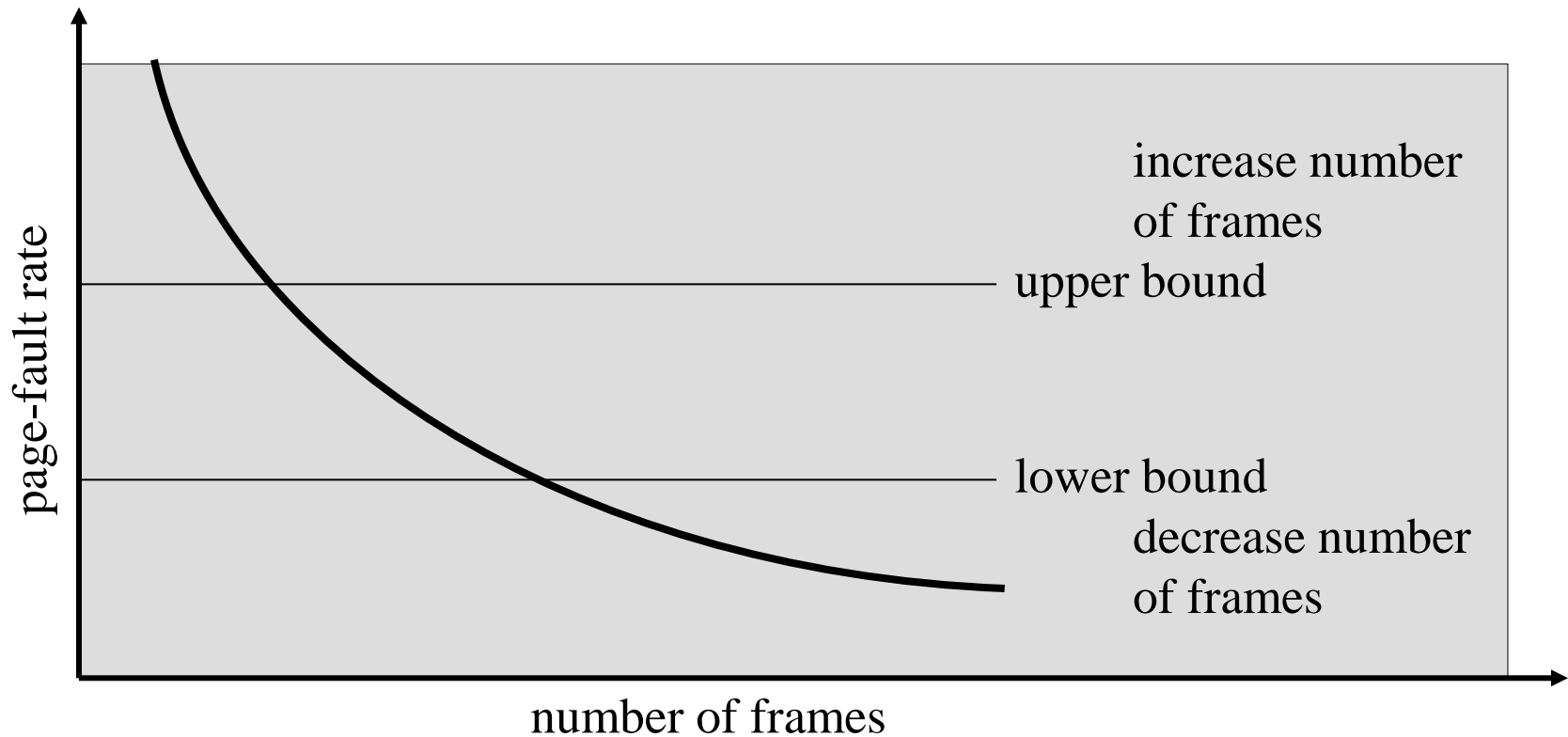


$$WS(t_2) = \{3, 4\}$$

Keeping Track of the Working Set

- **Approximate with interval timer + a reference bit**
- **Example: $\Delta = 10,000$ references**
 - **Timer interrupts after every 5000 time units.**
 - **Keep 2 in-memory bits for each page.**
 - **Whenever a timer interrupts, copy and set 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 history in-memory bits and interrupt every 1000 time units.**

Page-Fault Frequency Scheme

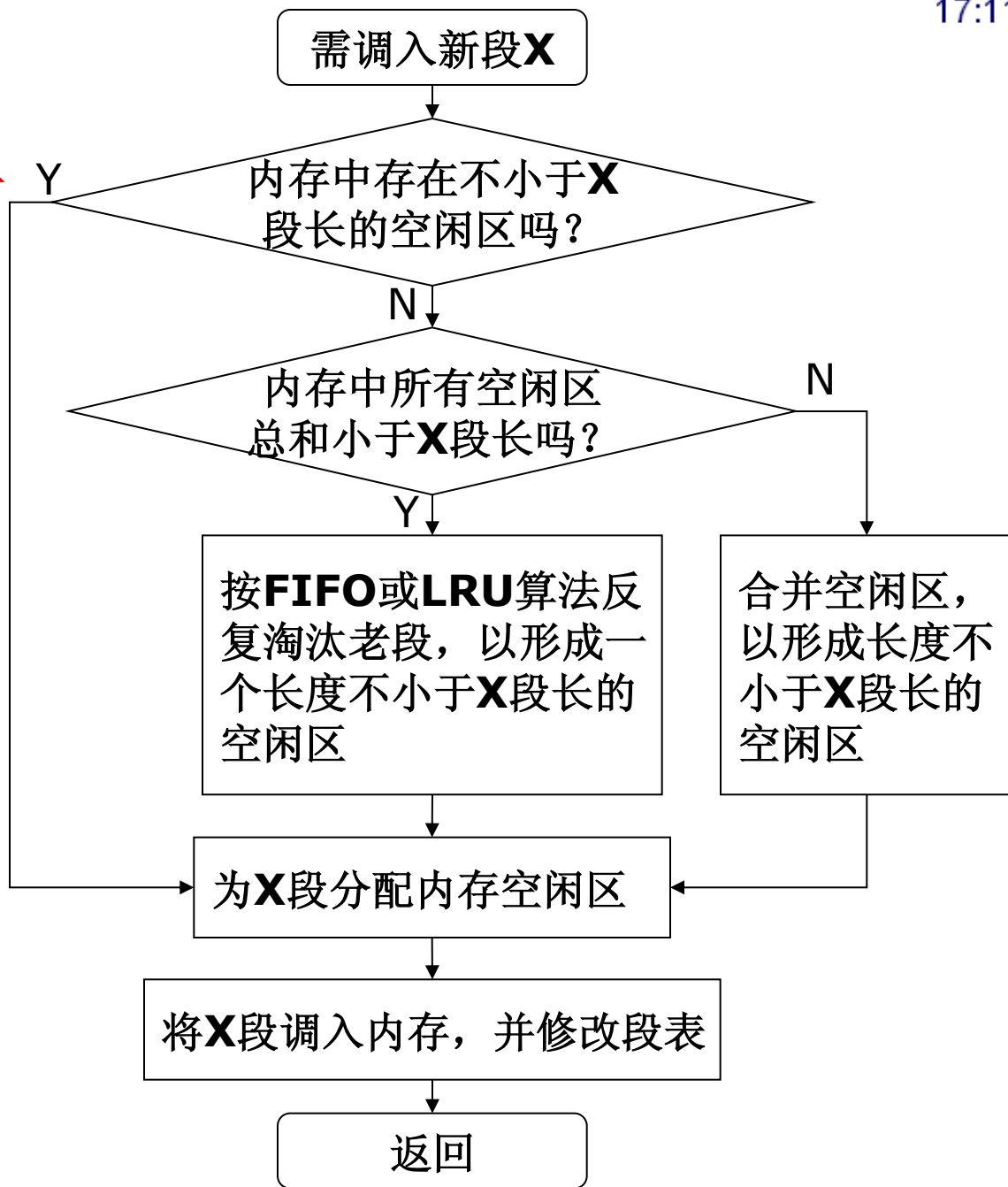


- Establish “acceptable” page-fault rate.
 - If actual rate too low, process loses frame.
 - If actual rate too high, process gains frame.

Supplement:

缺段中断及处理

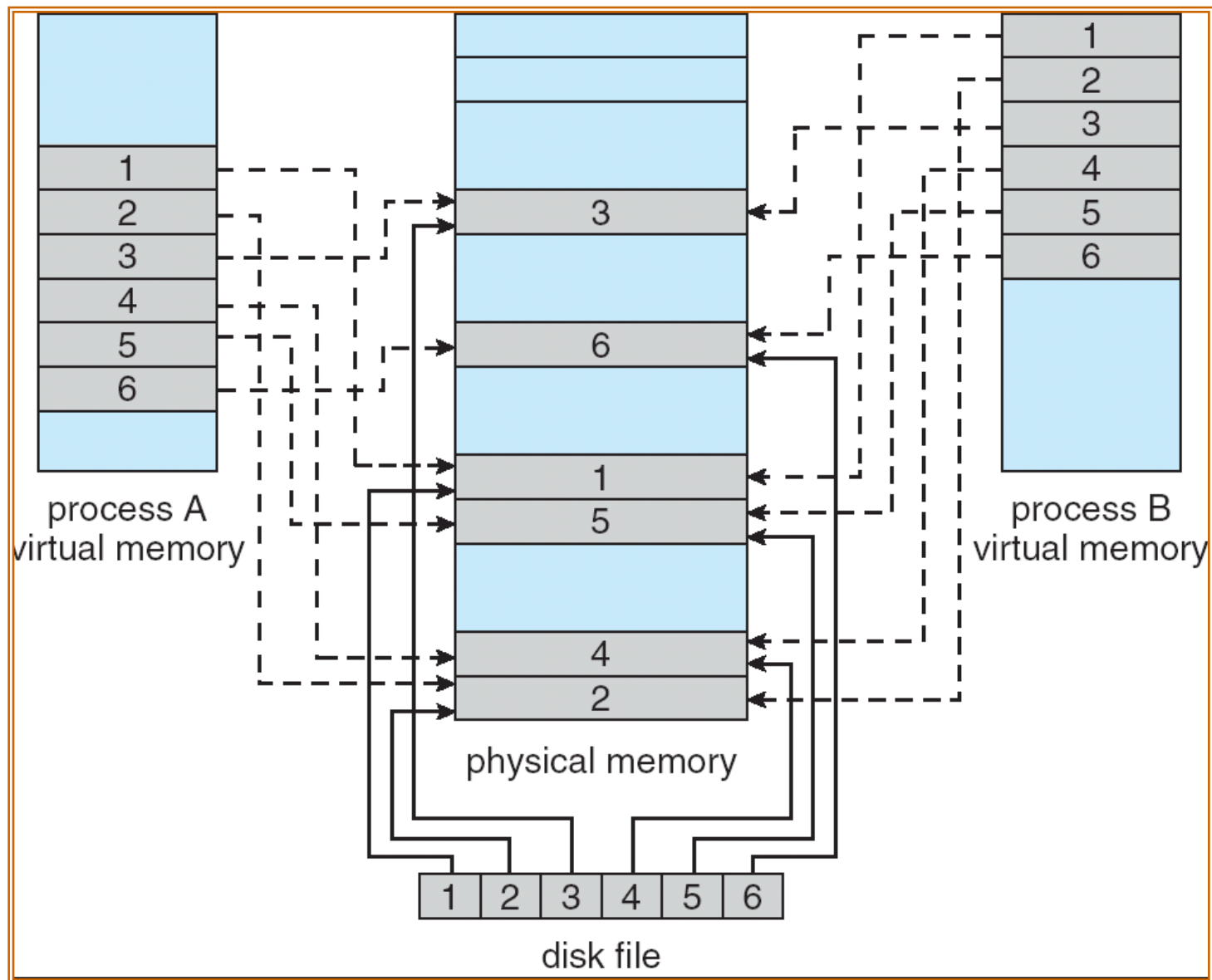
- 当**CPU**要访问的指令/数据不在内存中时，产生“缺段中断”。
- **OS**捕获中断，并调用相应的中断处理程序，进行处理。
- 缺段处理过程：



9.7 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 file access by treating 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

Memory Mapped Files



memory-mapping files in Win32--producer

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

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

```
int main(int argc, char *argv[])
```

```
{
```

```
    HANDLE hFile, hMapFile;
```

```
    LPVOID lpmapAddress;
```

```
    // first create/open the file
```

```
    hFile = CreateFile("temp.txt",    // file name
```

```
        GENERIC_READ | GENERIC_WRITE,    // read/write access
```

```
        0,    // no sharing of the file
```

```
        NULL,    // default security
```

```
        OPEN_ALWAYS,    // open new or existing file
```

```
        FILE_ATTRIBUTE_NORMAL,    // routine file attributes
```

```
        NULL);    // no file template
```

```
    if (hFile == INVALID_HANDLE_VALUE) {
```

```
        fprintf(stderr, "Could not open file temp.txt (%d).\n", GetLastError());
```

```
        return -1;
```

```
    }
```

```
// now obtain a mapping for it
hMapFile = CreateFileMapping(hFile, // file handle
    NULL, // default security
    PAGE_READWRITE, // read/write access to mapped pages
    0, // map entire file
    0,
    TEXT("SharedObject")); // named shared memory object
if (hMapFile == NULL) {
    fprintf(stderr, "Could not create mapping (%d).\n", GetLastError());
    return -1;
}

// now establish a mapped viewing of the file
lpmapAddress =
MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
if (lpmapAddress == NULL) {
    printf("Could not map view of file (%d).\n", GetLastError());
    return -1;
}
```

```
// write to shared memory  
sprintf(lpmapAddress,"%s","Shared memory message");  
  
while (1);  
// remove the file mapping  
UnmapViewOfFile(lpmapAddress);  
  
// close all handles  
CloseHandle(hMapFile);  
CloseHandle(hFile);  
}
```


memory-mapping files in Win32--consumer

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

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

```
int main(int argc, char *argv[]) {
```

```
    HANDLE hMapFile;
```

```
    LPVOID lpMapAddress;
```

```
    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,      //  
        read/write permission
```

```
        FALSE,          // Do not inherit the name
```

```
        TEXT("SharedObject")); // name of the mapped file object
```

```
    if (hMapFile == NULL)
```

```
{
```

```
    printf("Could not open file mapping object (%d).\n", GetLastError());
```

```
    return -1;
```

```
}
```

```
lpMapAddress = MapViewOfFile(hMapFile, // handle to mapping object  
FILE_MAP_ALL_ACCESS, // read/write permission  
0, // max. object size  
0, // size of hFile  
0); // map entire file
```

```
if (lpMapAddress == NULL)  
{  
    printf("Could not map view of file (%d).\n", GetLastError());  
    return -1;  
}
```

```
// read from shared memory
```

```
printf("read message: %s\n",lpMapAddress);
```

```
UnmapViewOfFile(lpMapAddress);
```

```
CloseHandle(hMapFile);
```

```
}
```

9.8 Allocating Kernel Memory

- **Treated differently from user memory .**
- **Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. Reasons are:**
 - **The kernel request memory for data structures of varying sizes, some of which are less than a page in size.**
 - **Some kernel memory needs to be contiguous. Certain hardware devices interact directly with physical memory, and may require memory residing in physically contiguous pages.**
- **Buddy system**
- **Slab allocation**

Buddy System (伙伴系统)

- Allocates memory from a fixed-size segment consisting of physically contiguous pages.
- Memory is allocated by using a **power-of-2 allocator**.
- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s \leq 2^U$, entire block is allocated
 - Otherwise block is split into two equal buddies
 - Process continues until smallest block greater than or equal to s is generated

Example of Buddy System

1MB block

A: Request 100KB

B: Request 240KB

C: Request 64KB

D: Request 256KB

Release B

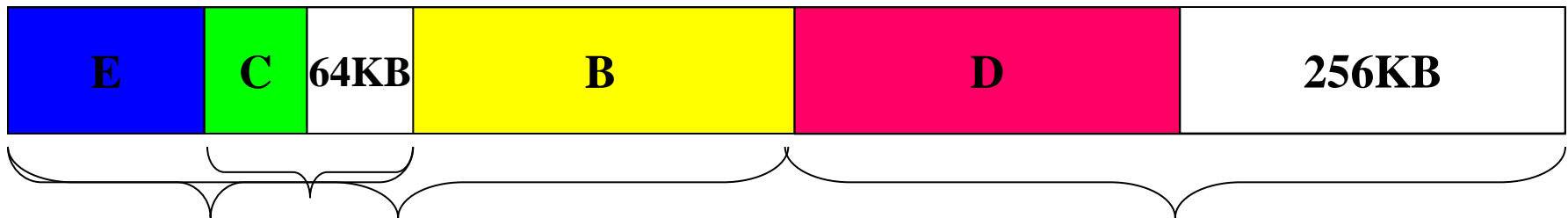
Release A

E: Request 75KB

Release C

Release E

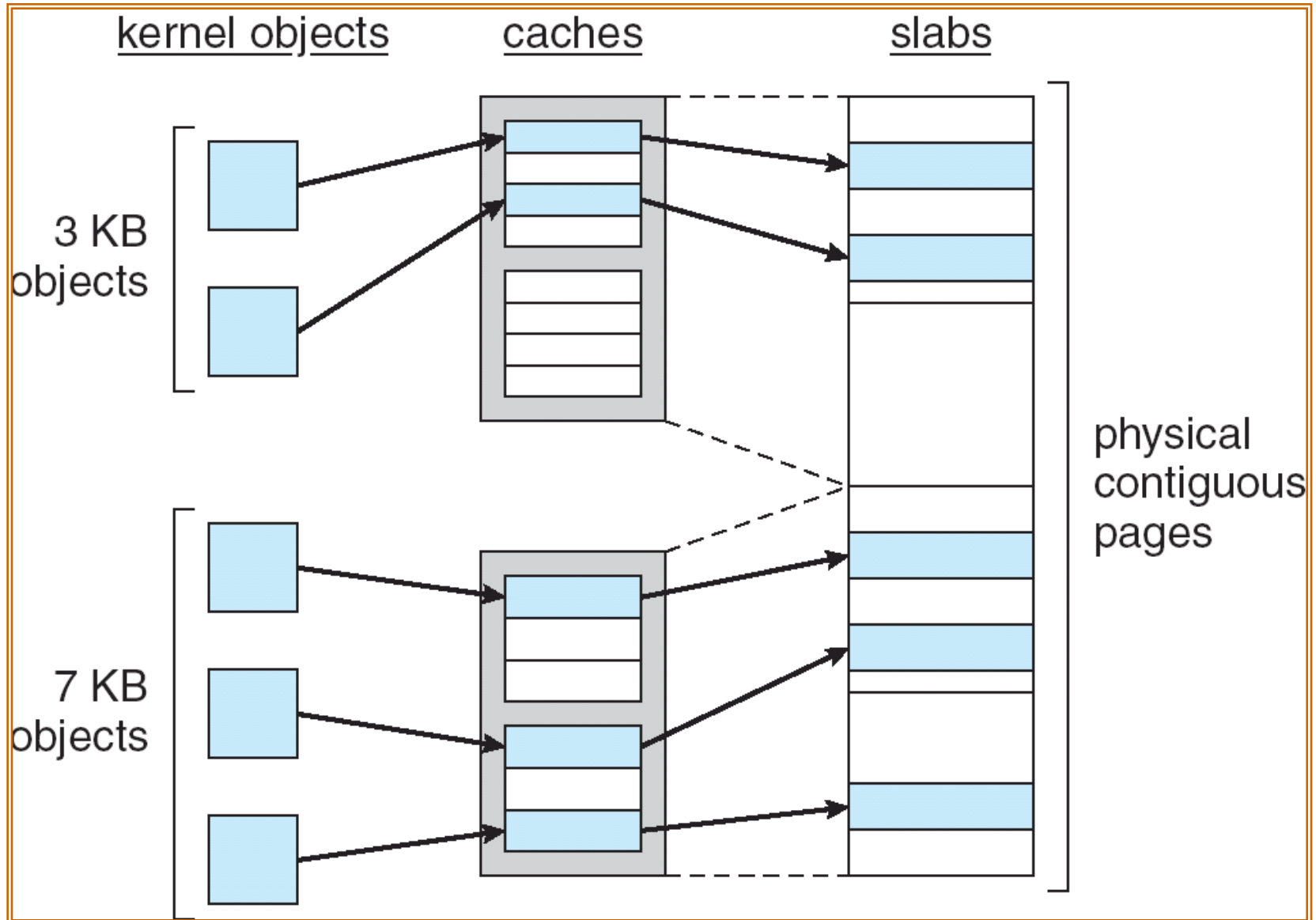
Release D



Slab allocation

- A **slab** is made up of one or more physically contiguous pages.
- A **cache** consists of one or more slabs.
- There is a single cache for each unique kernel data structure
 - Cache for PCBs
 - Cache for file objects
 - Cache for semaphores
- Each cache is filled with objects -- instantiations of the kernel data structure the cache represents.

The relationship between slabs, caches and objects



Slab allocation

- When cache created, filled with objects marked as **free**
 - The number of objects in the cache depends on the size of the associated slab.
- 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
 - no fragmentation
 - fast memory request satisfaction

9.9 Other Considerations-- 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.
- Prepaging may be an advantage in some cases.
 - whether the cost of using prepaging is less than the cost of servicing the corresponding page faults.
- Example
 - S pages are prepaged
 - a fraction α of these S pages is actually used ($0 < \alpha < 1$)
 - the cost of the $S \times \alpha$ saved page faults is greater or less than the cost of prepaging $S \times (1 - \alpha)$ unnecessary pages
 - if α is close to 0, prepaging loses; if α is close to 1, prepaging wins.

Other Issues --Page size selection

- **size of the page table**
 - each active process must have its own copy of the page table, a **large page size** is desirable.
- **fragmentation**
 - to minimize internal fragmentation, **small page size** is needed.
- **I/O overhead**
 - $\text{I/O time} = \text{seek time} + \text{latency time} + \text{transfer time}$
 - seek and latency time normally dwarf transfer time
 - to minimize I/O time, **large page size** is desired
- **locality**
 - with a small page size, locality will be improved, total I/O should be reduced
 - to minimize the number of page faults, a **large page size** is need.

Example Page Sizes

Computer	page size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes to 4 Mbytes
PowerPC	4 Kbytes

Other Issues --TLB Reach

- The amount of memory accessible from the TLB.
- **TLB Reach = (TLB Size) × (Page Size)**
- Ideally, the working set of each process is stored in the TLB. Otherwise the process will spend a considerable amount of time resolving memory references in the page table rather than TLB.
- **Increasing the Size of the TLB reach by**
 - **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 to have the opportunity to use them without an increase in fragmentation.

Other Issues --Program structure

- Page size = 1024 words
- `int A[][] = new int[1024][1024];`
- Each row is stored in one page
- The OS allocates fewer than 1024 frames to the entire program

■ Program 1:

```
for (j = 0; j < 1024; j++)  
    for (i = 0; i < 1024; i++)  
        A[i,j] = 0;
```

1024 × 1024 page faults

■ Program 2:

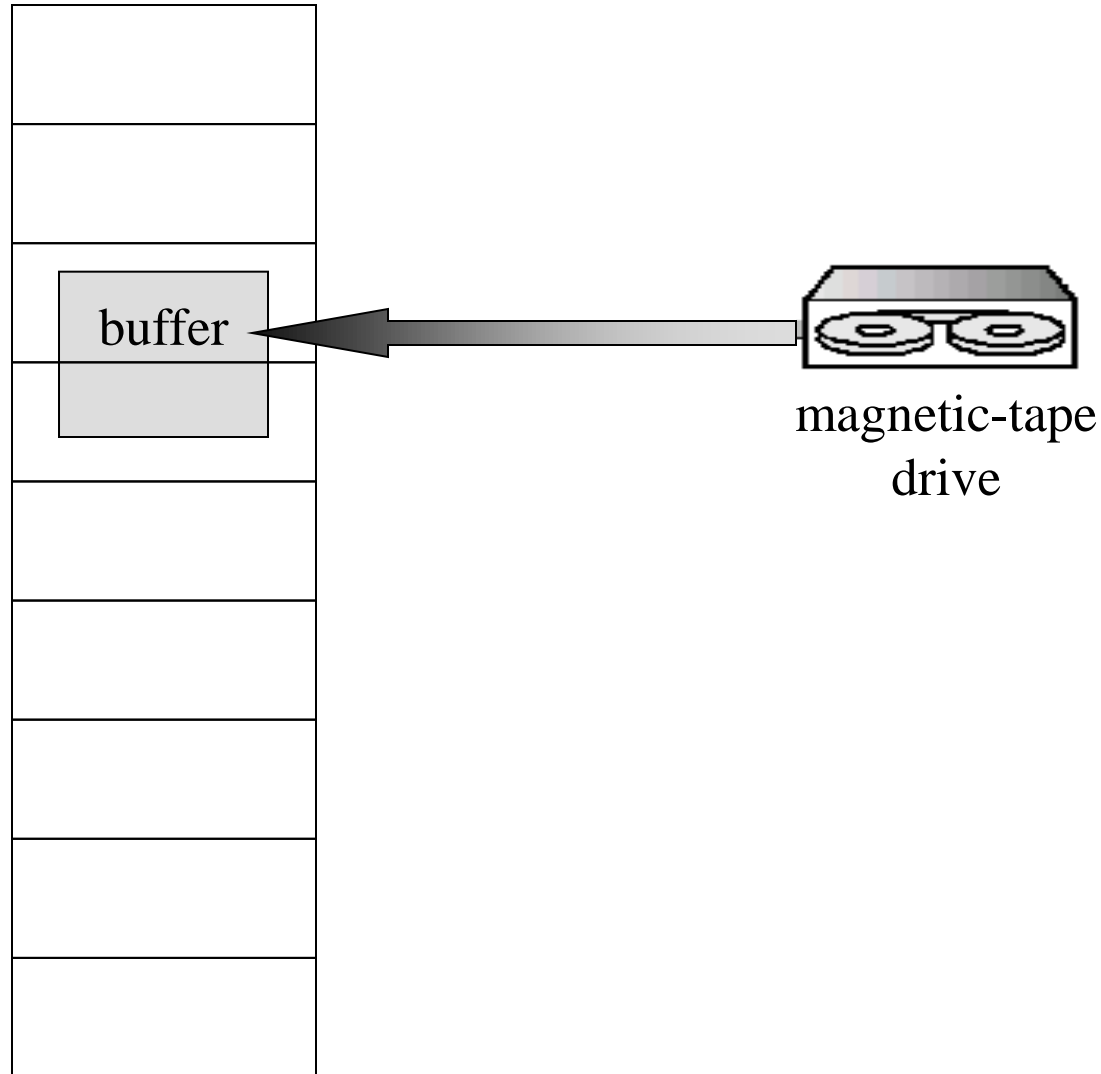
```
for (i = 0; i < 1024; i++)  
    for (j = 0; j < 1024; j++)  
        A[i,j] = 0;
```

1024 page faults

Other Issues --I/O Interlock

- Pages must sometimes be locked into memory.
- Consider I/O.
- Never to execute I/O to user memory.
 - I/O takes place only between system memory and I/O device.
 - Data are copied between system memory and user memory.
- Allow pages to be locked into memory.
 - A lock bit is associated with every frames.
 - Locked page cannot be replaced, when I/O complete, unlocked.
- 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



9.10 Operating System Examples

- **Windows XP**
- **Solaris**

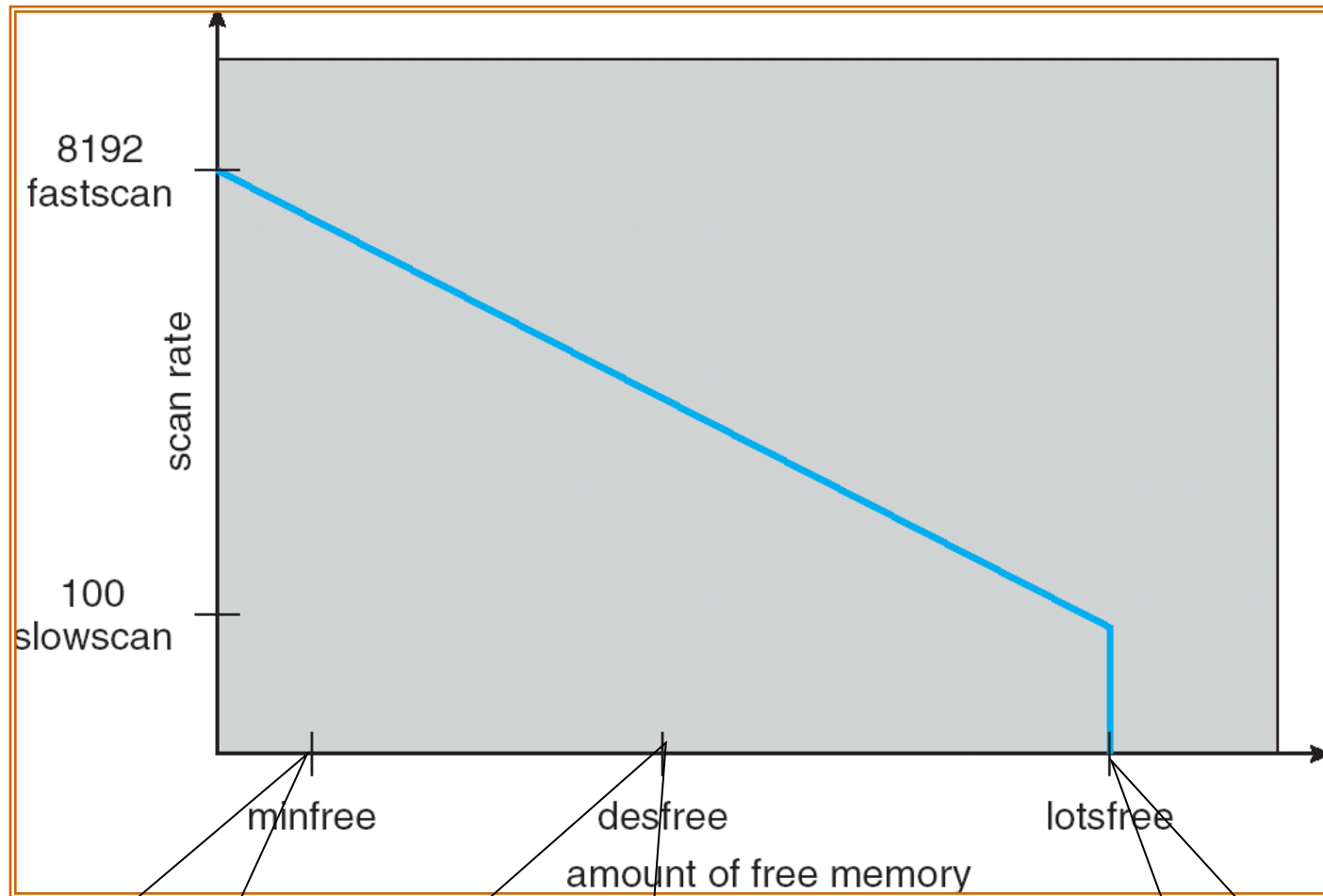
Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- When created, 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
- Thresholds:
 - *Lotsfree* – threshold parameter (amount of free memory) to begin paging
 - *Desfree* – threshold parameter to increasing paging
 - *Minfree* – threshold parameter to being 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

Solaris page scanner



Pageout is called for every request for a new page

Pageout check memory 100 times/s
If can't keep the amount at desfree for 30s, begins swapping

Pageout check memory 4 times/s

Homework (page 366)

9.4 9.5 9.11 9.21(experiment 3)

- Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.
- Consider the following page reference string:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
How many page faults would occur for the following replacement algorithms, assuming three, or four frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.
 1. LRU replacement
 2. FIFO replacement
 3. Optimal replacement

Homework (page 366)

	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
-	1	1	1																	
-	-	2	2																	
-	-	-	3																	
	F	F	F																	



Chapter 10 File-System Interface



LI Wensheng, SCS, BUPT

Teaching hours: 3h

Strong points:

File Concept

Access Methods

Directory Structure

Chapter 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**

Contents



Overview of file management

-
-
-
- 10.1 File Concept**
- 10.2 Access Methods**
- 10.3 Directory Structure**
- 10.4 File System Mounting**
- 10.5 File Sharing**
- 10.6 Protection**

10.1 File Concept

- **Contiguous logical address space.**
- **A uniform logical view of information storage.**
- **A file is a named collection of related information that is recorded on secondary storage.**
- **Commonly, files represent programs and data.**
 - **Data files**
 - **numeric**
 - **character**
 - **binary**
 - **Program**
 - **source**
 - **object**
- **files may be free form or may be formatted rigidly.**
- **The information in a file is defined by its creator.**

File Attributes

- **Name** – only information kept in human-readable form.
- **Identifier** – non-human-readable name for the file, a unique number that identifies the file within the system.
- **Type** – needed for systems that support different types.
- **Location** – a pointer to the location of the file on device.
- **Size** – current size of the file.
- **Protection** – access-control information, 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.

File Operations (1/2)

File is an abstract data type

- **Create** – find space, make entry in directory for the file.
- **Write** – file name, information to be written. *write* ptr.
- **Read** – file name, position in memory. *read* pointer.
- **Reposition** within file – file seek, not need I/O.
- **Delete** – name, release file space, erase directory entry.
- **Truncate** – reset length to 0, release file space.
- These primitive operations may be combined to perform other file operations.

File Operations (2/2)

- **Open-file table**
- **Open(F_i)** – search the directory structure on disk for entry F_i , and move the content of entry to open-file table. Return a pointer to the entry in the open-file table
- **Close (F_i)** – move the content of entry F_i in open-file table to directory structure on disk.
- Several users may open the file at the same time. Two levels of internal tables:
 - **Process open-file table**, a per-process table, tracks all files a process has open.
 - **System open-file table**, a system-wide table, contains process-independent information, such as file location on disk, access dates, file size.
- information associated with an open file: **file pointer, file open count, disk location of the file, access rights.**

Information associated with an Open File

- Several pieces of data are needed to manage 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
- Mediates(调停) access to a file
- Shared lock and exclusive lock (similar to reader lock, writer lock)
- Mandatory(强制的) or advisory(劝告的):
 - **Mandatory** – access is denied depending on locks held and requested
 - **Advisory** – processes can find status of locks and decide what to do

File types

file type	usual extension	function
executable	exe, com, bin or none	read to run machine- language program
object	obj, o	compiled, machine language, not linked
Source code	c, cpp, java, psa, 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, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
Print or view	arc, zip, tar mpeg, mov, rm	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files frouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	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



Access right?

