PREFACE

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer-science education. This field is undergoing change at a breathtakingly rapid rate, as computers are now prevalent in virtually every application, from games for children through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain fairly clear, and it is on these that we base this book.

We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level or at the first-year graduate level. It provides a clear description of the *concepts* that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C. The hardware topics required for an understanding of operating systems are included in Chapter 2. For code examples, we use predominantly C as well as some Java, but the reader can still understand the algorithms without a thorough knowledge of these languages.

The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. We present a large number of examples that pertain to the most popular operating systems, including Sun Microsystems' Solaris 2, Linux; Microsoft MS-DOS, Windows NT, and Windows 2000; DEC VMS and TOPS-20, IBM OS/2, and the Apple Macintosh Operating System.

Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are omitted. The bibliographical notes contain pointers to research papers in which results were first presented and proved, as well as references to material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

Content of this Book

The text is organized in seven major parts:

- **Overview:** Chapters 1 through 3 explain what operating systems *are,* what they *do,* and how they are *designed* and *constructed.* They explain how the concept of an operating system has developed, what the common features of an operating system are, what an operating system does for the user, and what it does for the computer-system operator. The presentation is motivational, historical, and explanatory in nature. We have avoided a discussion of how things are done internally in these chapters. Therefore, they are suitable for individuals or for students in lower-level classes who want to learn what an operating system is, without getting into the details of the internal algorithms. Chapter 2 covers the hardware topics that are important to an understanding of operating systems. Readers well-versed in hardware topics, including I/O, DMA, and hard-disk operation, may choose to skim or skip this chapter.

- **Process management:** Chapters 4 through 8 describe the process concept and concurrency as the heart of modern operating systems. A *process* is the unit of work in a system. Such a system consists of a collection of *concurrently* executing processes, some of which are operating-system processes (those that execute system code), and the rest of which are user processes (those that execute user code). These chapters cover methods for process scheduling, interprocess communication, process synchronization, and deadlock handling. Also included under this topic is a discussion of threads.

- **Storage management:** Chapters 9 through 12 deal with a process in main memory during execution. To improve both the utilization of CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes. These schemes reflect various approaches to memory management, and the effectiveness of the different algorithms depends on the situation. Since main memory is usually too small to accommodate all data and programs, and since it cannot store data permanently, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the primary on-line storage medium for information,

both programs and data. The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. These chapters describe the classic internal algorithms and structures of storage management. They provide a firm practical understanding of the algorithms used—the properties, advantages, and disadvantages.

- **I/O systems:** Chapters 13 and 14 describe the devices that attach to a computer and the multiple dimensions in which they vary. In many ways, they are also the slowest major components of the computer. Because devices differ so widely, the operating system needs to provide a wide range of functionality to applications to allow them to control all aspects of the devices. This section discusses system I/O in depth, including I/O system design, interfaces, and internal system structures and functions. Because devices are a performance bottleneck, performance issues are examined. Matters related to secondary and tertiary storage are explained as well.

- **Distributed systems:** Chapters 15 through 17 deal with a collection of processors that do not share memory or a clock—a *distributed system*. Such a system provides the user with access to the various resources that the system maintains. Access to a shared resource allows computation speedup and improved data availability and reliability. Such a system also provides the user with a distributed file system, which is a file-service system whose users, servers, and storage devices are dispersed among the sites of a distributed system. A distributed system must provide various mechanisms for process synchronization and communication, for dealing with the deadlock problem and the variety of failures that are not encountered in a centralized system.

- **Protection and security:** Chapters 18 and 19 explain the processes in an operating system that must be protected from one another's activities. For the purposes of protection and security, we use mechanisms that ensure that only those processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources. Protection is a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specification of the controls to be imposed, as well as a means of enforcement. Security protects the information stored in the system (both data and code), as well as the physical resources of the computer system, from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.

- **Case studies:** Chapters 20 through 22, in the book, and Appendices A through C, on the website, integrate the concepts described in this book by describing real operating systems. These systems include Linux, Windows 2000, FreeBSD, Mach, and Nachos. We chose Linux and FreeBSD because

UNIX—at one time—was almost small enough to understand, yet was not a "toy" operating system. Most of its internal algorithms were selected for *simplicity,* rather than for speed or sophistication. Both Linux and FreeBSD are readily available to computer-science departments, so many students have access to these systems. We chose Windows 2000 because it provides an opportunity for us to study a modern operating system that has a design and implementation drastically different from those of UNIX. We also cover the Nachos System, which allows students to get their hands *dirty*—to take apart the code for an operating system, to see how it works at a low level, to build significant pieces of the operating system themselves, and to observe the effects of their work. Chapter 22 briefly describes a few other influential operating systems.

The Sixth Edition

As we wrote this Sixth Edition, we were guided by the many comments and suggestions we received from readers of our previous editions, as well as by our own observations about the rapidly changing fields of operating systems and networking. We rewrote the material in most of the chapters by bringing older material up to date and removing material that was no longer of interest. We rewrote all Pascal code, used in previous editions to demonstrate certain algorithms, into C, and we included a small amount of Java as well.

We made substantive revisions and changes in organization in many of the chapters. Most importantly, we added two new chapters and reorganized the distributed systems coverage. Because networking and distributed systems have become more prevalent in operating systems, we moved some distributed systems material, client–server, in particular, out of distributed systems chapters and integrated it into earlier chapters.

- **Chapter 3, Operating-System Structures,** now includes a section discussing the Java virtual machine (JVM).

- **Chapter 4, Processes,** includes new sections describing sockets and remote procedure calls (RPCs).

- **Chapter 5, Threads,** is a new chapter that covers multithreaded computer systems. Many modern operating systems now provide features for a process to contain multiple threads of control.

- **Chapters 6 through 10** are the old Chapters 5 through 9, respectively.

- **Chapter 11, File-System Interface,** is the old Chapter 10. We have modified the chapter substantially, including the coverage of NFS from the Distributed File System chapter (Chapter 16).

- **Chapter 12 and 13** are the old Chapters 11 and 12, respectively. We have added a new section in Chapter 13, I/O Systems, covering STREAMS.

- **Chapter 14, Mass-Storage Structure,** combines old Chapters 13 and 14.

- **Chapter 15, Distributed System Structures,** combines old Chapters 15 and 16.

- **Chapter 19, Security,** is the old Chapter 20.

- **Chapter 20, The Linux System,** is the old Chapter 22, updated to cover new recent developments.

- **Chapter 21, Windows 2000,** is a new chapter.

- **Chapter 22, Historical Perspective,** is the old Chapter 24.

- **Appendix A** is the old Chapter 21 on UNIX updated to cover FreeBSD.

- **Appendix B** covers the Mach operating system.

- **Appendix C** covers the Nachos system.

The three appendices are provided online.

Teaching Supplements and Web Page

The web page for this book contains the three appendices, the set of slides that accompanies the book, in PDF and Powerpoint format, the three case studies, the most recent errata list, and a link to the authors home page. John Wiley & Sons maintains the web page at

> http://www.wiley.com/college/silberschatz/osc

To obtain restricted supplements, contact your local John Wiley & Sons sales representative. You can find your representative at the "Find a Rep?" web page: http://www.jsw-edcv.wiley.com/college/findarep

Mailing List

We provide an environment in which users can communicate among themselves and with us. We have created a mailing list consisting of users of our book with the following address: os-book@research.bell-labs.com. If you wish to be on the list, please send a message to avi@bell-labs.com indicating your name, affiliation, and e-mail address.

Suggestions

We have attempted to clean up every error in this new Edition, but—as happens with operating systems—a few obscure bugs may remain. We would appreciate hearing from you about any textual errors or omissions that you identify. If you would like to suggest improvements or to contribute exercises, we would also be glad to hear from you. Please send correspondence to Avi Silberschatz, Vice President, Information Sciences Research Center, MH 2T-310, Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974 (avi@bell-labs.com).

Acknowledgments

This book is derived from the previous editions, the first three of which were coauthored by James Peterson. Others who helped us with previous editions include Hamid Arabnia, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, P. C. Capon, John Carpenter, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Raşit Eskicioğlu, Hans Flack, Robert Fowler, G. Scott Graham, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Mark Holliday, Richard Kieburtz, Carol Kroll, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Yoichi Muraoka, Jim M. Ng, Banu Özden, Ed Posnak, Boris Putanec, Charles Qualline, John Quarterman, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, John Werth, and J. S. Weston.

We thank the following people who contributed to this edition of the book: Bruce Hillyer reviewed and helped with the rewrite of Chapters 2, 12, 13, and 14. Mike Reiter reviewed and helped with the rewrite of Chapter 18. Parts of Chapter 14 were derived from a paper by Hillyer and Silberschatz [1996]. Parts of Chapter 17 were derived from a paper by Levy and Silberschatz [1990]. Chapter 20 was derived from an unpublished manuscript by Stephen Tweedie. Chapter 21 was derived from an unpublished manuscript by Cliff Martin. Cliff Martin helped with updating the UNIX appendix to cover FreeBSD. Mike Shapiro reviewed the Solaris information and Jim Mauro answered several Solaris-related questions.

We thank the following people who reviewed this edition of the book: Rida Bazzi, Arizona State University; Roy Campbell, University of Illinois-Chicago; Gil Carrick, University of Texas at Arlington; Richard Guy, UCLA; Max Hailperin, Gustavus Adolphus College; Ahmed Kamel, North Dakota State University; Morty Kwestel, New Jersey Institute of Technology; Gustavo Rodriguez-Rivera, Purdue University; Carolyn J. C. Schauble, Colorado State University; Thomas P. Skinner, Boston University; Yannis Smaragdakis, Geor-

gia Tech; Larry L. Wear, California State University, Chico; James M. Westall, Clemson University; and Yang Xiang, University of Massassachusetts.

Our Acquisitions Editors, Bill Zobrist and Paul Crockett, provided expert guidance as we prepared this Edition. They were both assisted by Susannah Barr, who managed the many details of this project smoothly. Katherine Hepburn was our Marketing Manager. The Senior Production Editor was Ken Santor. The cover illustrator was Susan Cyr while the cover designer was Madelyn Lesure. Barbara Heaney was in charge of overseeing the copy-editing and Katie Habib copyedited the manuscript. The freelance proofreader was Katrina Avery; the freelance indexer was Rosemary Simpson. The Senior Illustration Coordinator was Anna Melhorn. Marilyn Turnamian helped generate figures and update the text, Instructors Manual, and slides.

Finally, we would like to add some personal notes. Avi would like to extend his gratitude to Krystyna Kwiecien, whose devoted care of his mother has given him the peace of mind he needed to focus on the writing of this book; Pete, would like to thank Harry Kasparian, and his other co-workers, who gave him the freedom to work on this project while doing his "real job"; Greg would like to acknowledge two significant achievements by his children during the period he worked on this text: Tom—age 5—learned to read, and Jay—age 2—learned to talk.

Abraham Silberschatz, Murray Hill, NJ, 2001
Peter Baer Galvin, Norton, MA, 2001
Greg Gagne, Salt Lake City, UT, 2001

CONTENTS

PART ONE ■ OVERVIEW

Chapter 1   Introduction

Chapter 2   Computer-System Structures

PART TWO   ■   PROCESS MANAGEMENT

Chapter 7   Process Synchronization

Chapter 8   Deadlocks

PART THREE   ■   STORAGE MANAGEMENT

Chapter 9   Memory Management

Chapter 10   Virtual Memory

Chapter 11   File-System Interface

Chapter 12   File-System Implementation

PART FOUR   ■   I/O SYSTEMS

Chapter 13   I/O Systems

Chapter 14   Mass-Storage Structure

PART FIVE ■ DISTRIBUTED SYSTEMS

Chapter 15   Distributed System Structures

Chapter 16   Distributed File Systems

Chapter 17   Distributed Coordination

PART SIX ■ PROTECTION AND SECURITY

Chapter 18   Protection

Chapter 19   Security

PART SEVEN   ■   CASE STUDIES

Chapter 20   The Linux System

Chapter 21   Windows 2000

Chapter 22   Historical Perspective

Appendix A   The FreeBSD System (contents online)

Appendix B   The Mach System (contents online)

Appendix C   The Nachos System (contents online)
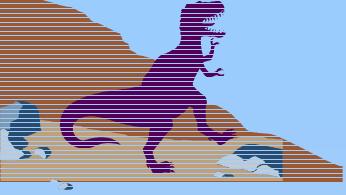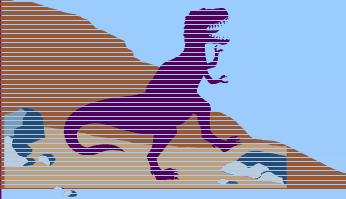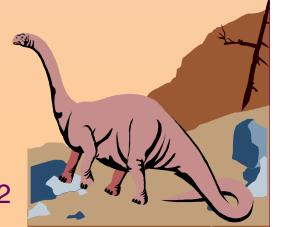
Chapter 1:   Introduction
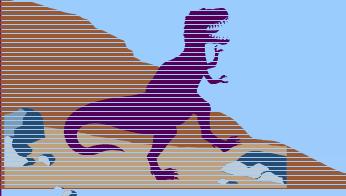
- What is an Operating System?
- Mainframe Systems
- Desktop Systems
- Multiprocessor Systems
- Distributed Systems
- Clustered System
- Real -Time Systems
- Handheld Systems
- Computing Environments

What is an Operating System?
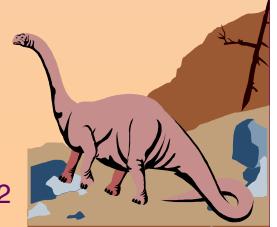
- A program that acts as an intermediary between a user of a computer and the computer hardware.

- Operating system goals:
  - Execute user programs and make solving user problems easier.
  - Make the computer system convenient to use.

- Use the computer hardware in an efficient manner.
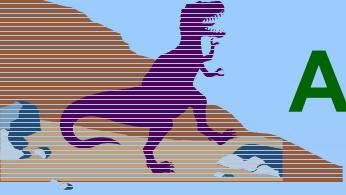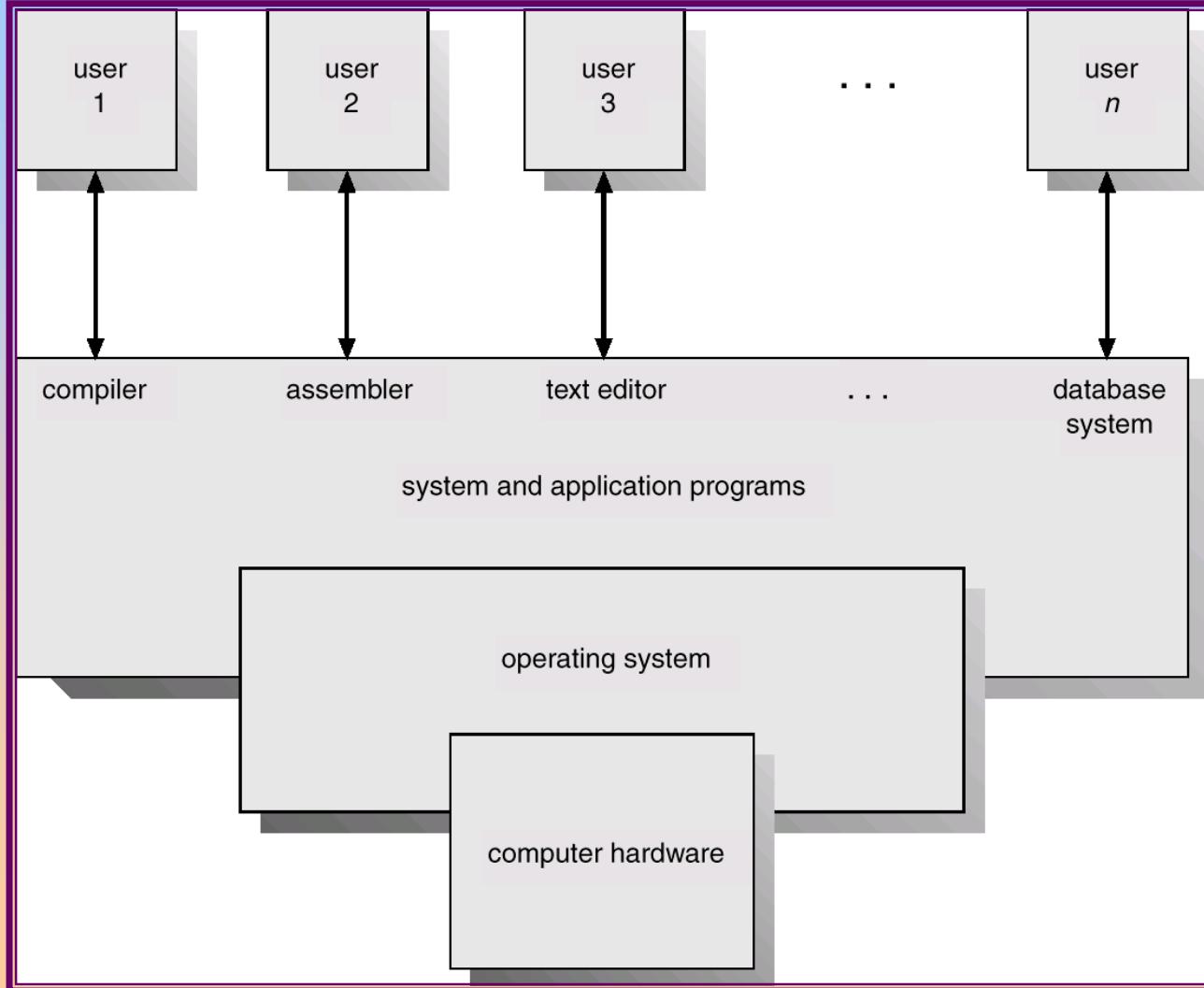
Computer System Components
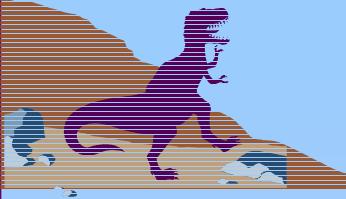
1. Hardware – provides basic computing resources (CPU, memory, I/O devices).

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 the computing problems of the users (compilers, database systems, video games, business programs).

4. Users (people, machines, other computers).
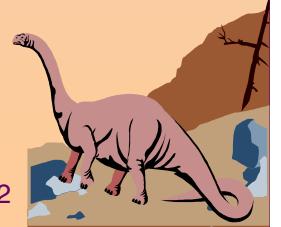
Abstract View of System Components
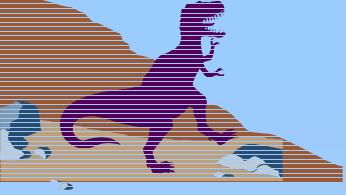
Operating System Definitions
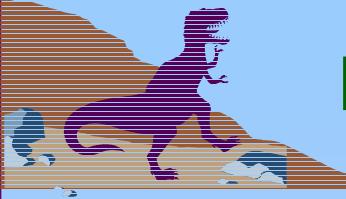
- Resource allocator – manages and allocates resources.

- Control program – controls the execution of user programs and operations of I/O devices .

- Kernel – the one program running at all times (all else being application programs).
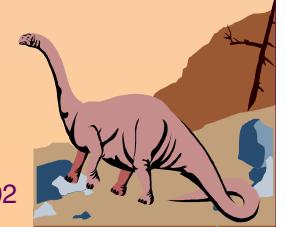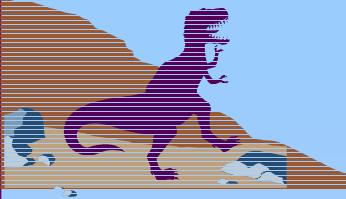
Mainframe Systems

- Reduce setup time by batching similar jobs

- Automatic job sequencing – automatically transfers control from one job to another. First rudimentary operating system.