Internal file structure

- **Logical structure**
 - **Two types**
 - **Text file:** a sequence of 8-bit bytes
 - **Record file:** a sequence of records with fixed or variable length
 - **Logical address**
 - Offset from the beginning of the file
 - Logic record no.
- **Physical structure:** A set of disk blocks
- **Basic I/O functions operate in terms of blocks**
- **Mapping logical address to physical address**
 - Logical block no.
 - Physical disk block no.
 - Offset in block
- **Internal fragmentation**

10.2 Access Methods

- **Reflect different file structures**
- **Different ways to store and process data**
- **Criteria for File Organization**
 - **Rapid access**
 - Needed when accessing a single record
 - Not needed for batch mode
 - **Ease of update**
 - File on CD-ROM will not be updated, so this is not a concern
 - **Economy of storage**
 - Should be minimum redundancy in the data
 - Redundancy can be used to speed access such as an index
 - **Simple maintenance**
 - **Reliability**

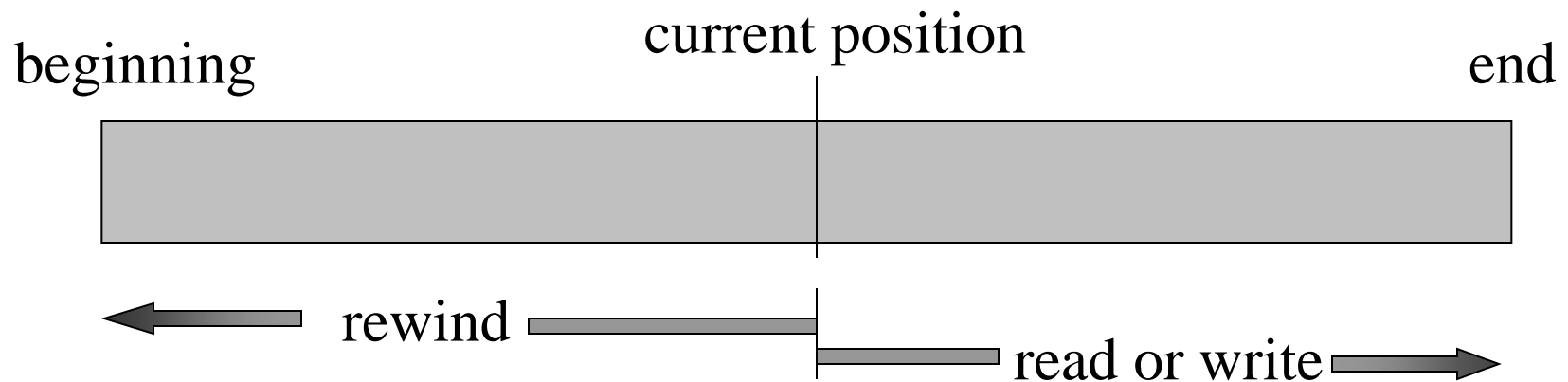
File organization?

Access Methods (1/3)

■ Sequential Access

- read next
- write next
- reset
- rewrite, no read after last write

■ sequential-access file



Access Methods (2/3)

■ Direct Access

- read n
- write n
- position to n
 - read next
 - write next
- rewrite n

n = relative block number

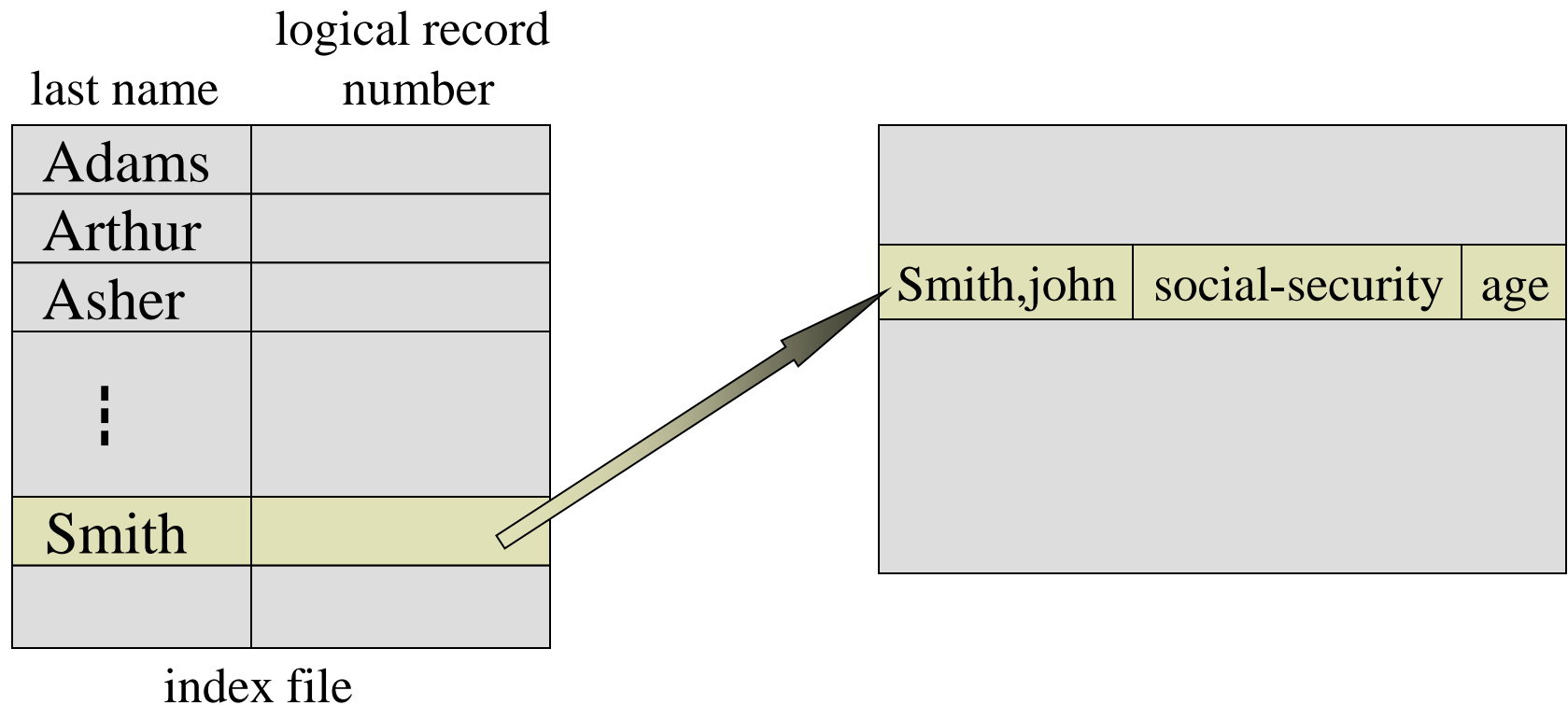
variable cp defines
current position

■ simulation of sequential access of a direct-access file.

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

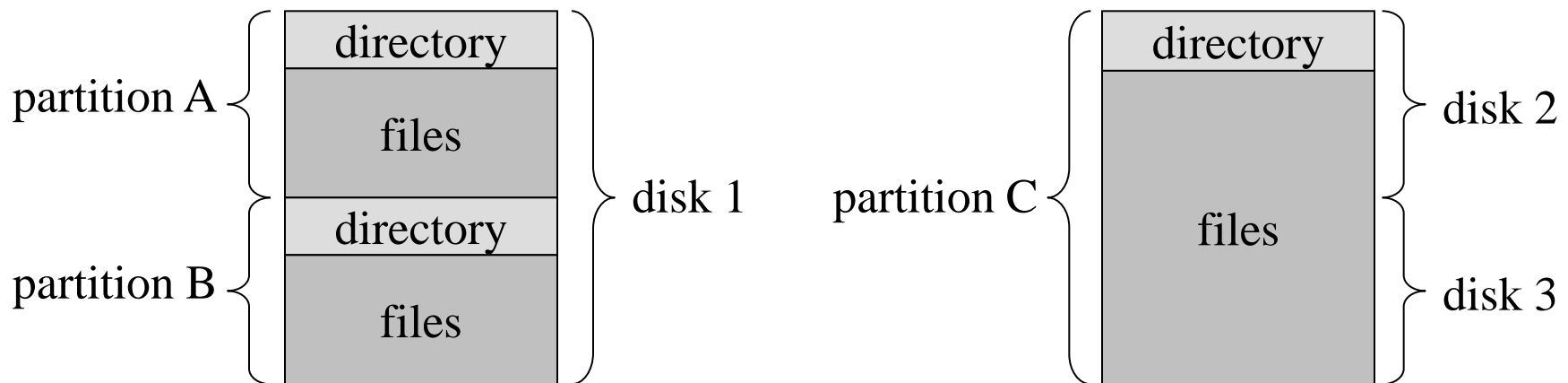
Other Access Methods (3/3)

- Can be built on top of a direct-access method.
- Usually involve the construction of an index for the file.
- Two- or multiple-level index files.
- Example of index and relative files.



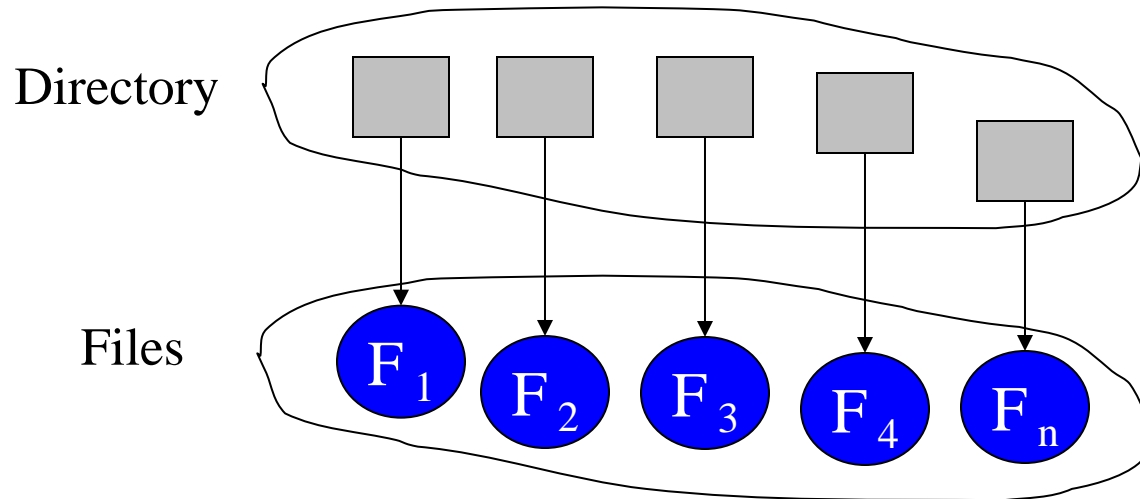
10.3 Directory Structure

- **Disks are split into one or more partitions.**
 - Each disk on a system contains at least one partition.
 - Some systems allow partitions to be larger than a disk to group disks into one logical structure.



- **Each partition contains information about files within it**
 - Information such as name, location, size, and type is kept in entries in a directory

Directory Structure (Cont.)



- A collection of nodes containing information about all files: Attributes, Location, Ownership
- Directory itself is a file owned by the operating system
- Provides mapping between file names and the files themselves
- Both the directory structure and the files reside on disk.

Supplement:

Information in a Directory Entry

- **Name**
- **Type**
- **Address**
- **Current length**
- **Maximum length**
- **Date last accessed (for archival)**
- **Date last updated (for dump)**
- **Owner ID (who pays)**
- **Protection information**

Operations Performed on a Directory

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

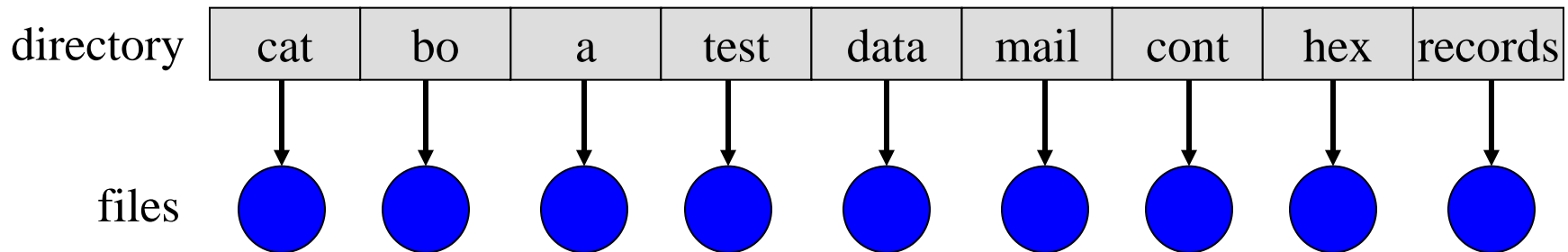
Supplement:

objects of organizing Directory (Logically)

- **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

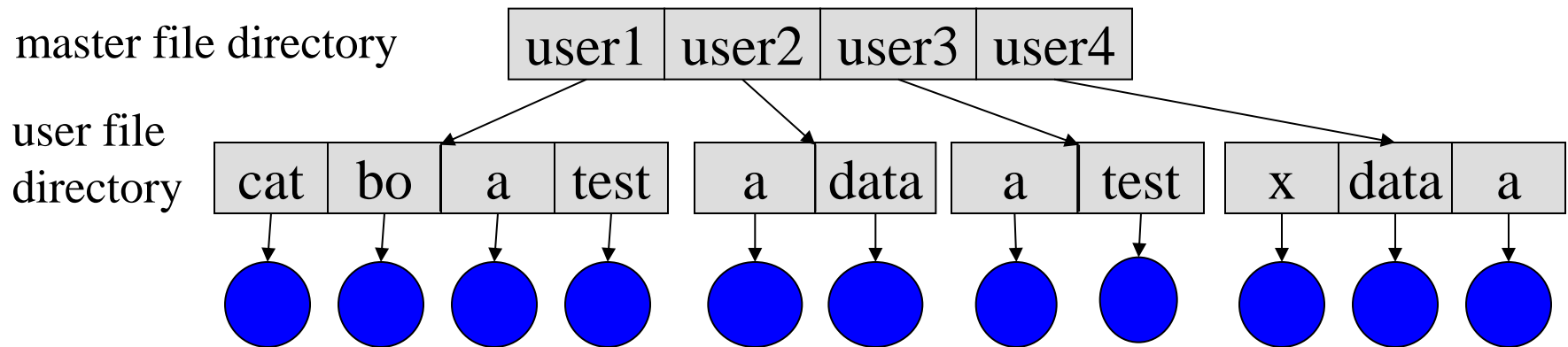
- All files are contained in the same directory.
- List of entries, one for each file
- Sequential file with the name of the file serving as the key



- Naming problem
 - Files must have unique names.
 - The length of a file name.
- Grouping problem

Two-Level Directory

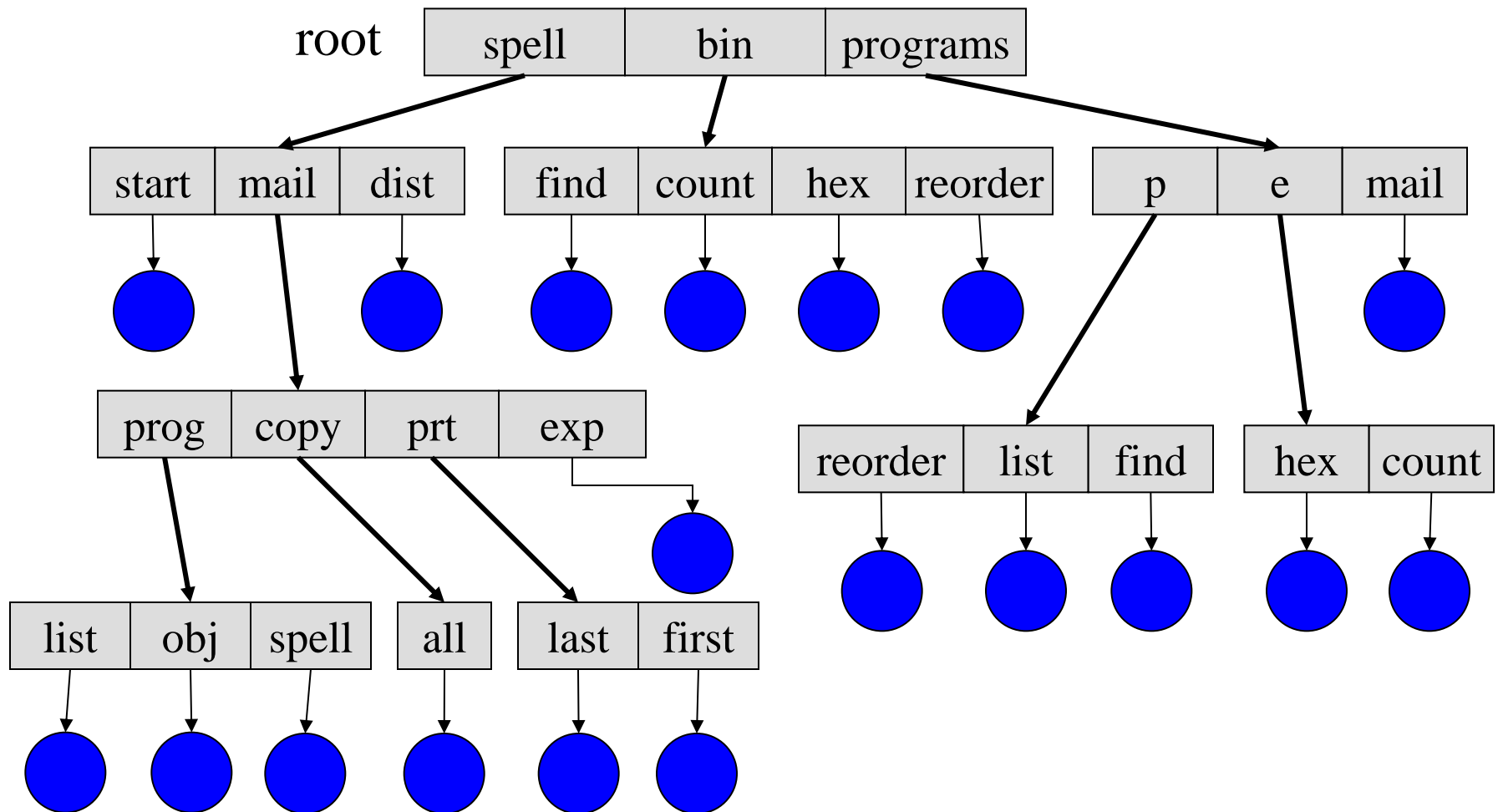
- A master directory and one directory for each user, master directory contains entry for each user, provides address and access control information
- Each user directory is a simple list of files for that user
- Create a separate directory for each user.



- Different users may have files with the same name
- Efficient searching
- Path name
- No grouping capability

Tree-Structured Directories (1/3)

- Master directory with user directories underneath it
- Each user directory may have subdirectories and files as entries



Tree-Structured Directories (2/3)

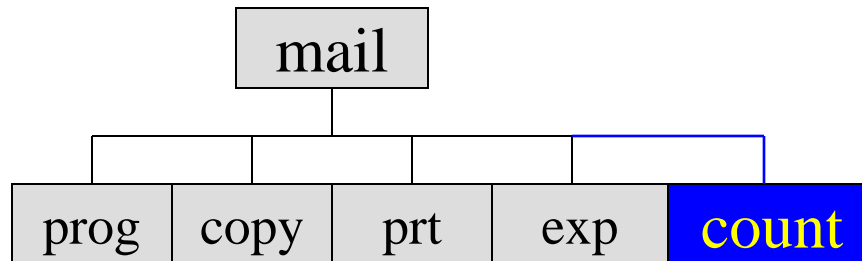
- **Efficient searching**
- **convenient Naming**
- **Grouping Capability**
- **Current directory** (working directory)
 - `cd /spell/mail/prog`
 - `type list`
- **Absolute** or **relative** path name
- **Creating a new file is done in current directory.**
- **Delete** a file
 - `rm <file-name>`

Tree-Structured Directories (3/3)

- **Creating a new subdirectory** is done in current directory.

mkdir <dir-name>

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



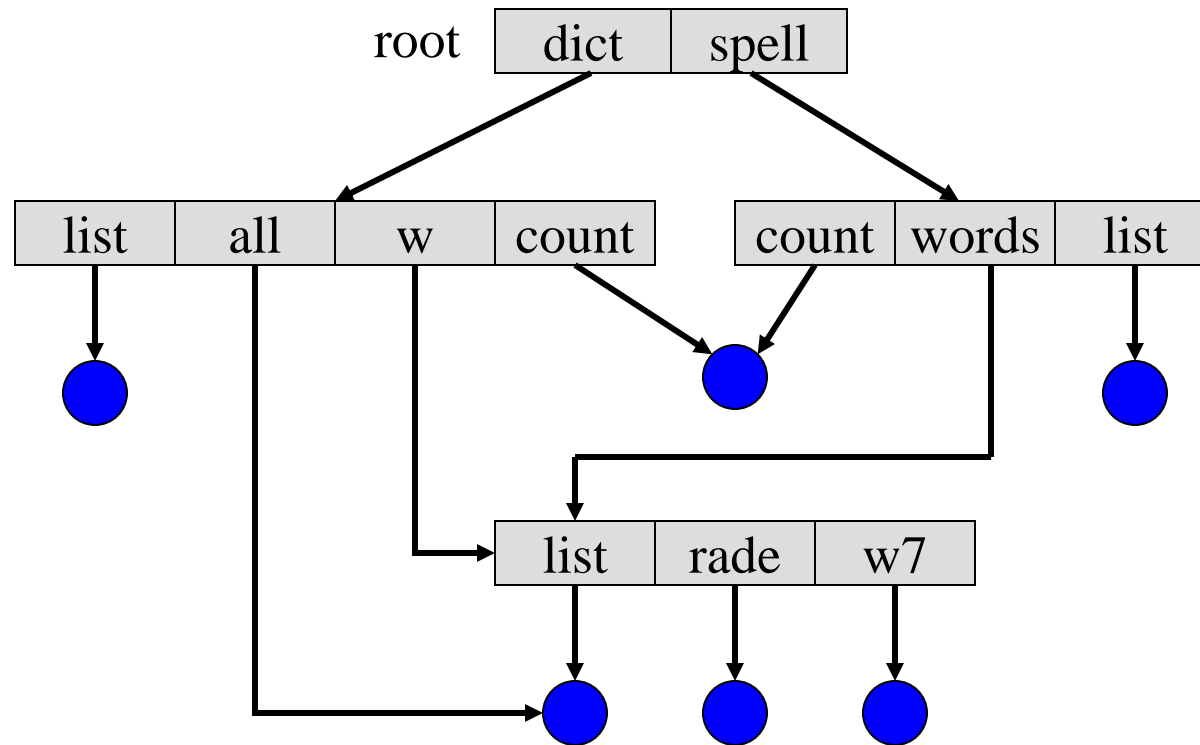
- **mkdir count**

- **Deleting a subdirectory**

- deleting a empty directory.
- deleting all files and subdirectories that it contains.

**How to
share files?**

- **Have shared subdirectories and files.**



Acyclic-Graph Directories (2/2)

■ Two different names (aliasing)

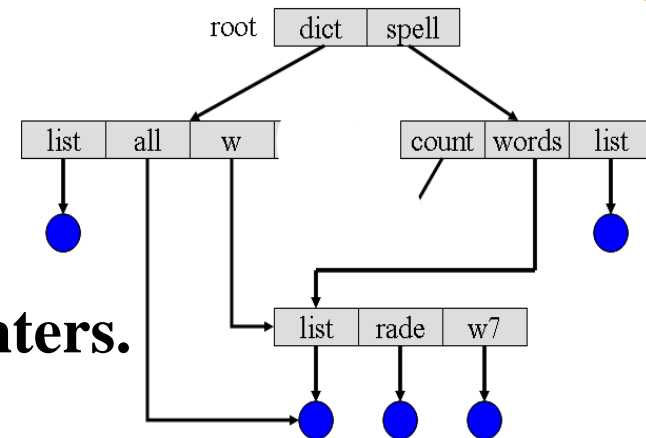
- **link**, a pointer to another file or directory. (e.g. path name)
 - **Resolve the link** – follow pointer to locate the file
- **duplicate all information** about shared files in all sharing directories.
 - Problem: the original and the copy indistinguishable.

■ If *dict* deletes *count*

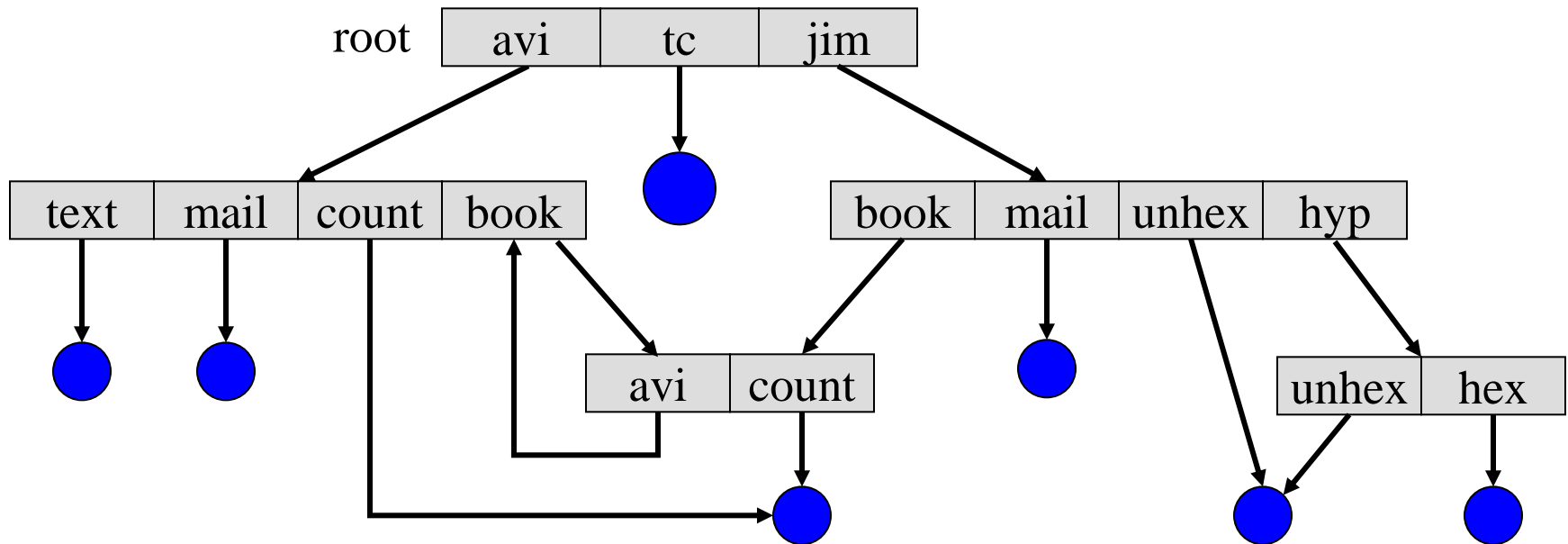
- dangling(悬空) pointer

Solutions:

- Backpointers, so we can delete all pointers.
- Variable size records a problem.
- File-reference list
- File-reference count



General Graph Directory



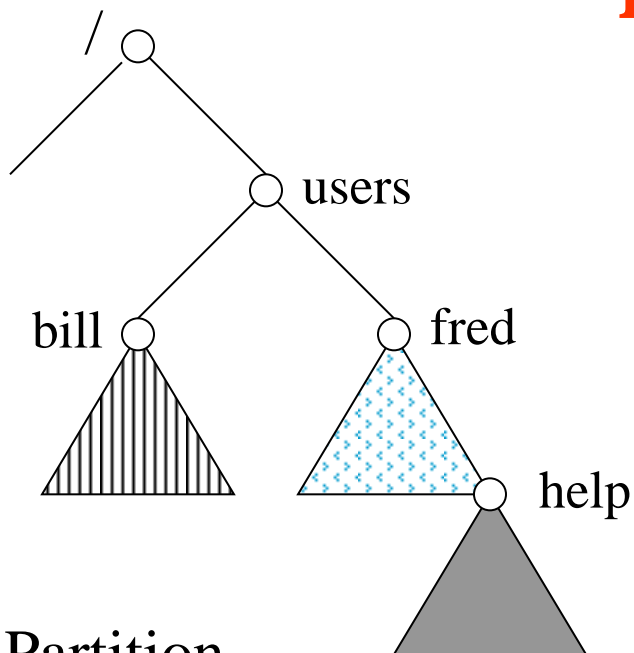
- **How do we guarantee no cycles?**
 - Allow only links to file not subdirectories.
 - Garbage collection.
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK.

10.4 File System Mounting

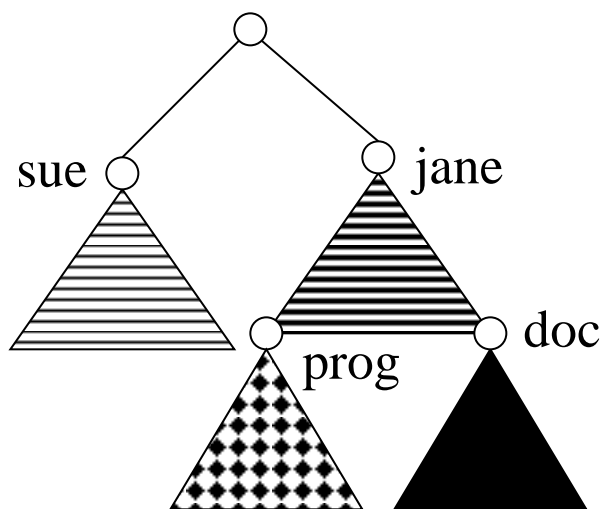
- A file system must be **mounted** before it can be accessed.
- The mount procedure
 - The operating system is given the name of the device, and the location within the file structure at which to attach the file system.
 - The operating system verifies that the device contains a valid file system by asking the device driver to read the device directory and verifying that the directory has the expected format.
 - The operating system notes in its directory structure that a file system is mounted at the specified **mount point**.

Example

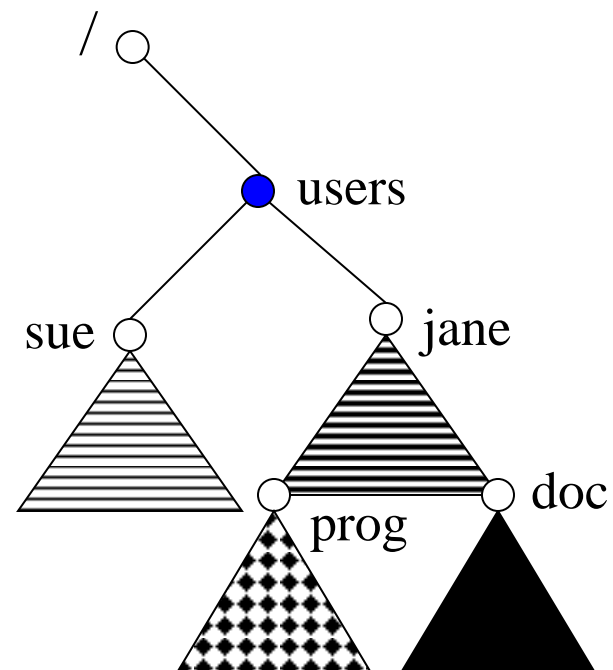
Existing



Unmounted Partition



Mount point: users



10.5 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.
 - ftp
 - DFS: distributed file system
 - WWW: World Wide Web
- Network File System (NFS) is a common distributed file-sharing method.

File Sharing – Multiple Users

- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights

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

- 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 include all information in each request, allowing easy recovery but less security

File Sharing – Consistency Semantics

- **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**

10.6 Protection

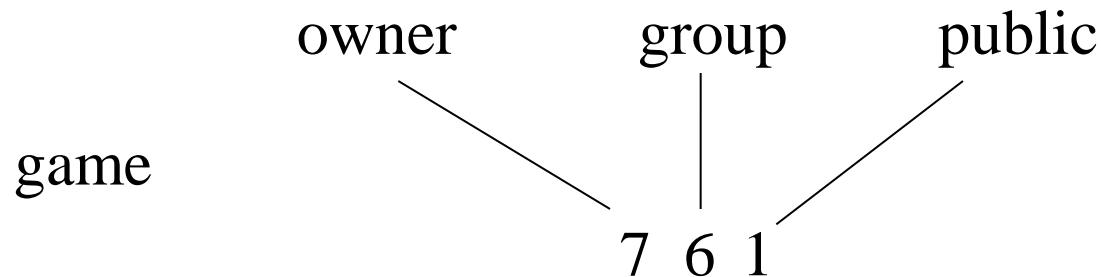
- **File owner/creator should be able to control:**
 - what can be done
 - by whom
- **Types of access**
 - **Read:** read from the file
 - **Write:** write or rewrite the file
 - **Execute:** load the file into memory and execute it
 - **Append:** write new information at the end of the file
 - **Delete:** delete the file and free its space for possible reuse
 - **List:** list the name and attributes of the file
 - Other operations, such as **renaming**, **copying**, or **editing** the file, may also be controlled.

Access control (1/2)

- **Mode of access:** **read, write, execute**
- **Access-control list (ACL):** associates with each file and specifies the user name and the types of access allowed for each user.
- **Three classes of users**
 - **Owner:** the user who created the file.
 - **Group:** a set of users who are sharing the file and need similar access.
 - **Universe (Public):** all other users in the system.
- **Access-control information**
 - **User access:** **RWX (111 – 7)**
 - **Group access:** **RW – (110 – 6)**
 - **Public access:** **– – X (001 – 1)**

Access control (2/2)

- 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.

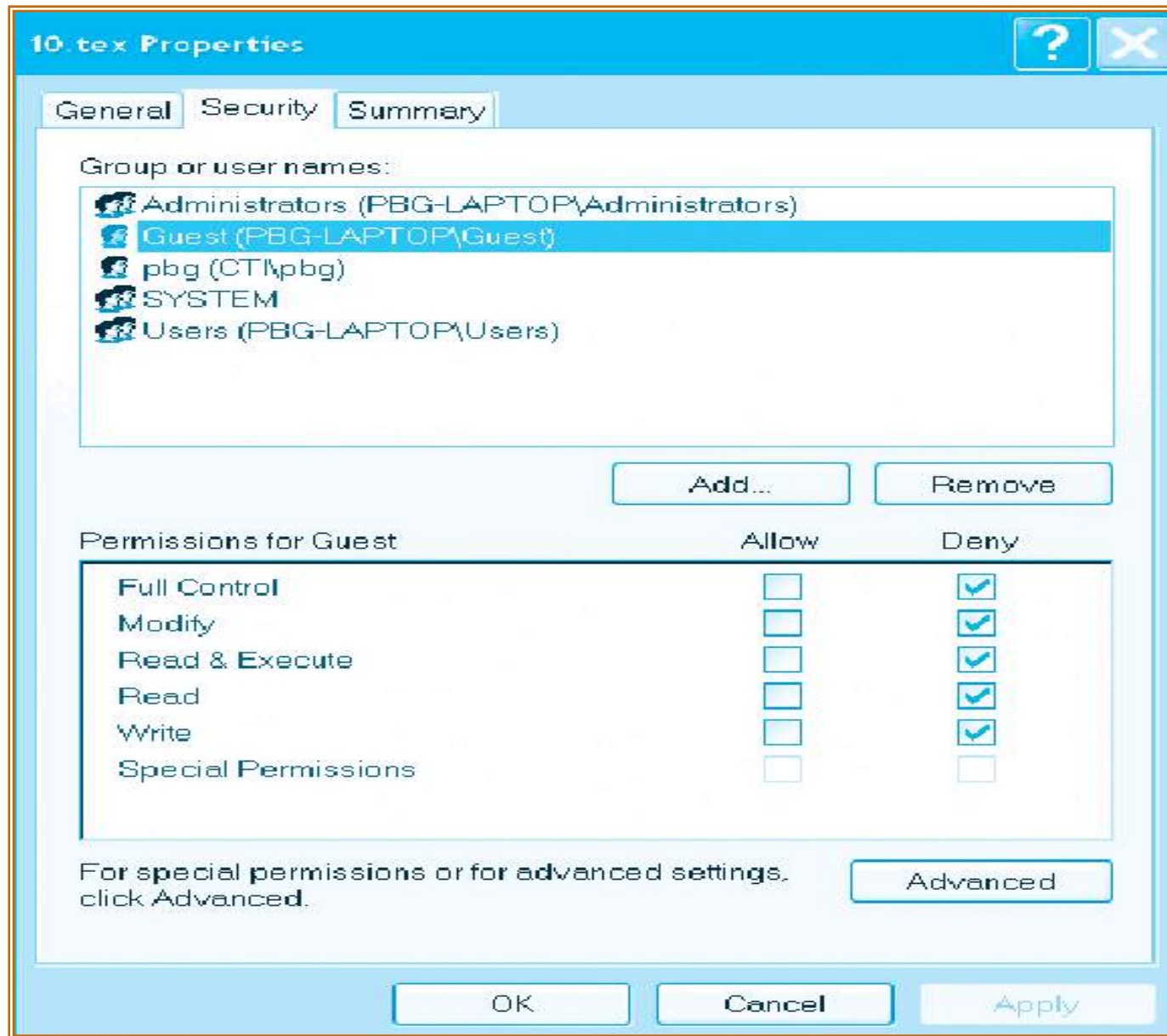


- Change user's access methods
`chmod go -RW file`
- Attach a group to a file
`chgrp G game.`

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/

Windows XP Access-control List Management



Homework (page 408)

10.1

10.2

10.9



Supplement:

Overview of File Management

■ File Management

- File management system is considered part of the operating system
- Input to applications is by means of a file
- Output is saved in a file for long-term storage

■ File Management System

- The way a user of application may access files
- Programmer does not need to develop file management software

Supplement:

Objectives for a File Management System

- Meet the data management needs and requirements of the user
- Guarantee that the data in the file are valid
- Optimize performance
- Provide I/O support for a variety of storage device types
- Minimize or eliminate the potential for lost or destroyed data
- Provide a standardized set of I/O interface routines
- Provide I/O support for multiple users

Supplement:

Minimal Set of Requirements

- Each user should be able to create, delete, read, and change files
- Each user may have controlled access to other users' files
- Each user may control what type of accesses are allowed to the users' files
- Each user should be able to restructure the user's files in a form appropriate to the problem
- Each user should be able to move data between files
- Each user should be able to back up and recover the user's files in case of damage
- Each user should be able to access the user's files by using symbolic names

Supplement:

File Management Functions

- Identify and locate a selected file
- Use a directory to describe the location of all files plus their attributes
- On a shared system describe user access control
- Blocking for access to files
- Allocate files to free blocks
- Manage free storage for available blocks

Access Rights (1/2)

■ None

- User may not know of the existence of the file
- User is not allowed to read the user directory that includes the file

■ Knowledge

- User can only determine that the file exists and who its owner is

■ Execution

- The user can load and execute a program but cannot copy it

■ Reading

- The user can read the file for any purpose, including copying and execution

■ Appending

- The user can add data to the file but cannot modify or delete any of the file's contents

Access Rights (2/2)

■ Updating

- The user can modify, delete, and add to the file's data. This includes creating the file, rewriting it, and removing all or part of the data

■ Changing protection

- User can change access rights granted to other users

■ Deletion

- User can delete the file

■ Owners

- Has all rights previously listed
- May grant rights to others using the following classes of users
 - Specific user
 - User groups
 - All for public files

Supplement:

File Organization (1/4)

■ The Pile (堆)

- Data are collected in the order they arrive
- Purpose is to accumulate a mass of data and save it
- Records may have different fields
- No structure
- Record access is by exhaustive (穷举) search

Supplement:

File Organization (2/4)

■ **The Sequential File**

- **Fixed format used for records**
- **Records are the same length**
- **All fields the same (order and length)**
- **Field names and lengths are attributes of the file**
- **One field is the key field**
 - **Uniquely identifies the record**
 - **Records are stored in key sequence**
- **New records are placed in a log file or transaction file**
- **Batch update is performed to merge the log file with the master file**

Supplement:

File Organization (3/4)

■ Indexed Sequential File

- Index provides a lookup capability to quickly reach the vicinity of the desired record
 - Contains key field and a pointer to the main file
 - Indexed is searched to find highest key value that is equal or less than the desired key value
 - Search continues in the main file at the location indicated by the pointer
- New records are added to an overflow (溢出) file
- Record in main file that precedes it is updated to contain a pointer to the new record
- The overflow is merged with the main file during a batch update
- Multiple indexes for the same key field can be set up to increase efficiency

Supplement:

File Organization (4/4)

■ Indexed File

- Uses multiple indexes for different key fields
- May contain an exhaustive index that contains one entry for every record in the main file
- May contain a partial index

■ The Direct, or Hashed File

- Directly access a block at a known address
- Key field required for each record

Chapter 11 File-System Implementation



LI Wensheng, SCS, BUPT

Teaching hours: 3h

Strong points:

File System Structure, File System Implementation

Directory Implementation

Allocation Methods, Free-Space Management

Chapter Objectives

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

Contents

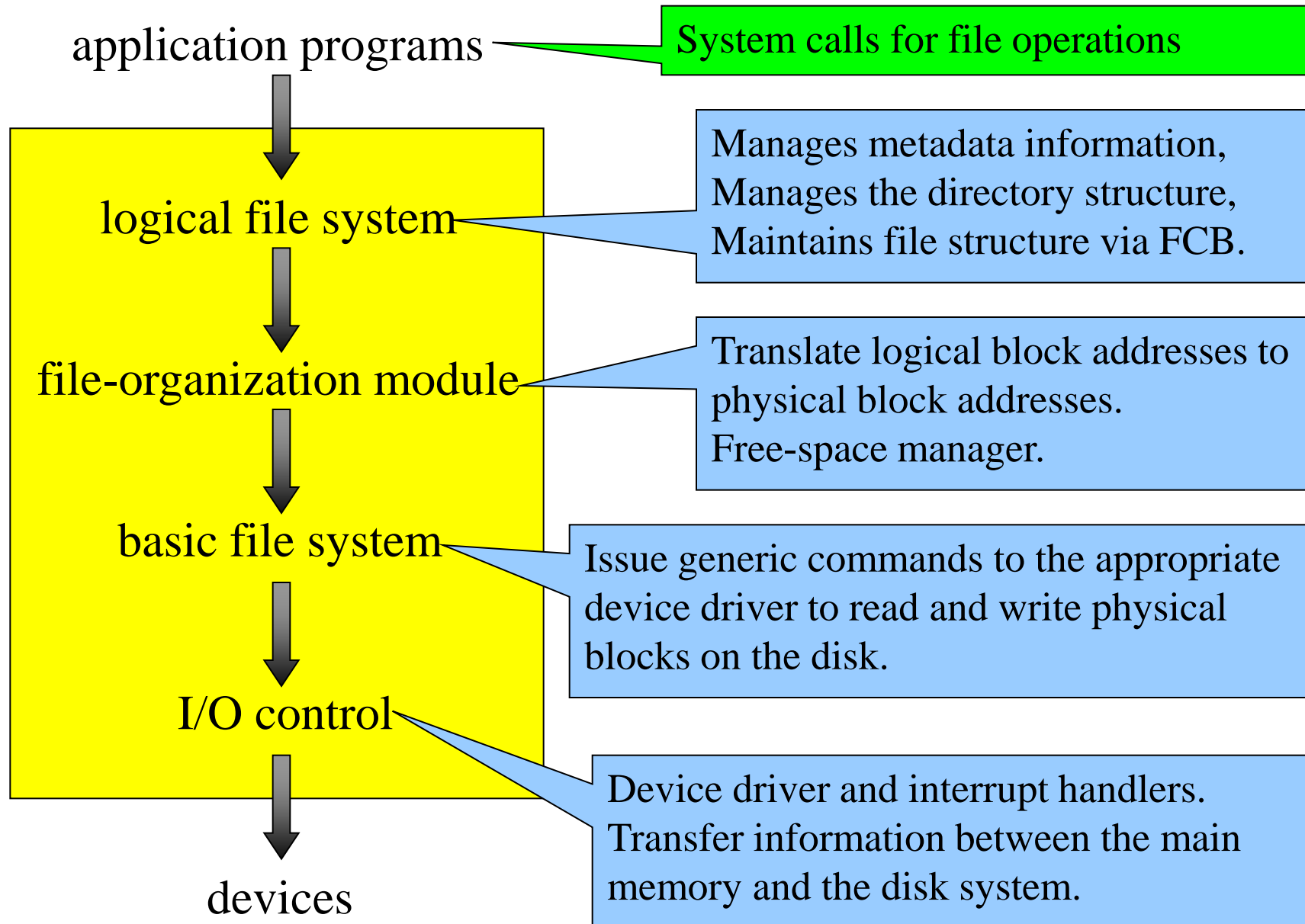
- 11.1 File System Structure**
- 11.2 File System Implementation**
- 11.3 Directory Implementation**
- 11.4 Allocation Methods**
- 11.5 Free-Space Management**

- 11.6 Efficiency and Performance**
- 11.7 Recovery**
- 11.8 Log-Structured File Systems**

11.1 File-System Structure

- **File structure**
 - Logical storage unit
 - Collection of related information
- **File system resides on secondary storage (disks)**
 - A disk can be written in place
 - A disk can access directly any given block of information
 - I/O transfers between memory and disk in units of blocks
- **Main design problems**
 - User Interface
 - Mapping the logical file system onto the physical secondary-storage device
- **File system organized into layers**

Layered File System



11.2 File System Implementation

- **On-disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.**
- **On-disk structures**
 - **A boot control block (per volume)**
 - In UFS, boot block, In NTFS, boot section
 - **A volume/partition control block (per volume)**
 - In UFS, superblock, In NTFS, Master File Table
 - **A directory structure (per file system)**
 - In UFS, file names and associated inode
 - In NTFS, in Master File Table
 - **A FCB (per file)**
 - In UFS, inode In NTFS, Master File Table (relational database)

A Typical File Control Block

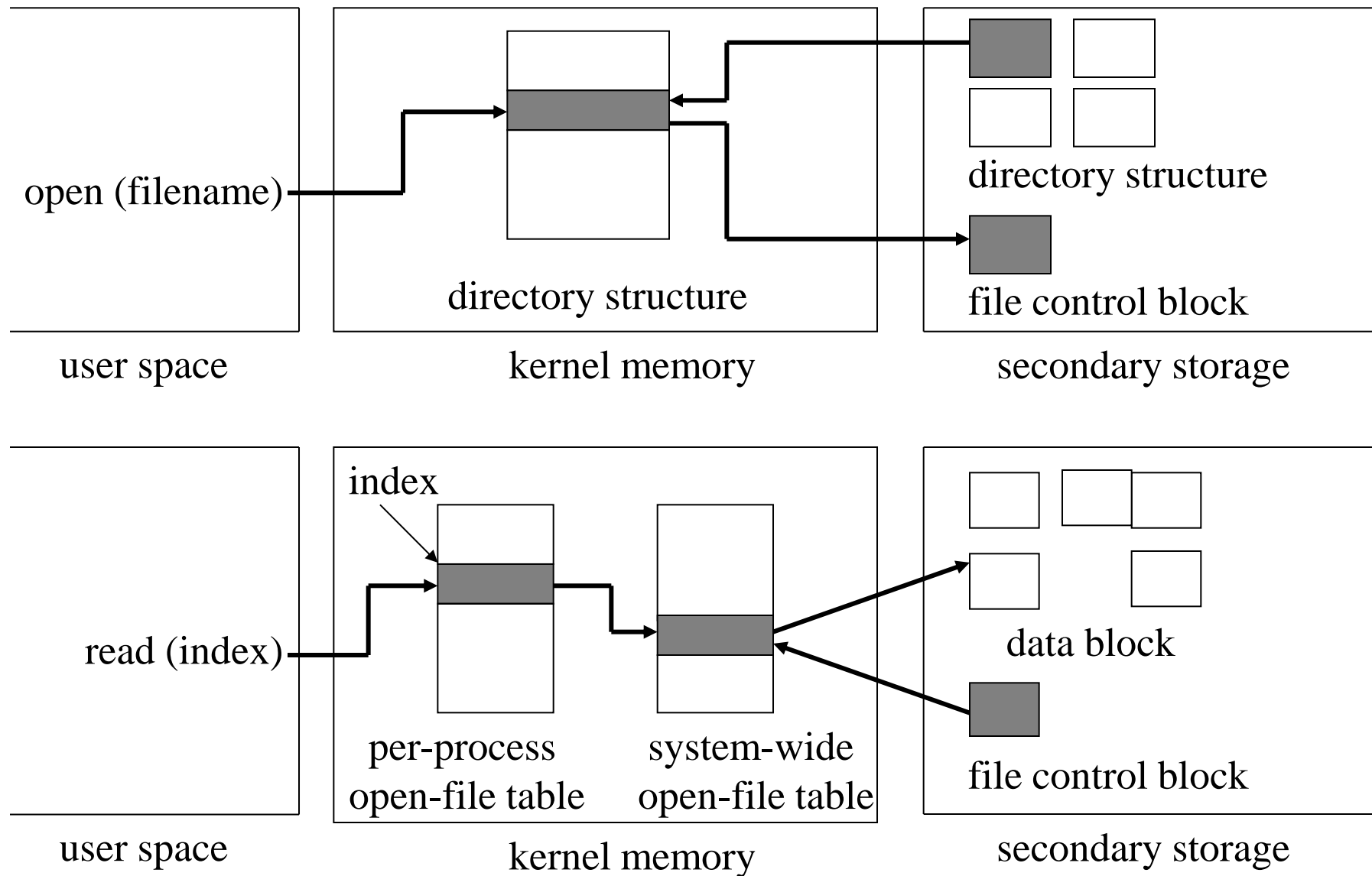
- *File control block* – storage structure consisting of information about a file, including ownership, permissions, and location of the file contents.

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

In-Memory File System Structures

- **An in-memory mount table (partition table)**
- **An in-memory directory structure cache**
- **The system-wide open-file table**
 - Contains a copy of the FCB of each open file
 - One entry for each open file
 - Only one table for the system
- **The per-process open-file table**
 - Contains a pointer to the appropriate entry in the system-wide open-file table
 - One entry for each open file
 - One table per process

In-Memory File System Structures



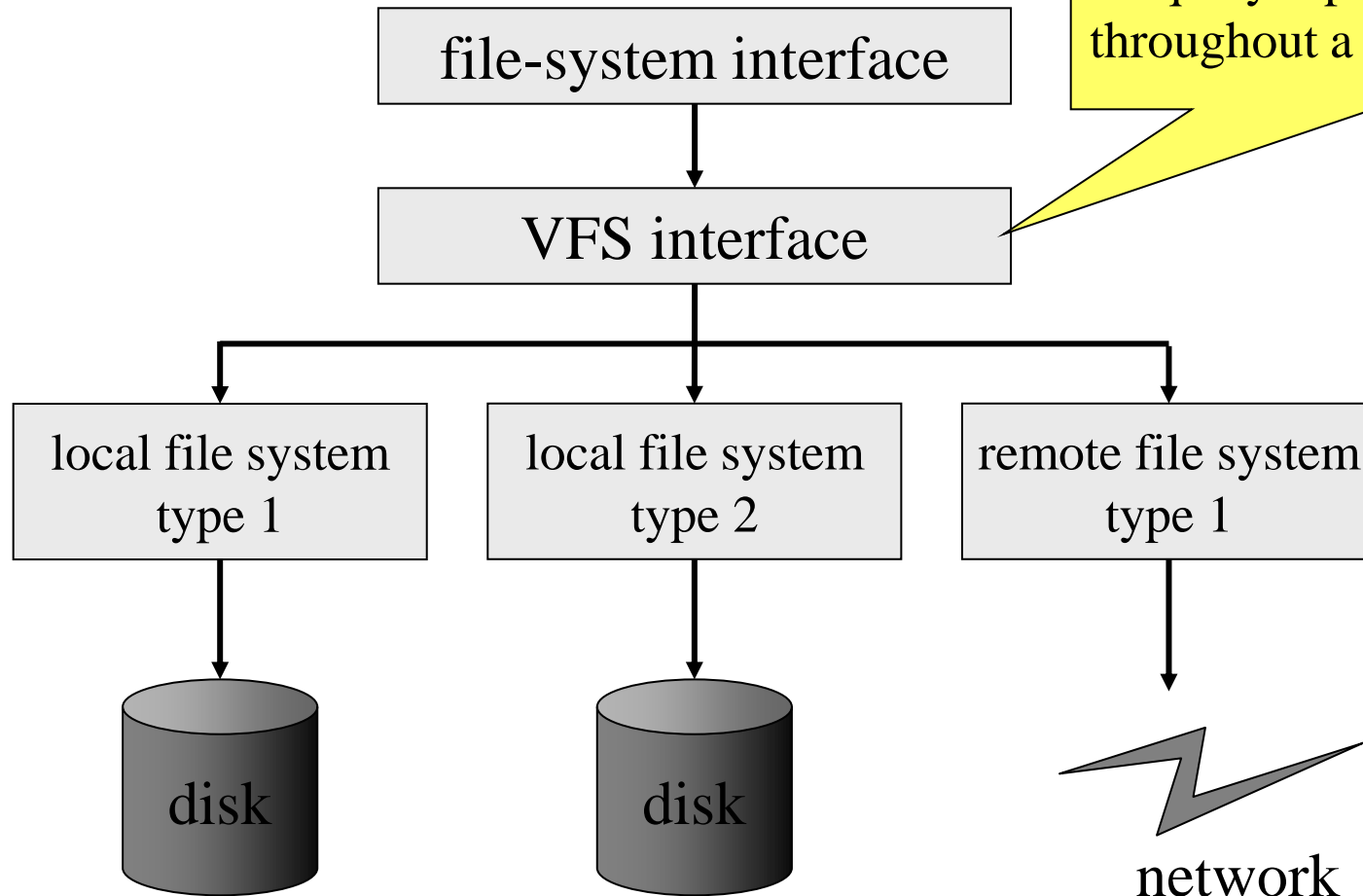
Partitions and Mounting

- A disk can be sliced into multiple partitions, or a partition can span multiple disks.
- Each partition can either be
 - “**raw**”: containing no file system
 - “**cooked**”: containing a file system.
- Boot information can be stored in a separate partition.
 - Boot information is loaded into memory as an image
- A boot loader understands multiple file systems and multiple operating systems can occupy the boot space.
- The root partition, which contains the operating-system kernel and potentially other system files, is mounted at boot time.
- **Mount table** – containing information about file systems that has been mounted.

Virtual File Systems

Separates file-system-generic operations from their implementation.

Provides a mechanism for uniquely representing a file throughout a network--vnode.



11.3 Directory Implementation

- **Linear list of file names with pointer to the data blocks.**
 - simple to program
 - time-consuming to execute
- **A sorted list allows a binary search and decreases the average search time.**
- **Hash Table – linear list with hash data structure.**
 - decreases directory search time
 - *collisions* – situations where two file names hash to the same location
 - **Difficulties: fixed size, hash function**

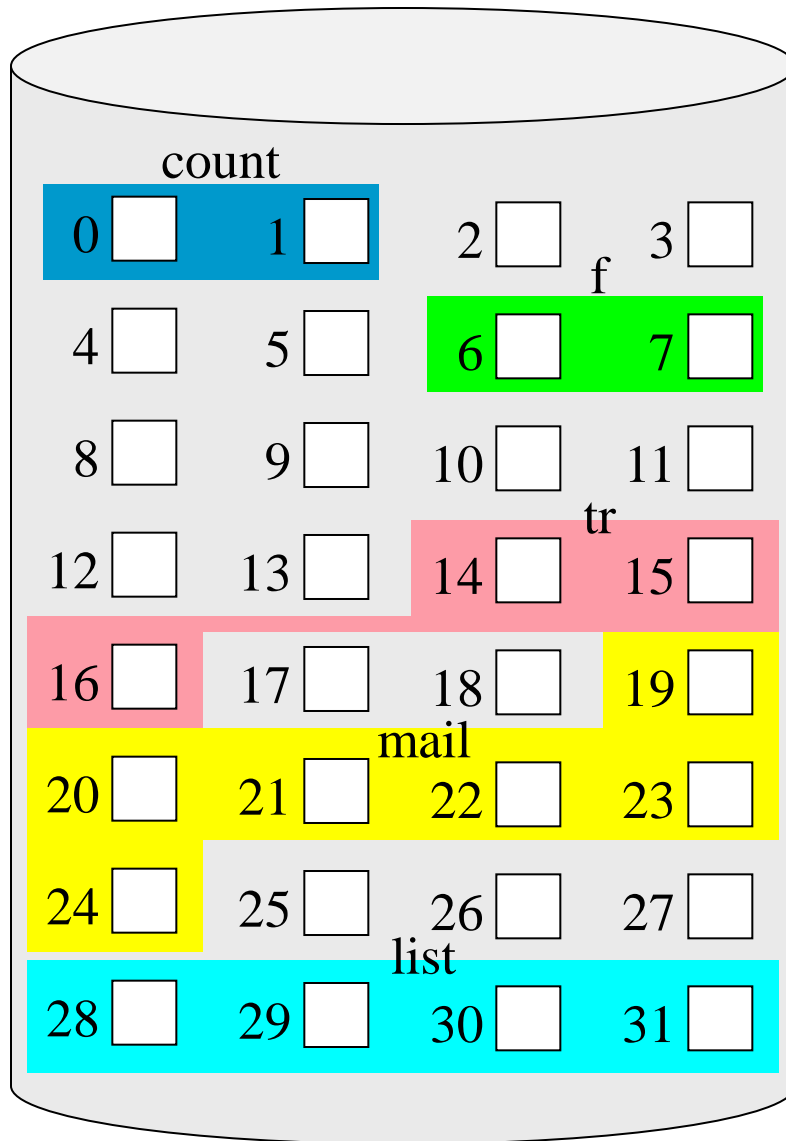
11.4 Allocation Methods

- **An allocation method refers to how disk blocks are allocated for files.**
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

Contiguous Allocation

- **Single set of blocks is allocated to a file at the time of creation**
 - Each file occupies a set of contiguous blocks on the disk.
- **Simple (see Fig. 11.5 on P421)**
 - Only a single entry in the file allocation table
 - only starting location (block #) and length (number of blocks) are required.
- **Both sequential and direct access can be supported.**
- **Problems**
 - Finding space for a new file(free-space-management).
 - Wasteful of space (dynamic storage-allocation problem).
 - External fragmentation.
 - How much space is needed for a file, Files cannot grow.

Contiguous Allocation of Disk Space



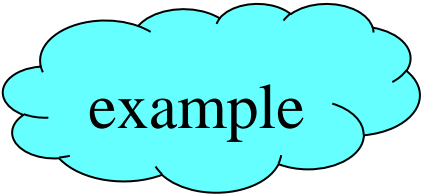
directory

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

Supplement:

Mapping from logical to physical



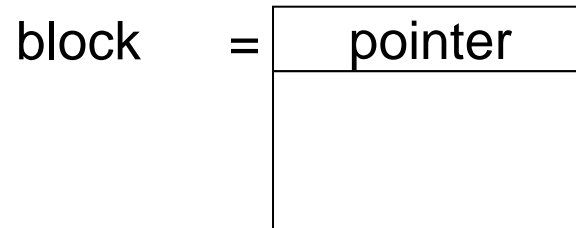
- Block to be accessed = $Q + \text{starting address}$
- Displacement into block = R
- Example
 - For text file . . . 
 - For record file
- Problems
 - Difficult to find space for a new file
 - External fragmentation
 - Determine how much space is needed for a file

Extent-Based Systems

- Many newer file systems (i.e. Veritas File System) use a modified contiguous allocation scheme.
- A contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, an **extent**, is added to the initial allocation.
- Extent-based file systems allocate disk blocks in **extents**.
- An **extent** is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents.

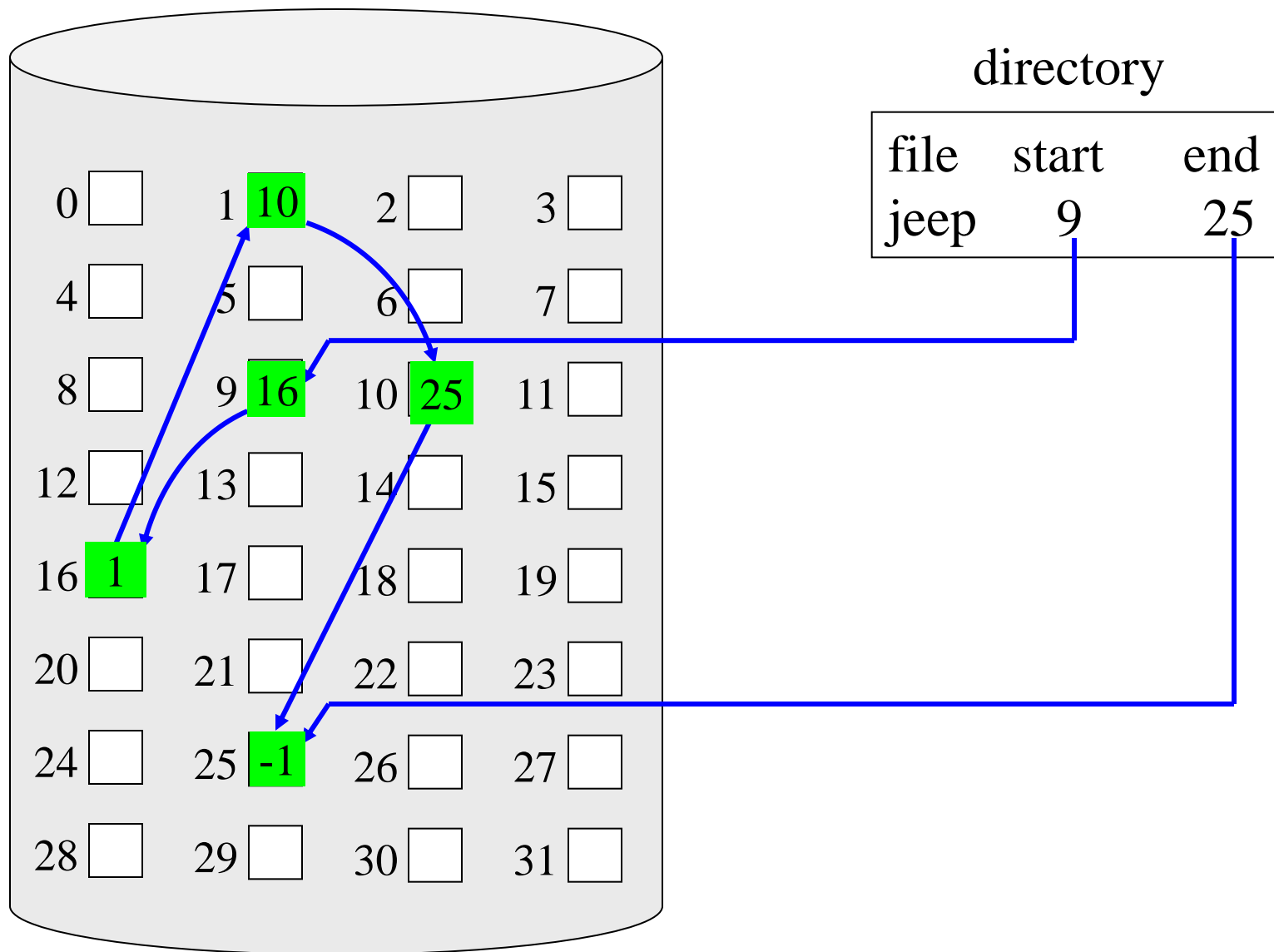
Linked Allocation

- Allocation on basis of individual block
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file.
- Each block contains a pointer to the next block in the chain



- There is no external fragmentation.
- Any free block on the free-space list can be used to satisfy a request.

Linked Allocation



Linked Allocation (Cont.)

■ Disadvantages

- No random access, Space for pointers, Reliability

■ Mapping from logical to physical

- Block to be accessed is the Qth block in the linked chain of blocks representing the file.
- Displacement into block = $R + \text{ptr_space}$

$$\text{LA} / (\text{block_size} - \text{ptr_space}) \begin{matrix} \swarrow Q \\ \searrow R \end{matrix}$$

■ File-allocation table (FAT)

- Simple but efficient method used by MS-DOS and OS/2
- Contained in a section at the beginning of each partition.
- One entry per block.
- Indexed by block number.
- Cached in memory

File-Allocation Table

directory entry

test	...	217
------	-----	-----

name

start block

217

339

618

no. of disk blocks

-1

FAT

0

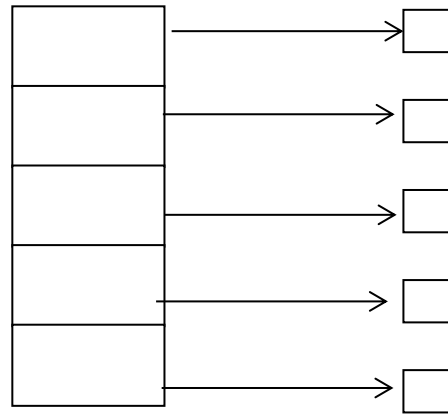
618

end of file

339

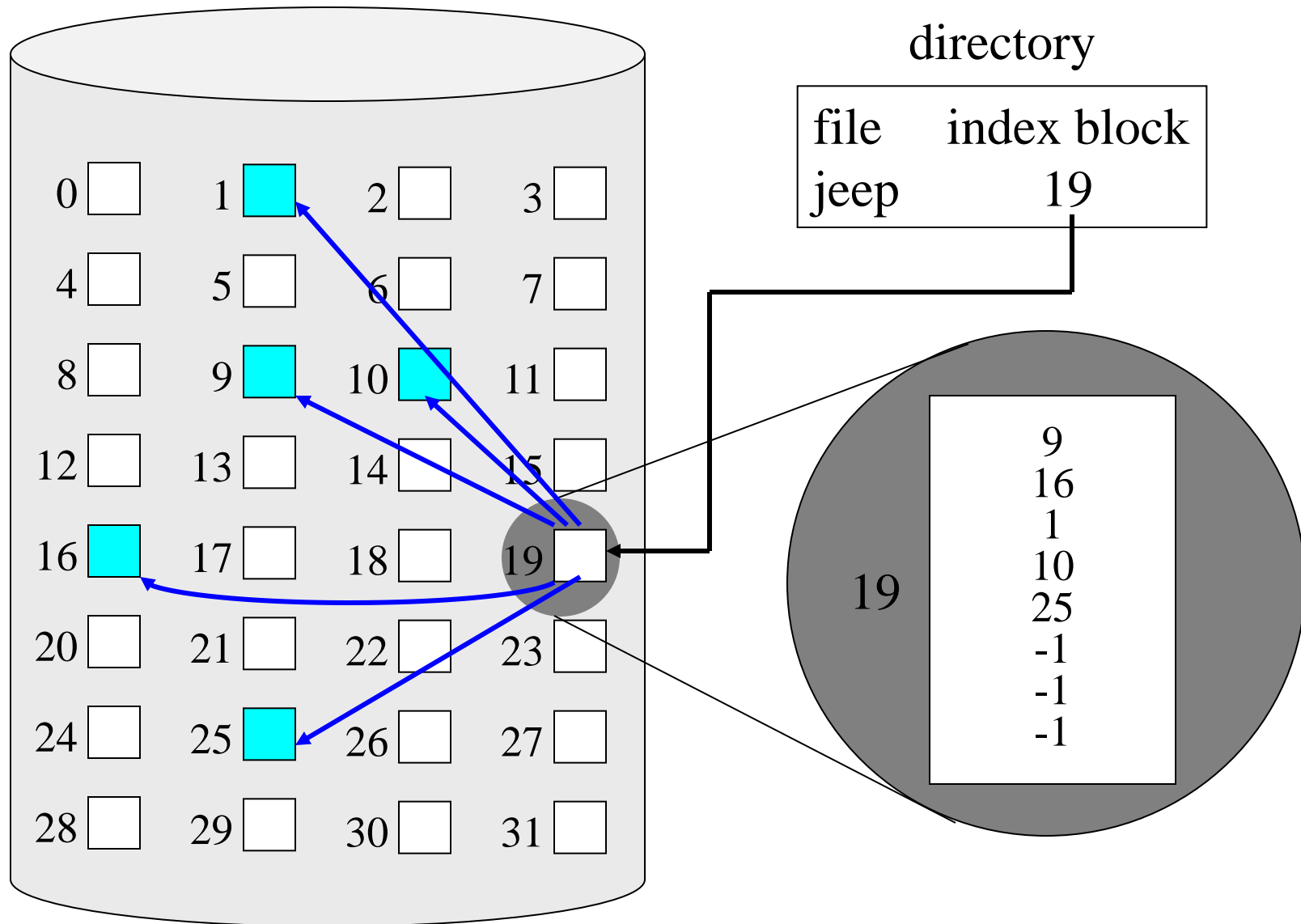
Indexed Allocation

- Brings all pointers together into the *index block*.
- Each file has its own index block, which is an array of disk-block addresses.
- The directory contains block number for the index block.
- Logical view.



index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

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

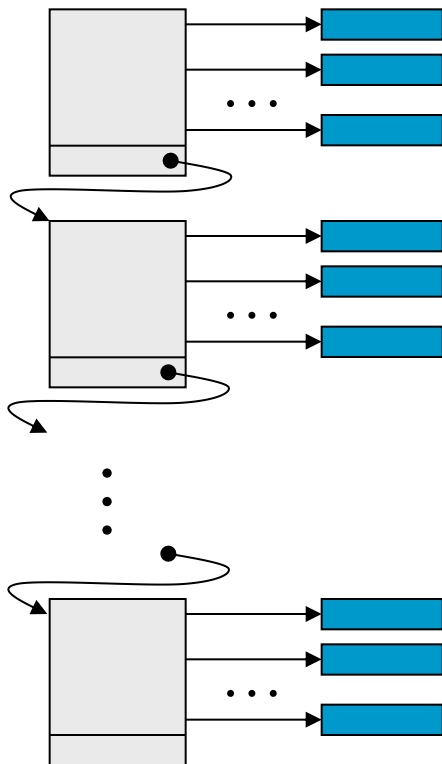
$$\text{LA} / 512 \begin{cases} Q \\ R \end{cases}$$

Q = displacement into index table

R = displacement into block

Indexed Allocation – Mapping (Cont.)