- Resident monitor
  - initial control in monitor
  - control transfers to job
  - when job completes control transfers pack to monitor
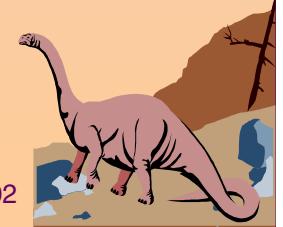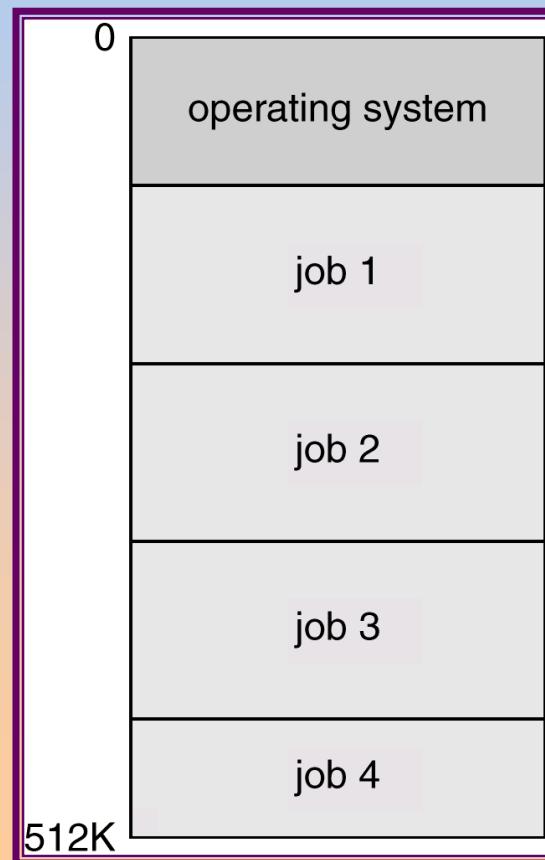
Memory Layout for a Simple Batch System

operating
system

user program
area

Multiprogrammed Batch Systems

Several jobs are kept in main memory at the same time, and the CPU is multiplexed among them.

```
0
┌─────────────────────┐
│  operating system   │
├─────────────────────┤
│                     │
│        job 1        │
│                     │
├─────────────────────┤
│                     │
│        job 2        │
│                     │
├─────────────────────┤
│                     │
│        job 3        │
│                     │
├─────────────────────┤
│                     │
│        job 4        │
│                     │
512K─────────────────────┘
```
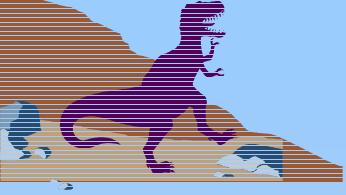
OS Features Needed for Multiprogramming

- I/O routine 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.
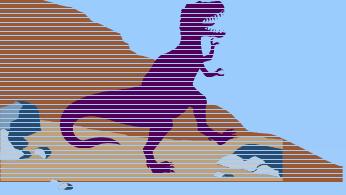
Time-Sharing Systems–Interactive Computing

- The CPU is multiplexed among several jobs that are kept in memory and on disk (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.

- On-line communication between the user and the system is provided; when the operating system finishes the execution of one command, it seeks the next "control statement" from the user's keyboard.

- On-line system must be available for users to access data and code.
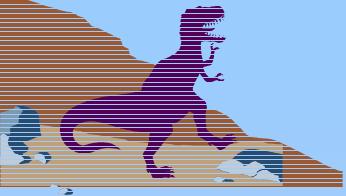
Desktop Systems

- *Personal computers* – computer system dedicated to a single user.

- I/O devices – keyboards, mice, display screens, small printers.

- User convenience and responsiveness.

- Can adopt technology developed for larger operating system' often individuals have sole use of computer and do not need advanced CPU utilization of protection features.

- May run several different types of operating systems (Windows, MacOS, UNIX, Linux)

Parallel Systems

- Multiprocessor systems with more than on CPU in close communication.

- *Tightly coupled system* – processors share memory and a clock; communication usually takes place through the shared memory.

- Advantages of parallel system:
  - Increased *throughput*
  - Economical
  - Increased reliability
    - ✔ graceful degradation
    - ✔ fail-soft systems
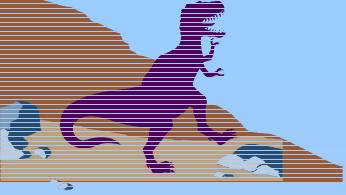
Parallel Systems (Cont.)

- *Symmetric multiprocessing (SMP)*
    - Each processor runs and identical copy of the operating system.
    - Many processes can run at once without performance deterioration.
    - Most modern operating systems support SMP
- *Asymmetric multiprocessing*
    - Each processor is assigned a specific task; master processor schedules and allocated work to slave processors.
    - More common in extremely large systems
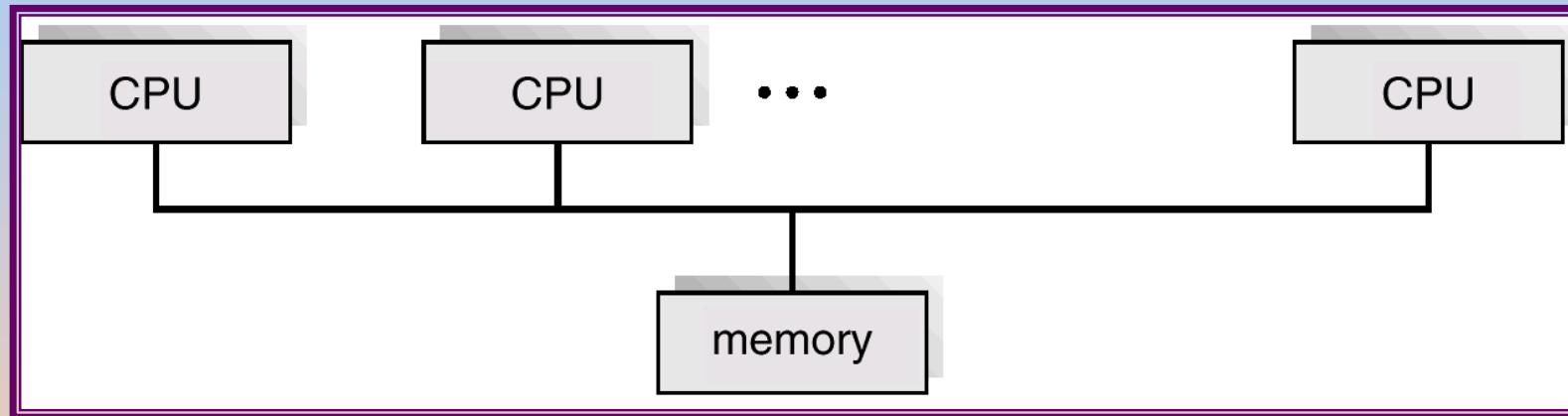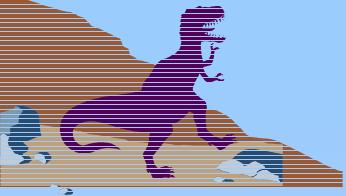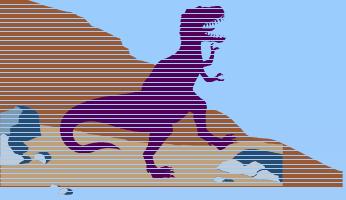
Symmetric Multiprocessing Architecture

Distributed Systems
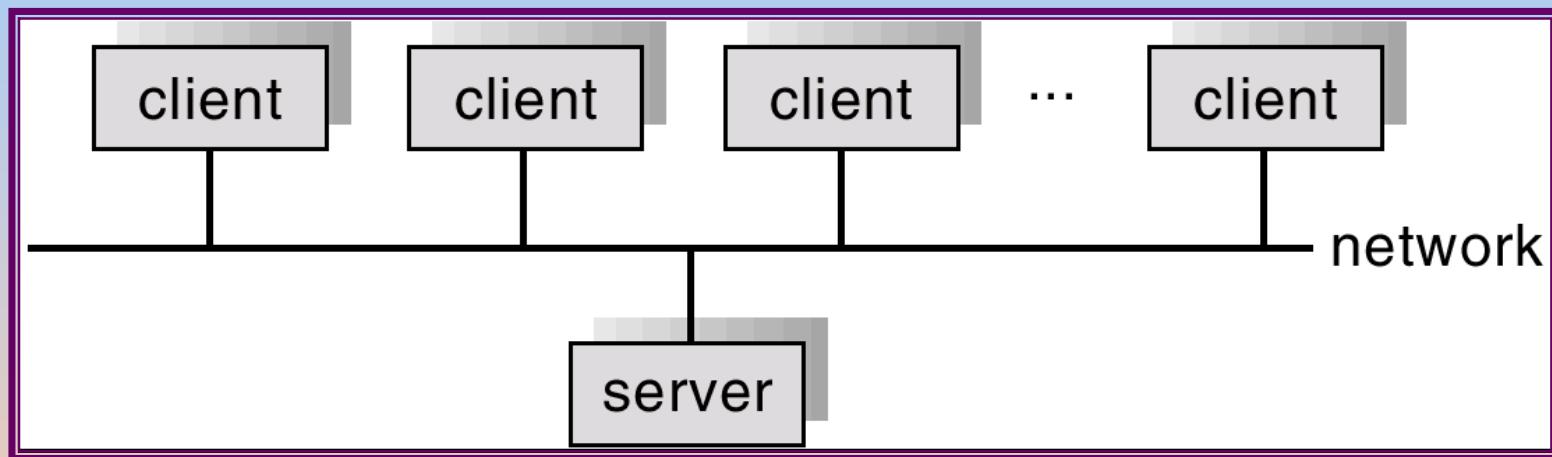
- Distribute the computation among several physical processors.

- *Loosely coupled 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
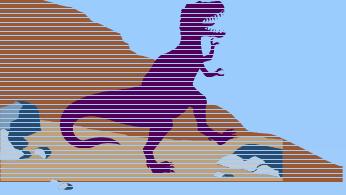
Distributed Systems (cont)

- Requires networking infrastructure.
- Local area networks (LAN) or Wide area networks (WAN)
- May be either client-server or peer-to-peer systems.
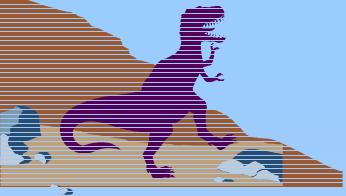
General Structure of Client-Server
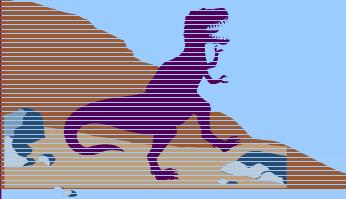
Clustered Systems

- Clustering allows two or more systems to share storage.
- Provides high reliability.
- *Asymmetric clustering*: one server runs the application while other servers standby.
- *Symmetric clustering*: all N hosts are running the application.

Real-Time Systems

- 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.

- Real-Time systems may be either *hard* or *soft* real-time.
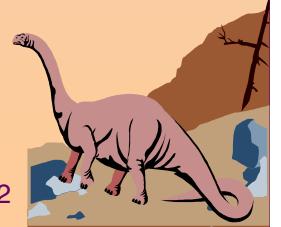
Real-Time Systems (Cont.)
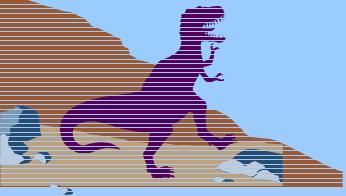
- **Hard real-time:**
  - Secondary storage limited or absent, data stored in short term memory, or read-only memory (ROM)
  - Conflicts with time-sharing systems, not supported by general-purpose operating systems.

- **Soft real-time**
  - Limited utility in industrial control of robotics
  - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.
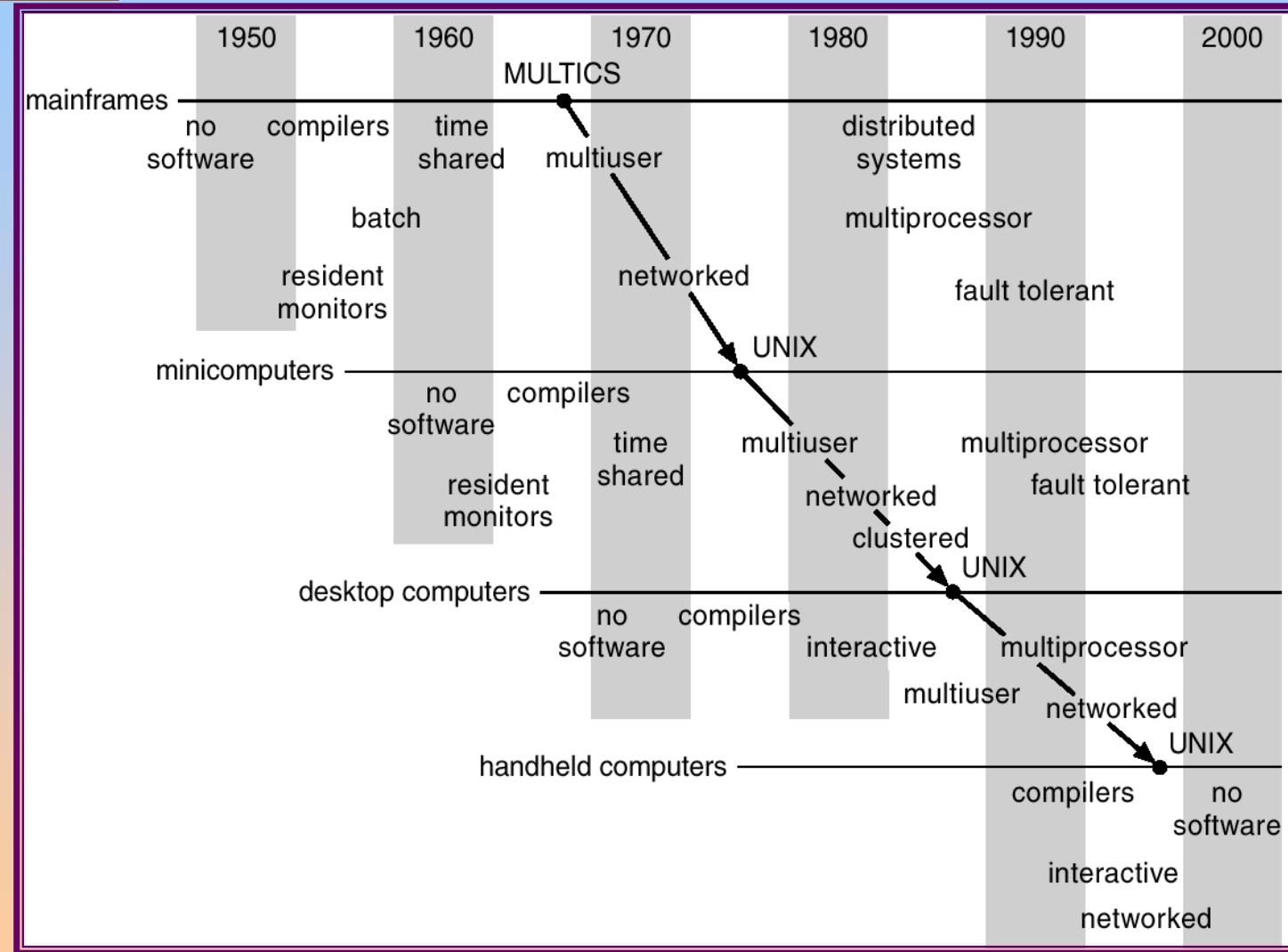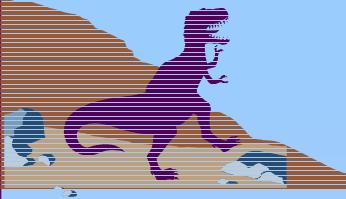
Handheld Systems

- Personal Digital Assistants (PDAs)
- Cellular telephones
- Issues:
  - Limited memory
  - Slow processors
  - Small display screens.
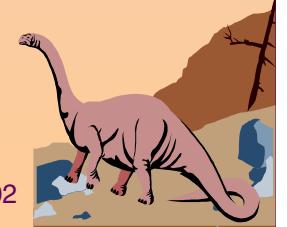
Migration of Operating-System Concepts and Features

	1950	1960	1970	1980	1990	2000

MULTICS

mainframes

no software · compilers · time shared · multiuser · distributed systems

batch · multiprocessor

resident monitors · networked · fault tolerant

UNIX

minicomputers

no software · compilers

time shared · multiuser · multiprocessor

resident monitors · networked · fault tolerant

clustered

UNIX

desktop computers

no software · compilers

interactive · multiprocessor

multiuser · networked

UNIX

handheld computers

compilers · no software

interactive

networked

Computing Environments

- Traditional computing
- Web-Based Computing
- Embedded Computing
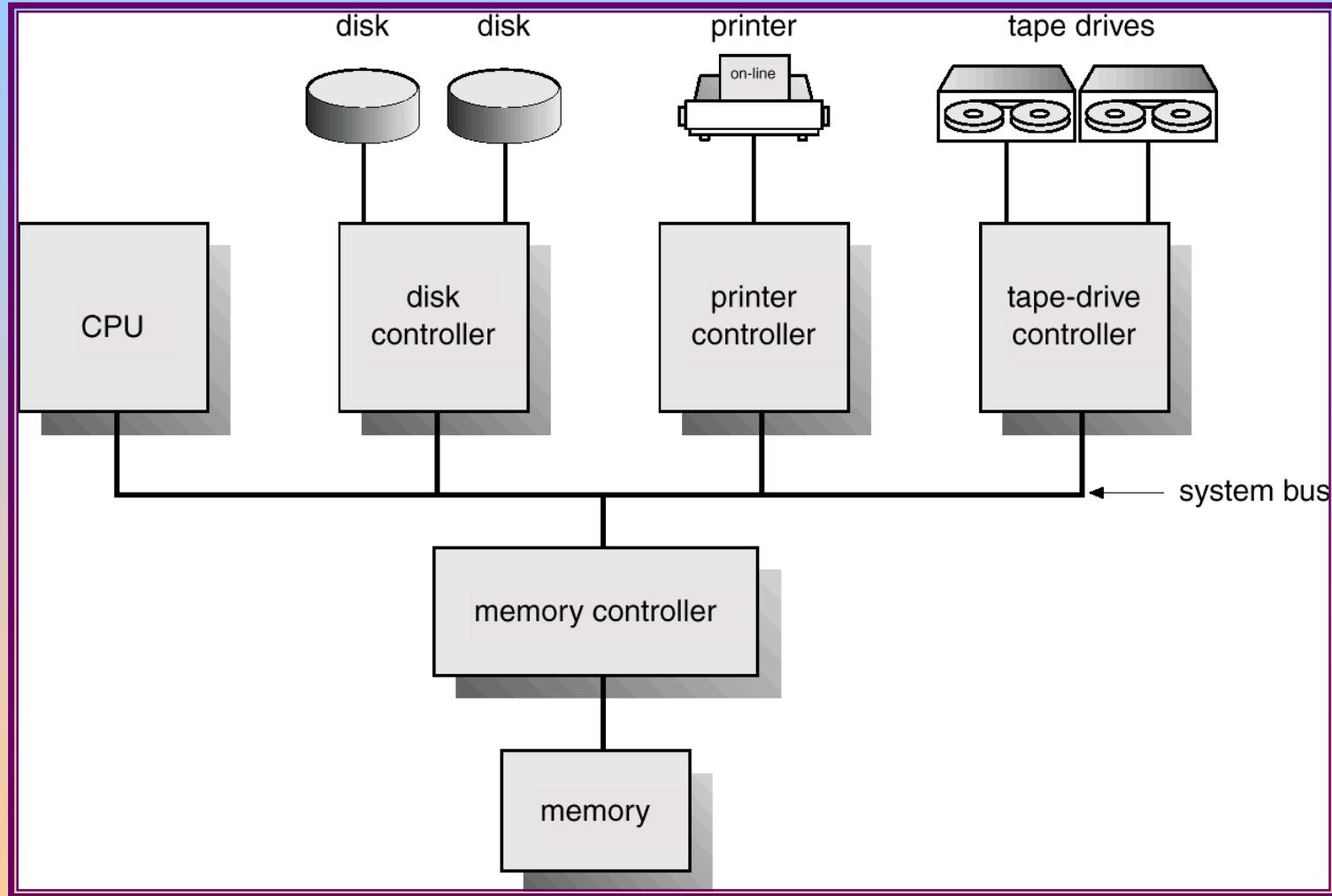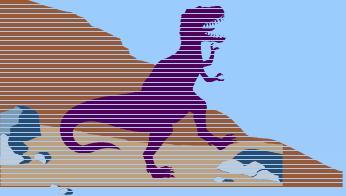
Chapter 2: Computer-System Structures

- ■ Computer System Operation
- ■ I/O Structure
- ■ Storage Structure
- ■ Storage Hierarchy
- ■ Hardware Protection
- ■ General System Architecture

2.1
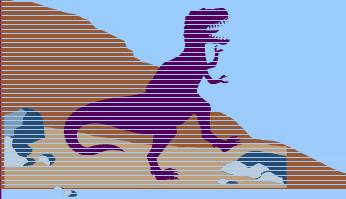
Computer-System Architecture
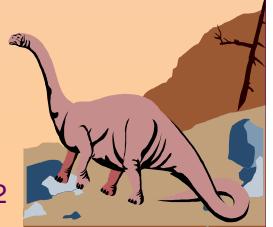
Computer-System Operation
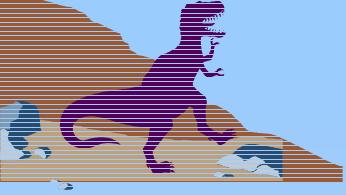
- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an *interrupt*.

Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the *interrupt vector*, which contains the addresses of all the service routines.

- Interrupt architecture must save the address of the interrupted instruction.

- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.

- A *trap* is a software-generated interrupt caused either by an error or a user request.

- An operating system is *interrupt* driven.
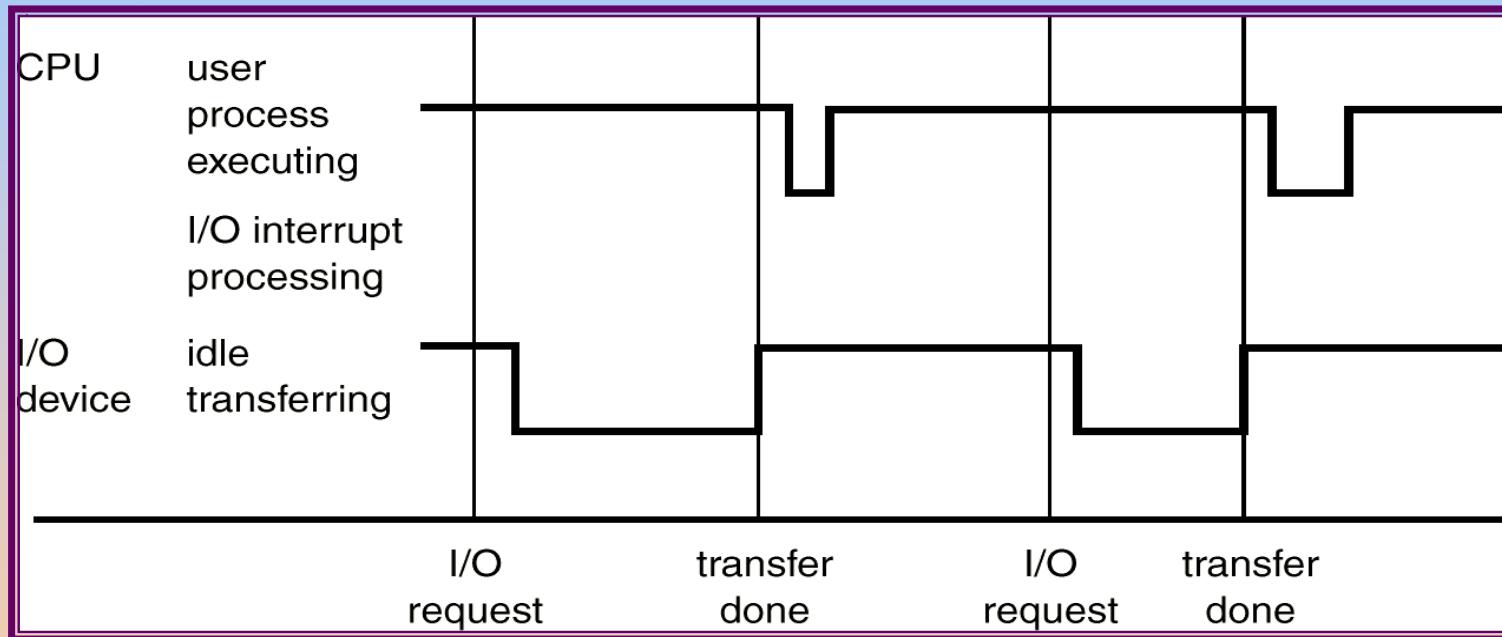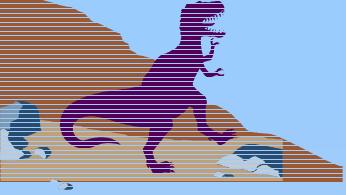
Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.

- Determines which type of interrupt has occurred:
  - *polling*
  - *vectored* interrupt system

- Separate segments of code determine what action should be taken for each type of interrupt

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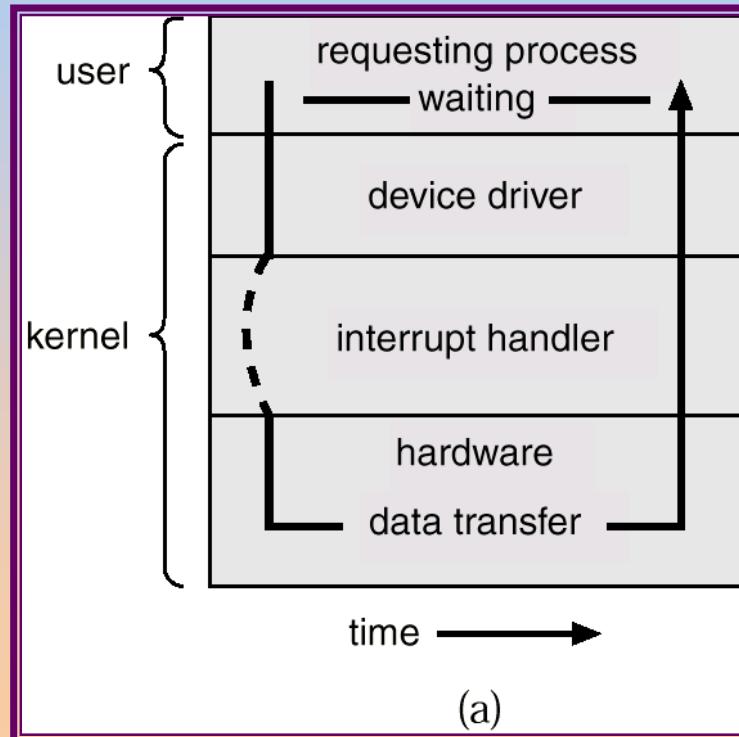
I/O Structure
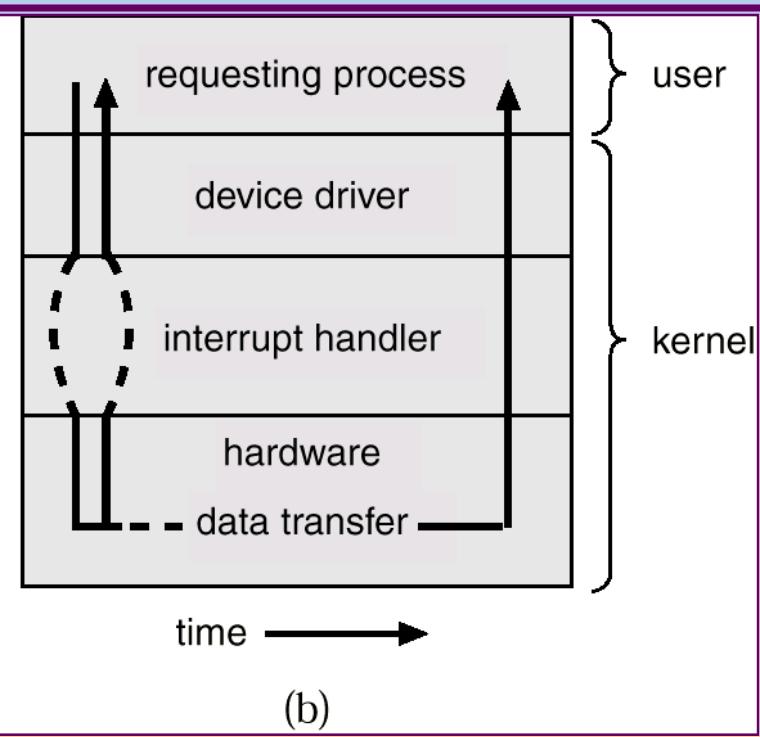
- After I/O starts, control returns to user program only upon I/O completion.
    - Wait instruction idles the CPU until the next interrupt
    - Wait loop (contention for memory access).
    - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control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

Two I/O Methods

Synchronous

Asynchronous

(a)

(b)

Device-Status Table

device: card reader 1 status: idle	
device: line printer 3 status: busy	→ request for line printer address: 38546 length: 1372
device: disk unit 1 status: idle	
device: disk unit 2 status: idle	
device: disk unit 3 status: busy	→ request for disk unit 3 file: xxx operation: read address: 43046 length: 20000 → request for disk unit 3 file: yyy operation: write address: 03458 length: 500
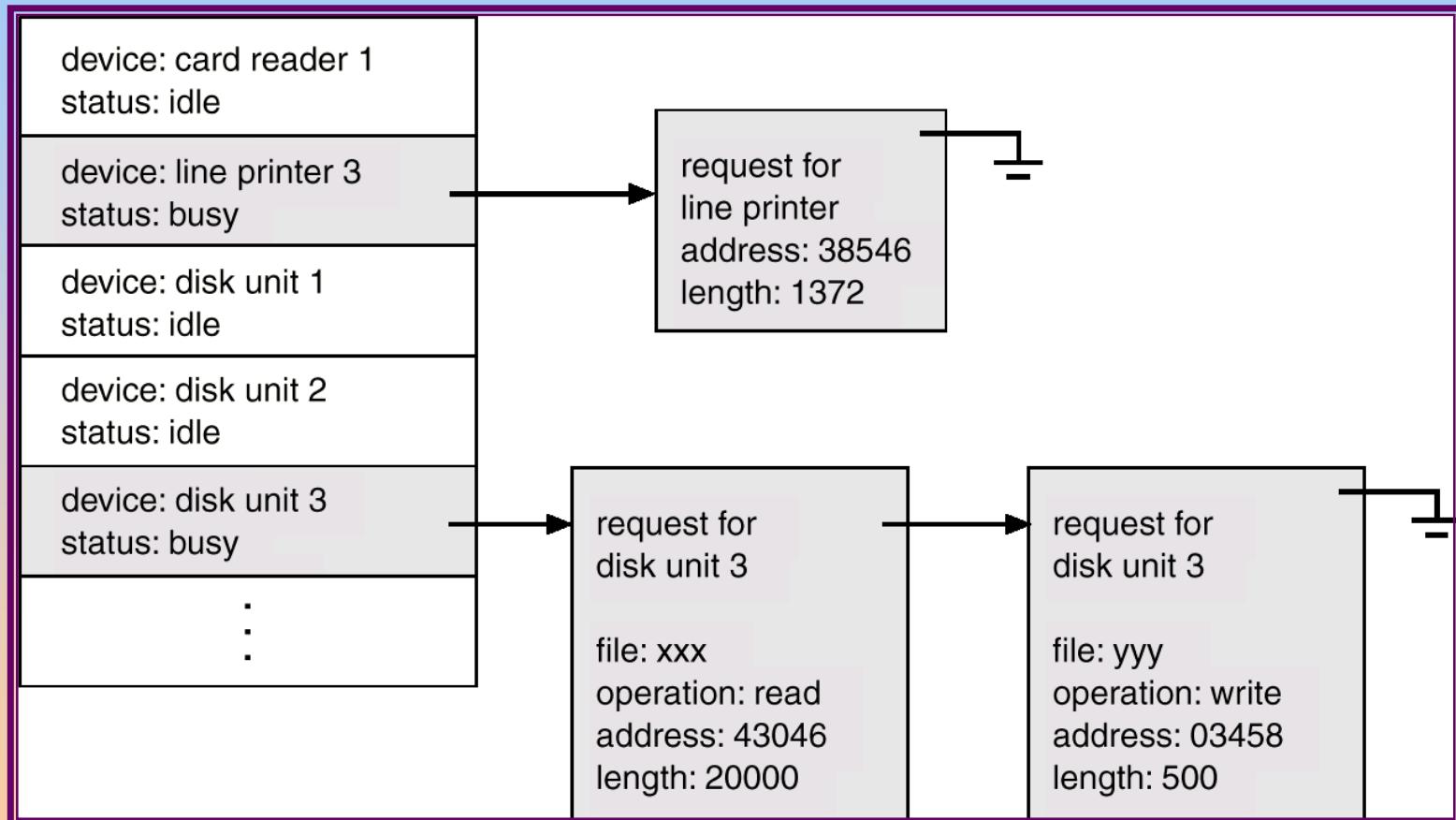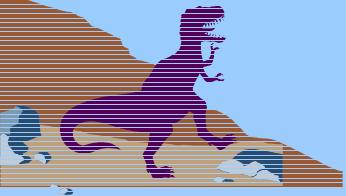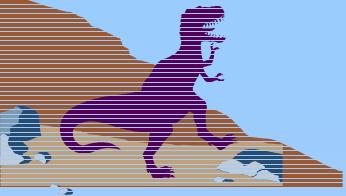
Direct Memory Access Structure
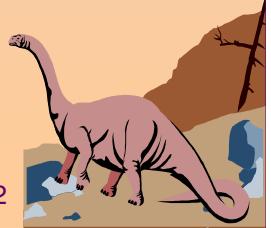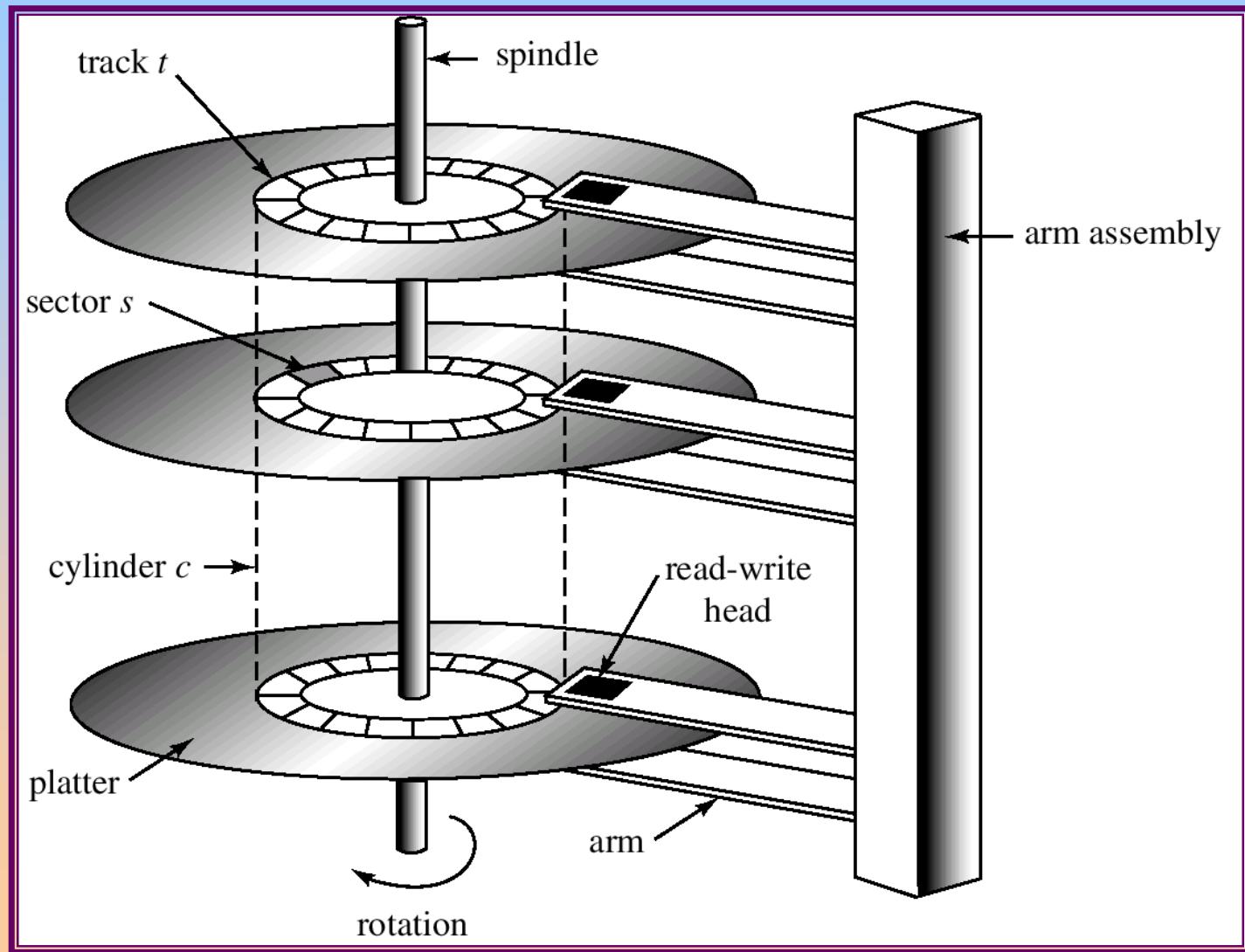
- Used for high-speed I/O devices able to transmit information at close to memory speeds.

- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.

- Only on interrupt is generated per block, rather than the one interrupt per byte.
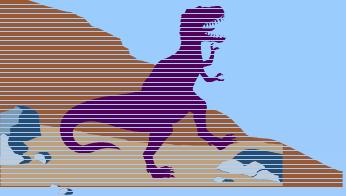
Storage Structure

- Main memory – only large storage media that the CPU can access directly.

- Secondary storage – extension of main memory that provides large nonvolatile storage capacity.

- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into *tracks*, which are subdivided into *sectors*.
  - The *disk controller* determines the logical interaction between the device and the computer.
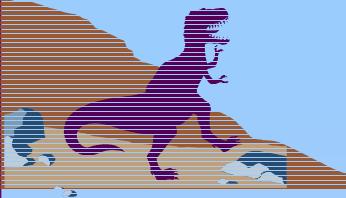
Moving-Head Disk Mechanism
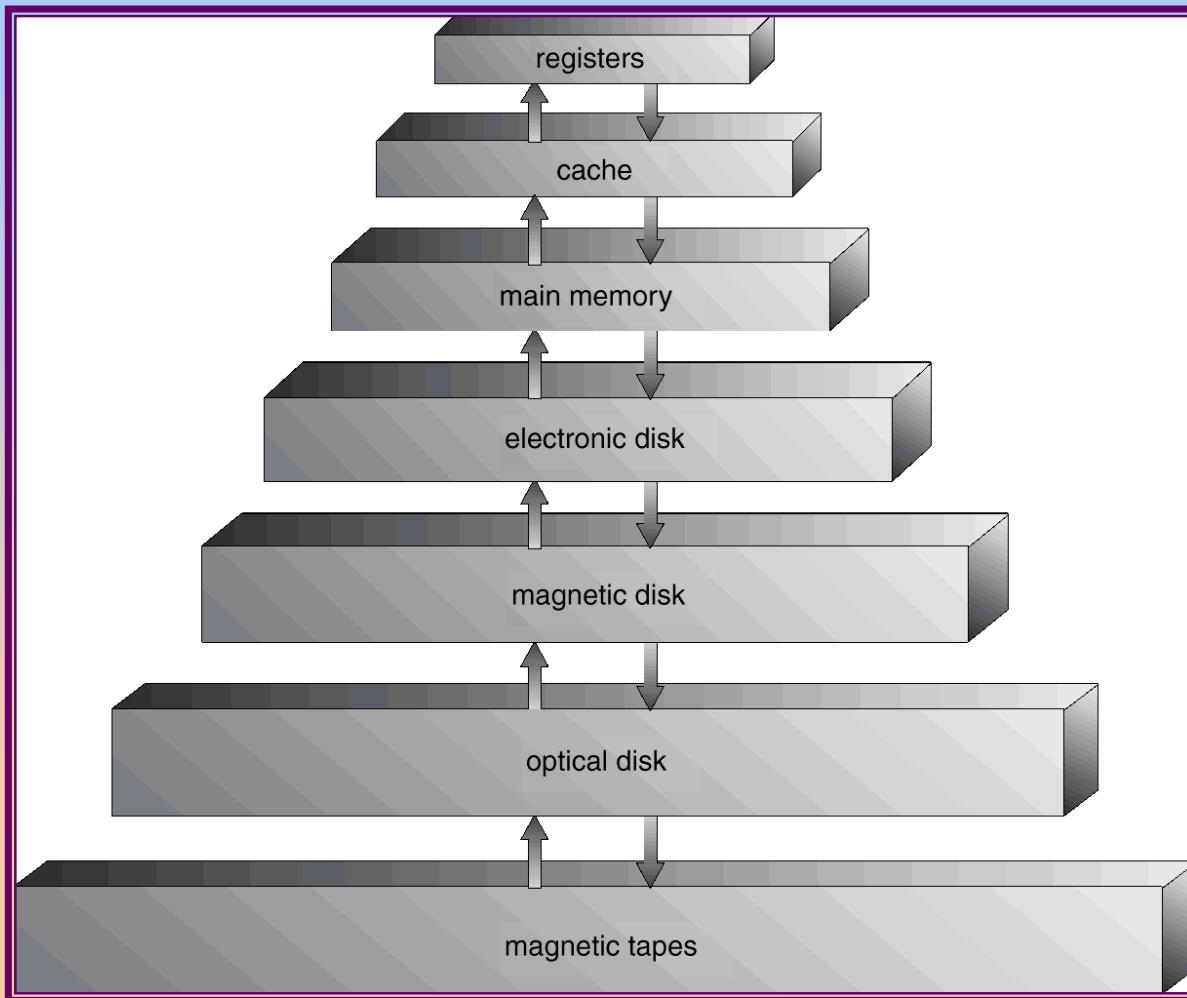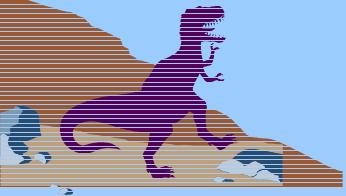
Storage Hierarchy

- Storage systems organized in hierarchy.
  - Speed
  - Cost
  - Volatility

- *Caching* – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage.
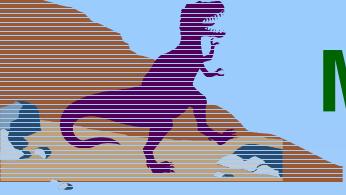
Storage-Device Hierarchy

registers

cache

main memory

electronic disk

magnetic disk

optical disk

magnetic tapes
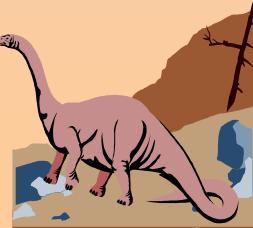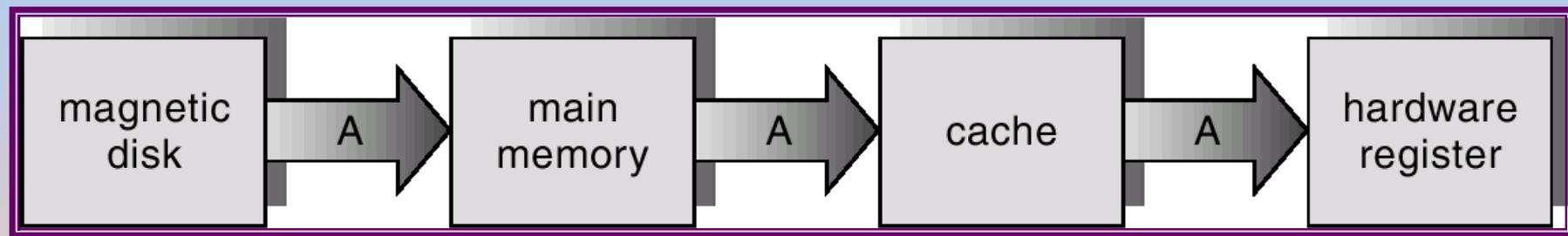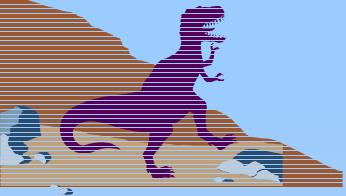
Caching

- Use of high-speed memory to hold recently-accessed data.