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



$$LA / (512 \times 511) \begin{cases} Q_1 \\ R_1 \end{cases}$$

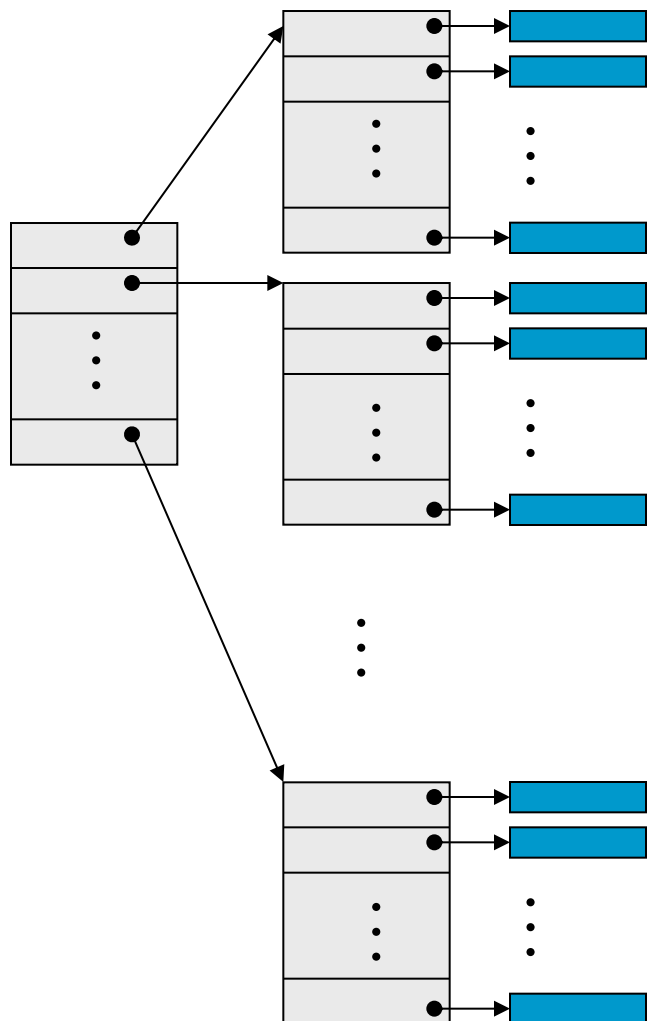
Q_1 = block of index table
 R_1 is used as follows:

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Q_2 = displacement into block of index table
 R_2 = displacement into block of file

Indexed Allocation – Mapping (Cont.)

- Two-level index (maximum file size is 512^3)



$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

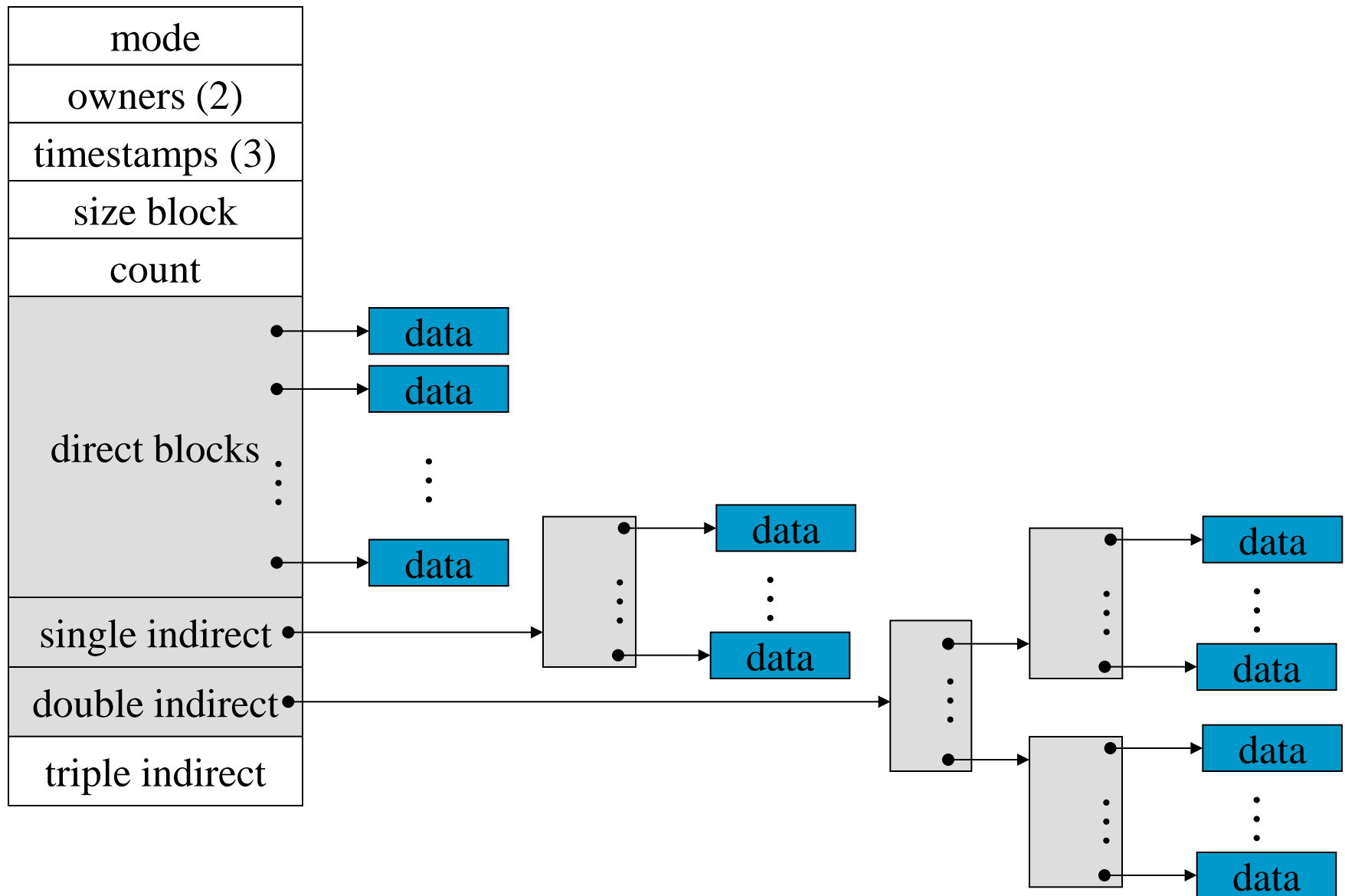
Q_1 = displacement into outer-index
 R_1 is used as follows:

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Q_2 = displacement into block of
 index table

R_2 = displacement into block of file

Combined Scheme: UNIX (4K bytes per block)



11.5 Free-Space Management

■ Bit map / Bit vector (n blocks)



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

word

bit

	31 30 2 1 0																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	0	1	1	1	0	0	1	0	1	1	0	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0
3																																

■ Block number calculation:

(number of bits per word) * (number of 0-value words) +
offset of first 1 bit

Free-Space Management (Cont.)

- Bit map requires extra space. Example:

block size = 2^{12} bytes

disk size = 2^{30} bytes (1 gigabyte)

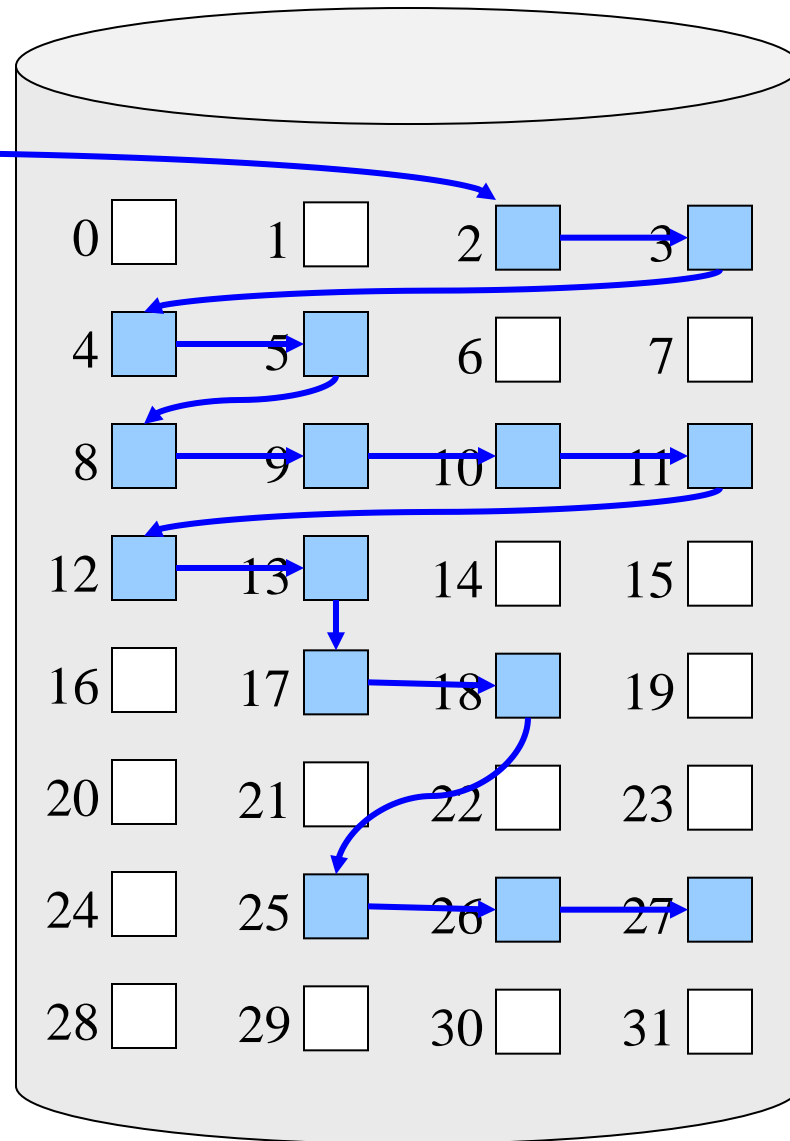
$n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

40G ---- 1MB

- Easy to get contiguous files space
- Linked list (free list)
 - Cannot get contiguous space easily
 - No waste of space
- Grouping
- Counting

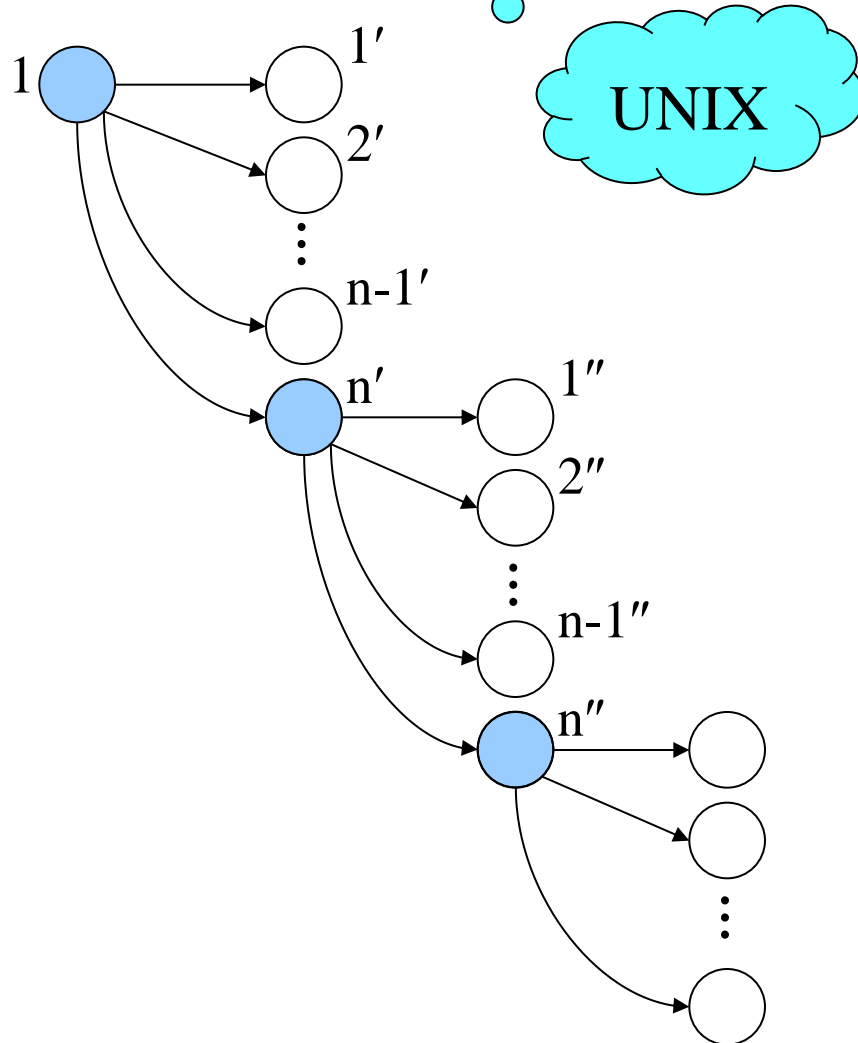
Linked Free Space List on Disk

free-space list head



Free-Space Management (Cont.)

■ Grouping



■ Counting

Free-space list

addr1	count1
addr2	count2
addr3	count3
...	...

Free-Space Management (Cont.)

■ Need to protect:

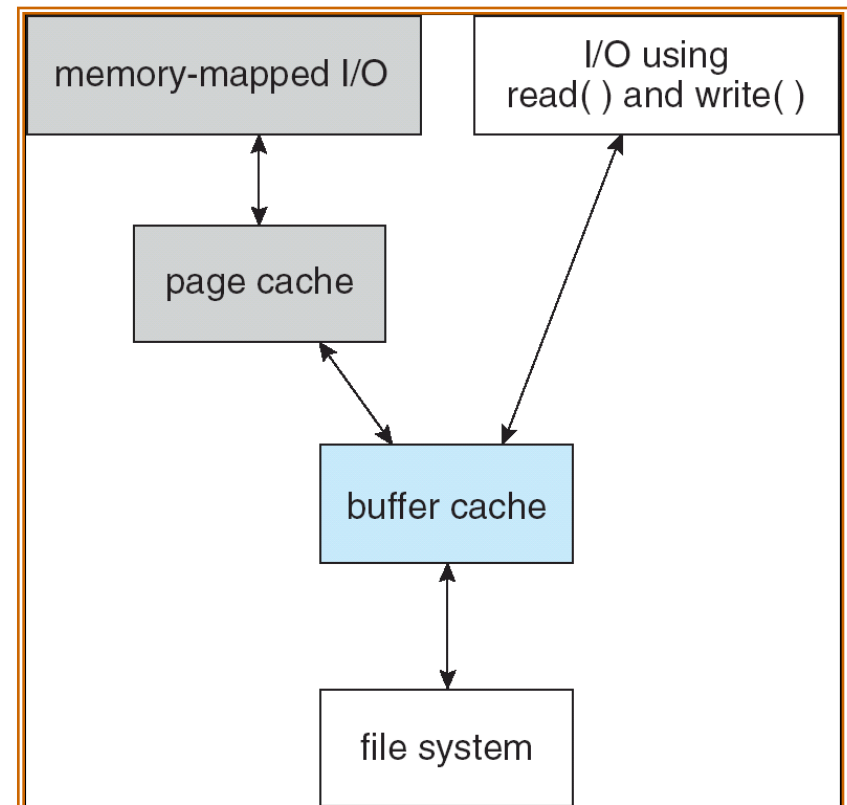
- **Pointer to free list**
- **Bit map**
 - **Must be kept on disk**
 - **Copy in memory and disk may differ.**
 - **Cannot allow for block[i] to have a situation where $\text{bit}[i] = 0$ in memory and $\text{bit}[i] = 1$ on disk.**
- **Solution:**
 - **Set $\text{bit}[i] = 0$ on disk.**
 - **Allocate block[i]**
 - **Set $\text{bit}[i] = 0$ in memory**

11.6* Efficiency and Performance

- **Efficiency dependent on:**
 - disk allocation and directory algorithms
 - types of data kept in file's directory entry
 - size of pointers used to access data
- **Performance**
 - **disk cache** – separate section of main memory for frequently used blocks
 - **free-behind and read-ahead** – techniques to optimize sequential access
 - improve PC performance by dedicating section of memory as **virtual disk**, or RAM disk.

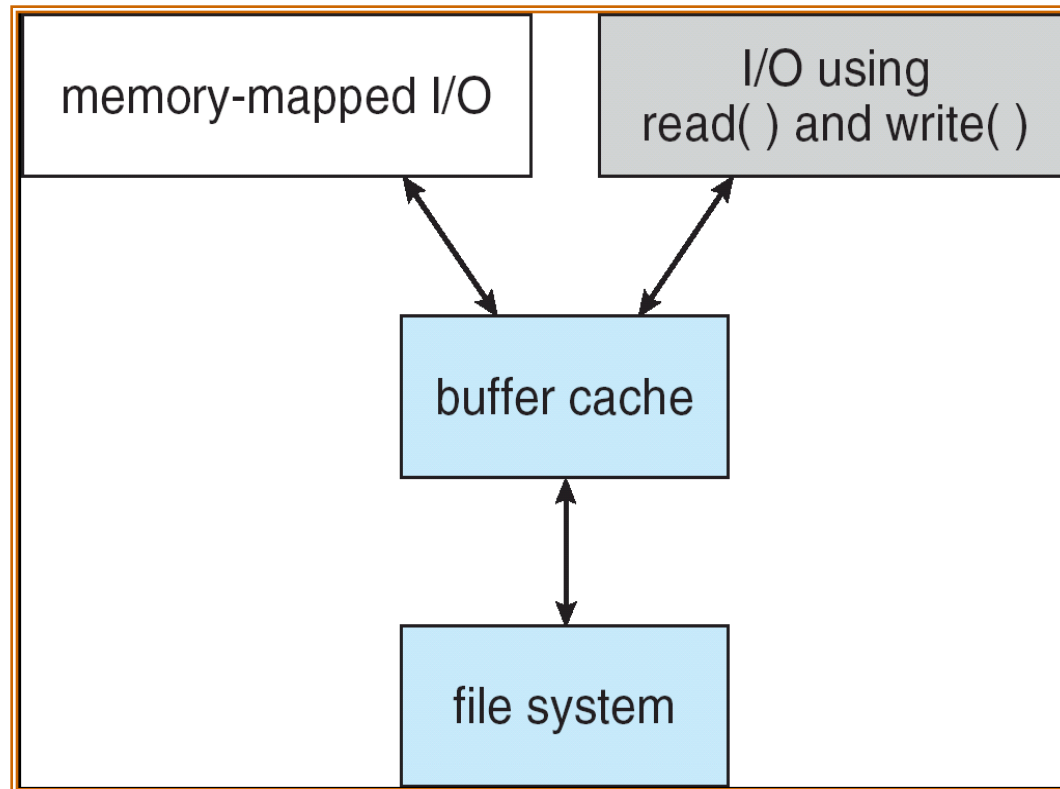
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques.
- Memory-mapped I/O uses a page cache.
- Routine I/O through the file system uses the buffer (disk) cache.



Unified Buffer Cache

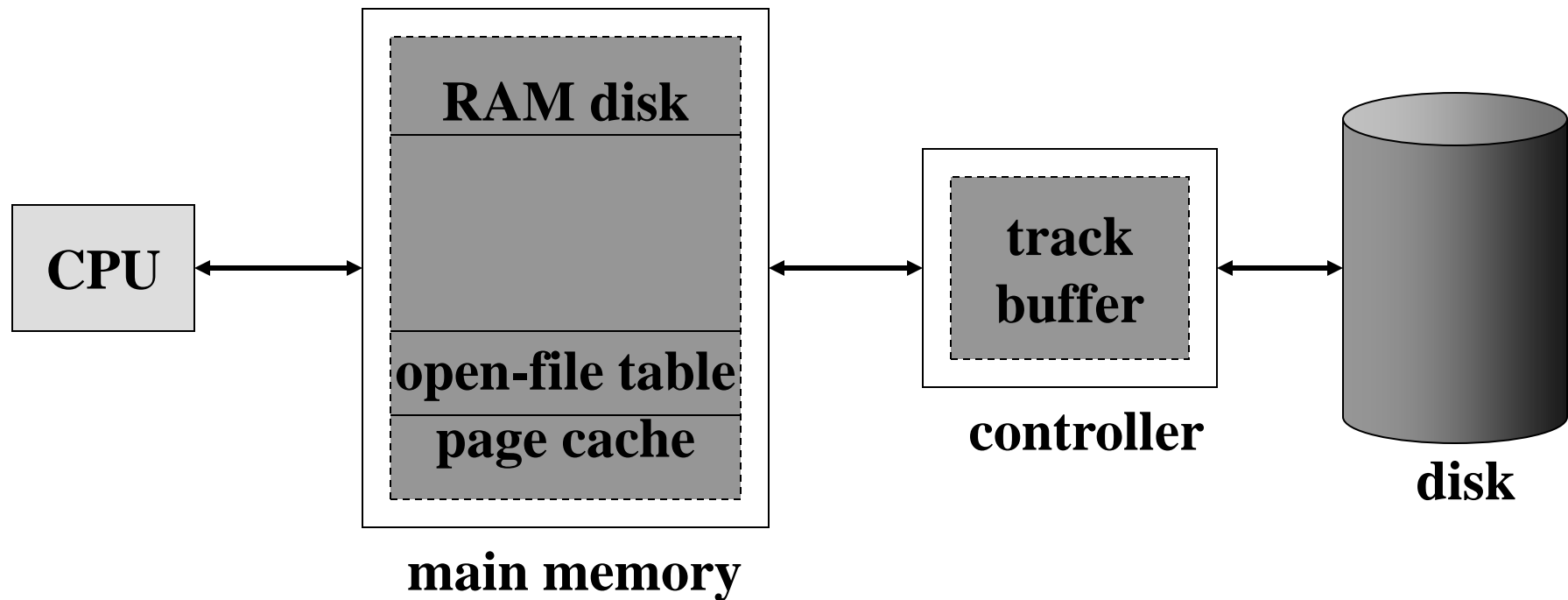
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.



Free-behind and read ahead

- **Synchronous writes** occur in the order in which the disk subsystem receives them, and the writes are not buffered.
- In an **asynchronous write** the data is stored in the cache and returns control to the caller.
- Using a flag in the open system call to allow a process to request that writes be performed synchronously.
- **Free-behind** removes a page from the buffer as soon as the next page is requested.
- With **read-ahead**, a requested page and several subsequent pages are read and cached.

Various Disk-Caching Locations



RAM disk: contents are controlled by user

Disk cache: controlled by OS

11.7* Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Use system programs to *back up* data from disk to another storage device (floppy disk, magnetic tape).
- Recover lost file or disk by *restoring* data from backup.
- A typical backup schedule
 - Day 1: full backup
 - Day 2: incremental backup
 - ...
 - Day N: incremental backup. Go back to Day 1.

11.8* Log Structured File Systems

- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**.
- All transactions are written to a **log**. A transaction is considered **committed** once it is written to the log. However, the file system may not yet be updated.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.

Homework (page 447)

■ 11.6

■ Thinking about: 11.1 11.2 11.3

- 1. Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?**
- 2. Explain how the VFS layer allows an operating system easily to support multiple types of file systems.**

- 3. Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.**
- a. The block is added at the beginning.**
 - b. The block is added in the middle.**
 - c. The block is added at the end.**
 - d. The block is removed from the beginning.**
 - e. The block is removed from the middle.**
 - f. The block is removed from the end.**



连续文件举例

例如：某文件系统，磁盘块大小为 512B

- 字符流文件A，长度为1980B
 - 文件A需要占用4个物理块，假如分配到30、31、32和33四个相邻的物理块中。
 - 第33块中实际使用了444字节，剩余的68字节形成“内部碎片”。
- 记录文件B，逻辑记录长100B，有23个记录
 - 逻辑记录不能跨物理块存放
 - 每个物理块中可以存放5个逻辑记录，需要5个物理块，假设分配到第6、7、8、9、10块
 - 前4块各有内部碎片12B，最后一块有212B没有使用

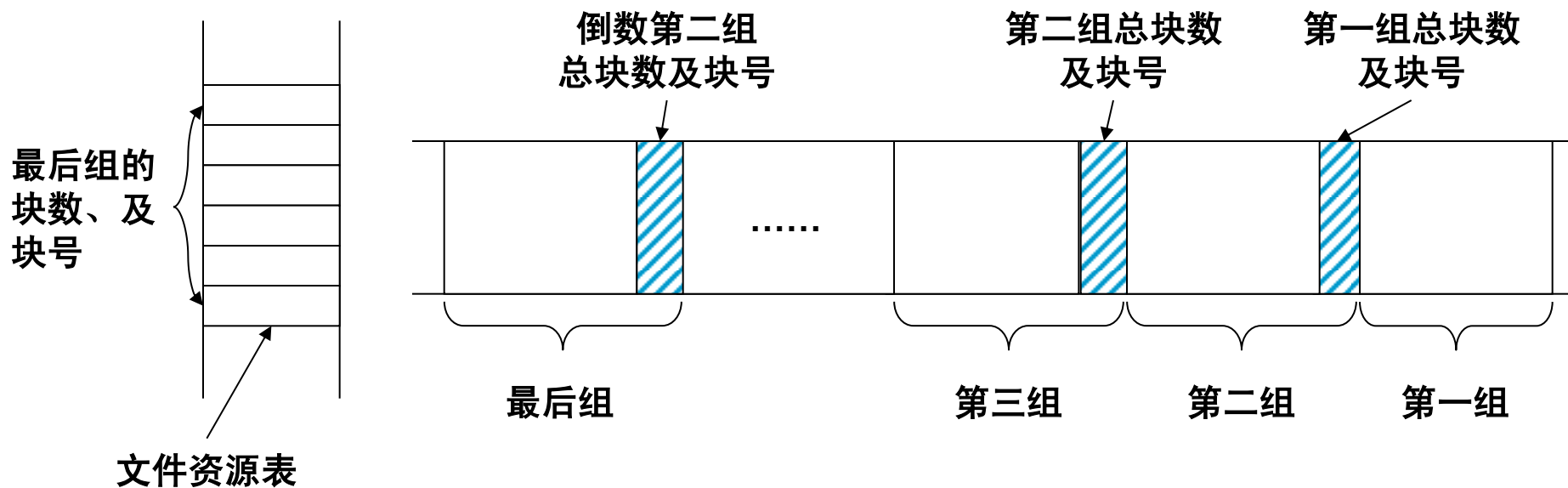
连续文件的地址转换

- 在文件说明信息中有关于存放位置的描述
 - 开始块号、总块数（物理文件的长度）
- 字符流文件中逻辑地址L
 - $L/\text{物理块大小}$ ，商s：逻辑块号，余数d：块内地址
 - 物理地址：块号：开始块号+s，块内地址：d
- 记录文件中的逻辑记录号n(记录不可跨块存放时)
 - 每个物理块中可以存放逻辑记录数m
 - $m = \text{物理块大小} / \text{逻辑记录大小}$
 - n/m ，商s：逻辑块号，余数w：块内记录号
 - 物理地址
 - 块号：开始块号+s，块内地址d=逻辑记录长度 $\times w$

UNIX: 成组链接的方法

- 把文件存储设备中的所有空闲块**从后向前**，按50块为一组进行划分。
- 每组的第一块用来存放**前一组**中各块的块号和总块数。
- 由于第一组前面已无其他组存在，因此第一组为49块。
- 由于存储空间不一定正好是50的整数倍，所以最后一组有可能不足50块
- 由于该组后边已经没有其他组了，所以最后一组的块号与总块数只能放在管理文件存储设备的文件资源表中。

成组链接示意图

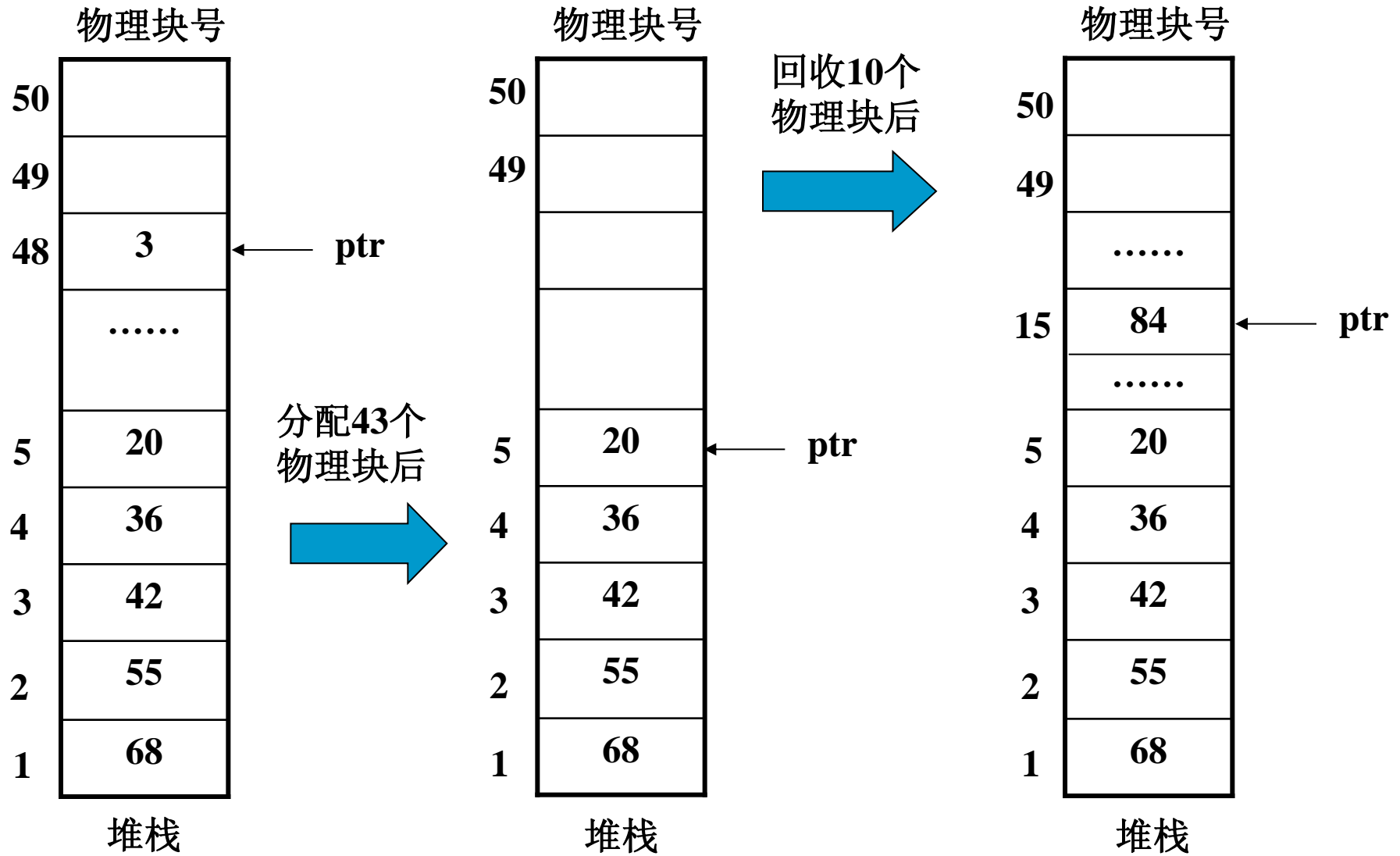


- 系统启动时把文件资源表复制到内存
 - 使文件资源表中的最后一组空闲块总数及块号的堆栈进入内存
- 设置一个用于空闲块分配与回收的堆栈
 - 栈指针ptr，ptr的初值等于该组空闲块的总块数。

成组链接法的分配方法

- 申请者要求的空闲块数 n
- 按照后进先出的原则分配栈中的空闲块
 - 将 ptr 所指的块号分配给请求者
 - $ptr \leftarrow ptr - 1$
- 重复此操作，直到所要求的 n 块都已分配完毕、或堆栈中只剩下最后一个空闲块的块号。
 - 若堆栈中只剩下最后一个空闲块号，弹出该块号，系统启动设备I/O，将该块中存放的下一组的块数和块号读入内存，压入栈中，将该块分配给请求者。
 - 重新设置 ptr =下一组的块数
 - 继续为申请者分配空闲块。
- 文件存储设备的最后一个空闲块中设置有尾部标识，指示空闲块分配完毕。

示例



成组链接法的回收方法

- 当用户删除某文件时，回收空闲块
- 空闲块号入栈
 - $ptr \leftarrow ptr + 1$
 - 把回收的物理块号放入ptr所指的位置
- 如果ptr=50，表示该组已经回收结束
 - 如果还有空闲物理块F需要回收，回收该块并启动设备I/O，把栈中记录的50个块号与块数50写入块F中。
 - 设置ptr=1，将块F的块号入栈，开始新的一组空闲块的回收。
- 对空闲块的分配和释放过程对栈的操作必须互斥进行，否则会发生数据混乱。

Chapter 12 Mass-Storage Structure



LI Wensheng, SCS, BUPT

Teaching hours: 2h

Strong points:

Disk Structure

Disk Scheduling

Disk Management

Chapter Objectives

- **Describe the physical structure of disk and disk scheduling algorithms**
- **Describe the policies of disk management**

Contents

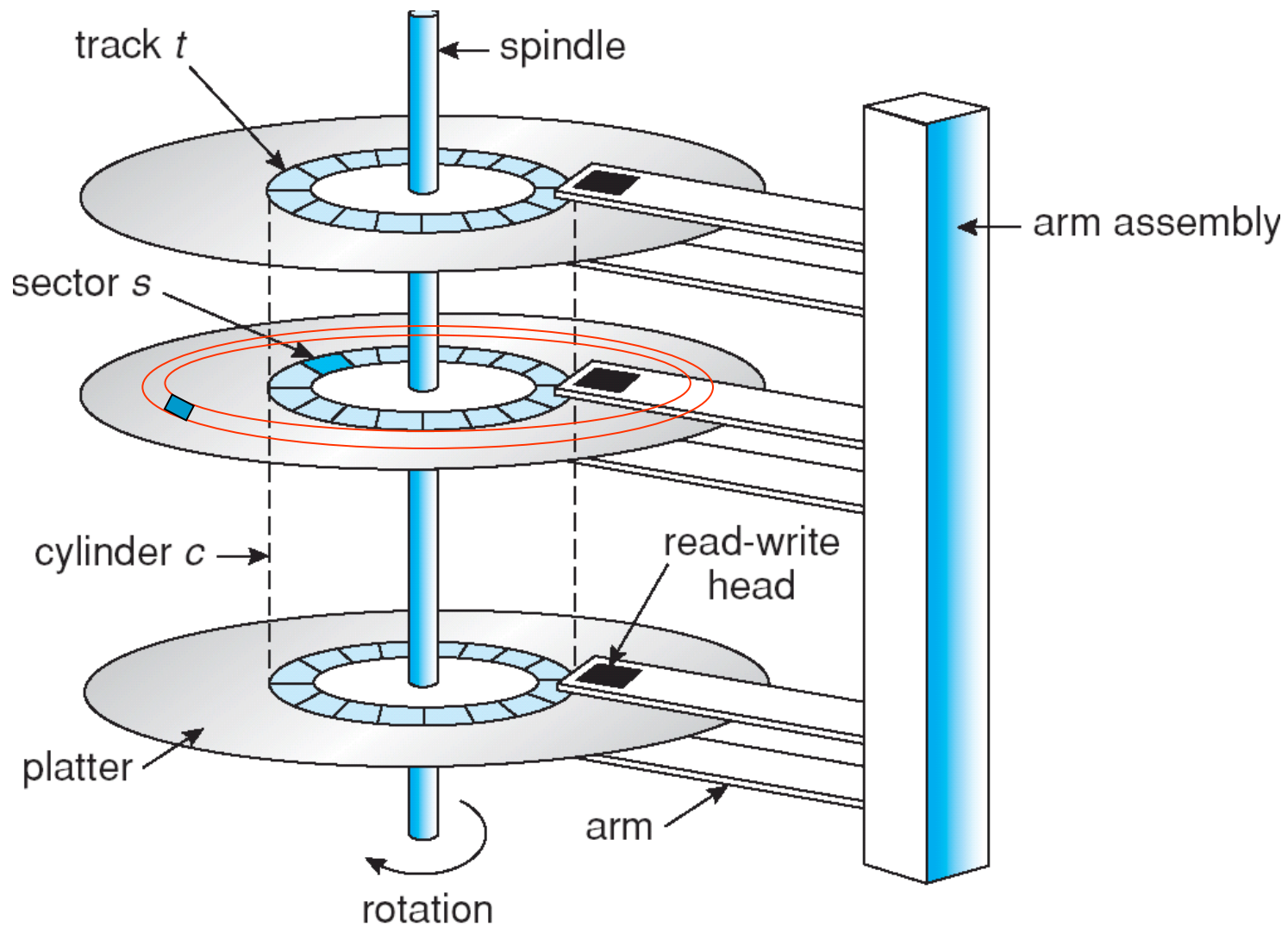
- 12.1 Overview of Mass Storage Structure**
- 12.2 Disk Structure**
- 12.3 Disk Attachment**
- 12.4 Disk Scheduling(√)**
- 12.5 Disk Management**
- 12.6 Swap-Space Management**

- 12.7 RAID Structure(*)**
- 12.8 Stable-Storage Implementation (*)**
- 12.9 Tertiary-Storage Structure (*)**

1. Overview of Mass Storage Structure

- **Magnetic disks provide bulk of secondary storage of modern computers**
 - Drives rotate at 60 to 200 times per second
 - **Transfer rate** is rate at which data flow between drive and computer
 - **Positioning time (random-access time)** is the time to move disk arm to desired cylinder (**seek time**) and the time for desired sector to rotate under the disk head (**rotational latency**)
 - **Head crash** results from disk head making contact with the disk surface
 - That's bad, cannot be repaired, the entire disk must be replaced
- **Disks can be removable**
- **Drive attached to computer via I/O bus**
 - Busses vary, including **EIDE, ATA, SATA, USB, Fiber Channel, SCSI**
 - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array

Moving-head Disk Mechanism

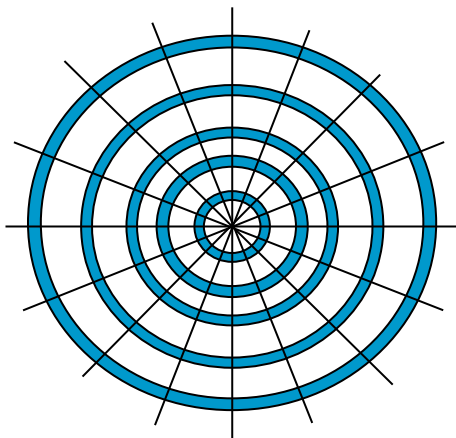
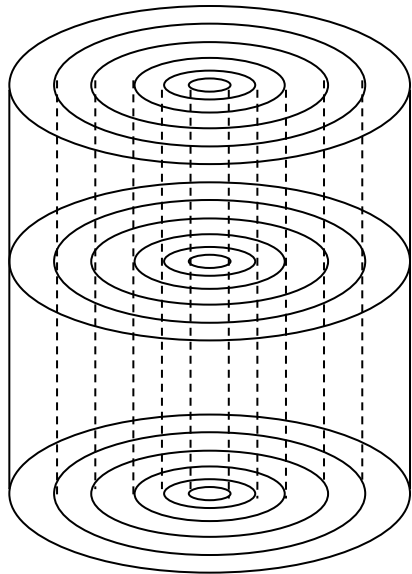


Overview of Mass Storage Structure (Cont.)