- Requires a *cache management* policy.

- Caching introduces another level in storage hierarchy. This requires data that is simultaneously stored in more than one level to be *consistent*.
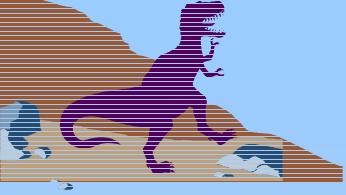
Migration of A From Disk to Register
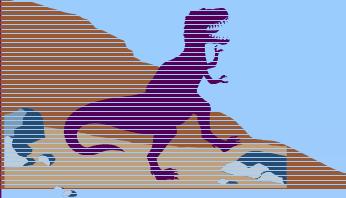
Hardware Protection

- Dual-Mode Operation
- I/O Protection
- Memory Protection
- CPU Protection
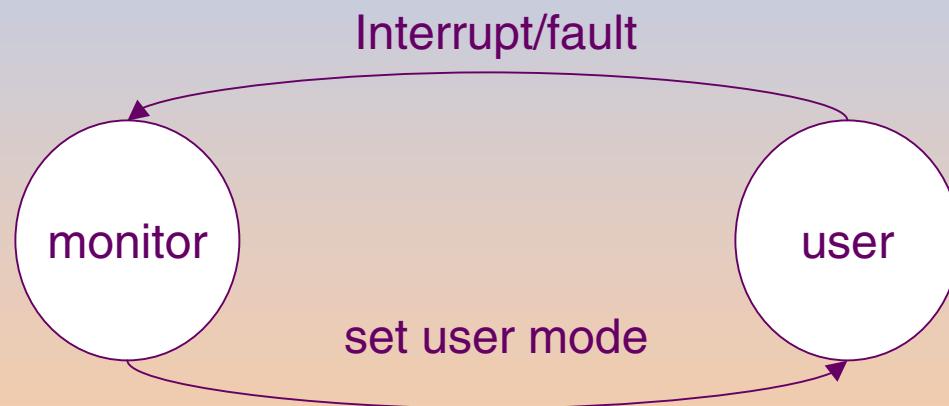
Dual-Mode Operation

■ Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly.

■ 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.
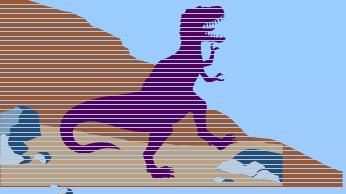
Dual-Mode Operation (Cont.)

■ *Mode bit* added to computer hardware to indicate the current mode: monitor (0) or user (1).

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

Interrupt/fault

monitor   user

set user mode

Privileged instructions can be issued only in monitor mode.
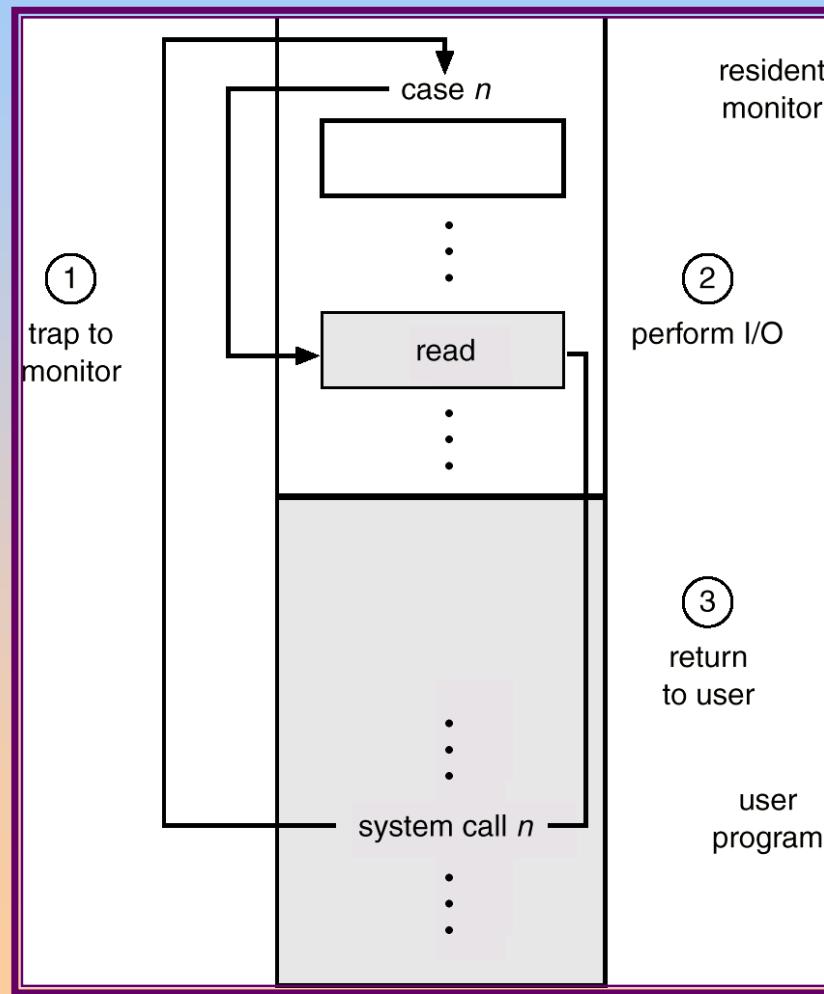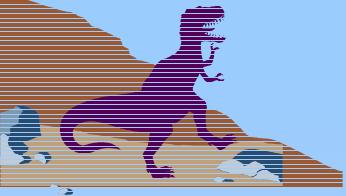
I/O Protection

- All I/O instructions are privileged instructions.

- Must ensure that a user program could never gain control of the computer in monitor mode (I.e., a user program that, as part of its execution, stores a new address in the interrupt vector).

Use of A System Call to Perform I/O

resident monitor

case *n*

① trap to monitor

② perform I/O

read

③ return to user

system call *n*

user program
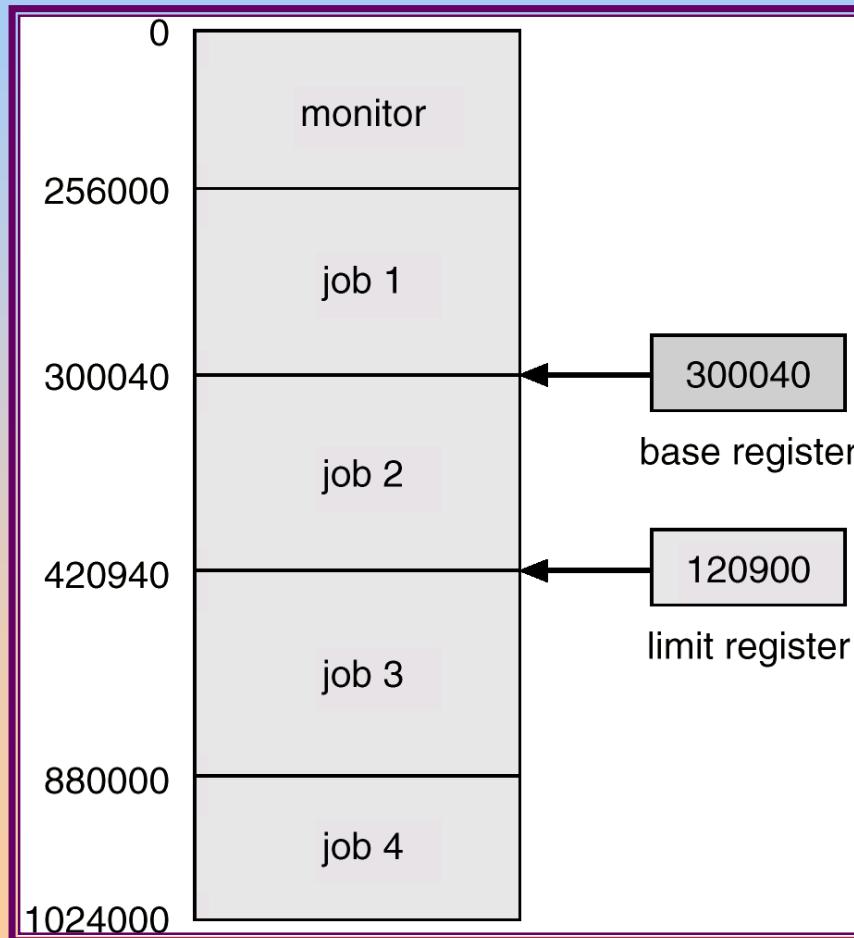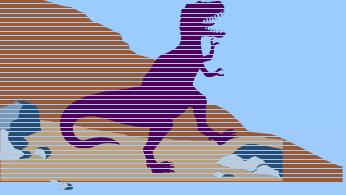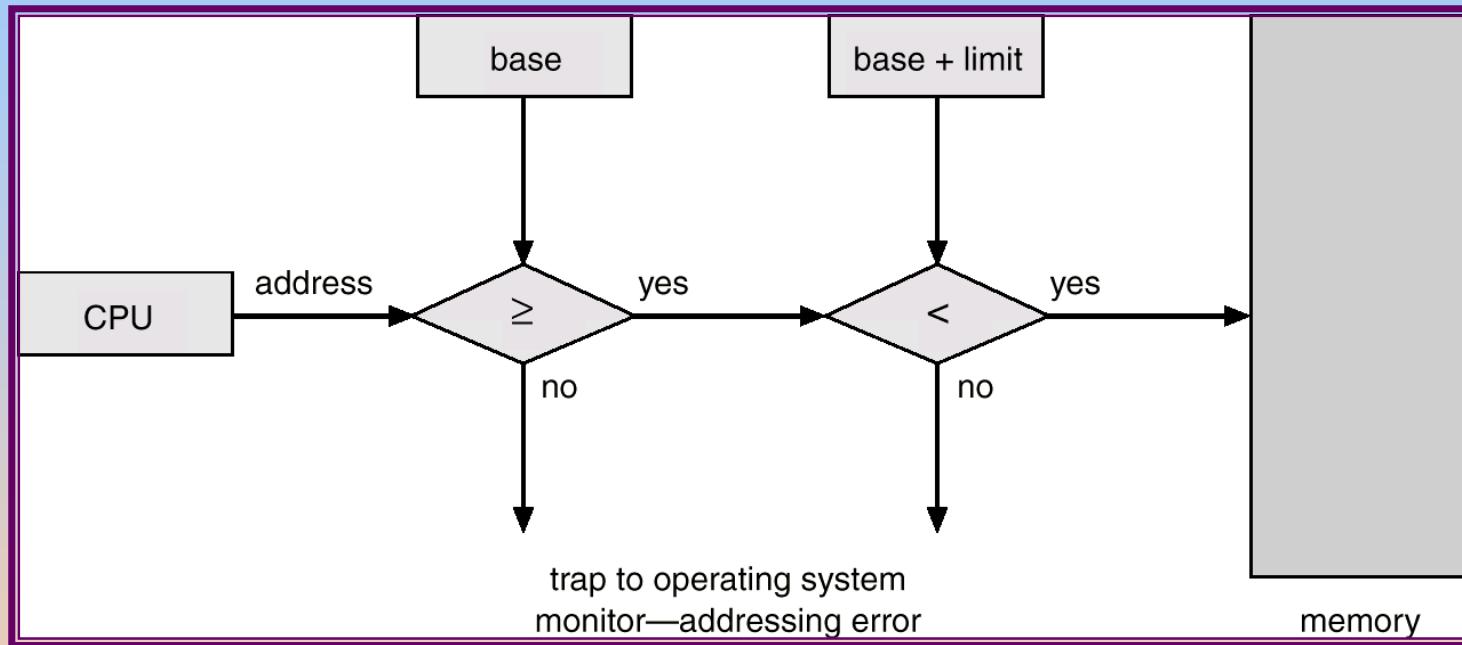
Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
    - **Base register** – holds the smallest legal physical memory address.
    - **Limit register** – contains the size of the range

- Memory outside the defined range is protected.

2.22
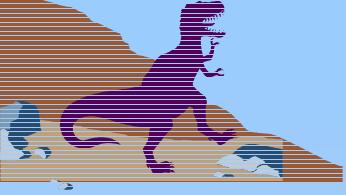
Use of A Base and Limit Register

0	monitor
256000	job 1
300040	job 2
420940	job 3
880000	job 4
1024000	

300040 → base register

120900 → limit register
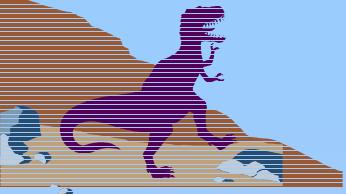
Hardware Address Protection

The diagram shows: base, base + limit, CPU, address, ≥, yes, no, <, yes, no, trap to operating system monitor—addressing error, memory.
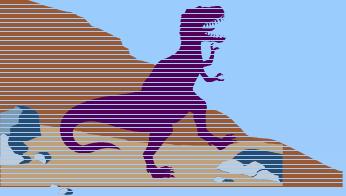
Hardware Protection

■ When executing in monitor mode, the operating system has unrestricted access to both monitor and user's memory.

■ The load instructions for the *base* and *limit* registers are privileged instructions.

CPU Protection
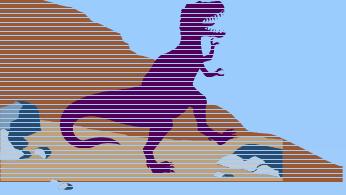
- *Timer* – interrupts computer after specified period to ensure operating system maintains control.
    - ✦ Timer is decremented every clock tick.
    - ✦ When timer reaches the value 0, an interrupt occurs.
- Timer commonly used to implement time sharing.
- Time also used to compute the current time.
- Load-timer is a privileged instruction.
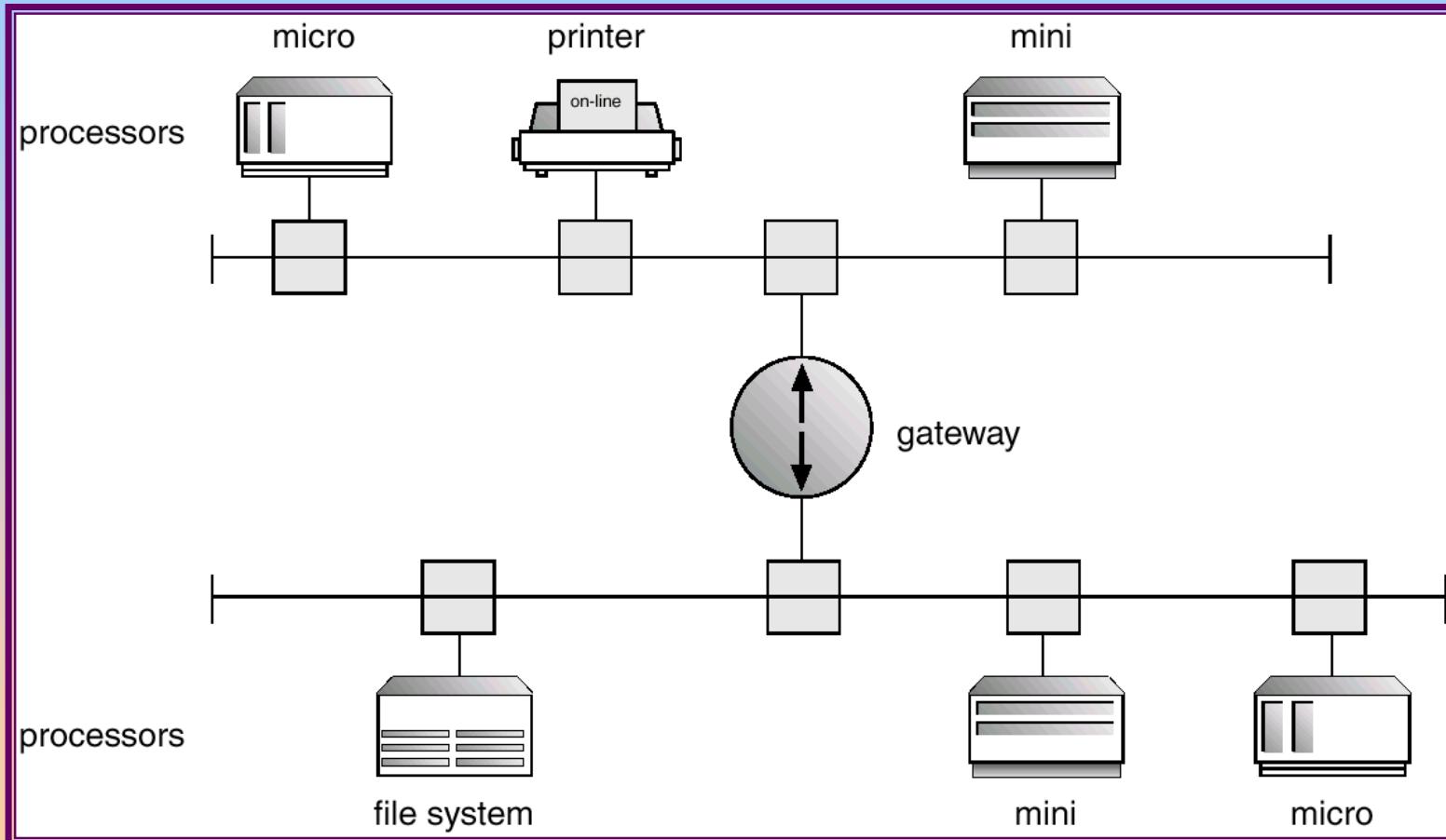
Network Structure