■ Magnetic tape

- Was early secondary-storage medium
- Relatively permanent and holds large quantities of data
- Access time slow
- Random access ~1000 times slower than disk
- Mainly used for backup, storage of infrequently-used data, transfer medium between systems
- Kept in spool and wound or rewound past read-write head
- Once data under head, transfer rates comparable to disk
- 20-200GB typical storage
- Common technologies are 4mm, 8mm, 19mm, 1/4 inch, 1/2 inch, named according to technology, LTO-2 and SDLT

12.2 Disk Structure



- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer. (block size is usually 512 bytes)
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
 - Sector 0 is the first sector of the first track on the outermost cylinder.
 - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.
- Constant Linear Velocity (常数线性周转率)
 - 位密度相同, CD-ROM, DVD-ROM
- Constant Angular Velocity (常数角周转率)
 - 位密度不同, hard disk

12.3 Disk Attachment

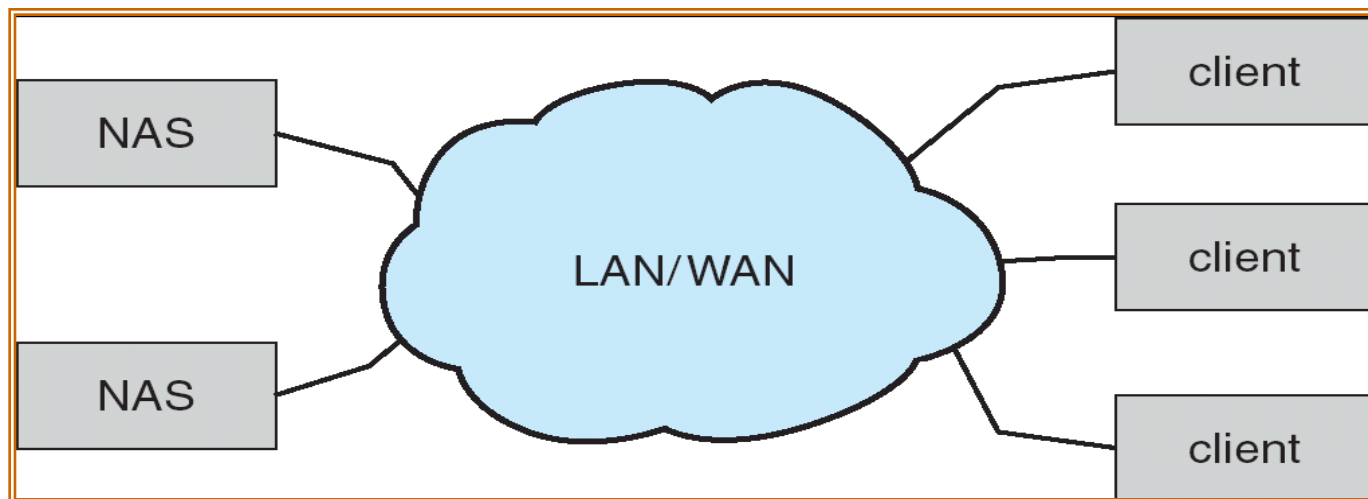
- **Computer access disk storage in two ways**
 - **Via I/O ports (host-attached storage)**
 - **Via a remote host in a distributed file system (NAS)**

Host-attached storage

- Is accessed through I/O ports talking to I/O busses
- **IDE, ATA**, support maximum of 2 drives per I/O bus
- **SATA (Serial ATA)**
- **SCSI** itself is a bus, up to 16 devices on one cable, **SCSI initiator (controller card in the host)** requests operation and **SCSI targets (storage devices)** perform tasks
 - A SCSI disk is a common SCSI target
 - Each target can have up to 8 **logical units** (disks attached to device controller)
- **FC** is a high-speed serial architecture, can operate over optical fiber or over a 4-conductor copper cable, two variants:
 - Can be switched fabric with 24-bit address space – the basis of storage area networks (SANs) in which many hosts attach to many storage units
 - Can be **arbitrated loop (FC-AL, Fibre Channel Arbitrated Loop)** of **126 devices**

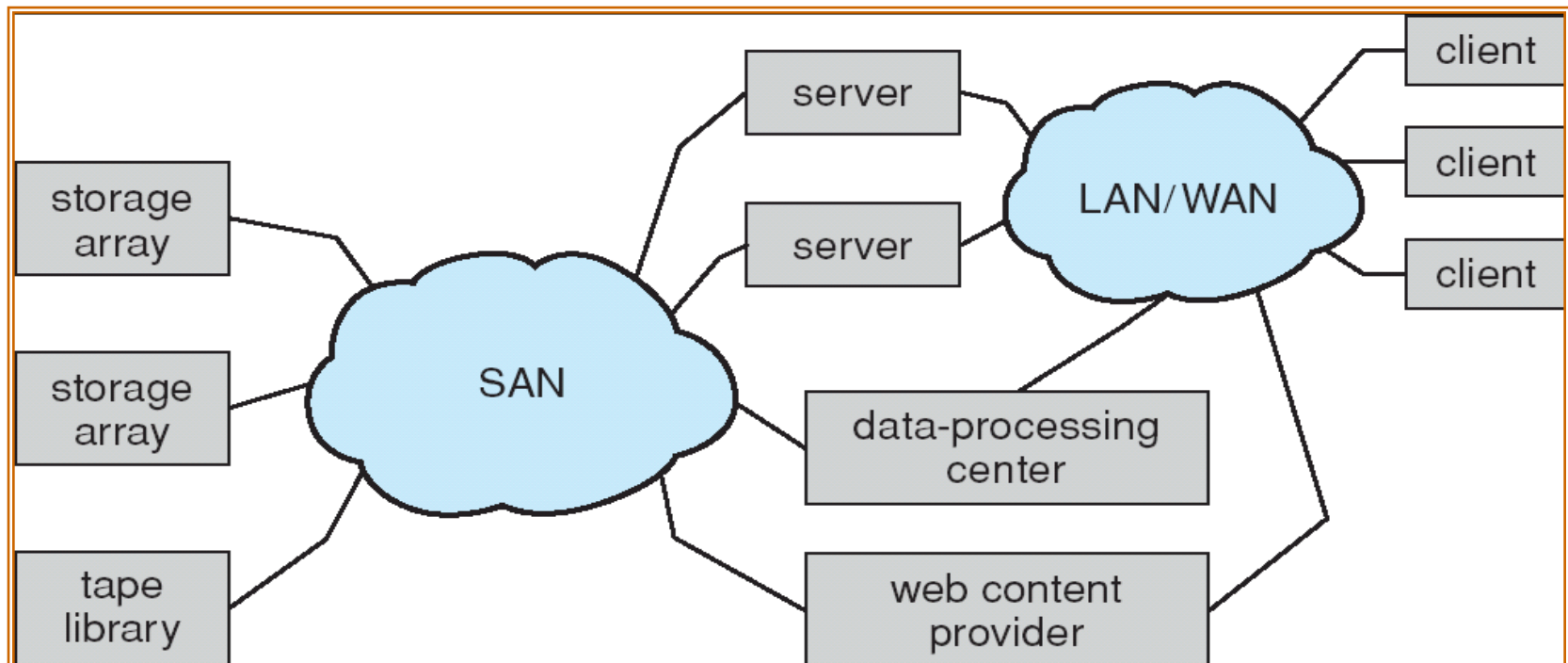
Network-Attached Storage

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



Storage Area Network

- Private network, connecting servers and storage units
- Common in large storage environments (and becoming more common)
- Multiple hosts attached to multiple storage arrays – flexible
- FC is the most common SAN interconnect.



12.4 Disk Scheduling

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

Disk Scheduling (Cont.)

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

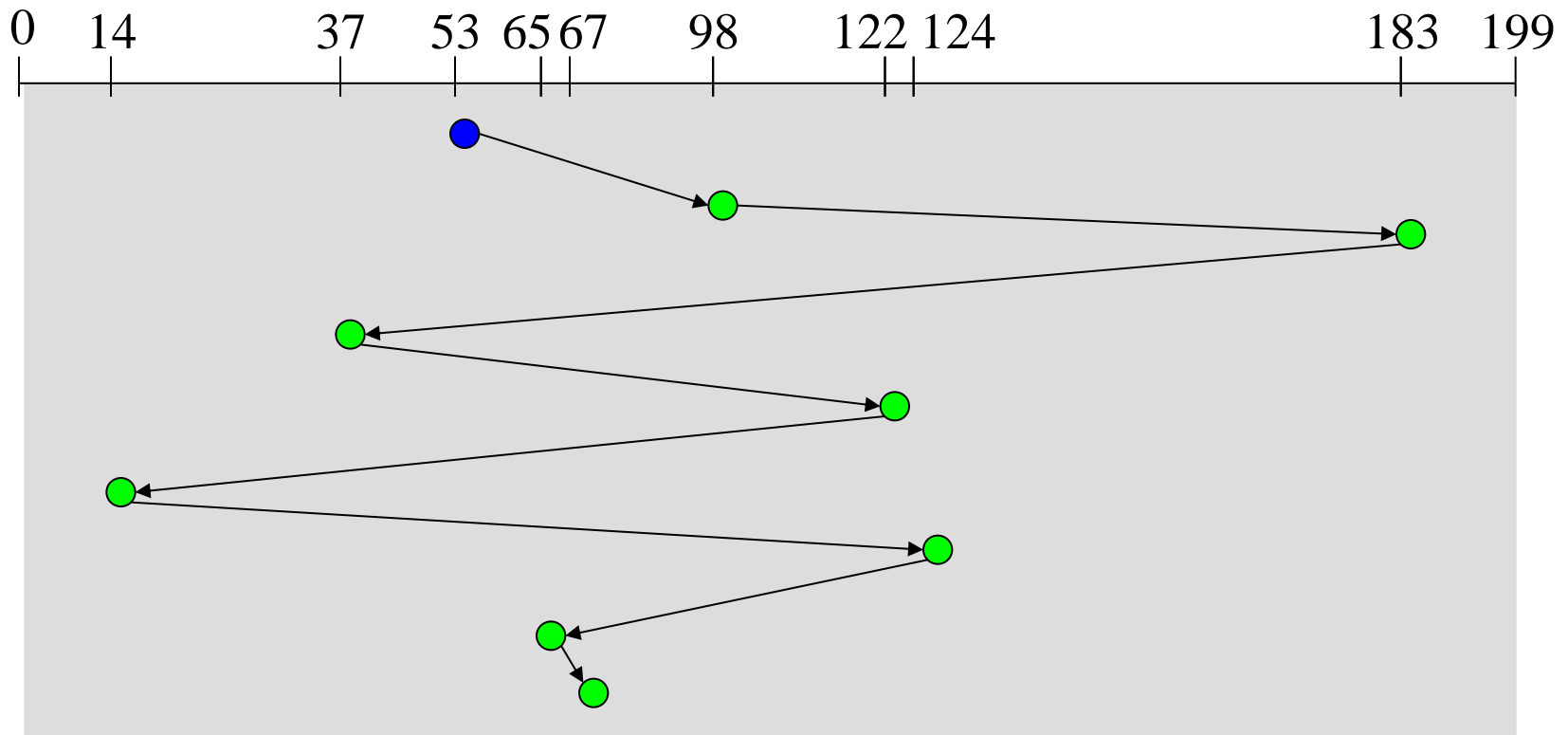
98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

- **FCFS Scheduling**
- **SSTF Scheduling**
- **SCAN Scheduling**
- **C-SCAN Scheduling**
- **LOOK Scheduling**

FCFS — First Come First Served

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

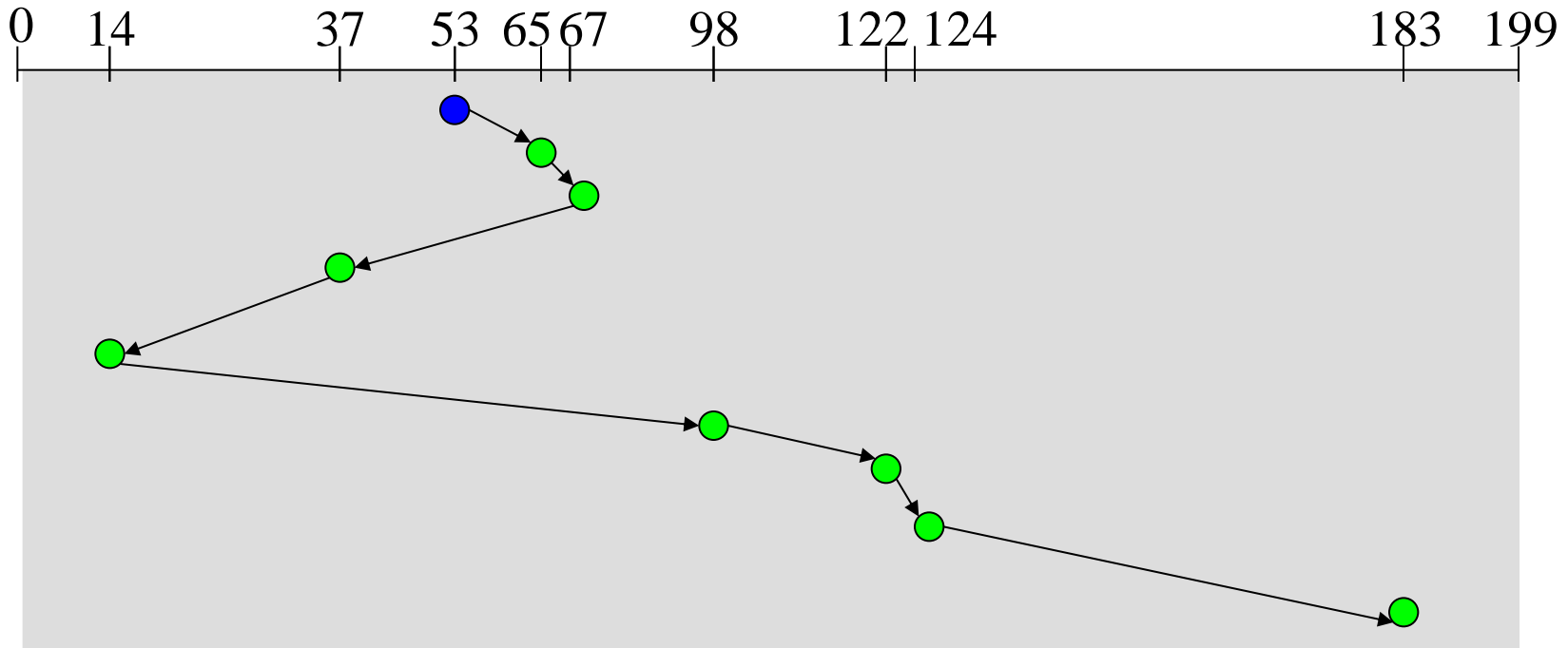


total head movement of 640 cylinders.

SSTF — Shortest Seek Time First

- Selects the request with the minimum seek time from the current head position.

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



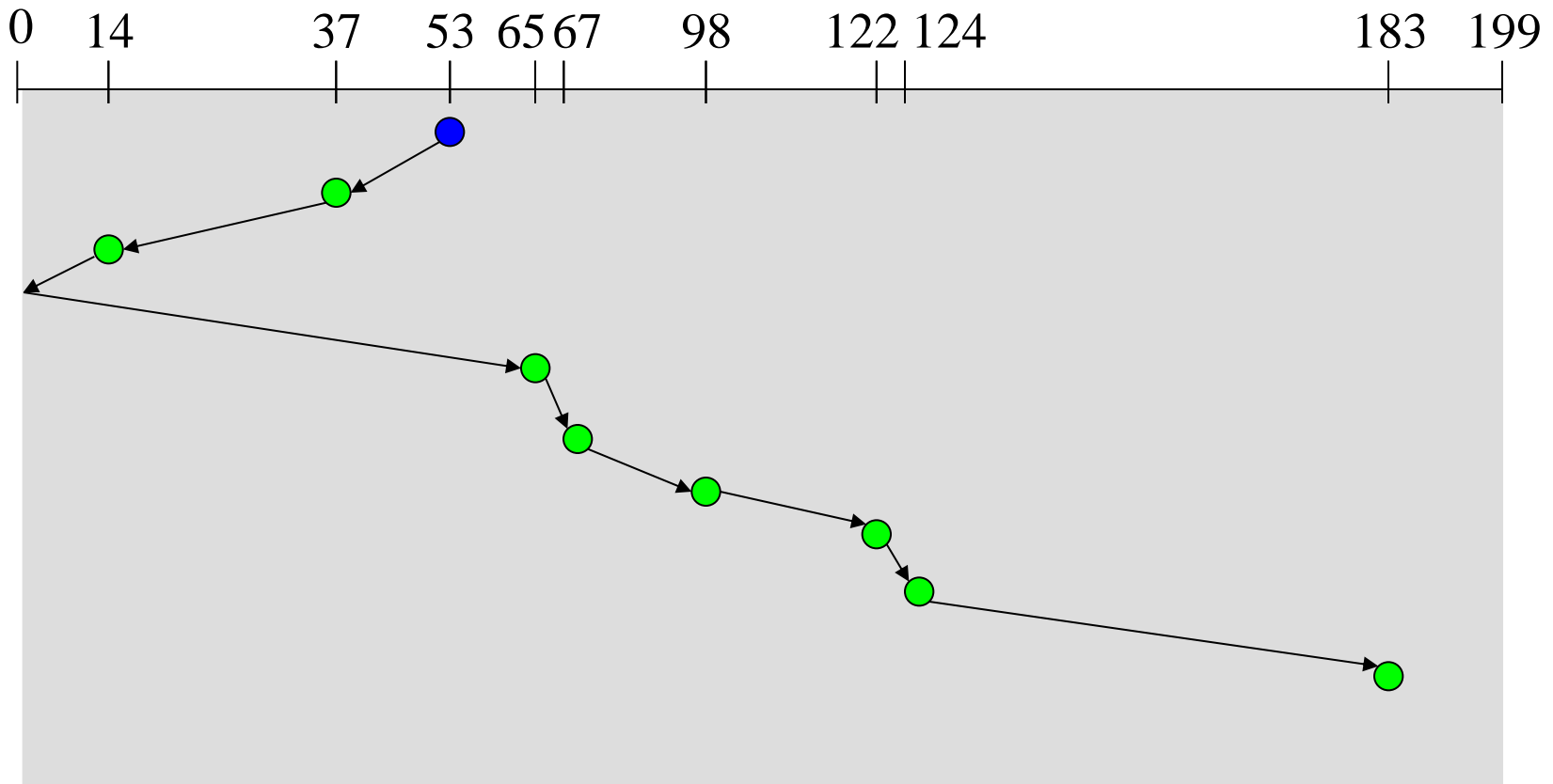
- Head movement of 236 cylinders.
- starvation

SCAN Scheduling

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

SCAN (Cont.)

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



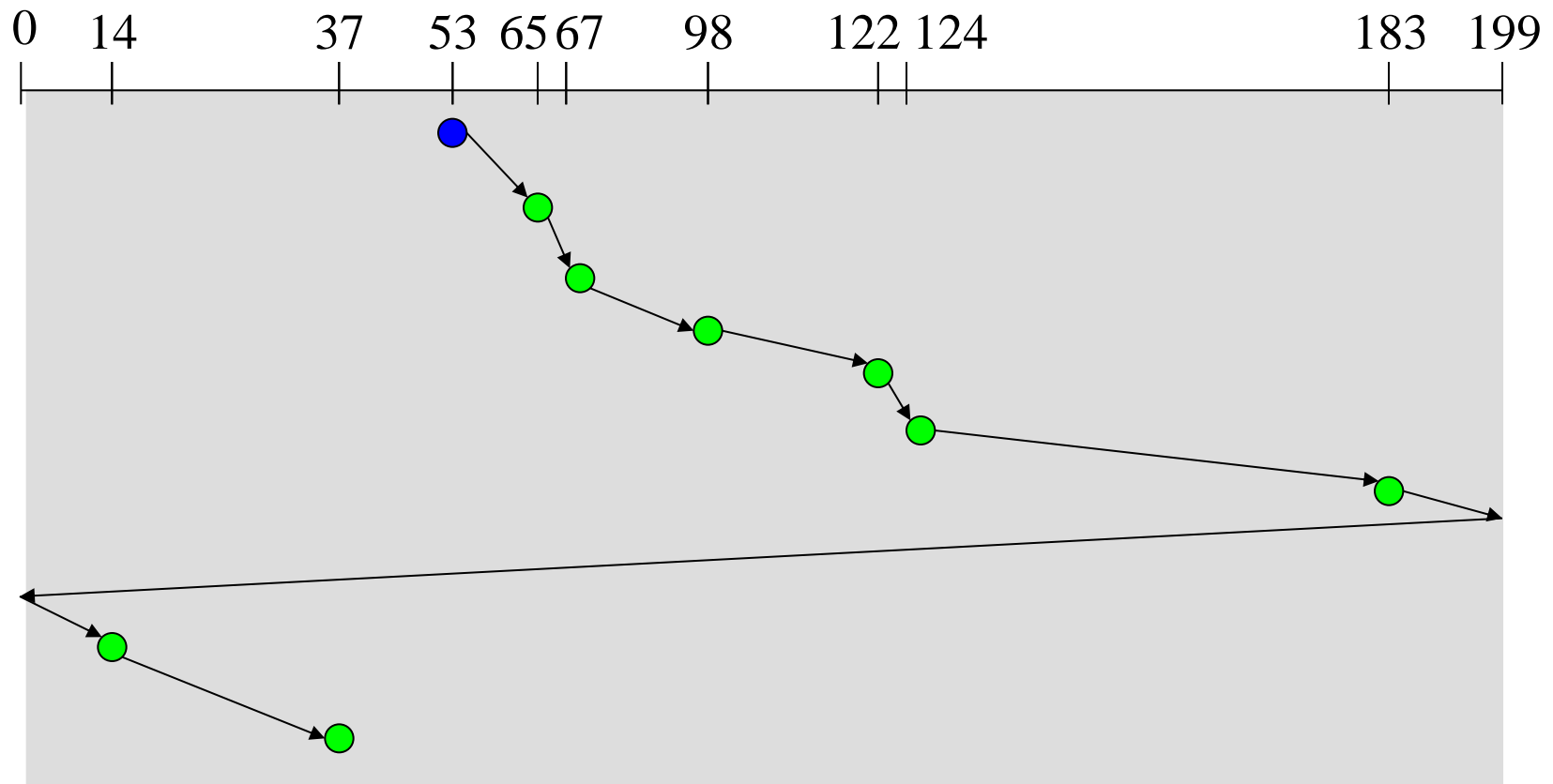
total head movement of 236 cylinders.

C-SCAN Scheduling— Circular SCAN

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

C-SCAN (Cont.)

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

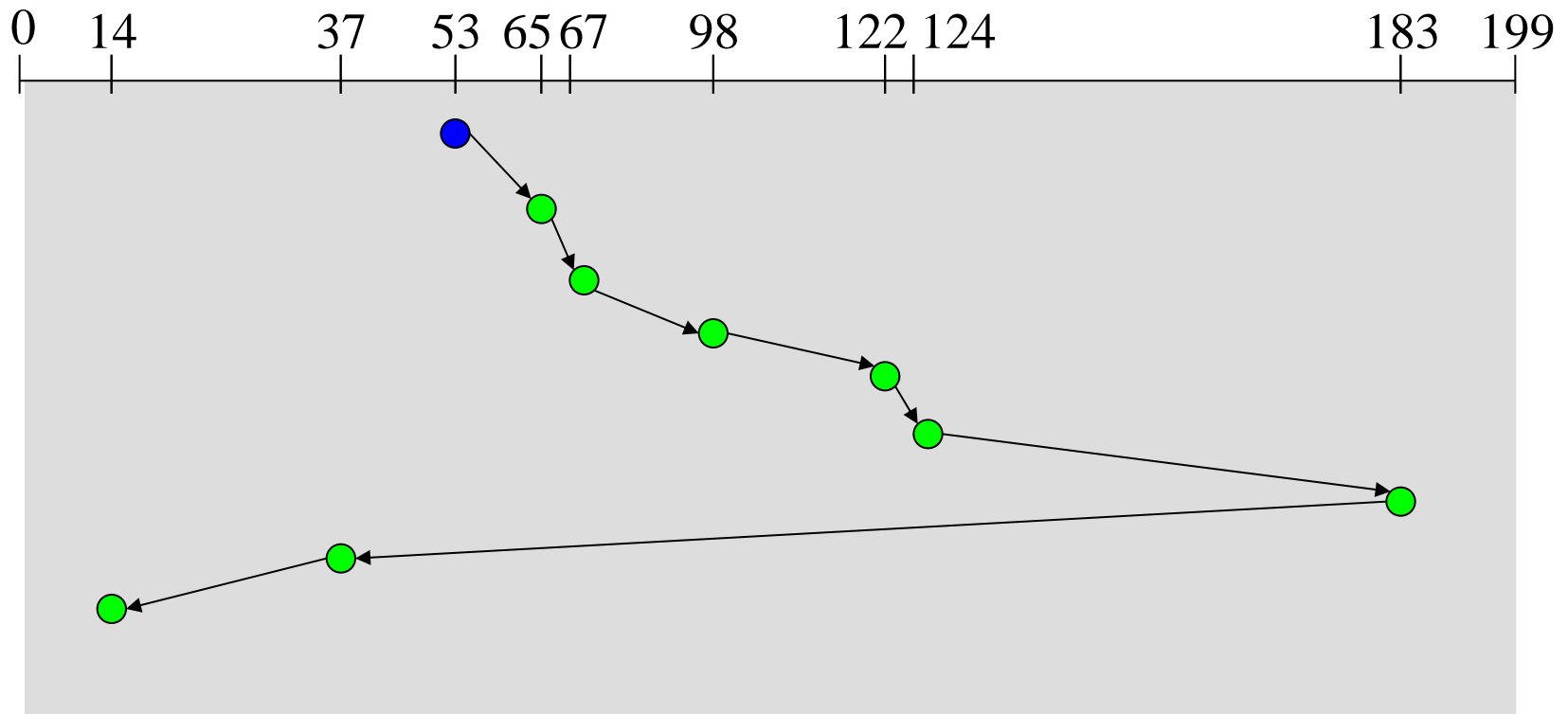


LOOK and C-LOOK Scheduling

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

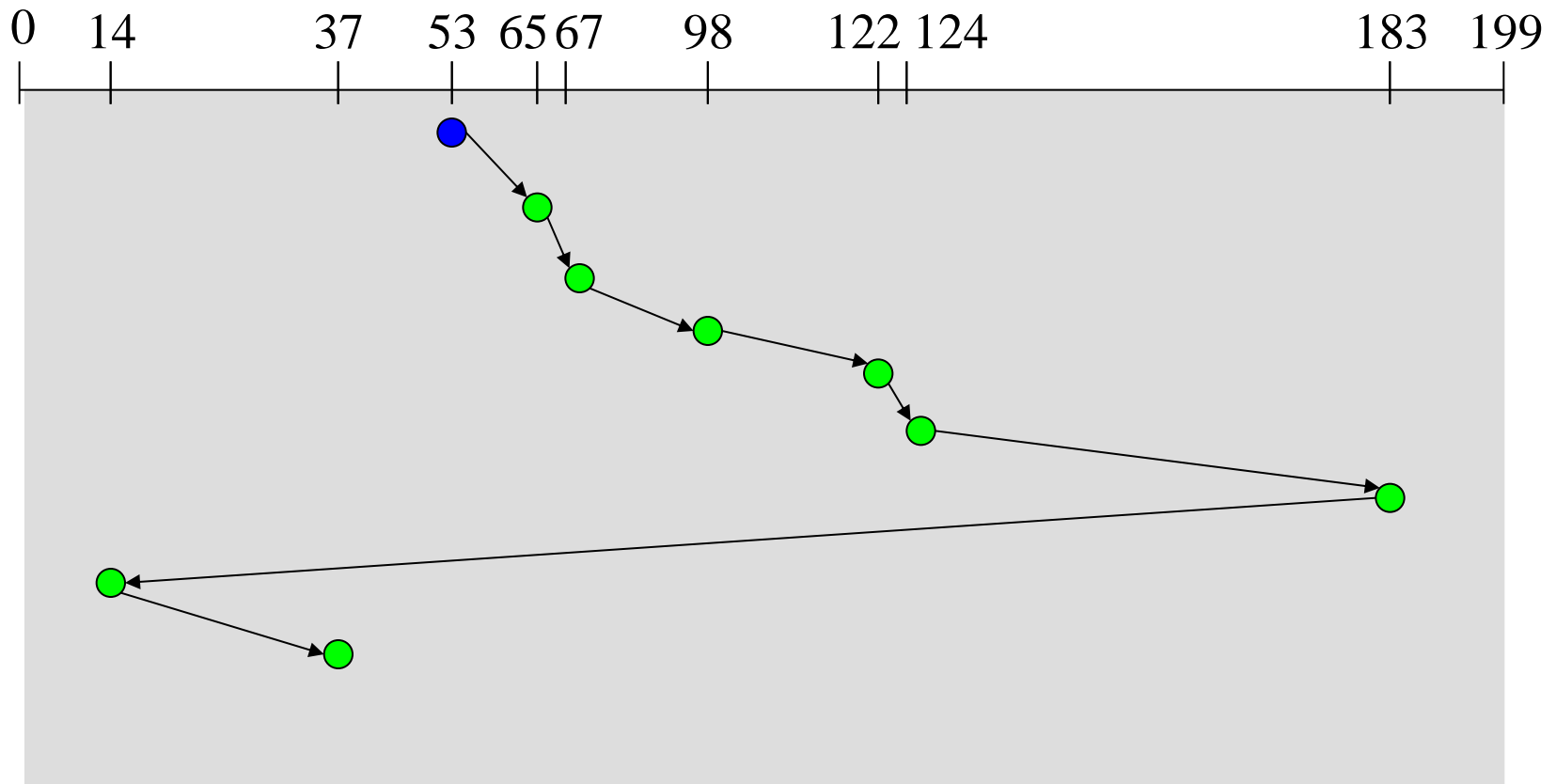
LOOK (Cont.)