- Local Area Networks (LAN)
- Wide Area Networks (WAN)
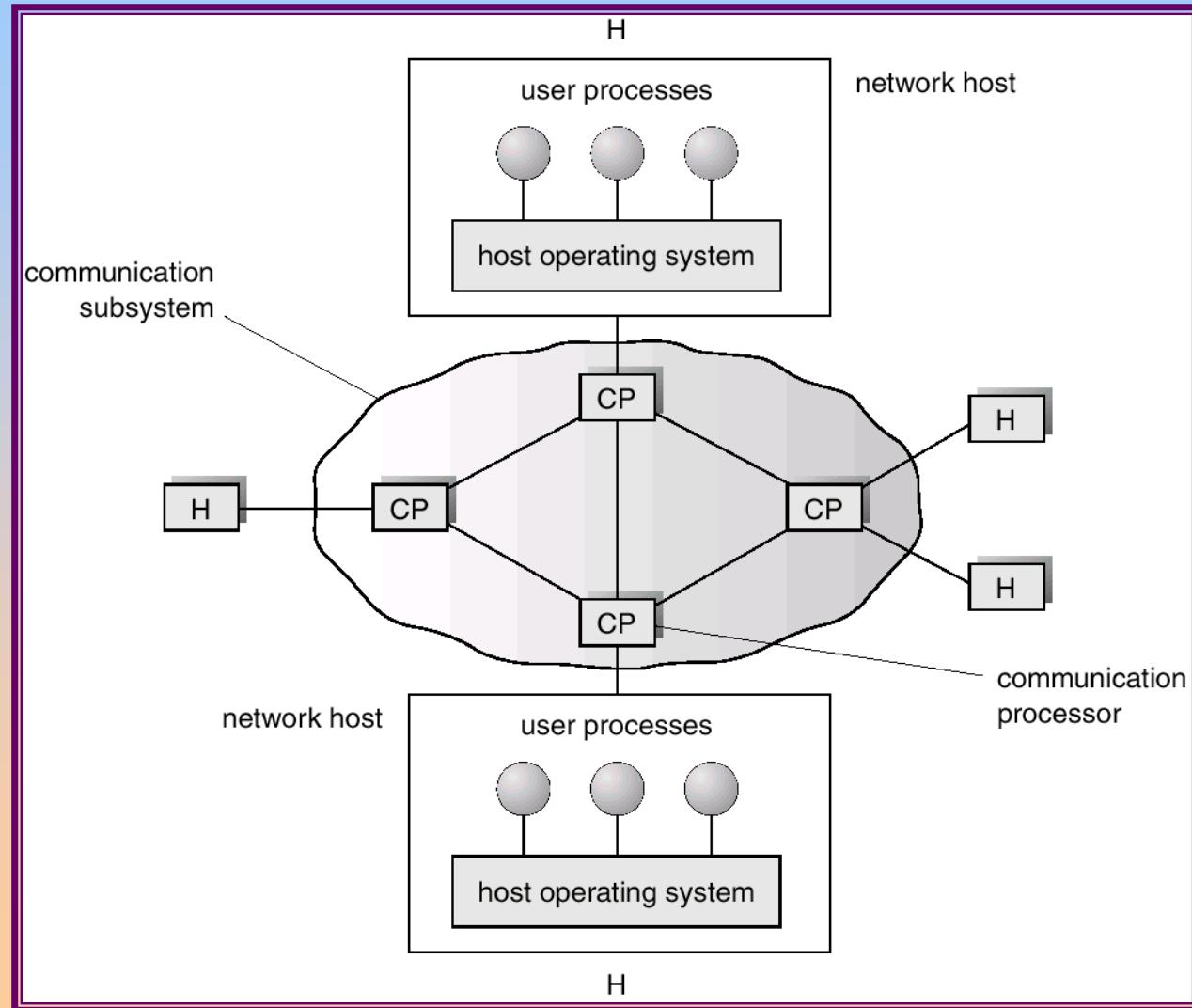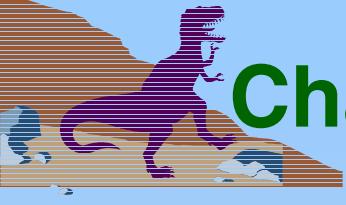
Local Area Network Structure

Wide Area Network Structure
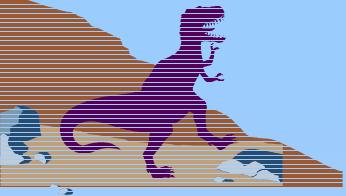
Chapter 3: Operating-System Structures
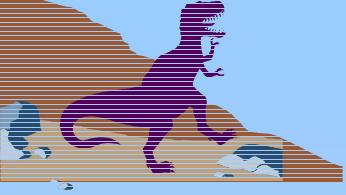
- System Components
- Operating System Services
- System Calls
- System Programs
- System Structure
- Virtual Machines
- System Design and Implementation
- System Generation
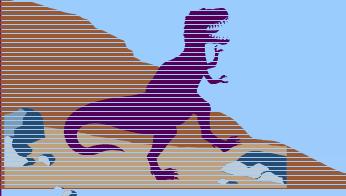
Common System Components

- Process Management

- Main Memory Management

- File Management

- I/O System Management

- Secondary Management

- Networking

- Protection System

- Command-Interpreter System
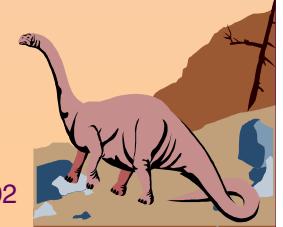
Process Management
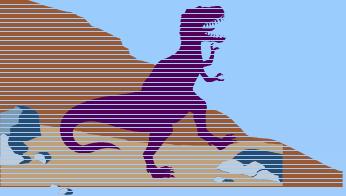
- A *process* is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

- The operating system is responsible for the following activities in connection with process management.
  - Process creation and deletion.
  - process suspension and resumption.
  - Provision of mechanisms for:
    - process synchronization
    - process communication

Main-Memory Management
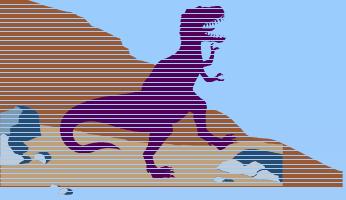
- Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

- Main memory is a volatile storage device. It loses its contents in the case of system failure.

- The operating system is responsible for the following activities in connections with memory management:
  - Keep track of which parts of memory are currently being used and by whom.
  - Decide which processes to load when memory space becomes available.
  - Allocate and deallocate memory space as needed.

File Management
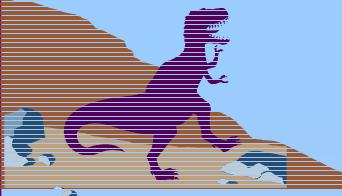
- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

- The operating system is responsible for the following activities in connections with file management:
  - File creation and deletion.
  - Directory creation and deletion.
  - Support of primitives for manipulating files and directories.
  - Mapping files onto secondary storage.
  - File backup on stable (nonvolatile) storage media.

I/O System Management
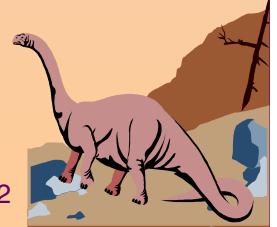
- The I/O system consists of:
  - A buffer-caching system
  - A general device-driver interface
  - Drivers for specific hardware devices
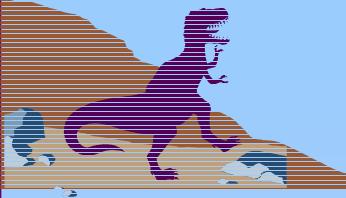
Secondary-Storage Management
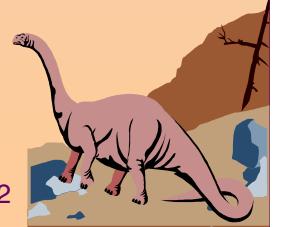
- Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory.

- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.

- The operating system is responsible for the following activities in connection with disk management:
  - Free space management
  - Storage allocation
  - Disk scheduling
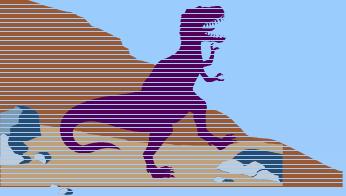
Networking (Distributed Systems)
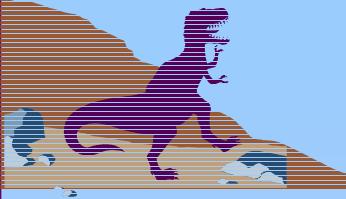
- A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory.

- The processors in the system are connected through a communication network.

- Communication takes place using a *protocol.*

- A distributed system provides user access to various system resources.

- Access to a shared resource allows:
  - Computation speed-up
  - Increased data availability
  - Enhanced reliability
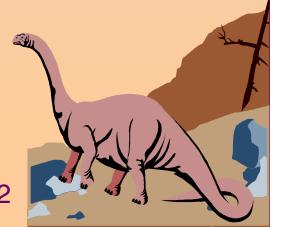
Protection System
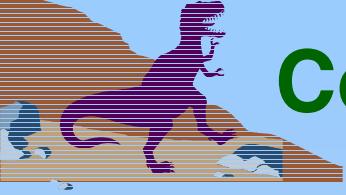
- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.

- The protection mechanism must:
    - distinguish between authorized and unauthorized usage.
    - specify the controls to be imposed.
    - provide a means of enforcement.

Command-Interpreter System

■ Many commands are given to the operating system by control statements which deal with:

✦ process creation and management

✦ I/O handling

✦ secondary-storage management

✦ main-memory management

✦ file-system access

✦ protection

✦ networking
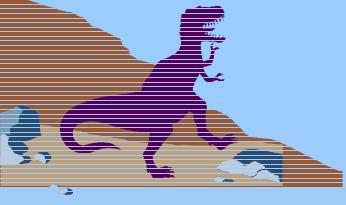
Command-Interpreter System (Cont.)

■ The program that reads and interprets control statements is called variously:
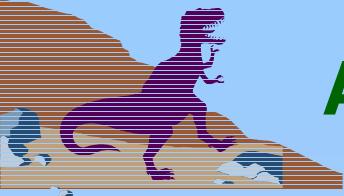
   ✦ command-line interpreter
   ✦ shell (in UNIX)

   Its function is to get and execute the next command statement.

Operating System Services
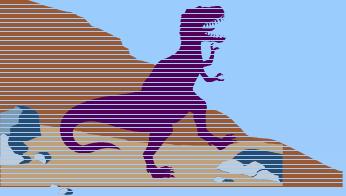
- Program execution – system capability to load a program into memory and to run it.

- I/O operations – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.

- File-system manipulation – program capability to read, write, create, and delete files.

- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.

- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

Additional Operating System Functions
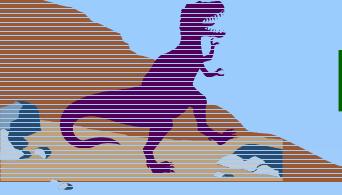
Additional functions exist not for helping the user, but rather for ensuring efficient system operations.

- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.

- Accounting – keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.

- Protection – ensuring that all access to system resources is controlled.
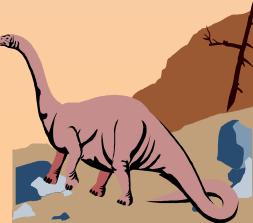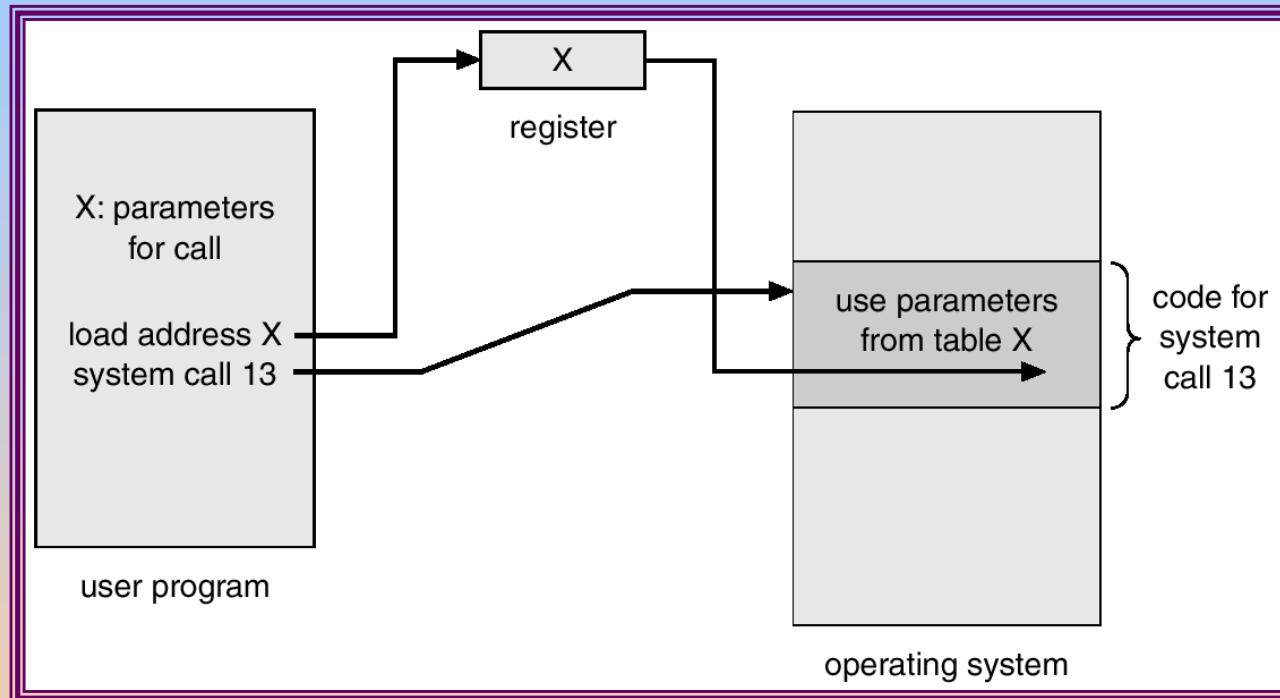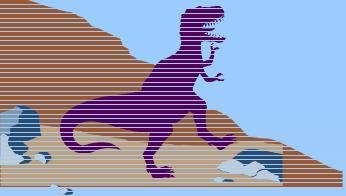
System Calls

- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)
- Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in *registers*.
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.
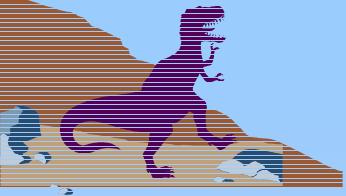
Passing of Parameters As A Table
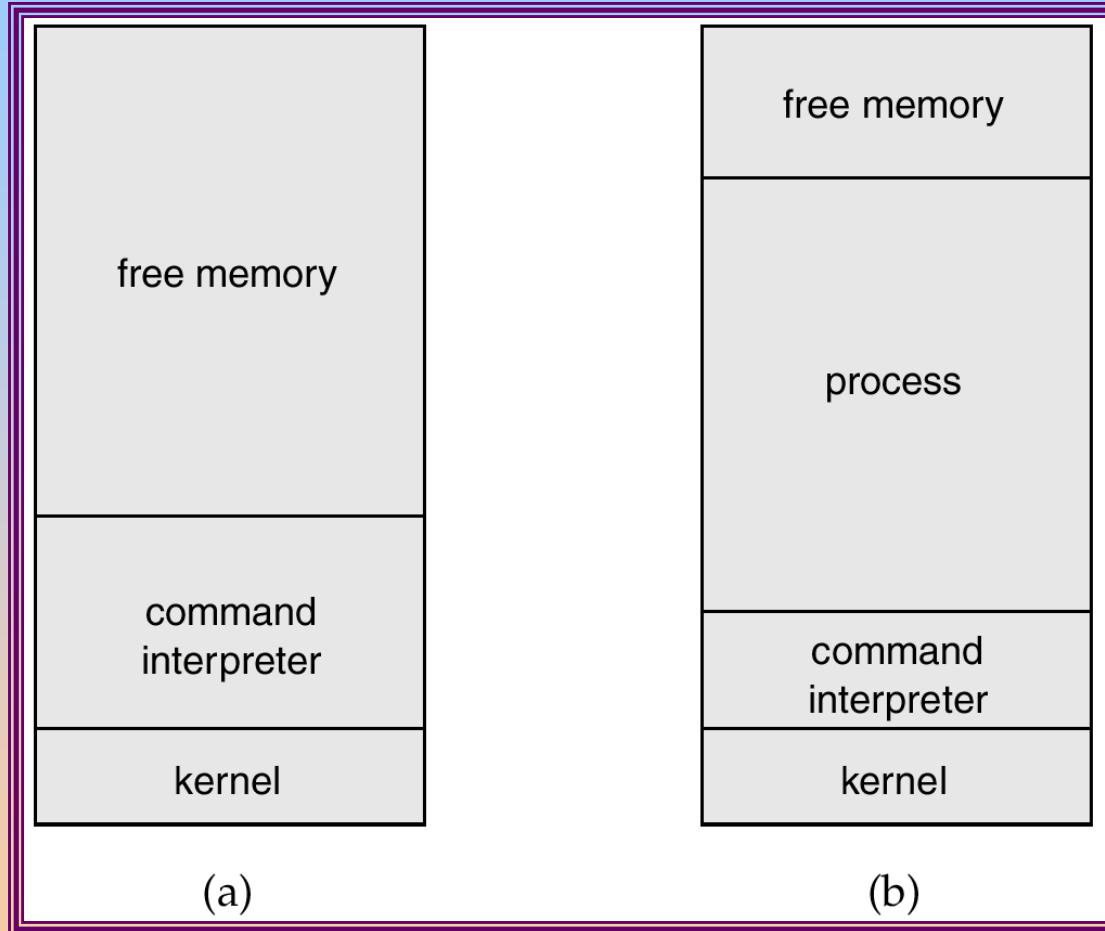
Types of System Calls

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

MS-DOS Execution

free memory			free memory
command interpreter			process
kernel			command interpreter
			kernel

(a)

(b)
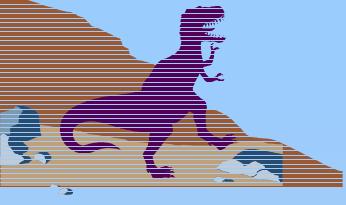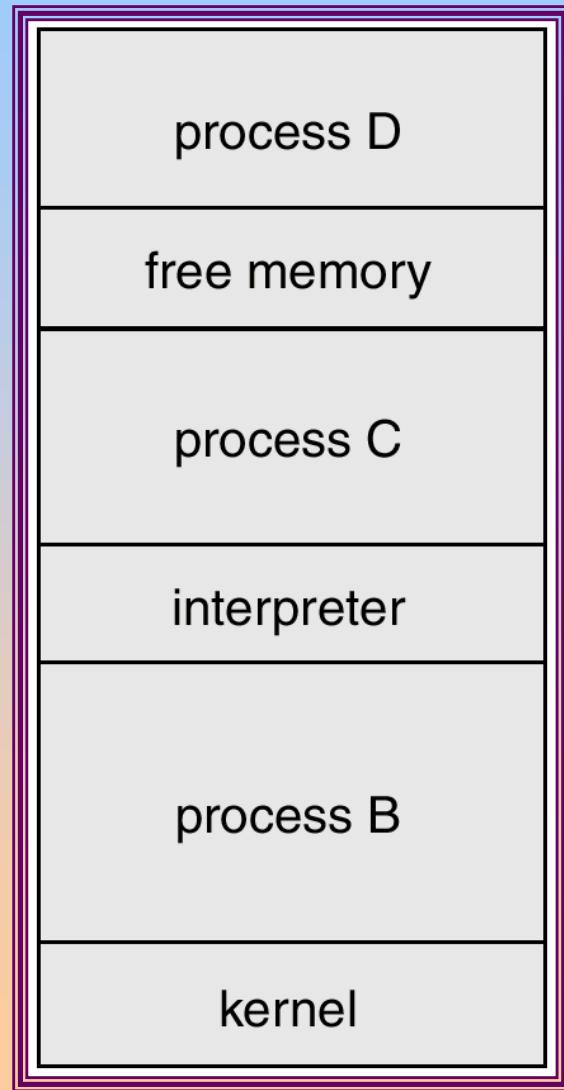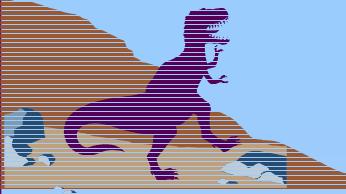
At System Start-up   Running a Program
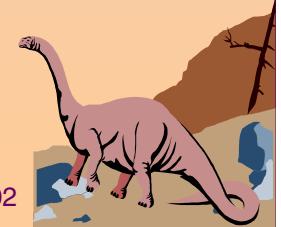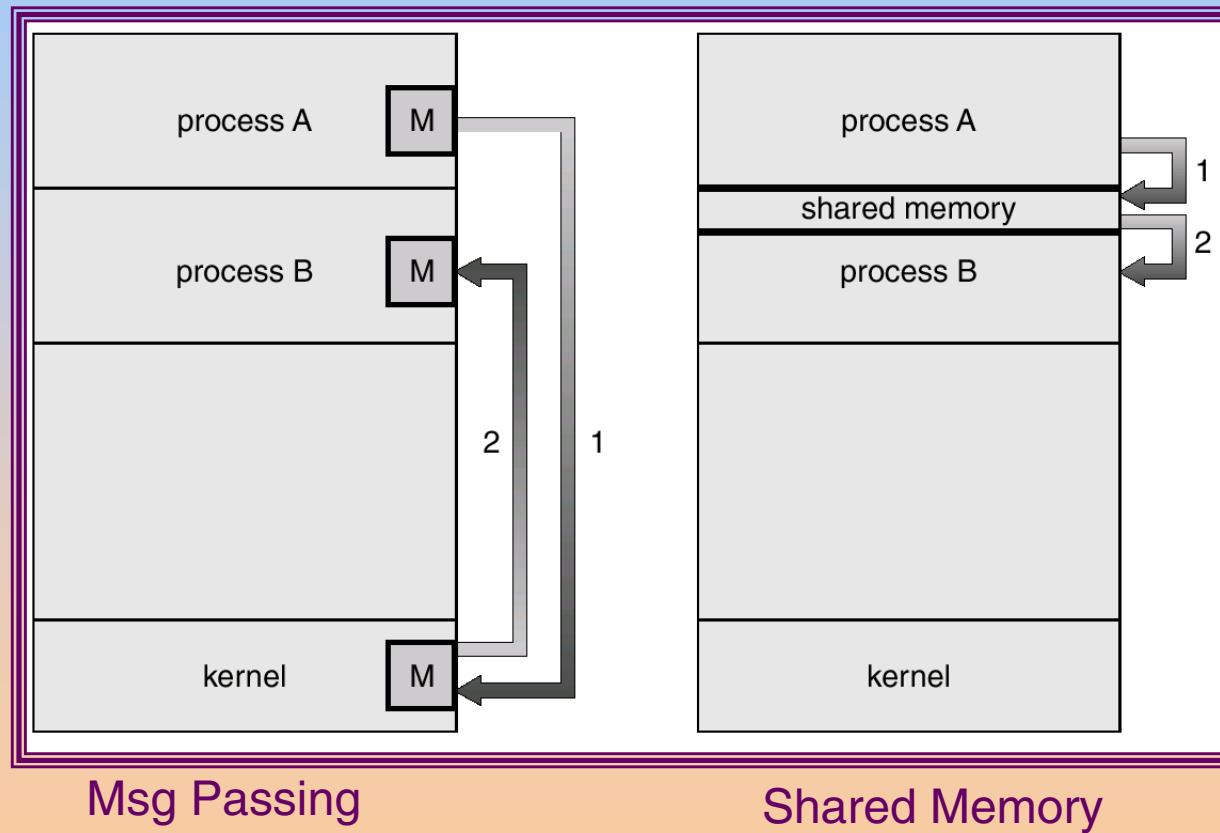
UNIX Running Multiple Programs

process D
free memory
process C
interpreter
process B
kernel
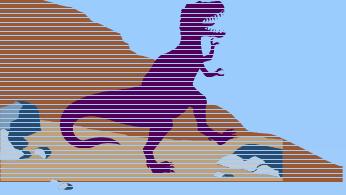
Communication Models
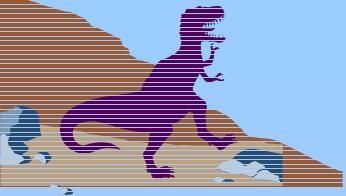
- Communication may take place using either message passing or shared memory.