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



C-LOOK (Cont.)

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



Selecting a Disk-Scheduling Algorithm

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

12.5 Disk Management

- **Formatting**
- **booting from disk**
- **recovery from bad block**

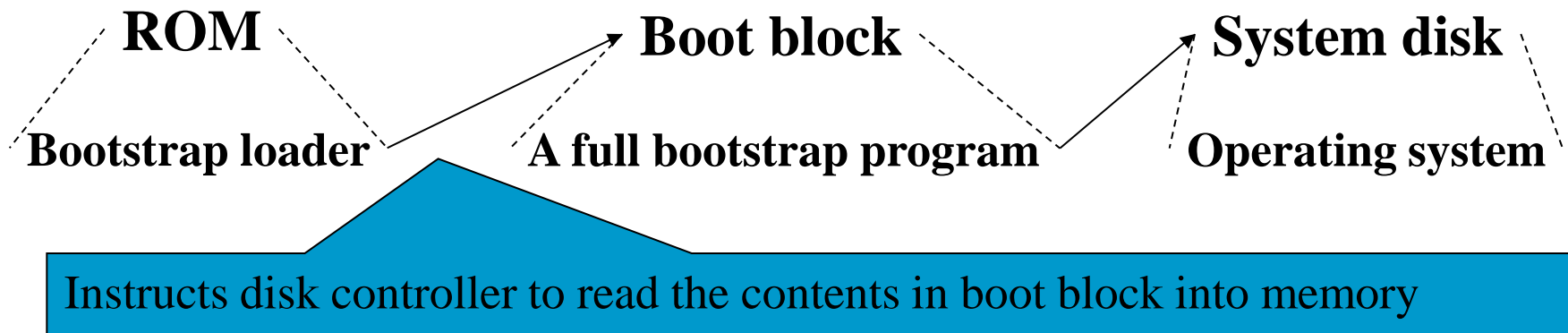
Disk formatting

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- *Low-level formatting* fills the disk with a special data structure for each sector. The data structure typically consists of a header, a data area, and a trailer.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
 - *Partition* the disk into one or more groups of cylinders.
 - *Logical formatting* or “making a file system”.

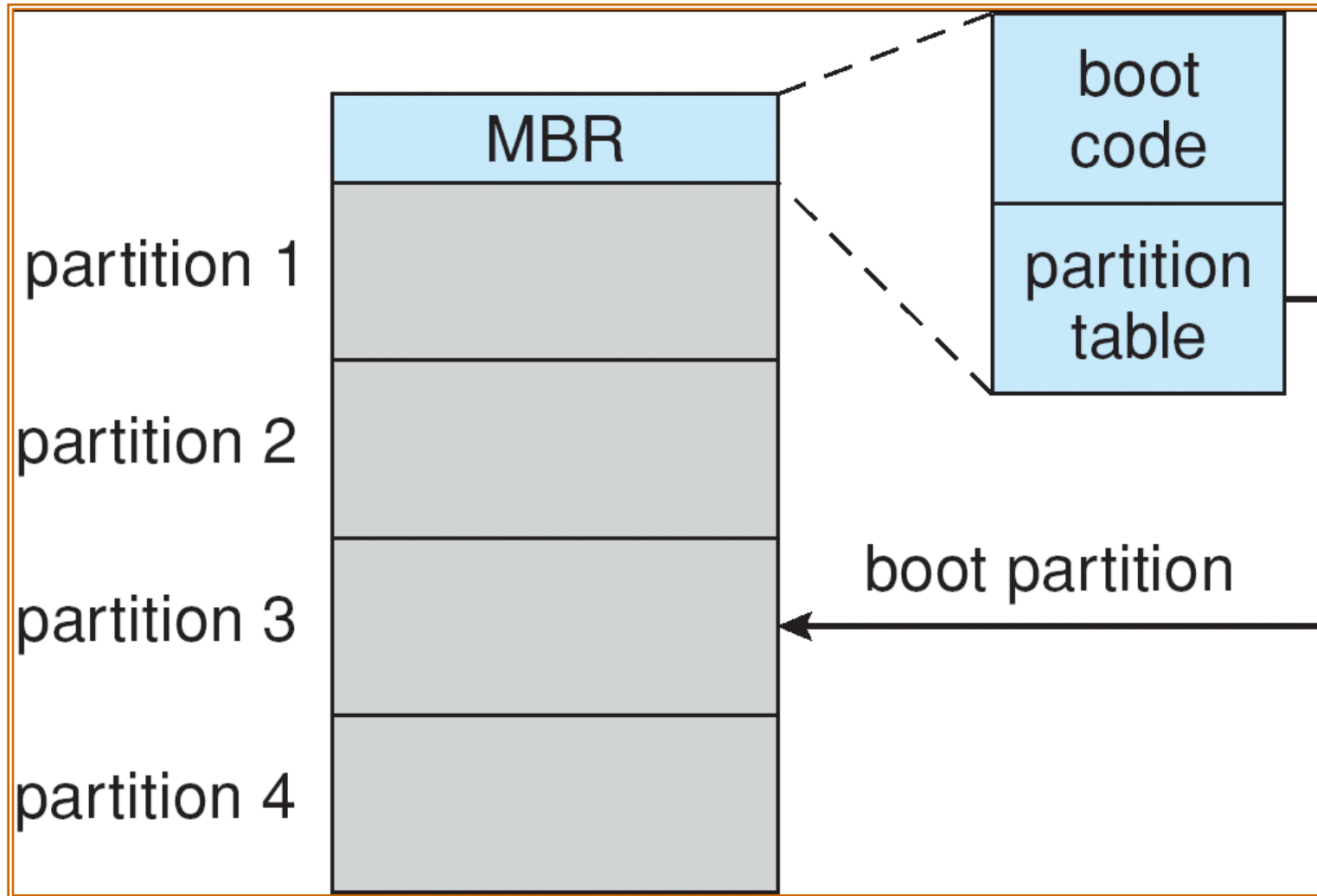
Boot block

■ Boot block initializes system

- Initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.
- The **bootstrap** is stored in ROM.
- Most systems store a tiny *bootstrap loader* program in the boot ROM.
- The full bootstrap program is stored in a partition called the **boot blocks**, at a fixed location on the disk.



Booting from a Disk in Windows 2000



Bad blocks

Methods used to handle bad blocks depends on the disk and controller in use.

- On simple disks, such as some disks with IDE controllers, bad blocks are handled manually.
 - MS-DOS, *format* command, *chkdsk* command
- For more sophisticated disks
 - methods such as *sector sparing* (or *forwarding*) are used to handle bad blocks.
 - Some controllers can be instructed to replace a bad block by *sector slipping*.

12.6 Swap-Space Management

- **Swap-space — Virtual memory uses disk space as an extension of main memory.**
- **Swap-space Use**
- **Swap-space Location**
- **Swap-space management**

Swap-space Use

- **Depending on the memory-management algorithms in use.**
 - Use swap space to hold an entire process image
 - Store pages that have been pushed out of memory
- **The amount of swap space needed depends on**
 - the amount of physical memory
 - the amount of virtual memory it is backing
 - the way in which the virtual memory is used

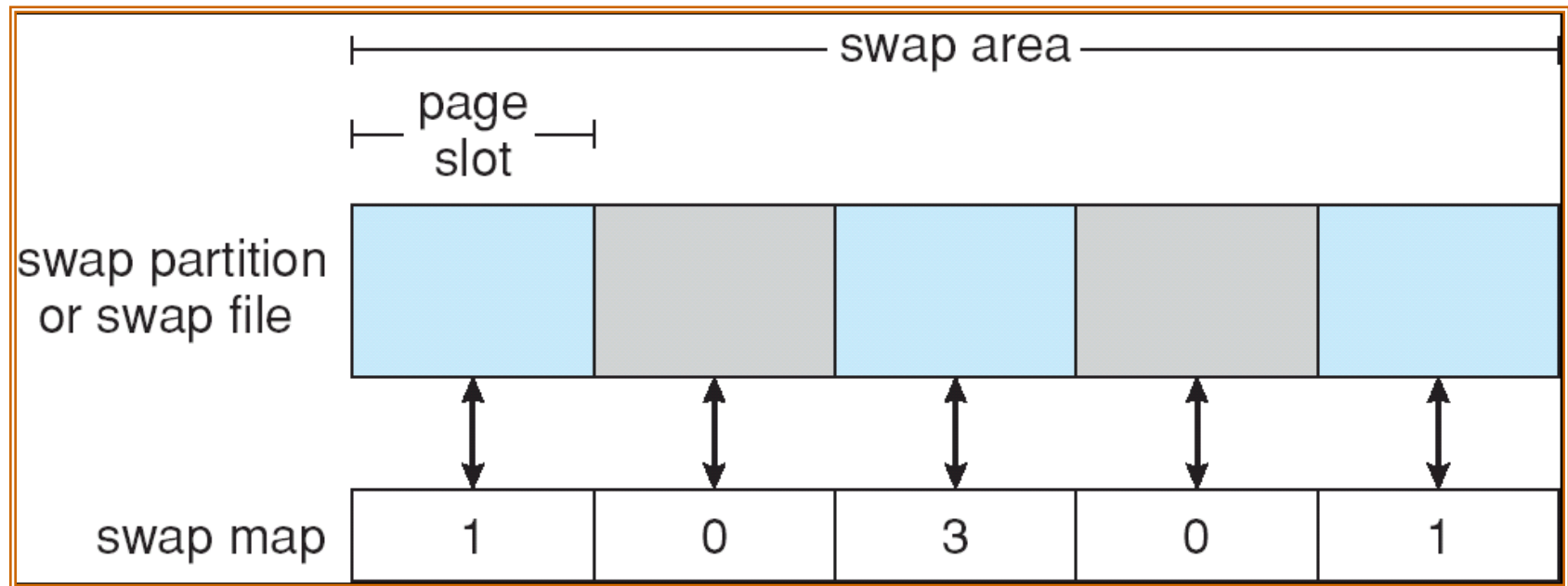
Swap-space Location

- A swap space can reside in one of two places:
 - Carved out of the normal file system, **a large file**
 - In a separate disk partition, **raw partition**
- A large file
 - Normal file-system routines can be used to create it, name it, and allocate its space.
 - Inefficient
- Raw partition
 - Swap-space storage manager, allocate and deallocate
 - A fixed amount of swap space is created during disk partitioning, adding more swap space requires repartitioning the disk.

Swap-space management

- 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment*.
- Kernel uses *swap maps* to track swap-space use.
- Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.

Data Structures for Swapping on Linux Systems



Homework (page 489)

■ 12.2

Thinking:
12.1



Chapter 13 I/O Systems



LI Wensheng, SCST, BUPT

Teaching hours: 2h

Strong points:

I/O Hardware

Application I/O Interface, Kernel I/O Subsystem

**Transforming I/O Requests to Hardware
Operations**

Chapter Objectives

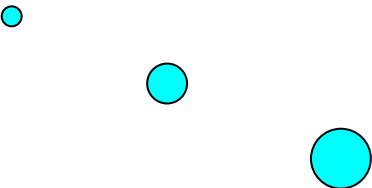
- **Explore the structure of an operating system's I/O subsystem**
- **Discuss the principles of I/O hardware and its complexity**
- **Provide details of the performance aspects of I/O hardware and software**

Contents

- 13.1 Overview**
- 13.2 I/O Hardware**
- 13.3 Application I/O Interface**
- 13.4 Kernel I/O Subsystem**
- 13.5 Transforming I/O Requests to Hardware Operations**
- 13.6* Streams**
- 13.7* Performance**

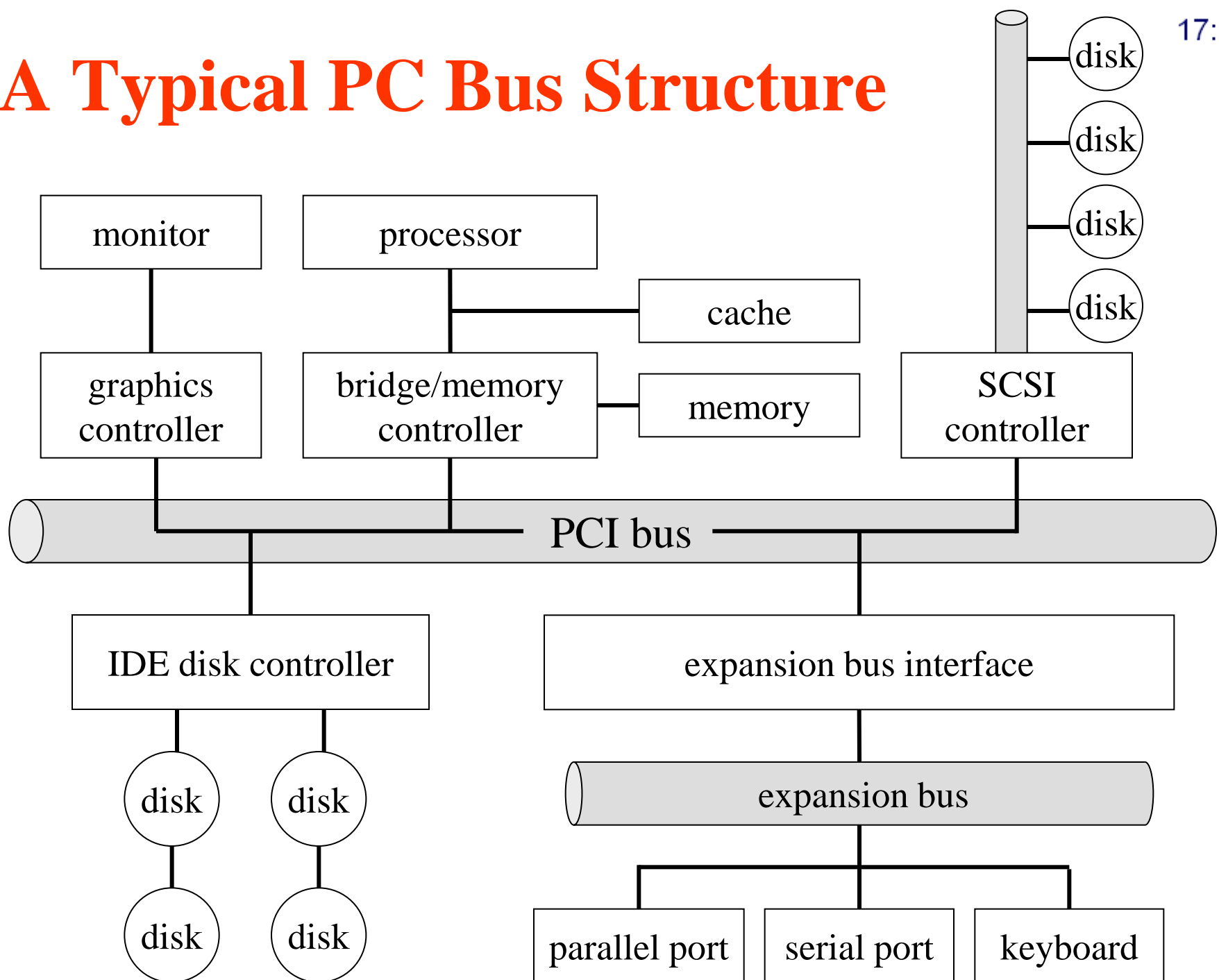
13.1 Overview

- **Incredible variety of I/O devices**
- **Two trends**
 - **Increasing standardization of software and hardware interfaces**
 - **Increasingly broad variety of I/O devices**



**I/O device?
difference?
Functions?**

A Typical PC Bus Structure



I/O device control

- **Processor give commands and data to a controller to accomplish an I/O transfer.**
- **I/O instructions control devices**
- **Devices have addresses, used by**
 - **Direct I/O instructions**
 - **Memory-mapped I/O**

Device I/O Port Locations on PCs (partial)

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

13.2 I/O Hardware

■ Common concepts

- **Port** : connection point, e.g. serial port.
- **Bus**: a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.
(daisy chain or shared direct access)
- **Controller** : a collection of electronics that can operate a port, a bus, or a device.
Serial-port controller, SCSI bus controller
host adapter: controller implemented as a separate circuit board.

I/O port

■ 4 registers

- **Status:** contains bits that can be read by the host.
Idle, busy, ready, error
- **Control:** can be written by the host to start a command or to change the mode of a device.
- **Data-in:** read by the host to get input
- **Data-out:** written by the host to send output

■ Data registers are typically 1 to 4 bytes in size.

■ Some controllers have FIFO chips to expand the capacity of the controller beyond the size of the data register.

FIFO chips can hold several bytes of input or output data

Techniques for Performing I/O

■ Programmed I/O

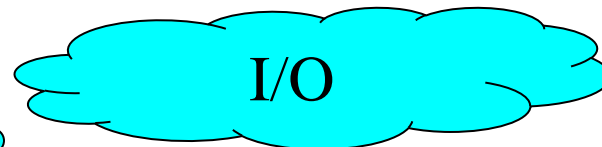
- Process is busy-waiting for the operation to complete

■ Interrupt-driven I/O

- I/O command is issued
- Processor continues executing instructions
- I/O module sends an interrupt when done

■ Direct Memory Access (DMA)

- DMA module controls exchange of data between main memory and the I/O device
- Processor interrupted only after entire block has been transferred

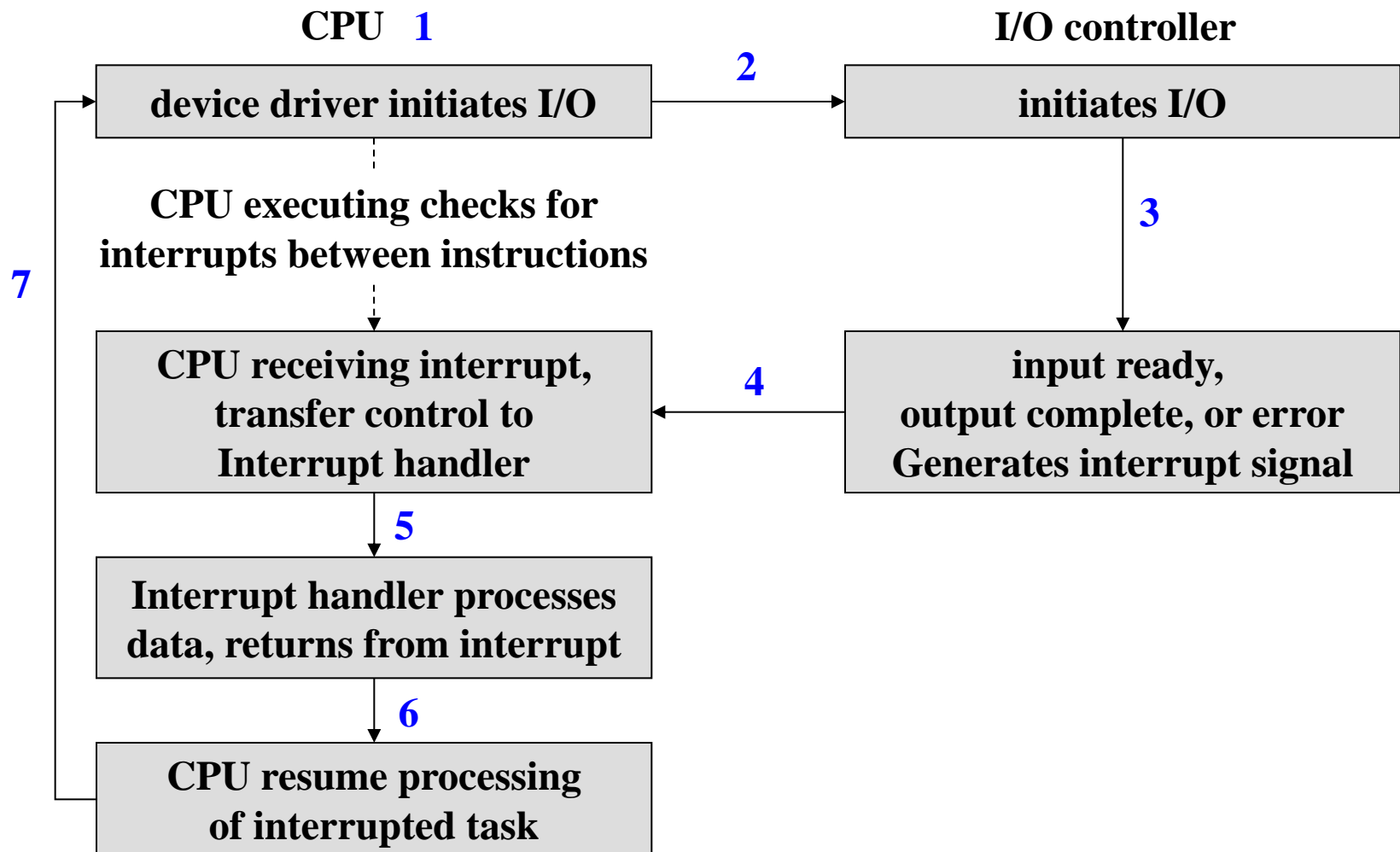


Polling

- **Determines state of device**
 - command-ready, busy, Error
- **Busy-wait** cycle to wait for I/O from device
- **Handshaking** (between host and controller)
 - host repeatedly reads the **busy bit** until that bit becomes clear.
 - host sets the **write bit**, and writes a byte into **data-out register**.
 - host sets the **command-ready bit**.
 - When the controller notices that the command-ready bit is set, it sets the **busy bit**.
 - controller **reads** the command register and sees the write command. It **reads** the data-out register to get the byte, and **does** the **I/O** to the device.
 - controller **clears** the **command-ready bit**, **clears error bit** in the status register, and **clears the busy bit**.

Interrupts

- CPU Interrupt-request line triggered by I/O device
- Interrupt handler receives interrupts
- Interrupt-Driven I/O Cycle



Two interrupts request lines

- Nonmaskable, reserved for critical events.
- Maskable, turned off to ignore or delay some interrupts.
- Intel Pentium Processor Event-Vector Table:

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

Interrupts (cont.)

- **Interrupt vector to dispatch interrupt to correct handler**
 - Based on priority
 - Some unmaskable
- **Interrupt mechanism also used for exceptions**
 - Dividing by zero
 - accessing a nonexistent memory address
 - Attempting to execute a privileged instruction from user mode
- **Good uses:**
 - Virtual memory paging (page fault)
 - System call

Direct Memory Access

- **Used to avoid programmed I/O for large data movement**
- **Requires DMA controller, a special-purpose processor**
- **Bypasses CPU to transfer data directly between I/O device and memory**
 - **Host writes a DMA command block into memory**
 - **CPU writes the address of this command block to the DMA controller**
 - **DMA controller operates the memory bus directly, placing address on the bus to perform transfers**
- **DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.**

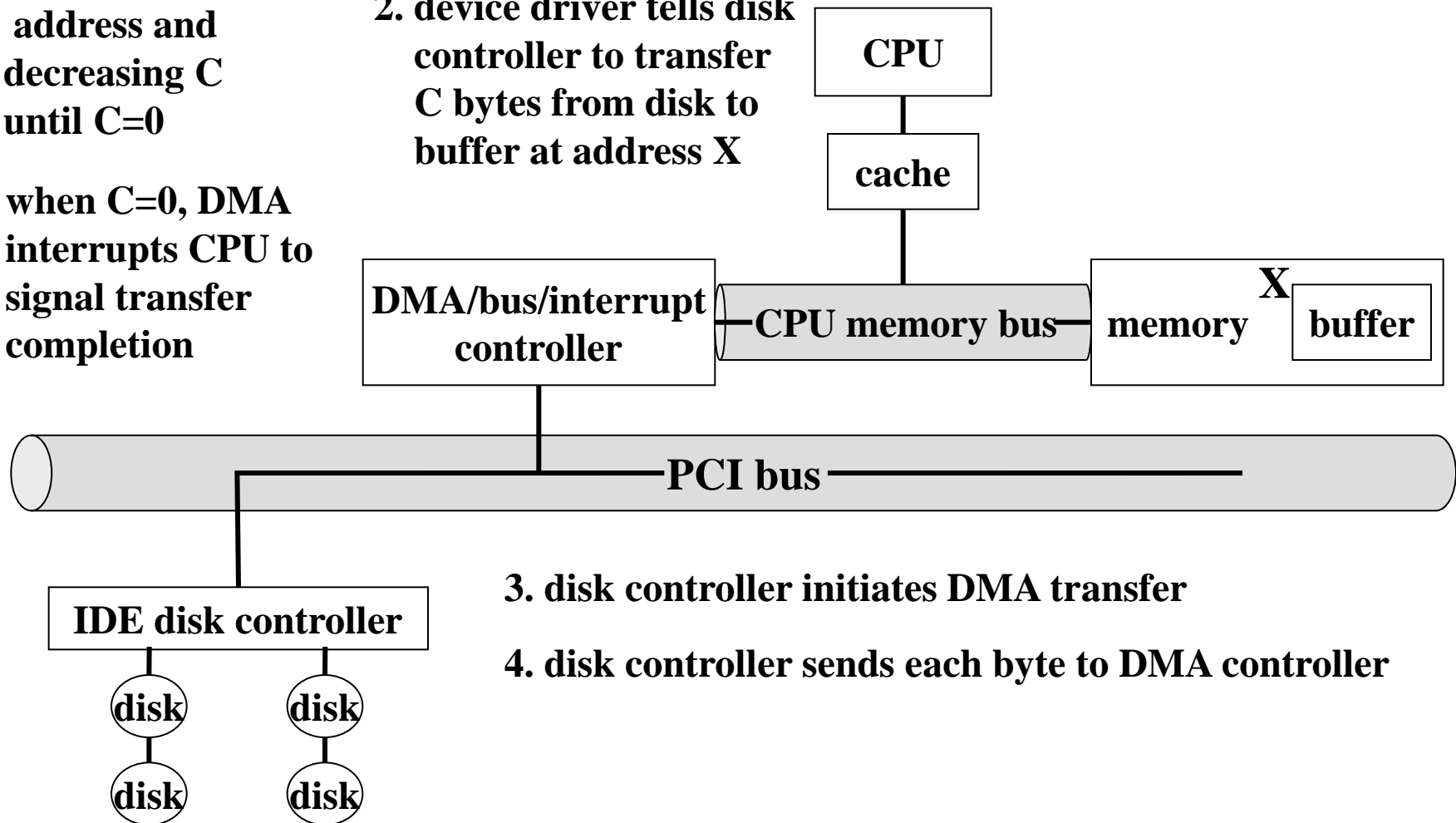
Six Step Process to Perform DMA Transfer

5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C=0

6. when C=0, DMA interrupts CPU to signal transfer completion

1. device driver is told to transfer disk data to buffer at address X

2. device driver tells disk controller to transfer C bytes from disk to buffer at address X



3. disk controller initiates DMA transfer

4. disk controller sends each byte to DMA controller

Direct Memory Access (Cont.)

- Takes control of the system from the CPU to transfer data to and from memory over the system bus
- Cycle stealing (周期挪用) is used to transfer data on the system bus
- The instruction cycle is suspended so data can be transferred
- The CPU pauses one bus cycle
- No interrupts occur
 - Do not save context

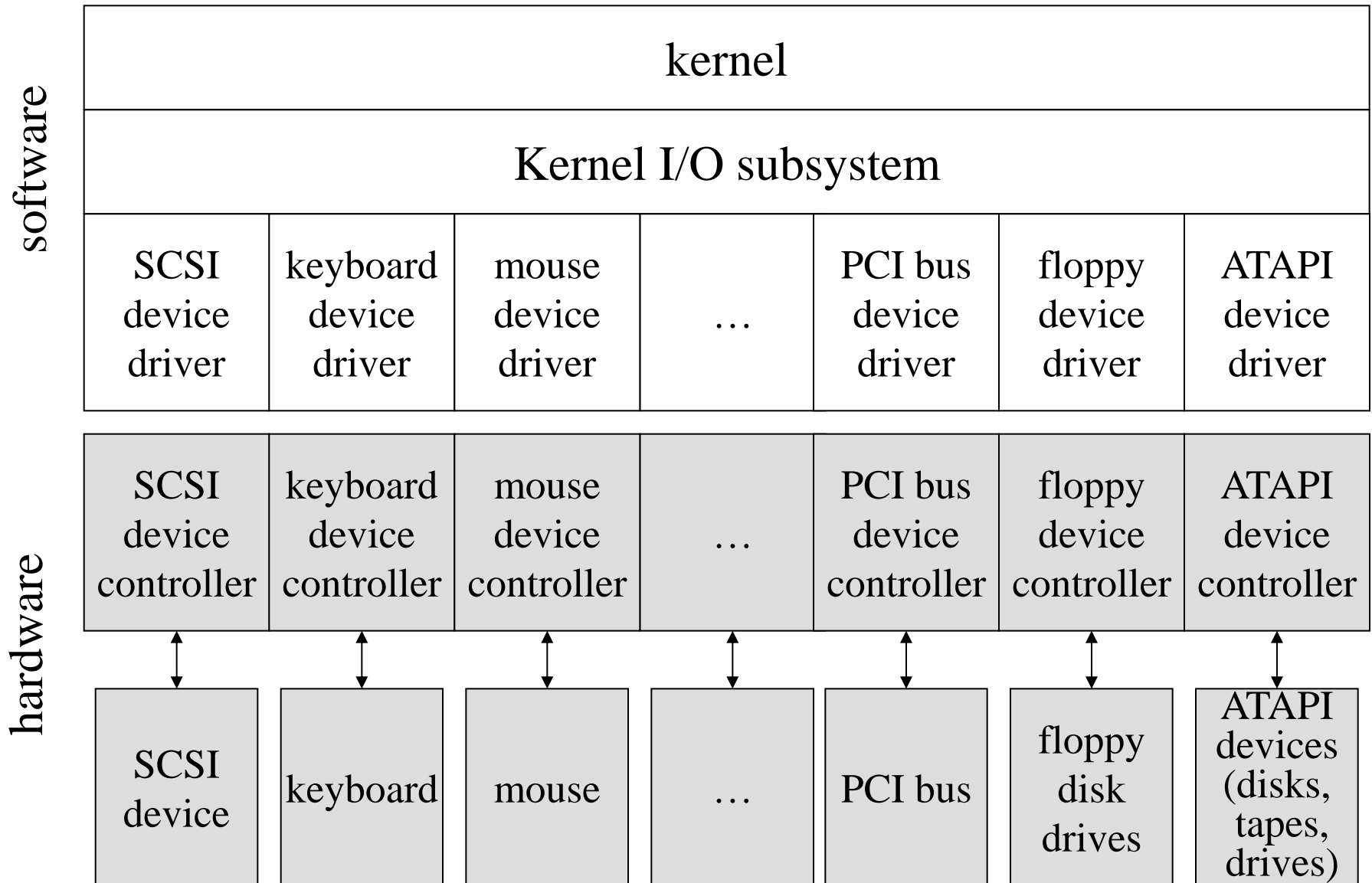


About DMA?

13.3 Application I/O Interface

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

A Kernel I/O Structure



Characteristics of I/O Devices

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

Block and Character Devices

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

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9i/2000 include **socket** interface
 - Separates network protocol from network operation
 - The system calls in the **socket interface** enable an application
 - To create a socket
 - To connect a local socket to a remote address
 - To listen for any remote application to plug into the local socket
 - To send and receive packets over the connection
 - Includes **select** functionality: manages a set of sockets
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

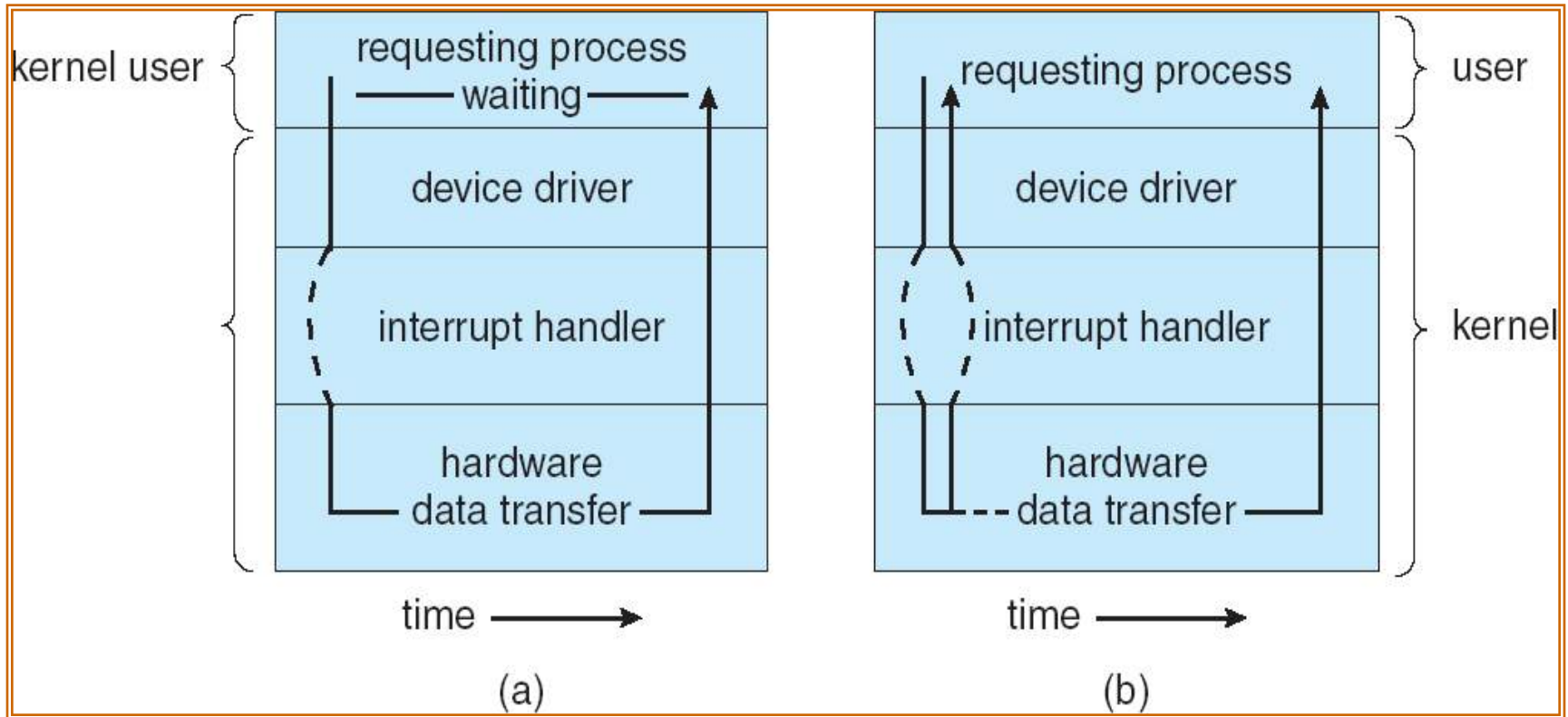
Clocks and Timers

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

Blocking and Nonblocking I/O

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

Two I/O Methods



Synchronous

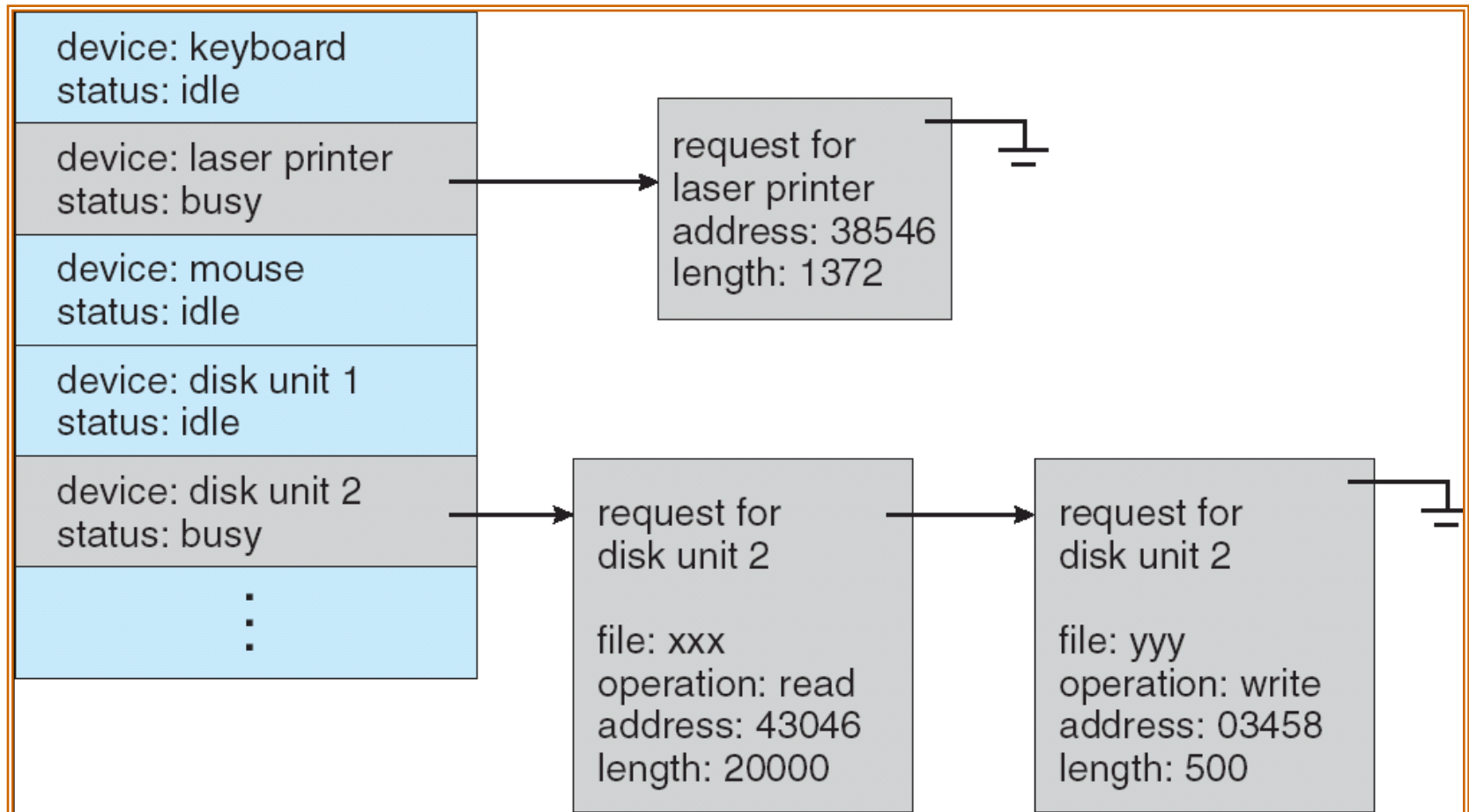
Asynchronous

13.4 Kernel I/O Subsystem

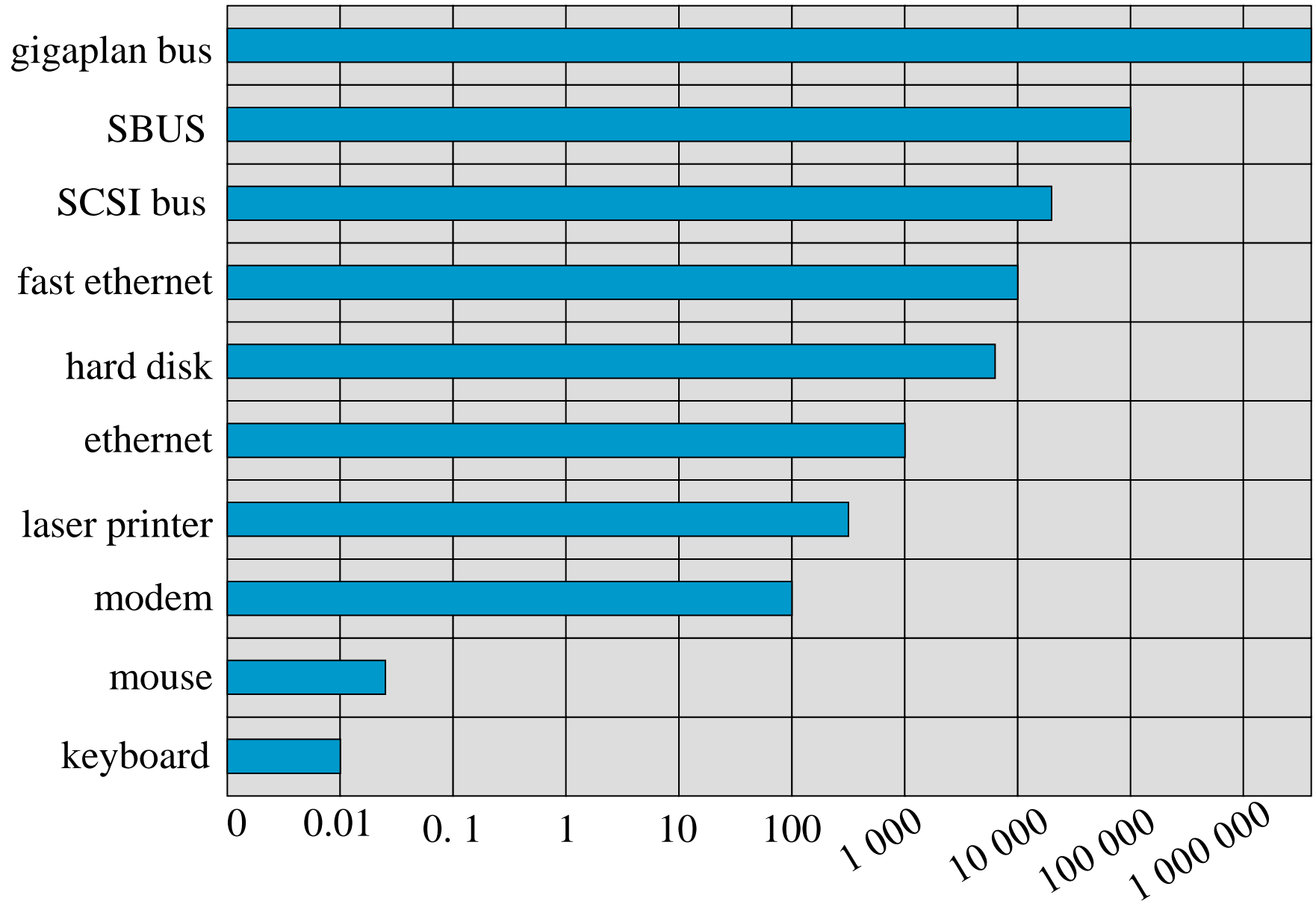
- **Services provided by kernel I/O subsystem:**
 - **Scheduling**
 - **Buffering**
 - **Caching**
 - **Spooling**
 - **device reservation**
 - **error handling**

I/O Scheduling

- Some I/O request ordering via per-device queue
- Some OSs try fairness



Sun Enterprise 6000 Device-Transfer Rates



buffering

- **Buffering - store data in memory while transferring between devices**
 - **To cope with device speed mismatch**
e.g. a file is being received via modem for storage on the hard disk.
 - **To cope with device transfer size mismatch**
e.g. fragmentation and reassembly of message.
 - **To maintain “copy semantics”**
e.g. an application write a buffer of data to disk.



I/O buffering?

caching

- **Caching - fast memory holding copy of data**
Key to performance
- **difference between a buffer and a cache**
 - A buffer may hold the only existing copy of a data item.
 - A cache just holds a copy on faster storage of an item that resides elsewhere.
- **Sometimes, a region of memory can be used for both purposes.**

Spooling

- **Spooling - hold output for a device**
 - If device can serve only one request at a time
 - i.e., Printing

Device reservation

- **Device reservation(预约) - provides exclusive access to a device**
 - System calls for allocation and deallocation
 - Watch out for deadlock

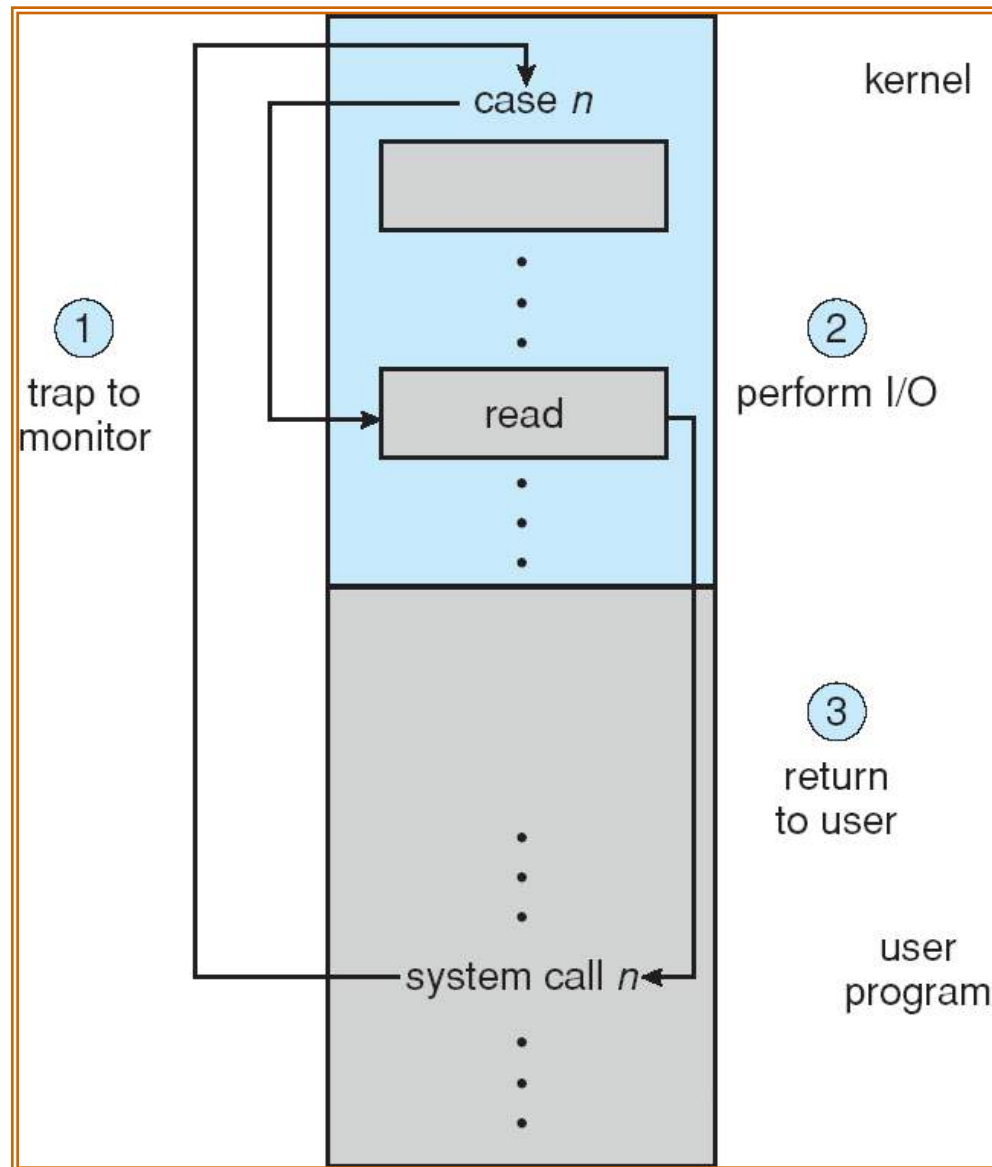
Error Handling

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

I/O Protection

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

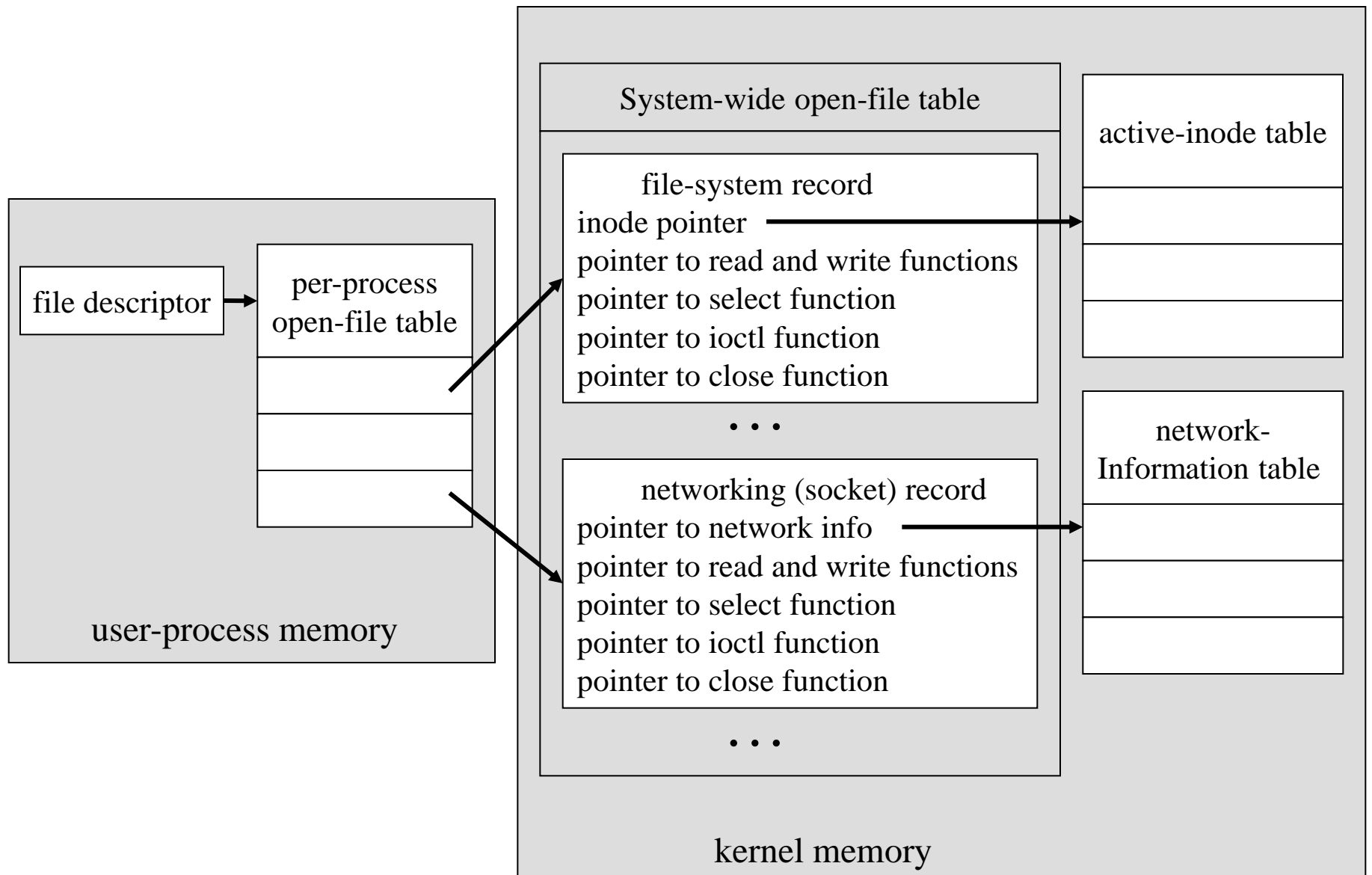
Use of a System Call to Perform I/O



Kernel Data Structures

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

UNIX I/O Kernel Structure

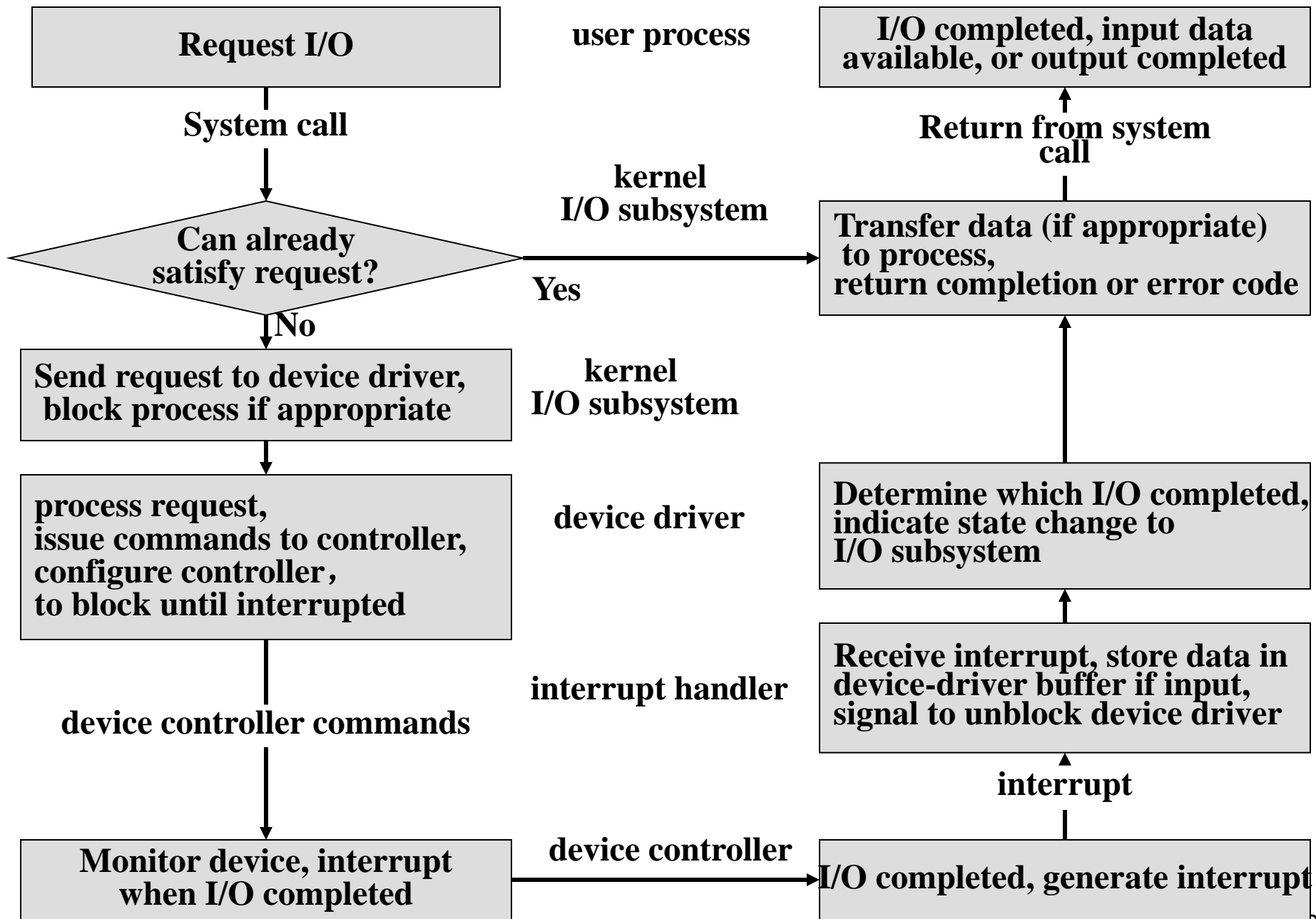


13.5 Transforming I/O Requests to Hardware Operations

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

Life Cycle of An I/O Request

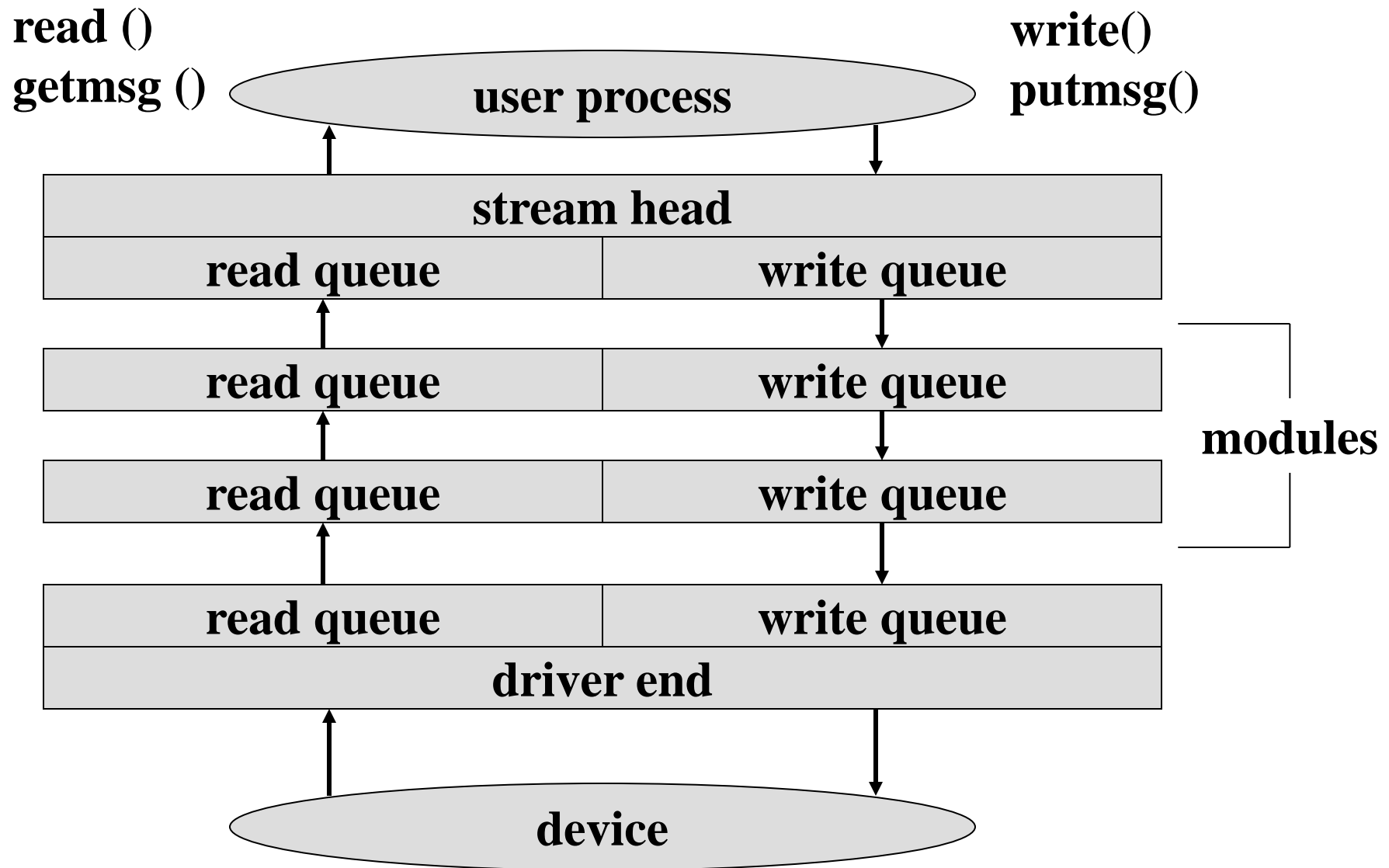
17:58



13.6 STREAMS

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

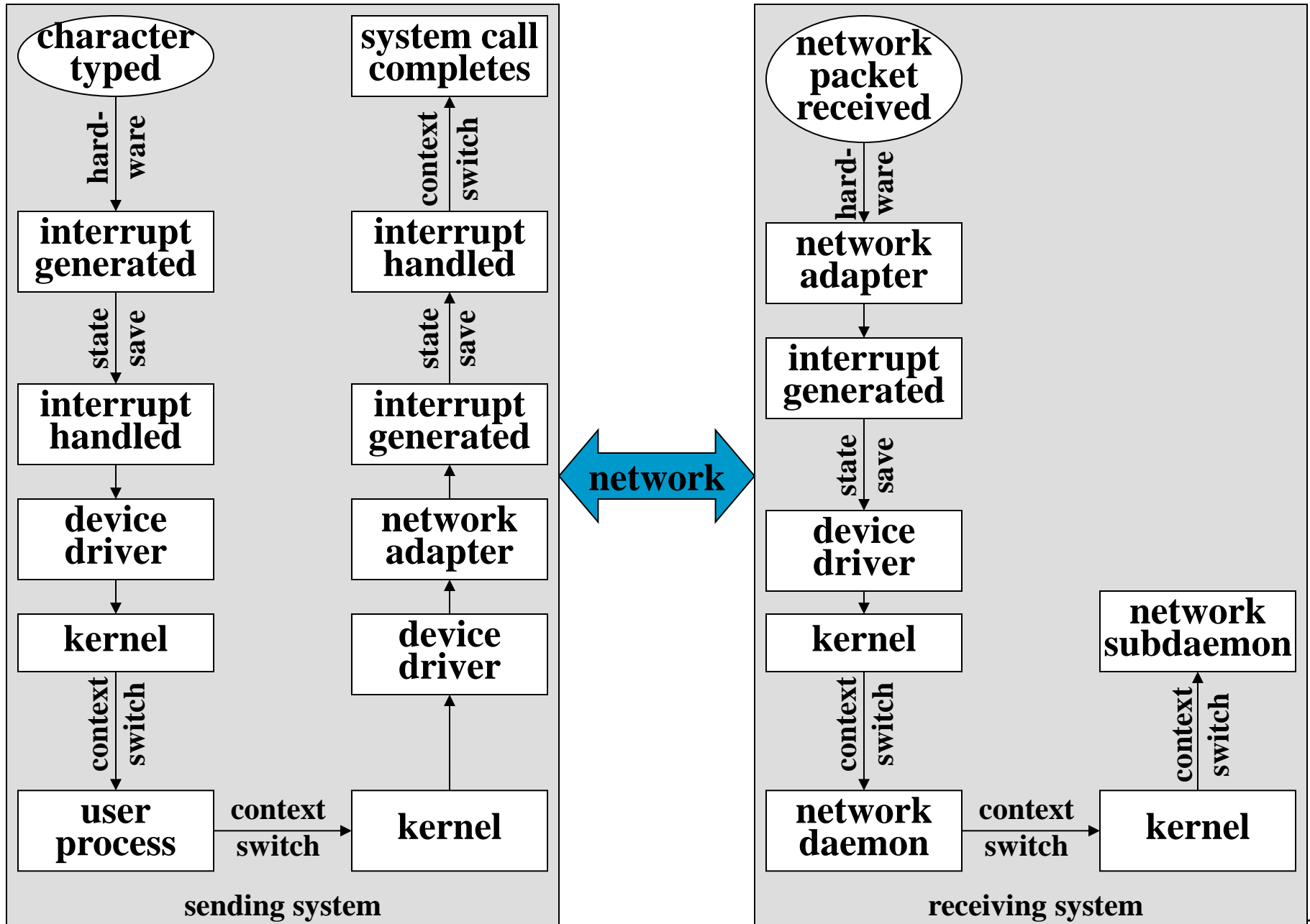
The STREAMS Structure



13.7 Performance

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

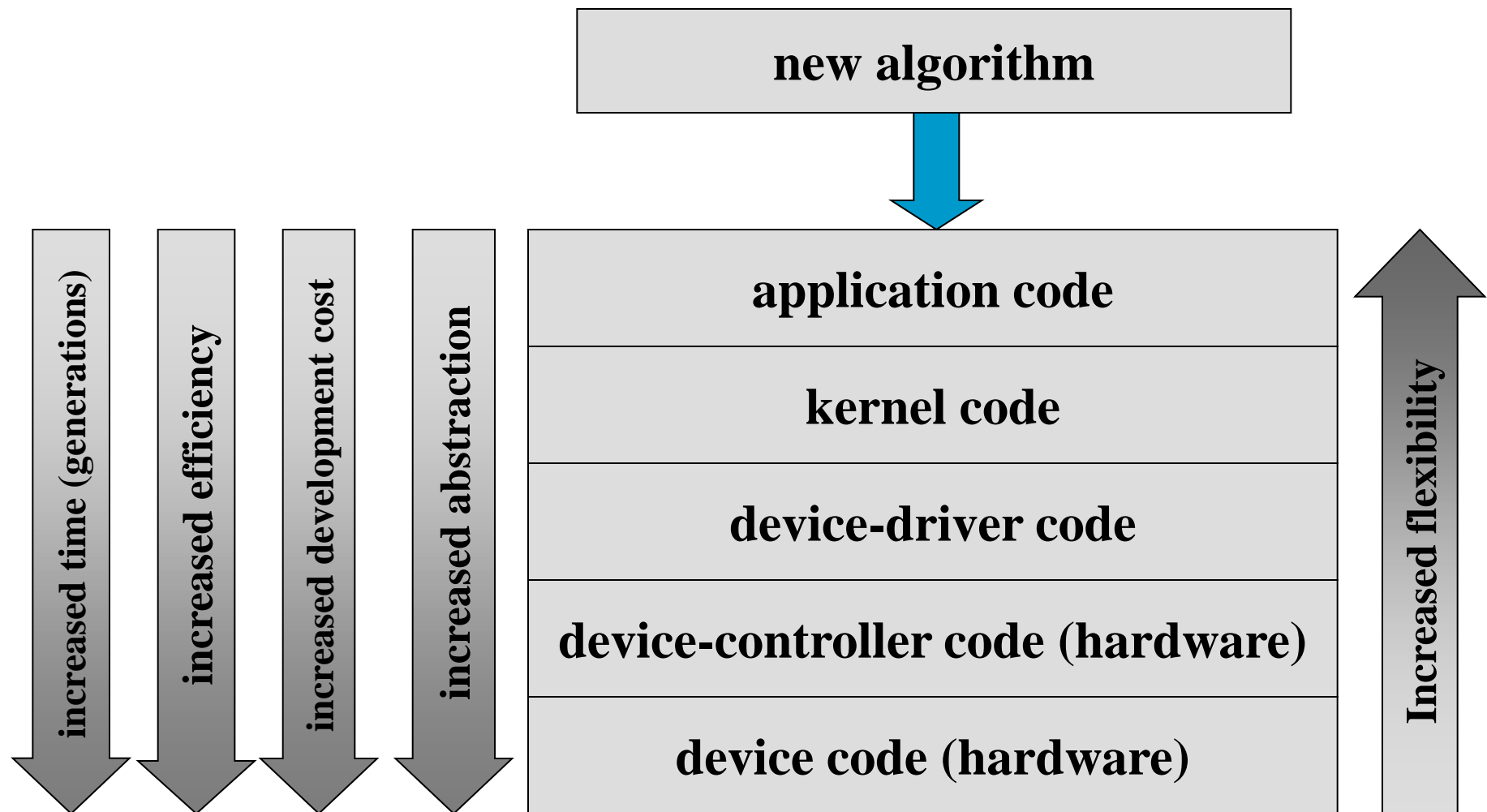
Intercomputer Communications



Improving Performance

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

Device-Functionality Progression



Homework (page 526)

- 13.6
- The example of handshaking in Section 13.2.1 used 2 bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only 1 bit? If it is, describe the protocol. If it is not, explain why 1 bit is insufficient.
- How does DMA increase system concurrency? How does it complicate hardware design?