Msg Passing   Shared Memory

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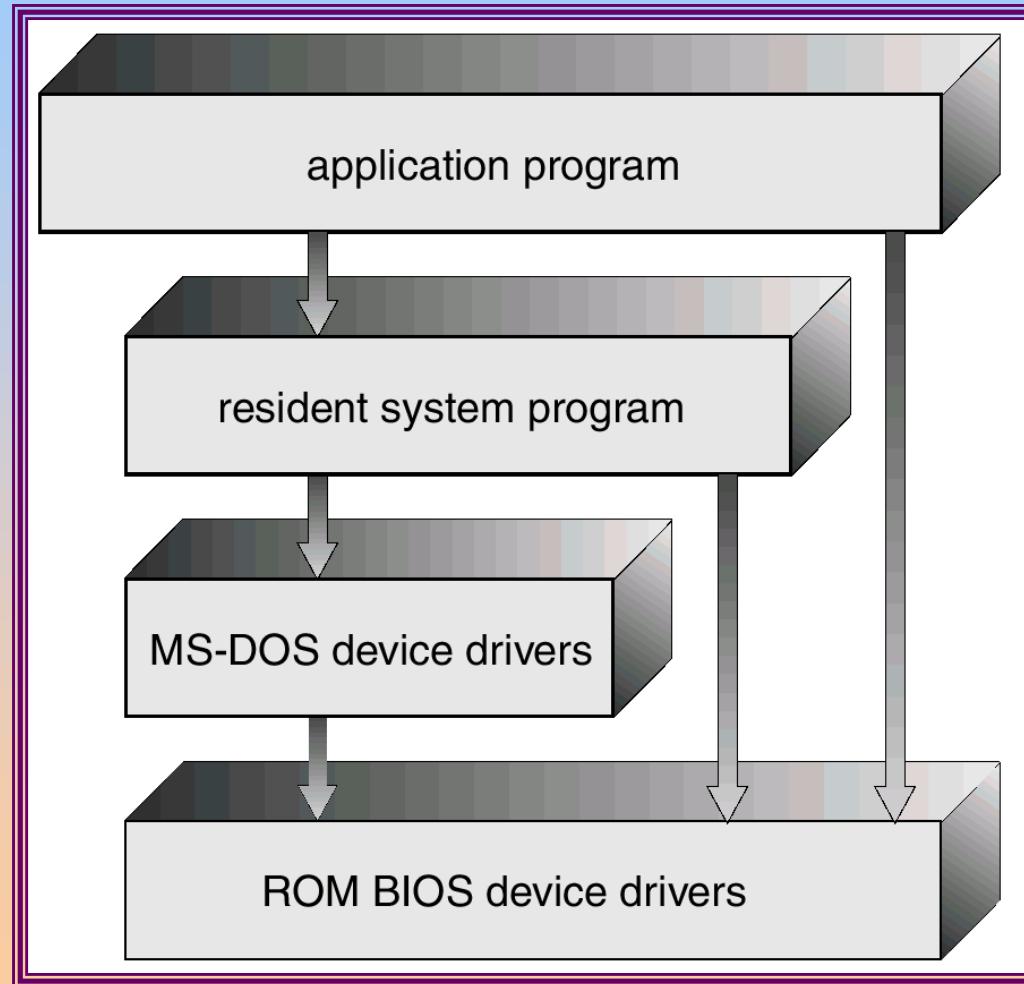
- System programs provide a convenient environment for program development and execution. The can be divided into:
    - File manipulation
    - Status information
    - File modification
    - Programming language support
    - Program loading and execution
    - Communications
    - Application programs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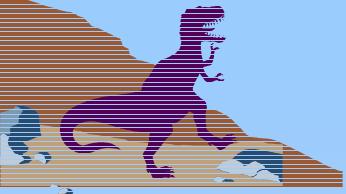
MS-DOS System Structure

■ MS-DOS – written to provide the most functionality in the least space

 ✦ not divided into modules

 ✦ Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
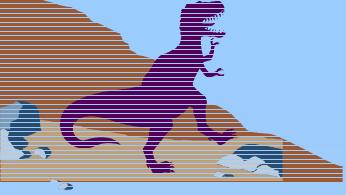
MS-DOS Layer Structure
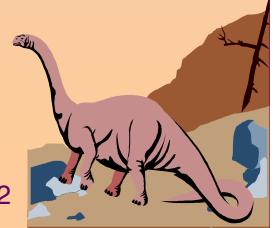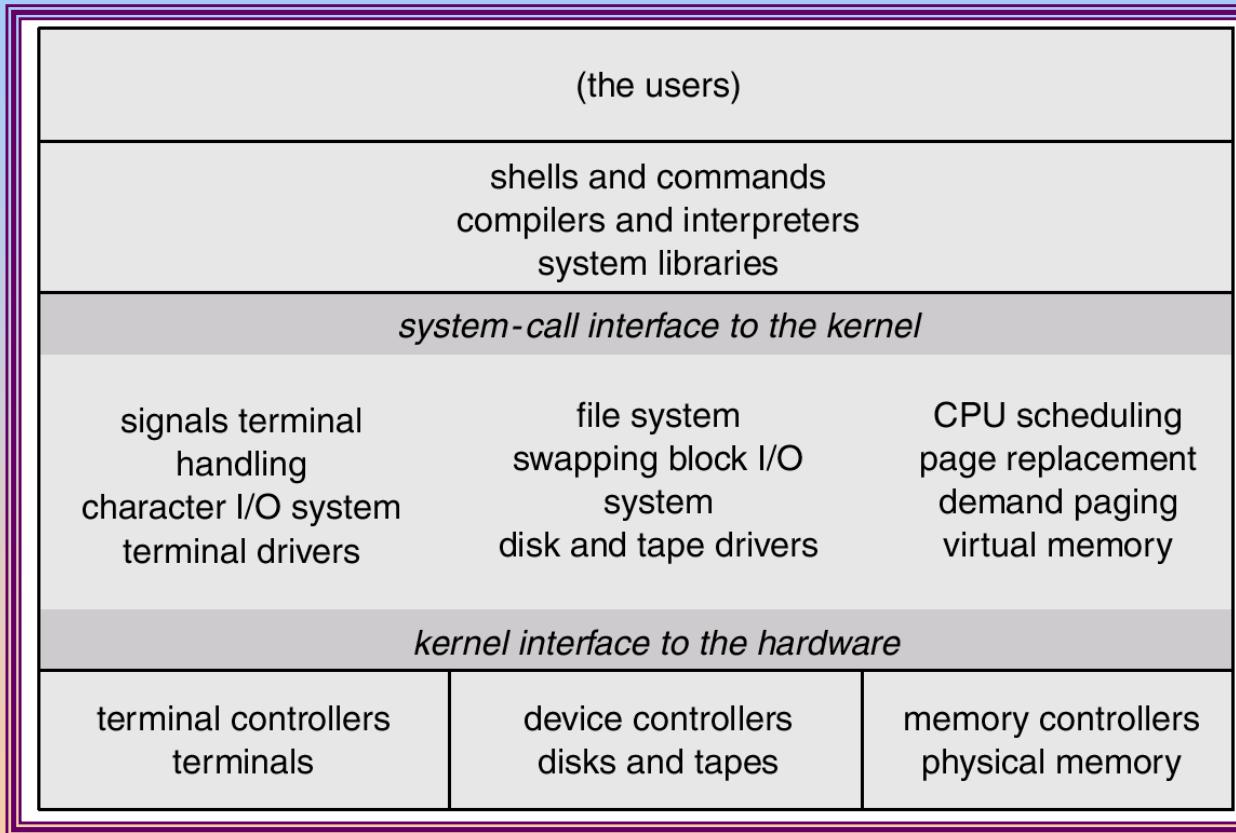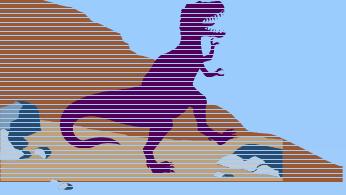
UNIX System Structure

■ UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.

   ✦ Systems programs

   ✦ The kernel

      ✔ 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.

UNIX System Structure

(the users)
shells and commands compilers and interpreters system libraries
system-call interface to the kernel

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

kernel interface to the hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

Layered Approach

■ The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

■ With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

An Operating System Layer

OS/2 Layer Structure

application	application	application

application-programming interface	API extension

subsystem	subsystem	subsystem

system
kernel
- memory management
- task dispatching
- device management

device driver	device driver	device driver	device driver
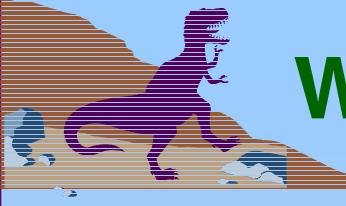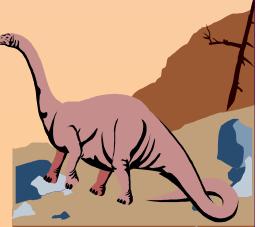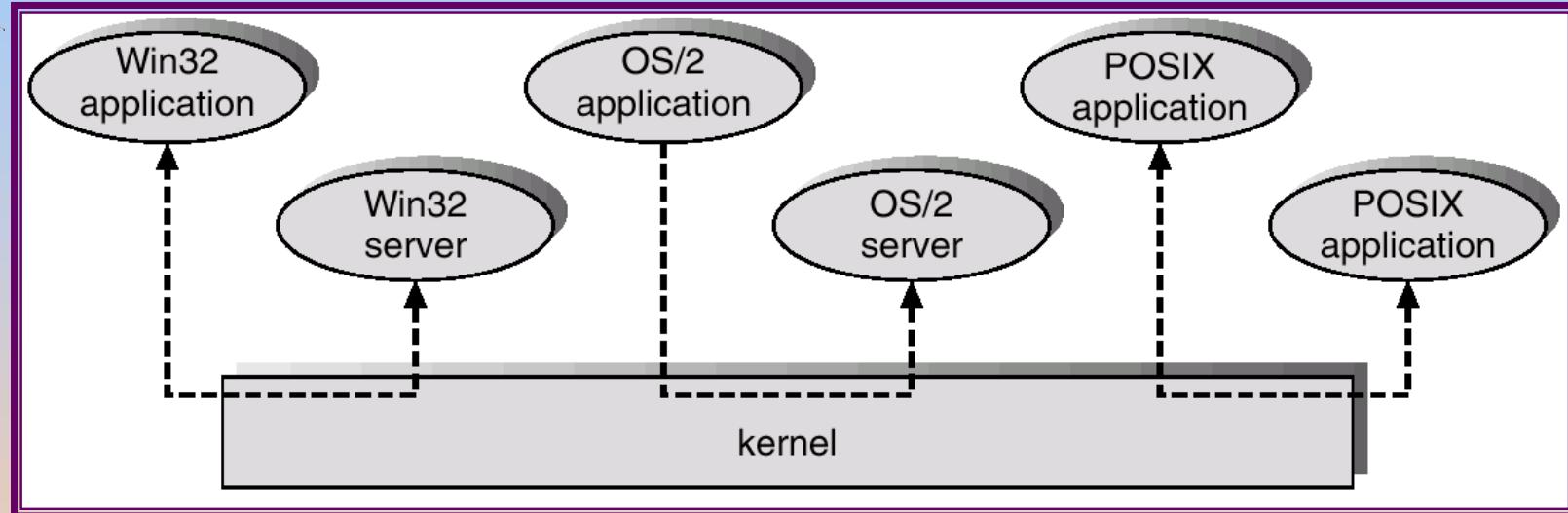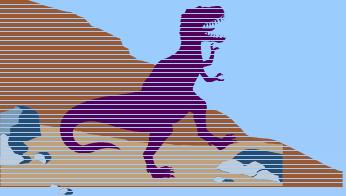
Microkernel System Structure

- Moves as much from the kernel into "*user*" space.
- Communication takes place between user modules using message passing.
- Benefits:

  - easier to extend a microkernel

  - easier to port the operating system to new architectures

  - more reliable (less code is running in kernel mode)

  - more secure
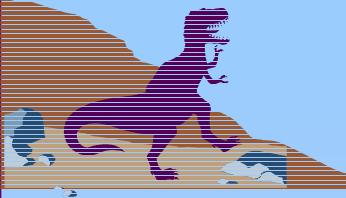
Windows NT Client-Server Structure
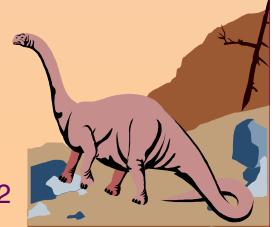
Virtual Machines

- 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.
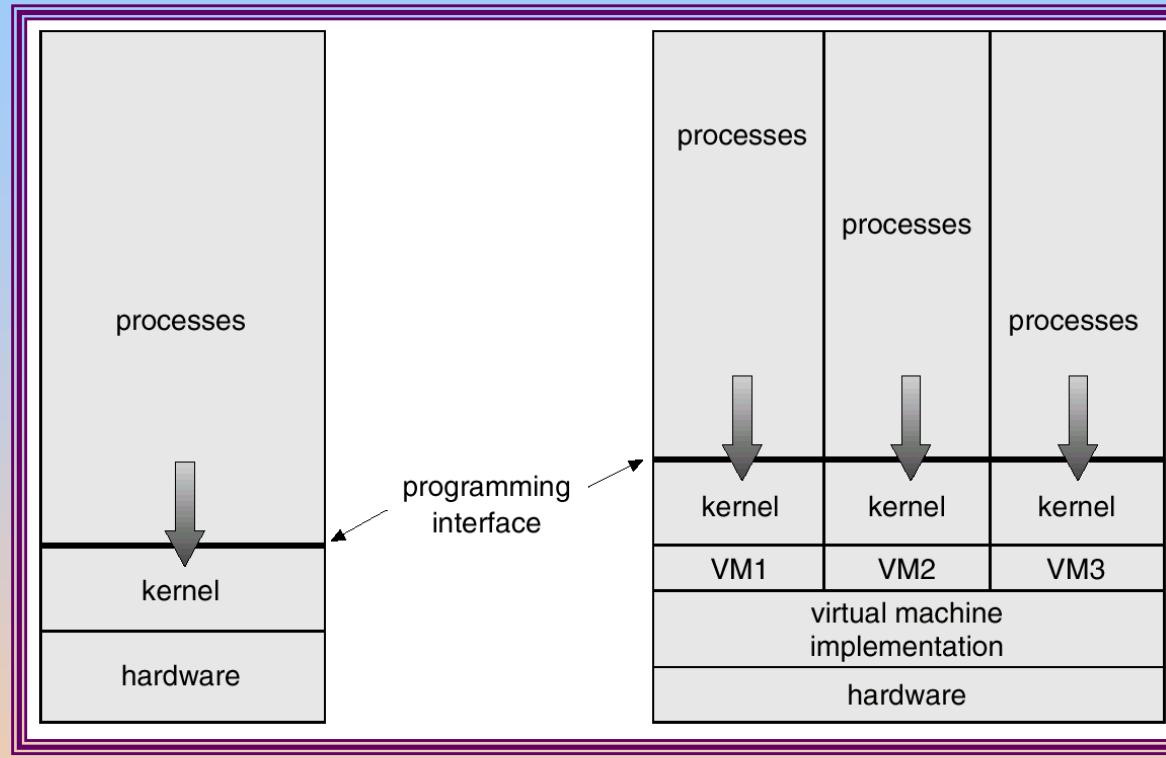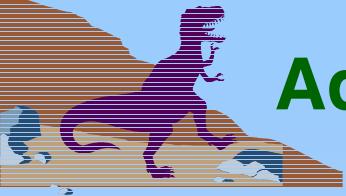
Virtual Machines (Cont.)

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

System Models

Non-virtual Machine   Virtual Machine
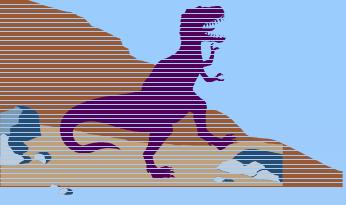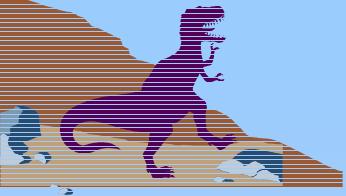
Advantages/Disadvantages of Virtual Machines

- 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.

- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.
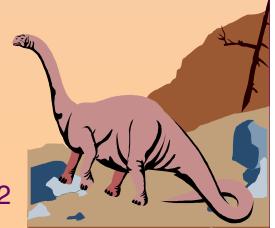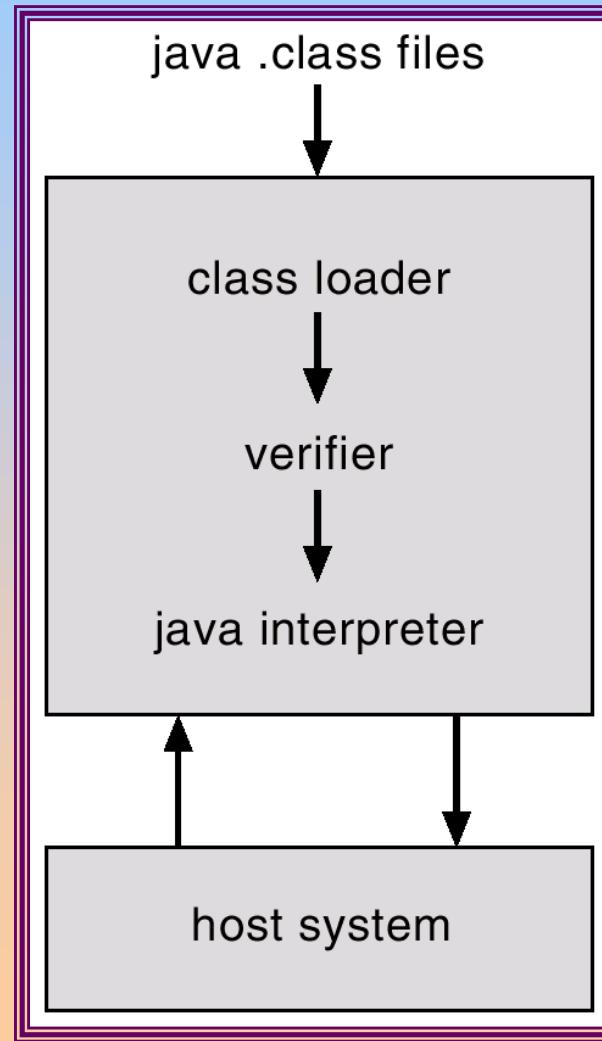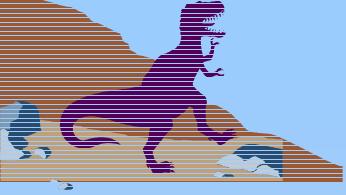
Java Virtual Machine

- Compiled Java programs are platform-neutral bytecodes executed by a Java Virtual Machine (JVM).