Supplement: Categories of I/O Devices

- **Human readable:** Used to communicate with the user
 - Printers
 - Video display terminals:
Display, Keyboard, Mouse
- **Machine readable:** Used to communicate with electronic equipment
 - Disk and tap drives
 - Sensors (传感器)
 - Controllers
 - Actuators (传动器)
- **Communication:** Used to communicate with remote devices
 - Digital line drivers
 - Modems

Differences in I/O Devices

■ Data rate

- May be differences of several orders of magnitude between the data transfer rates

■ Application

- Disk used to store files requires file-management software
- Disk used to store virtual memory pages needs special hardware and software to support it
- Terminal used by system administrator may have a higher priority

■ Complexity of control

■ Unit of transfer

- Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk

■ Data representation

- Encoding schemes

■ Error conditions

- Devices respond to errors differently

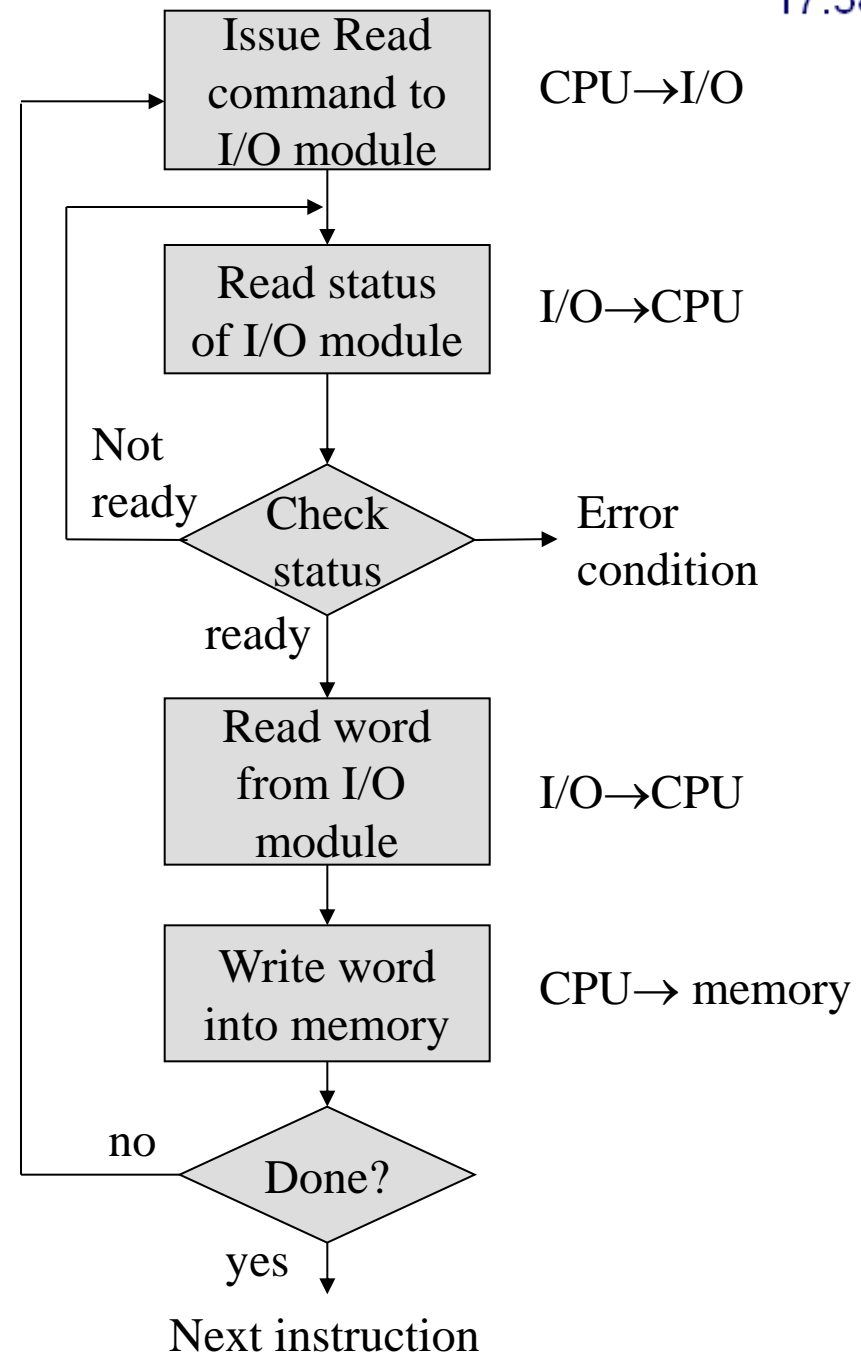
Evolution of the I/O Function

- **Processor directly controls a peripheral device**
- **Controller or I/O module is added**
 - Processor uses programmed I/O without interrupts
 - Processor does not need to handle details of external devices
- **Controller or I/O module with interrupts**
 - Processor does not spend time waiting for an I/O operation to be performed
- **Direct Memory Access**
 - Blocks of data are moved into memory without involving the processor
 - Processor involved at beginning and end only
- **I/O module is a separate processor**
- **I/O processor**
 - I/O module has its own local memory
 - Its a computer in its own right

Review:

1) Programmed I/O

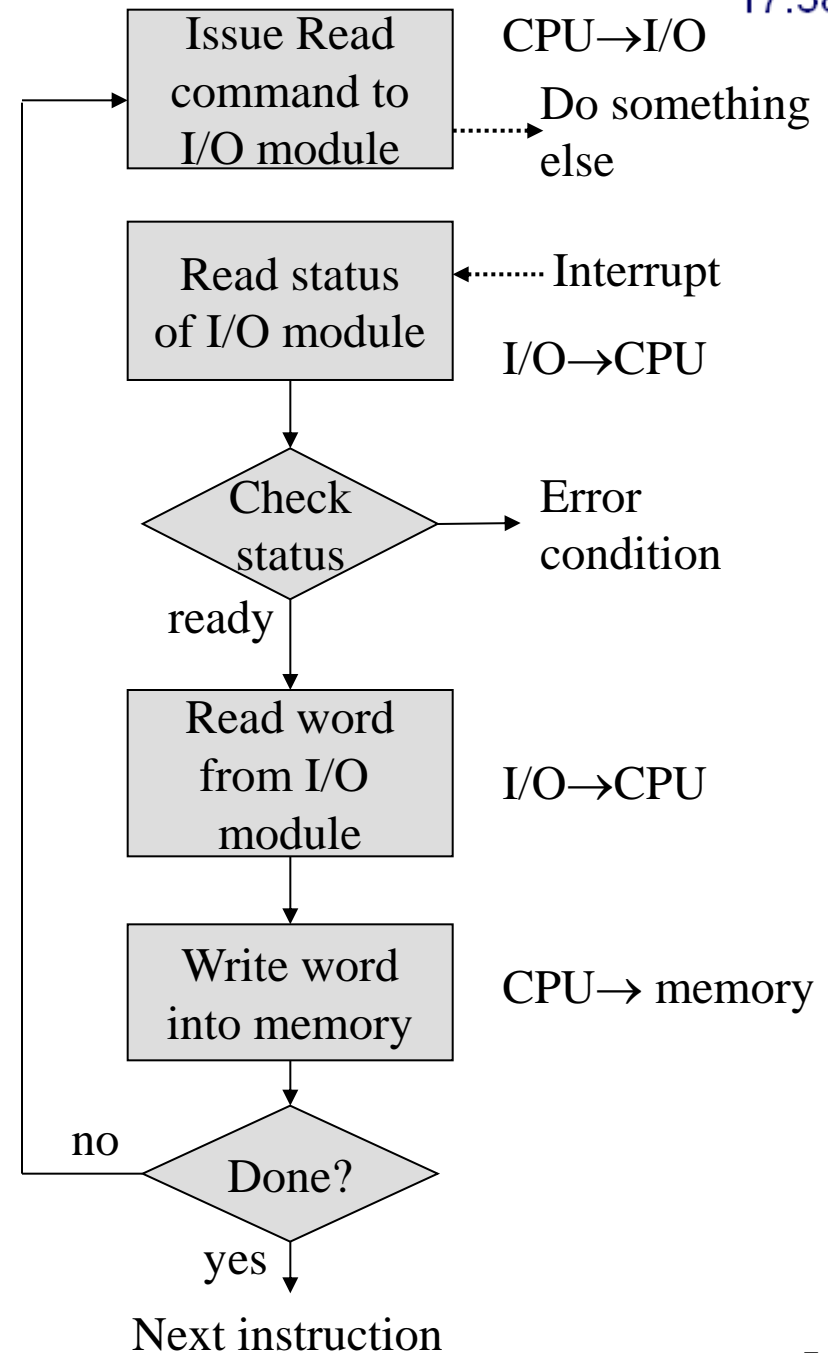
- I/O module performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete



Review:

2) Interrupt-Driven I/O

- Processor is interrupted when I/O module ready to exchange data
- Processor is free to do other work
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor



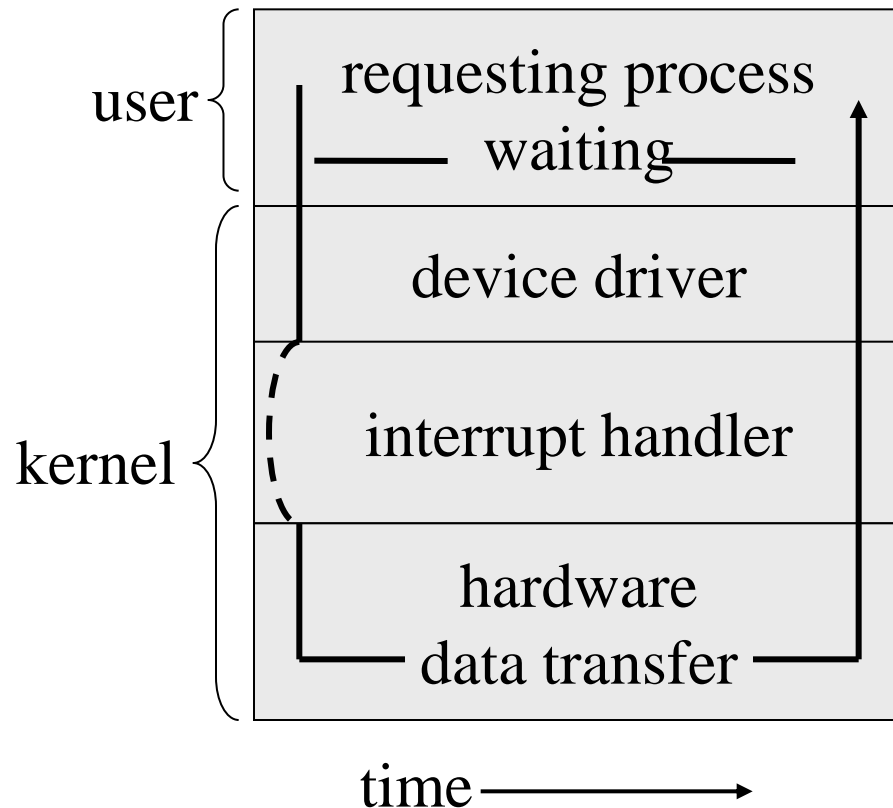
Review: Interrupt-Driven I/O (Cont.)

- **Synchronous I/O**----After I/O starts, control returns to user program only upon I/O completion.
- **Asynchronous I/O**---- After I/O starts, control returns to user program without waiting for I/O completion.
- **For synchronous I/O**, two methods can be used for CPU to wait for I/O completion:
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention (竞争) for memory access).

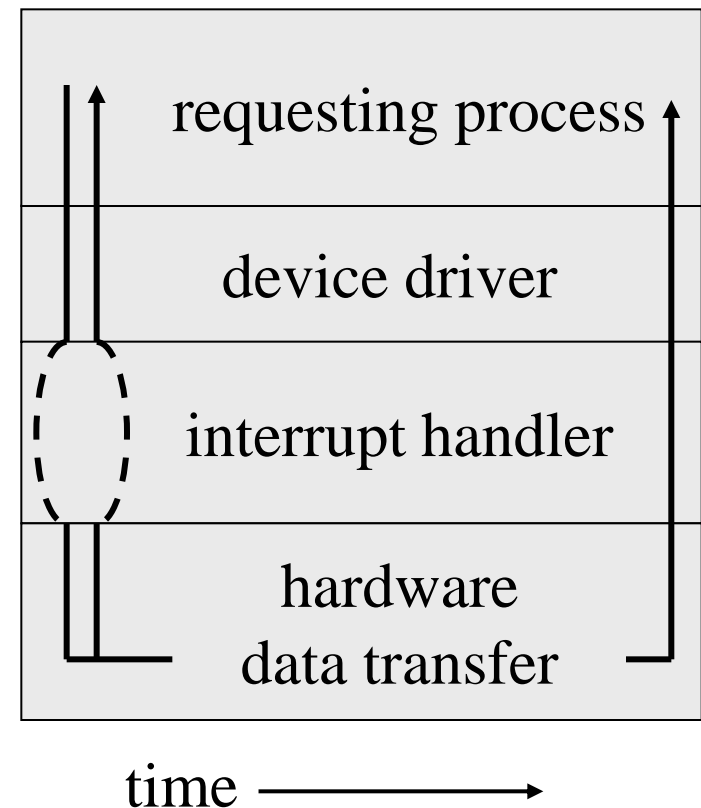
At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- In order to increase the utilization of CPU and I/O device, We need:
 - *System call* – request to the operating system to allow user to wait for I/O completion.
 - *Device-status table* contains entry for each I/O device indicating its type, address, and state.
 - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Review: Two Methods

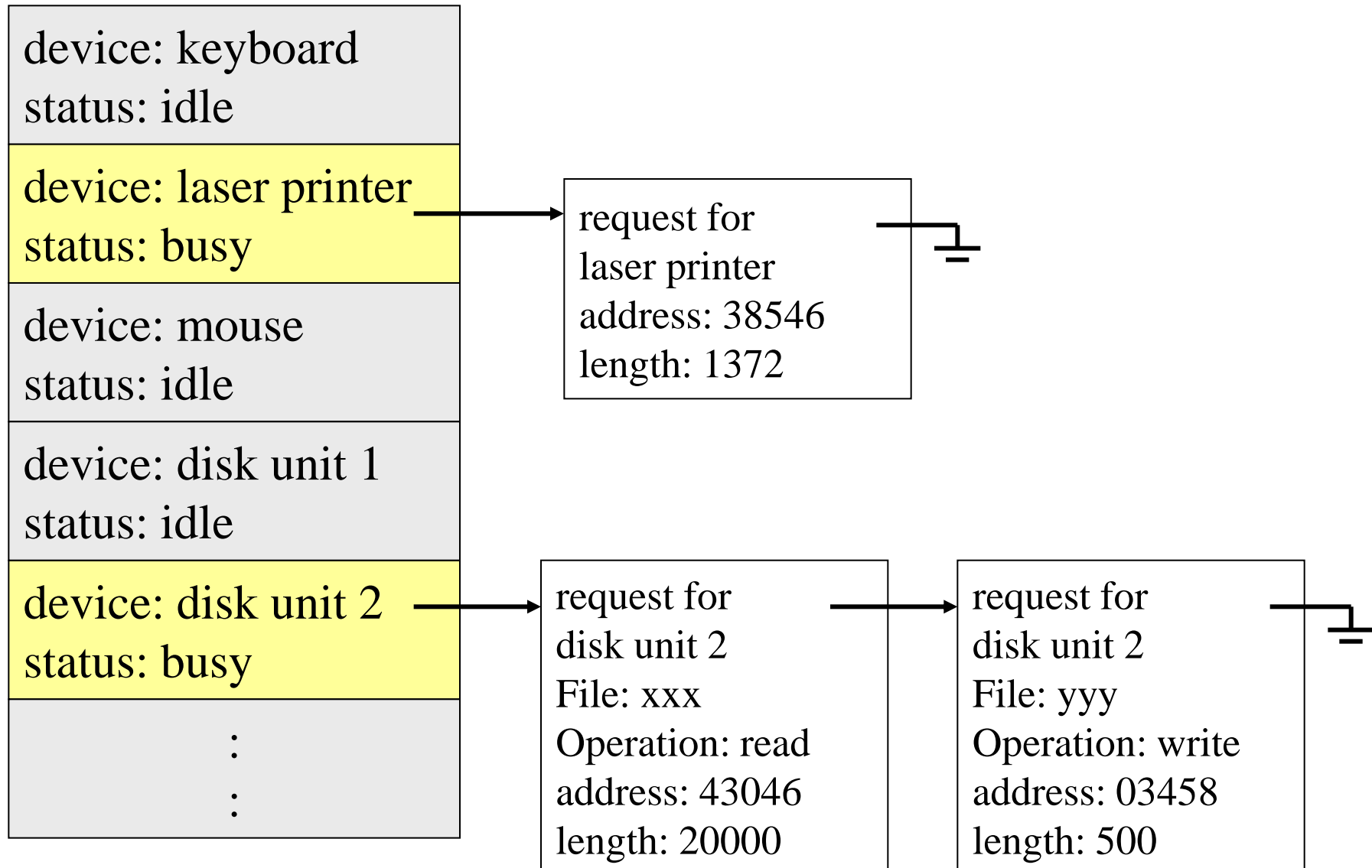
Synchronous



Asynchronous



Review: Device-Status Table

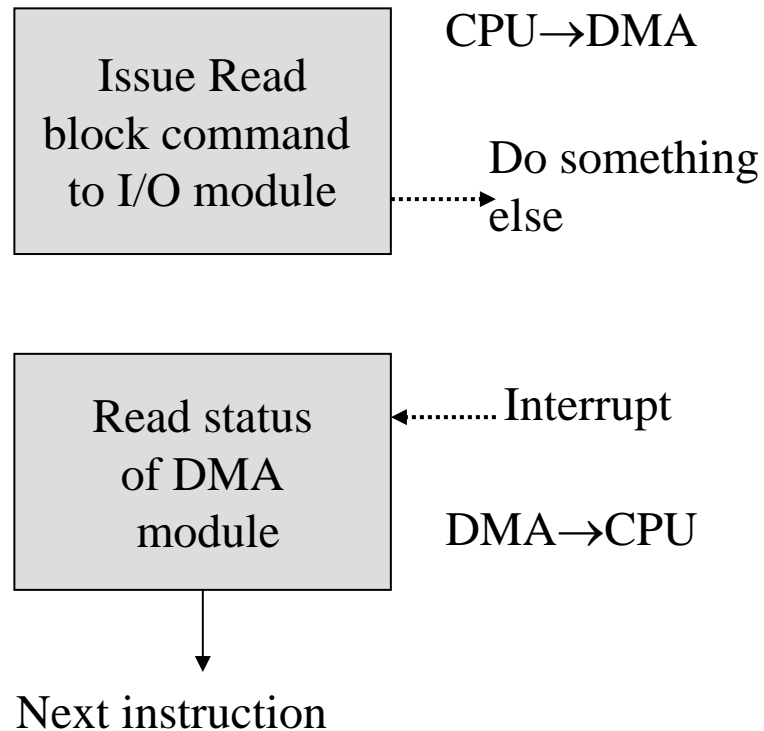


Review:

3) Direct Memory Access Structure

- **Transfers a block of data directly to or from memory.**
- **Used for high-speed I/O devices able to transmit information at close to memory speeds.**
- **After setting up buffers, pointers, and counters for the I/O devices , 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.**
 - **An interrupt is sent when the task is complete**
 - **The processor is only involved at the beginning and end of the transfer**

Review: Direct Memory Access (Cont.)



- **CPU is available to accomplish other work while the device controller is performing the DMA transfer**

Review: 4) I/O通道控制方式

- 是DMA方式的发展，进一步减少了CPU的干预。
- 完成一组数据块的读(或写)操作之后产生一次中断。
- 实现了CPU、通道和I/O设备三者的并行操作。
- CPU对通道的管理是通过I/O指令实现的。
 - I/O指令属于特权指令，仅能由操作系统使用。
 - 在I/O指令中需指定通道号、设备号、以及通道程序的内存地址。
- 当CPU要完成一组相关的读(或写)操作及有关控制时，只需向I/O通道发送一条I/O指令，以给出其所要执行的通道程序的首址和要访问的I/O设备，通道接到该指令后，通过执行通道程序便可完成CPU指定的I/O任务。

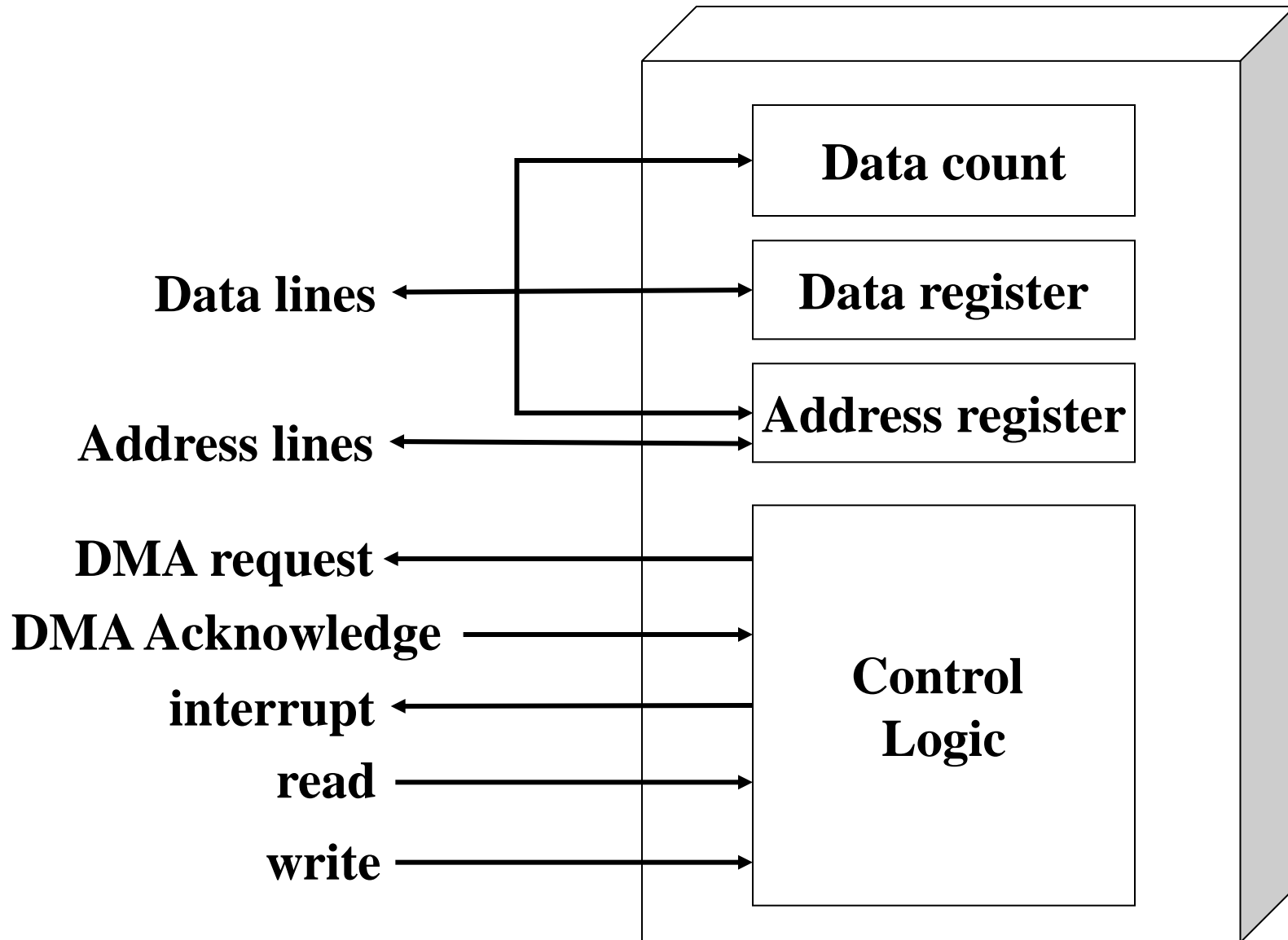
Review:通道指令

- 通道指令也称为通道命令字 (CCW, Channel Command Word)
 - 对设备进行初始化、读、写、查询、转移等
 - 使用CCW可以编写对各种不同设备进行管理和控制的通道程序
 - 一个通道可以以分时方式同时执行几个通道程序
 - 通道指令在进程要求数据时自动生成
- 指令格式: OP P R 计数 内存地址
 - OP: 读、写、或控制
 - P: 通道程序结束标志
P='1', 本条指令为最后一条指令。
 - R: 记录结束标志
R='0', 本指令与下一条指令所处理的数据属于同一条记录
R='1', 本指令是处理某记录的最后一条指令。
 - 计数: 本条指令所要处理数据的字节数
 - 内存地址:
 - 读操作: 数据要写入的内存地址
 - 写操作: 要写出的数据在内存中的起始地址

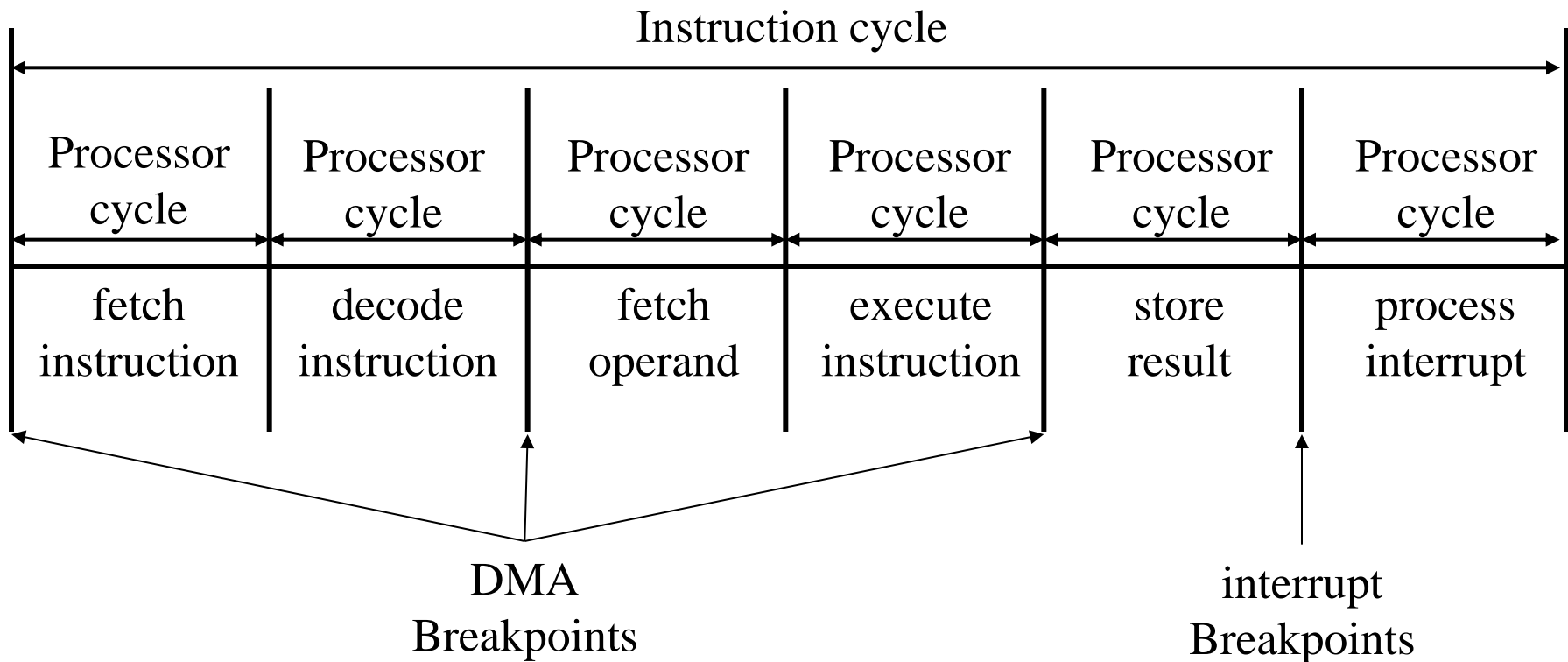
Review:通道程序举例

■	OP	P	R	计数	内存地址
■	read	0	0	250	1850
	read	1	1	250	720
■	write	0	0	80	813
	write	0	0	140	1034
	write	0	1	60	5830
	write	0	1	300	2000
	write	0	0	200	1650
	write	1	1	300	2720

Supplement: Typical DMA Block Diagram



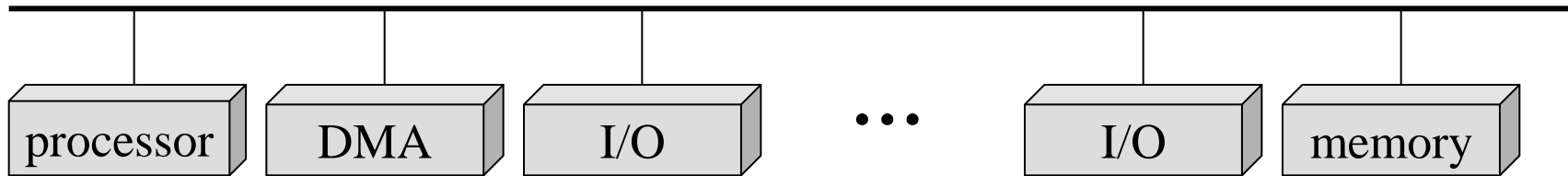
Supplement: instruction cycle



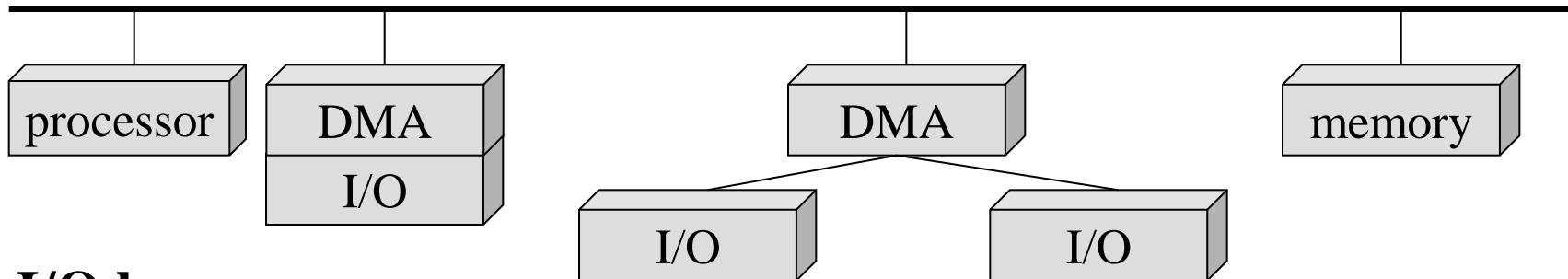
- Cycle stealing causes the CPU to execute more slowly

Supplement: DMA Scheme

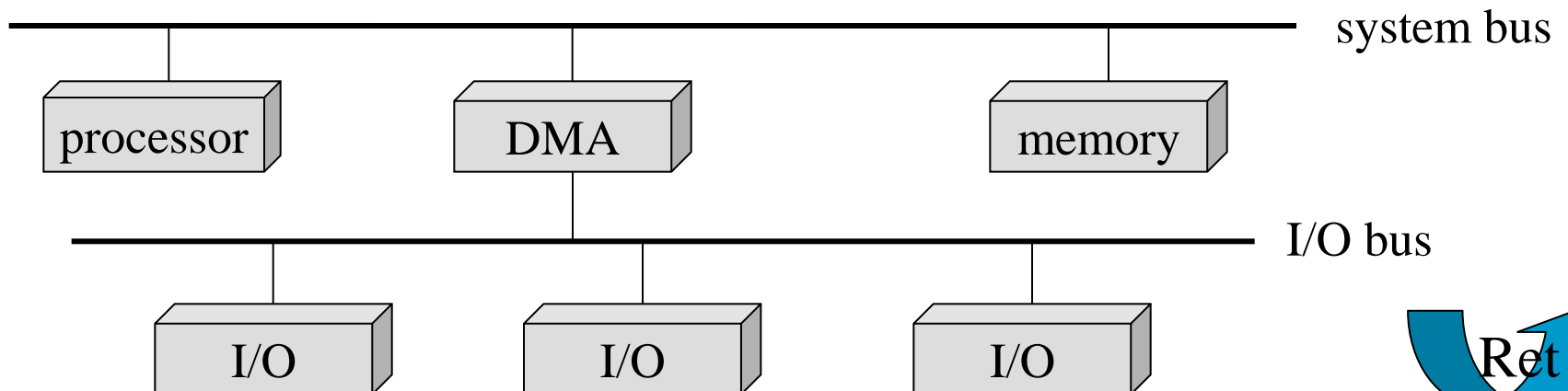
■ Single bus, detached DMA



■ Single bus, integrated DMA-I/O



■ I/O bus



Susplement: I/O Buffering

■ **Block-oriented**

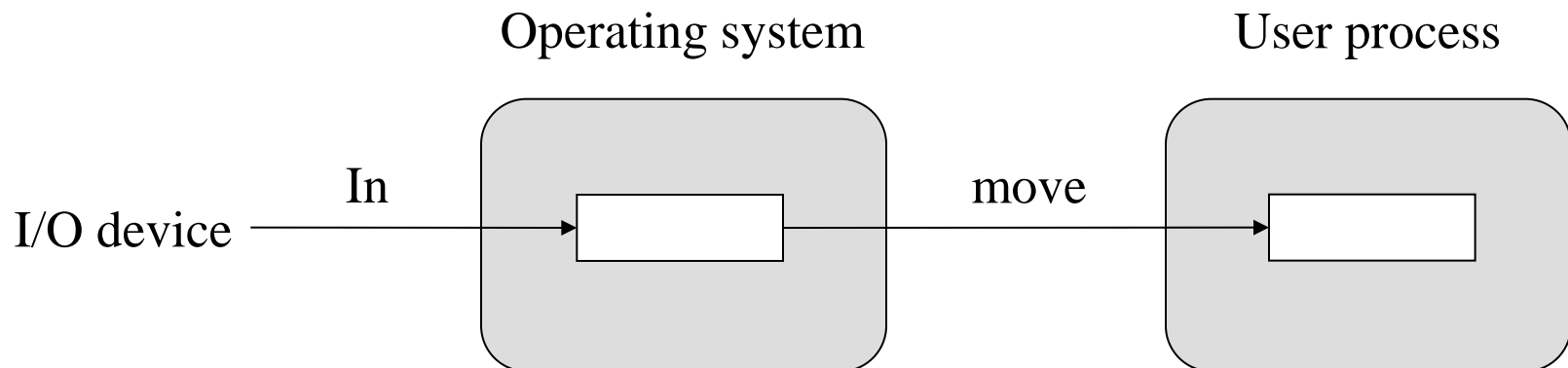
- **Information is stored in fixed sized blocks**
- **Transfers are made a block at a time**
- **Used for disks and tapes**

■ **Stream-oriented**

- **Transfer information as a stream of bytes**
- **Used for terminals, printers, communication ports, mouse, and most other devices that are not secondary storage**

Susplement: Single Buffer

- Operating system assigns a buffer in main memory for an I/O request
- Block-oriented
 - Input transfers made to buffer
 - Block moved to user space when needed
 - Another block is moved into the buffer
 - Read ahead



Susplement: Single Buffer (Cont.)

■ **Block-oriented**

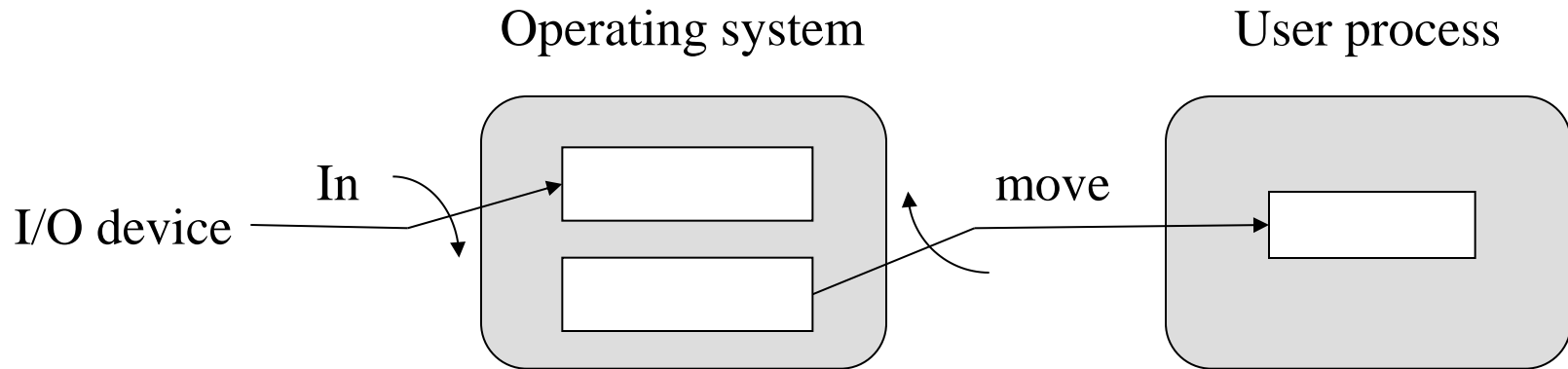
- User process can process one block of data while next block is read in
- Swapping can occur since input is taking place in system memory, not user memory
- Operating system keeps track of assignment of system buffers to user processes

■ **Stream-oriented**

- Used a line at time
- User input from a terminal is one line at a time with carriage return signaling the end of the line
- Output to the terminal is one line at a time

Susplement: Double Buffer

- Use two system buffers instead of one
- A process can transfer data to or from one buffer while the operating system empties or fills the other buffer



Susplement: Circular Buffer

- More than two buffers are used
- Each individual buffer is one unit in a circular buffer
- Used when I/O operation must keep up with process