- JVM consists of

  - class loader

  - class verifier

  - runtime interpreter

- Just-In-Time (JIT) compilers increase performance

Java Virtual Machine

java .class files

↓

class loader

↓

verifier

↓

java interpreter
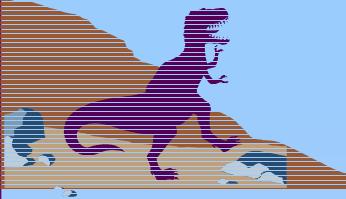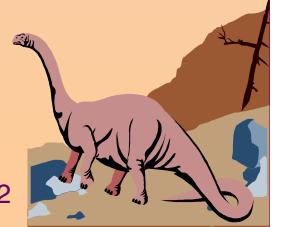
host system

System Design Goals
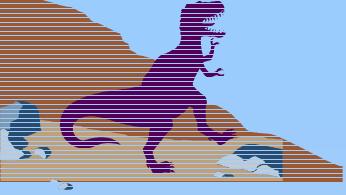
- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast.

- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

Mechanisms and Policies
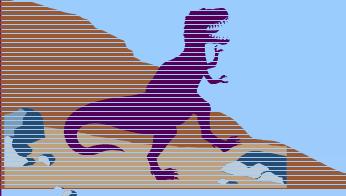
■ Mechanisms determine how to do something, policies decide what will be done.

■ The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
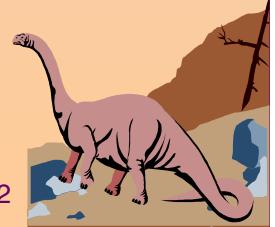
System Implementation
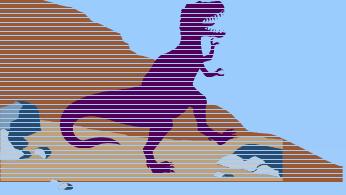
- Traditionally written in assembly language, operating systems can now be written in higher-level languages.

- Code written in a high-level language:
  - can be written faster.
  - is more compact.
  - is easier to understand and debug.

- An operating system is far easier to *port* (move to some other hardware) if it is written in a high-level language.

System Generation (SYSGEN)
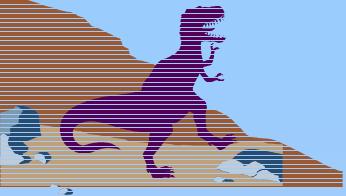
- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.

- 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.
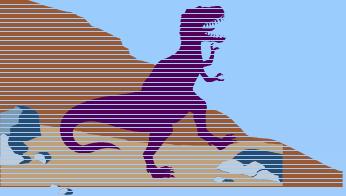
Chapter 4: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

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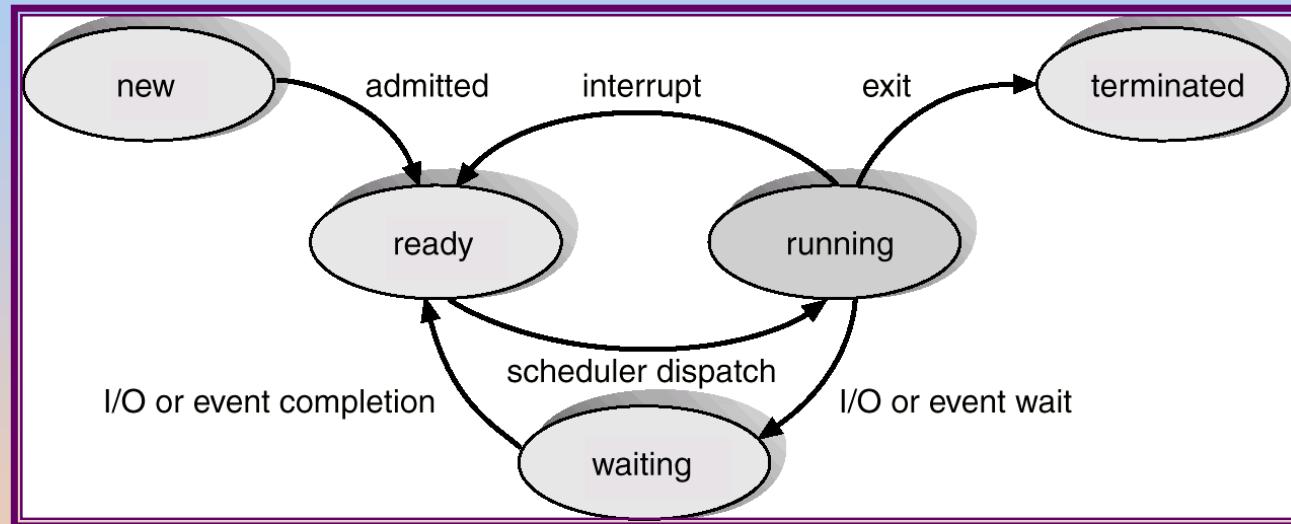
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes: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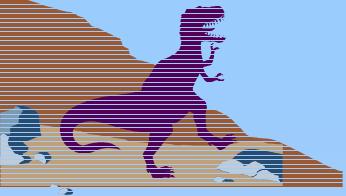
Process State

- 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 process.
  - **terminated**: The process has finished execution.

Diagram of Process State
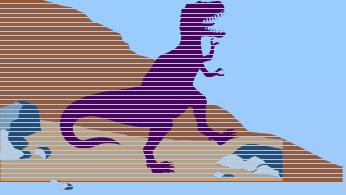
Process Control Block (PCB)
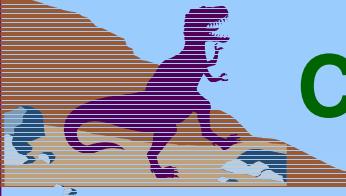
Information associated with each process.

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

Process Control Block (PCB)

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
· · ·	
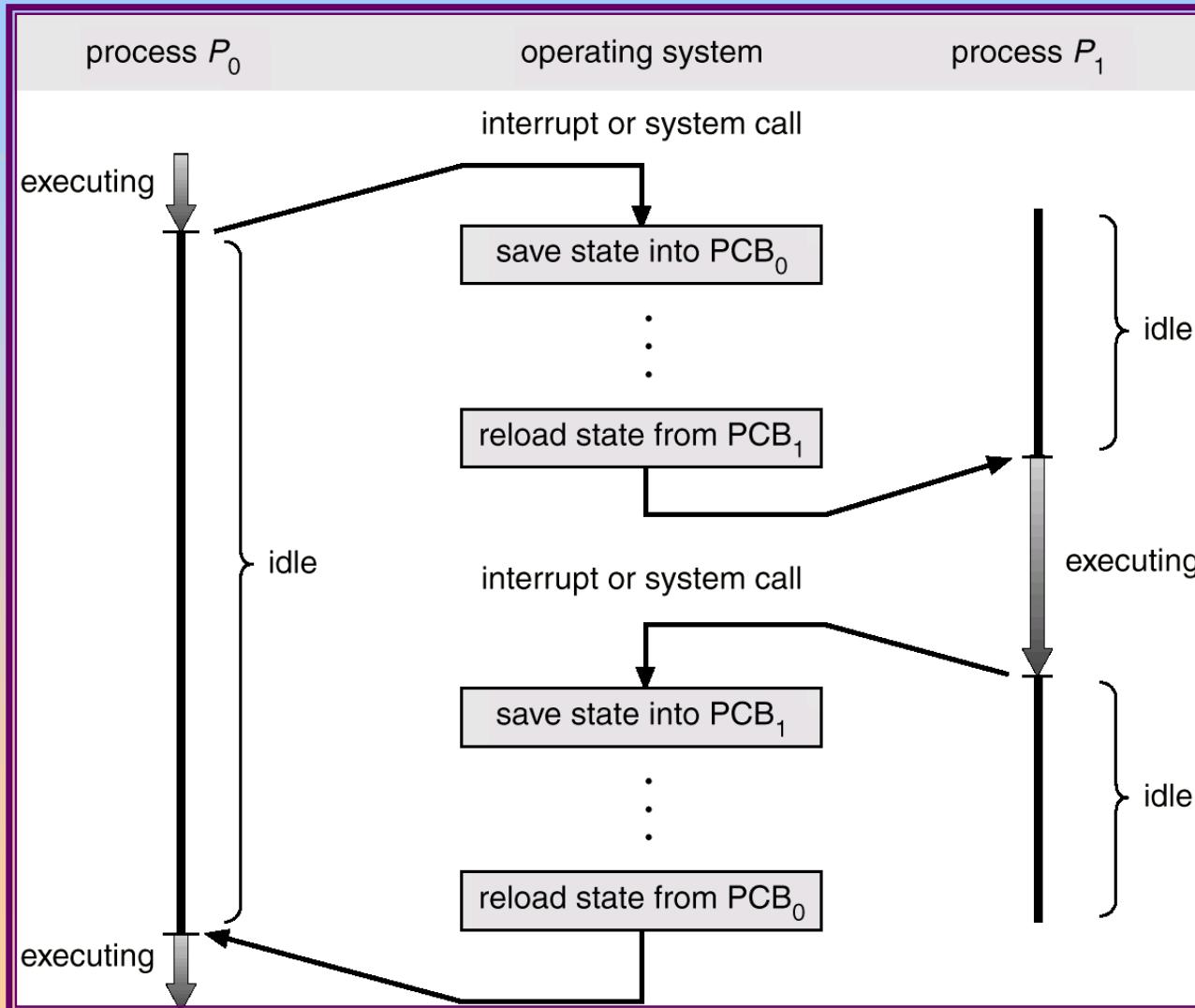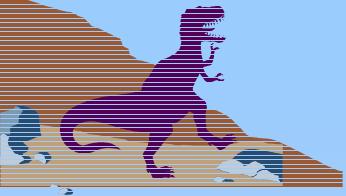
CPU Switch From Process to Process

process P_0	operating system	process P_1
	interrupt or system call	

executing

save state into PCB_0

reload state from PCB_1

idle

executing

interrupt or system call

idle

save state into PCB_1

reload state from PCB_0

idle
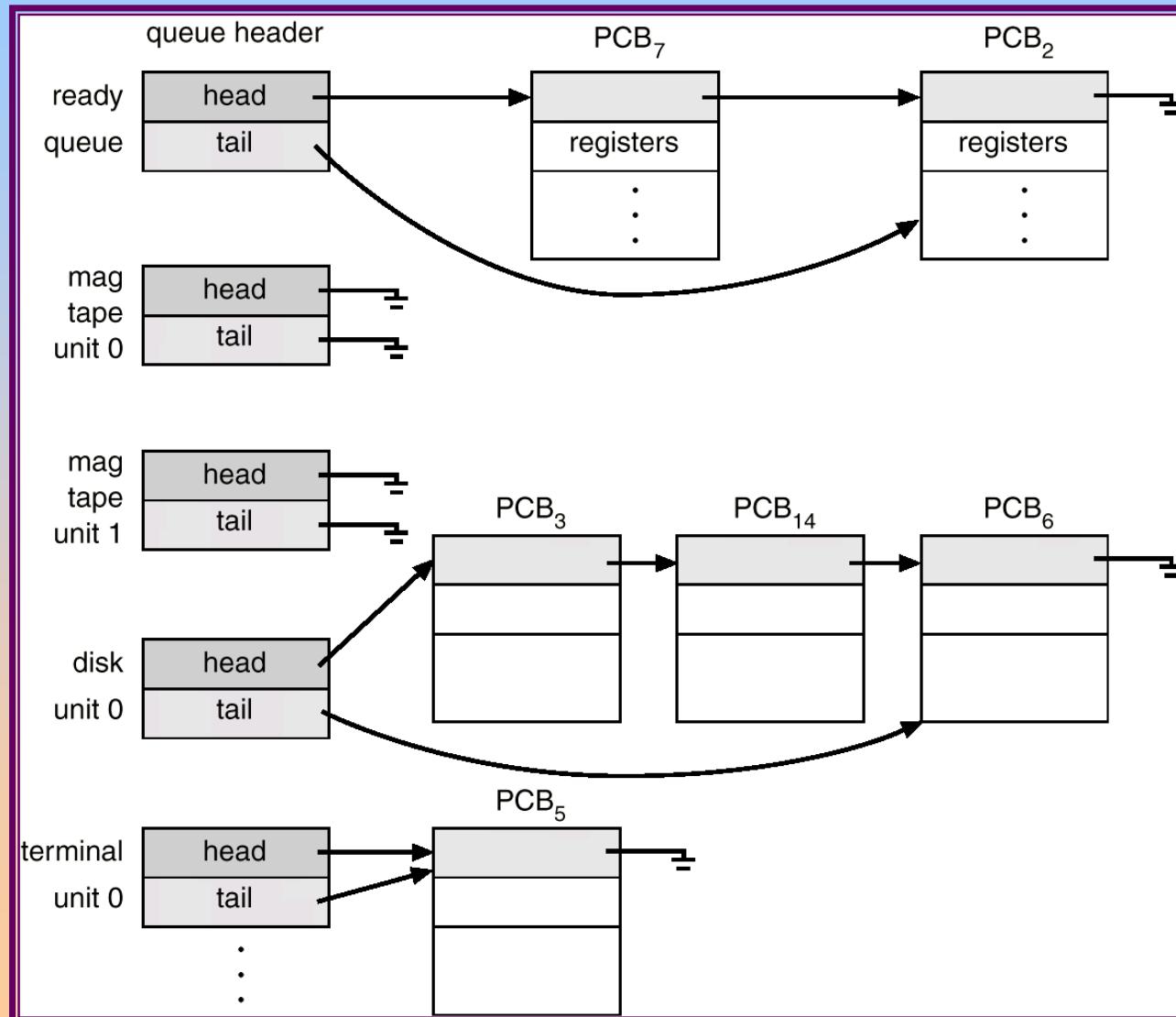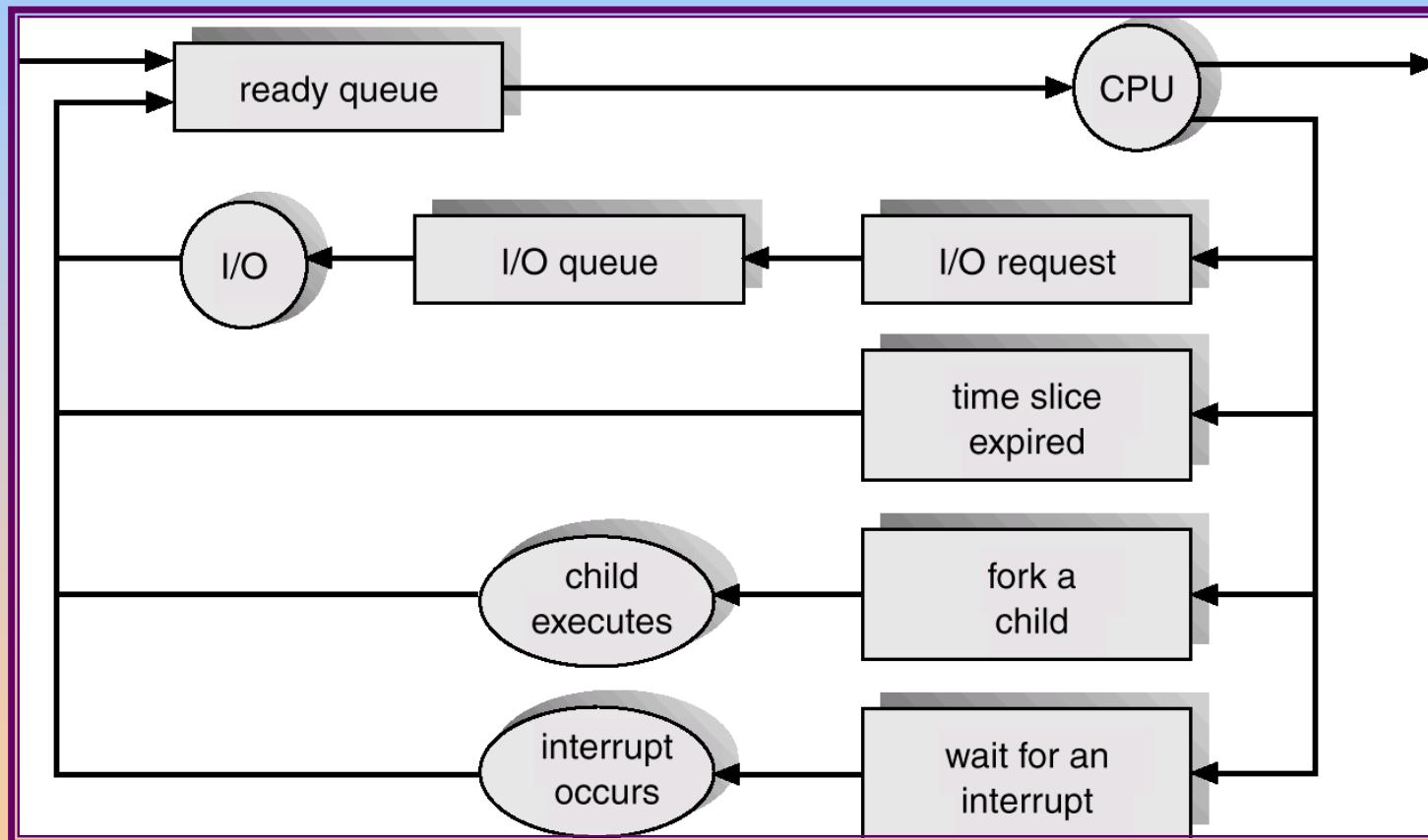
executing

Process Scheduling Queues
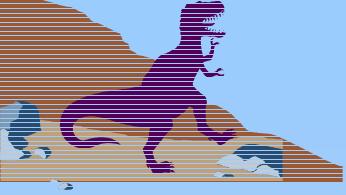
- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.

Representation of Process Scheduling
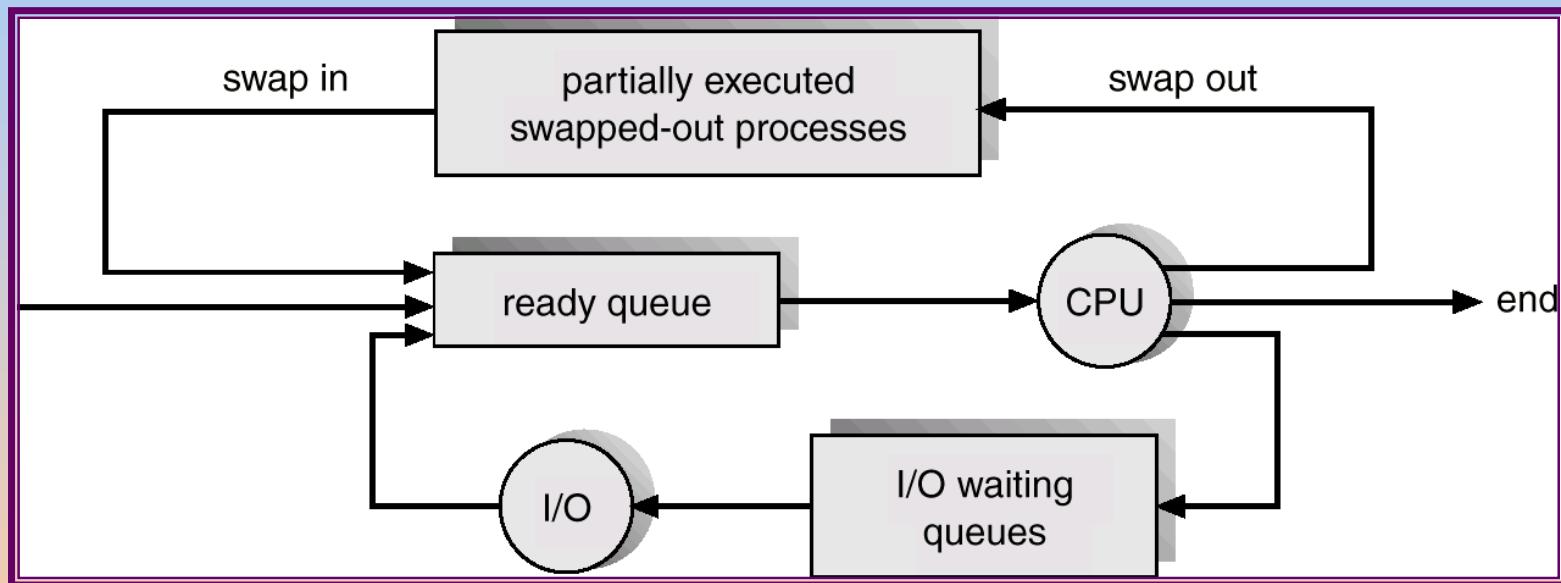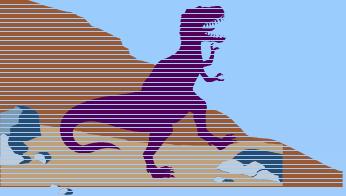
Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.
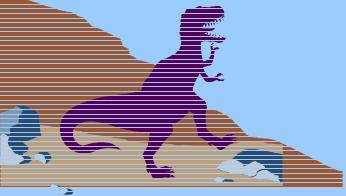
Addition of Medium Term Scheduling
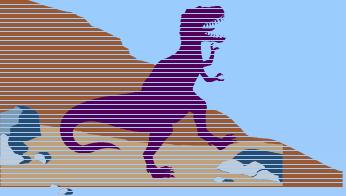
Schedulers (Cont.)

■ Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).

■ Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow).

■ The long-term scheduler controls the *degree of multiprogramming.*

■ Processes can be described as either:

  ✦ I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts.

  ✦ *CPU-bound process* – spends more time doing computations; few very long CPU bursts.
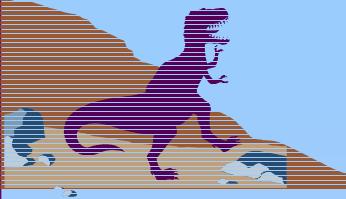
Context Switch

- 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.

- Context-switch time is overhead; the system does no useful work while switching.

- Time dependent on hardware support.

Process Creation

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

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

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
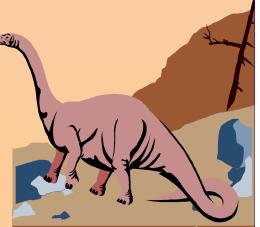
Process Creation (Cont.)
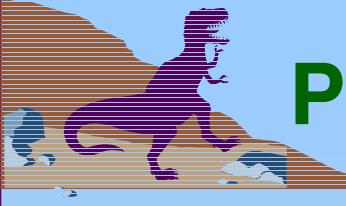
- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program.
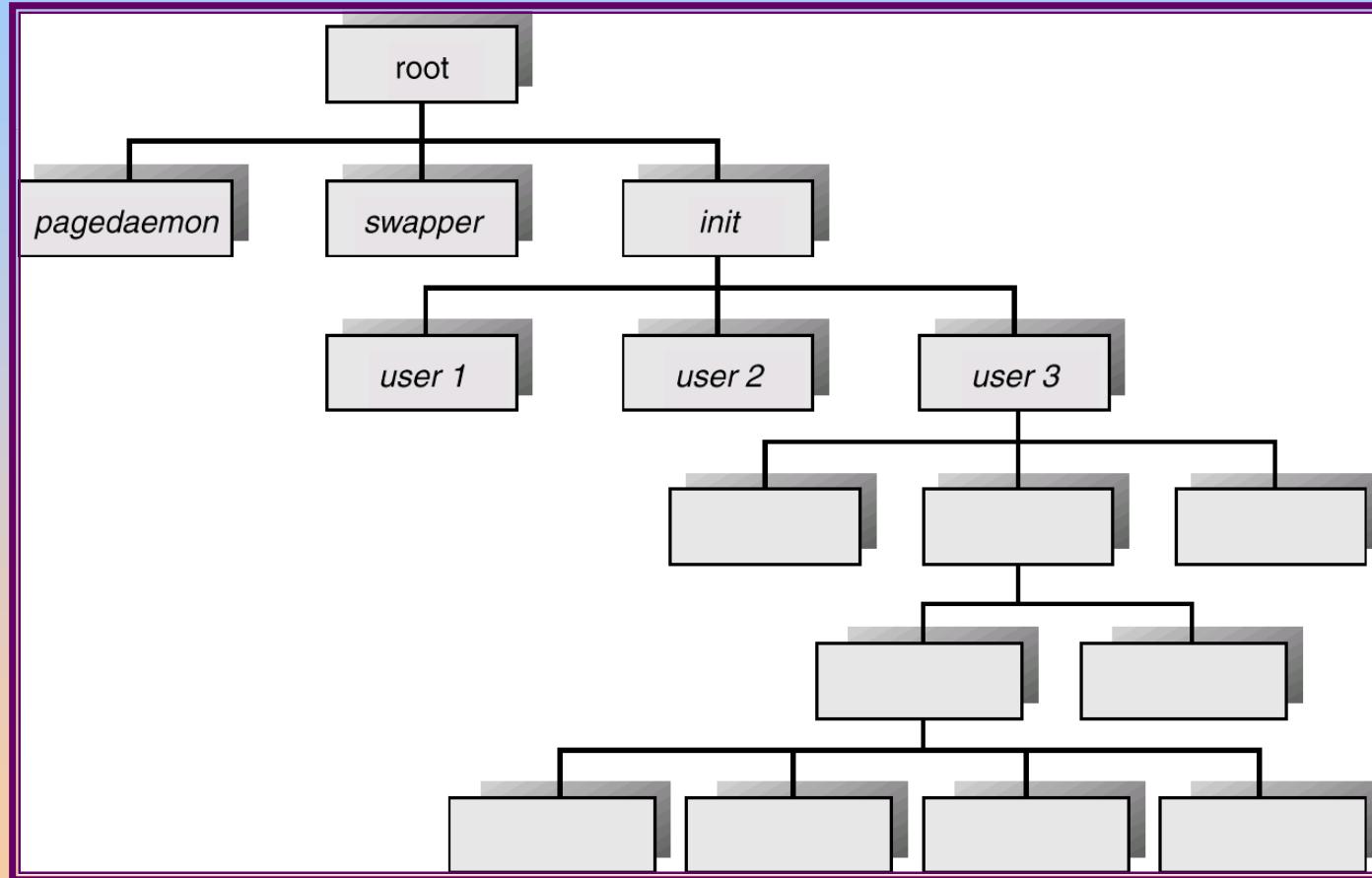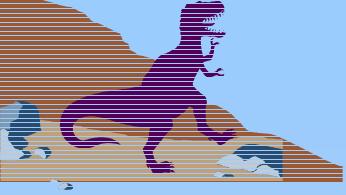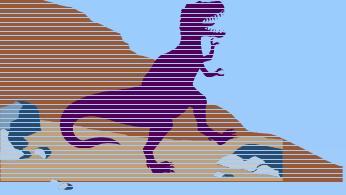
Processes Tree on a UNIX System
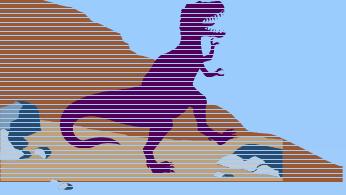
Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
    - ✦ Output data from child to parent (via **wait**).
    - ✦ Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
    - ✦ Child has exceeded allocated resources.
    - ✦ Task assigned to child is no longer required.
    - ✦ Parent is exiting.
        - ✔ Operating system does not allow child to continue if its parent terminates.
        - ✔ Cascading termination.
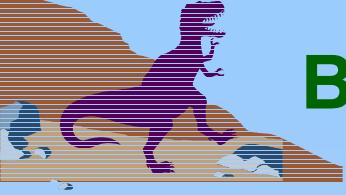
Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.

- *Cooperating* process can affect or be affected by the execution of another process

- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

■ Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.

✦ *unbounded-buffer* places no practical limit on the size of the buffer.

✦ *bounded-buffer* assumes that there is a fixed buffer size.
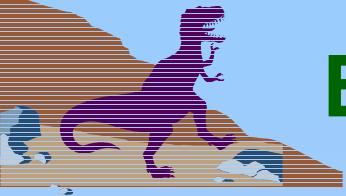
Bounded-Buffer – Shared-Memory Solution

- Shared data

  #define BUFFER_SIZE 10

  Typedef struct {

  . . .
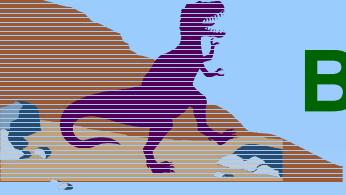
  } item;

  item buffer[BUFFER_SIZE];

  int in = 0;

  int out = 0;

- Solution is correct, but can only use BUFFER_SIZE-1 elements
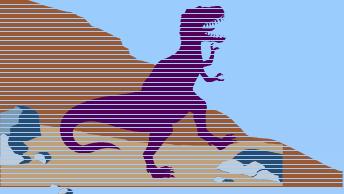
Bounded-Buffer – Producer Process

```
item nextProduced;

while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```
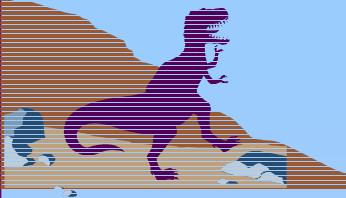
Bounded-Buffer – Consumer Process

```
item nextConsumed;

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

Interprocess Communication (IPC)
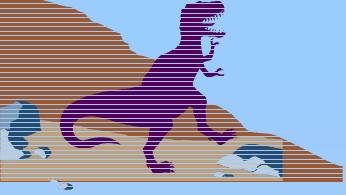
- Mechanism for processes to communicate and to synchronize their actions.

- Message system – processes communicate with each other without resorting to shared variables.

- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)

- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions
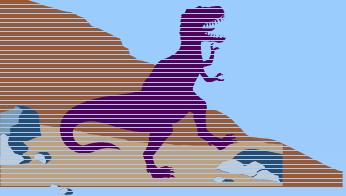
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
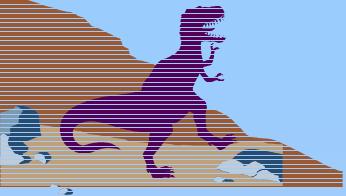
Direct Communication

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

Indirect Communication

■ Messages are directed and received from mailboxes (also referred to as ports).

✦ Each mailbox has a unique id.

✦ Processes can communicate only if they share a mailbox.

■ Properties of communication link

✦ Link established only if processes share a common mailbox

✦ A link may be associated with many processes.

✦ Each pair of processes may share several communication links.

✦ Link may be unidirectional or bi-directional.

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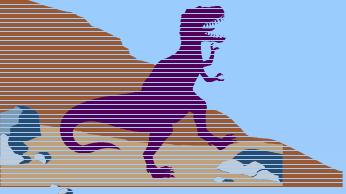
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

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

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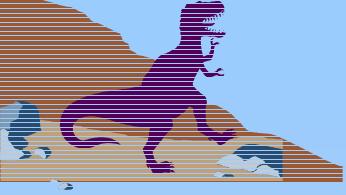
Indirect Communication

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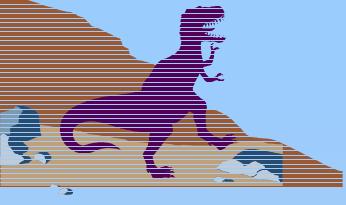
- Solutions
    - Allow a link to be associated with at most two processes.
    - Allow only one process at a time to execute a receive operation.
    - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
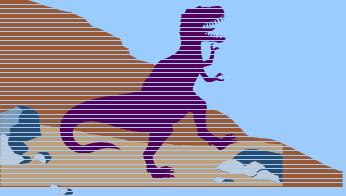
Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.

Buffering
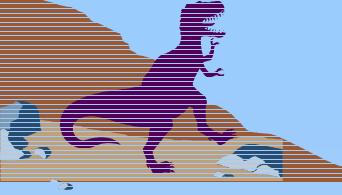
■ Queue of messages attached to the link; implemented in one of three ways.

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous).

2. Bounded capacity – finite length of n messages
   Sender must wait if link full.

3. Unbounded capacity – infinite length
   Sender never waits.

Client-Server Communication
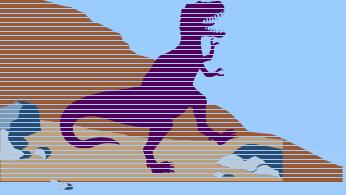
- Sockets

- Remote Procedure Calls

- Remote Method Invocation (Java)
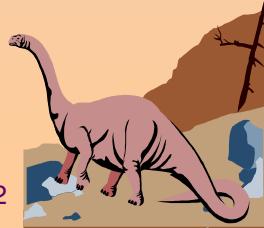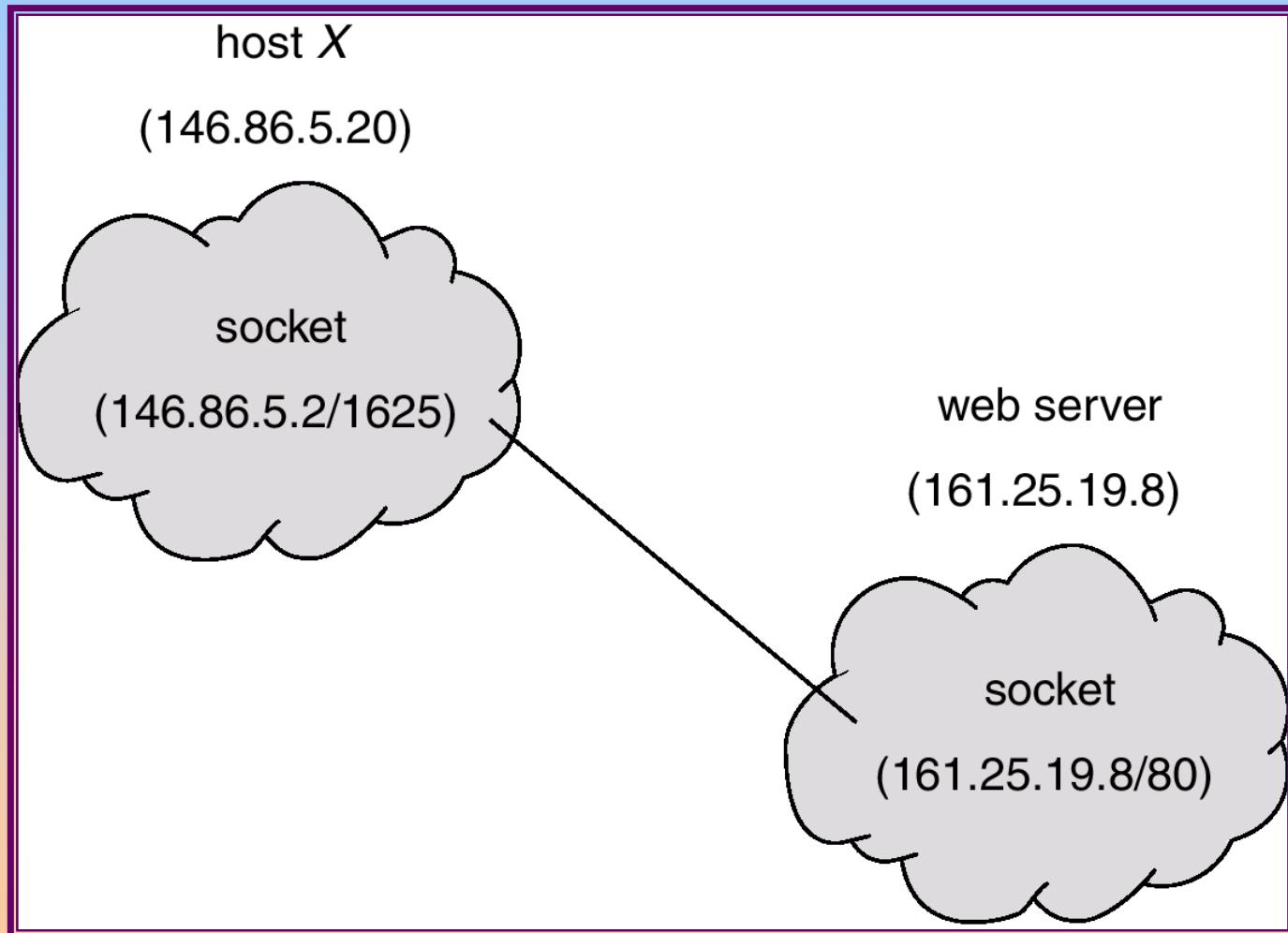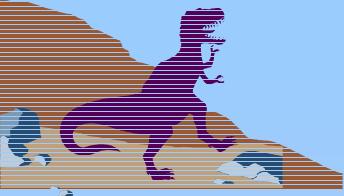
Sockets

- 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.

Socket Communication

host *X*

(146.86.5.20)

socket

(146.86.5.2/1625)

web server

(161.25.19.8)

socket

(161.25.19.8/80)
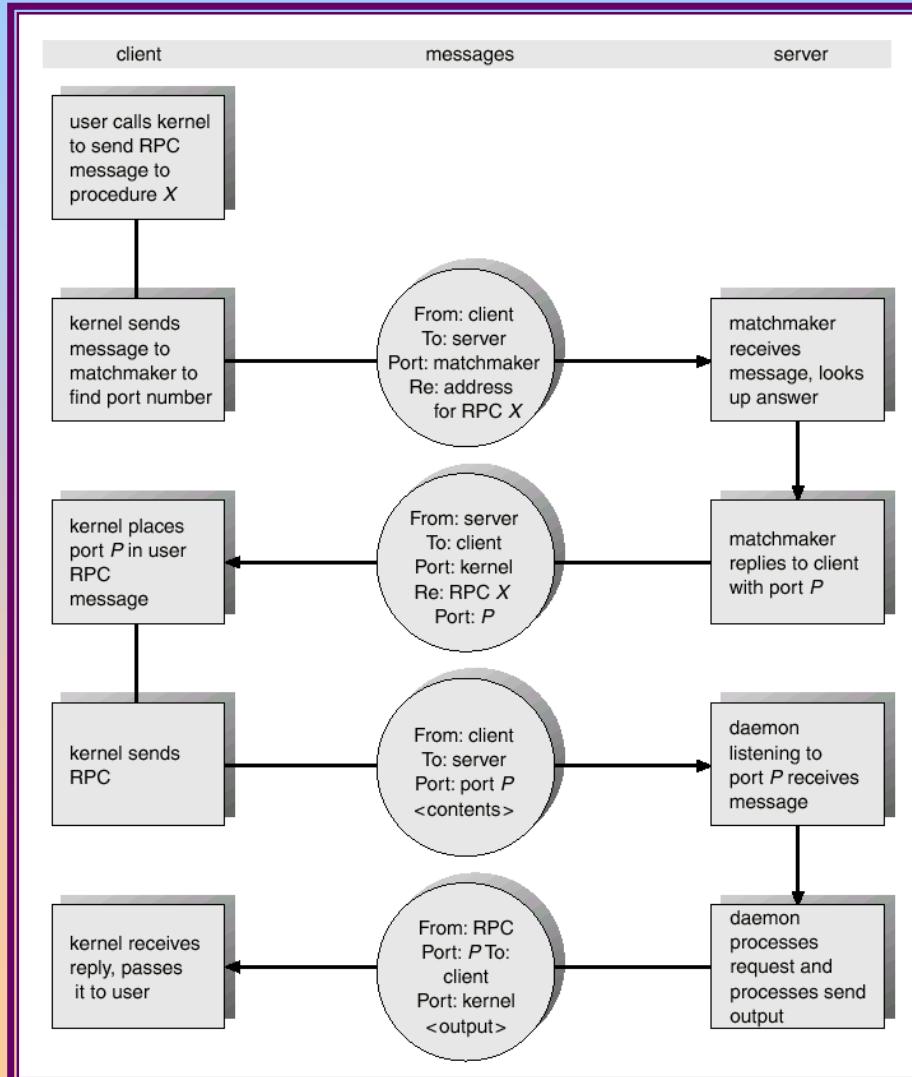
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.

- **Stubs** – client-side proxy for the actual procedure on the server.

- The client-side stub locates the server and *marshalls* the parameters.

- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

Execution of RPC

client	messages	server
user calls kernel to send RPC message to procedure X		
kernel sends message to matchmaker to find port number	From: client To: server Port: matchmaker Re: address for RPC X	matchmaker receives message, looks up answer
kernel places port P in user RPC message	From: server To: client Port: kernel Re: RPC X Port: P	matchmaker replies to client with port P
kernel sends RPC	From: client To: server Port: port P <contents>	daemon listening to port P receives message
kernel receives reply, passes it to user	From: RPC Port: P To: client Port: kernel <output>	daemon processes request and processes send output

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.
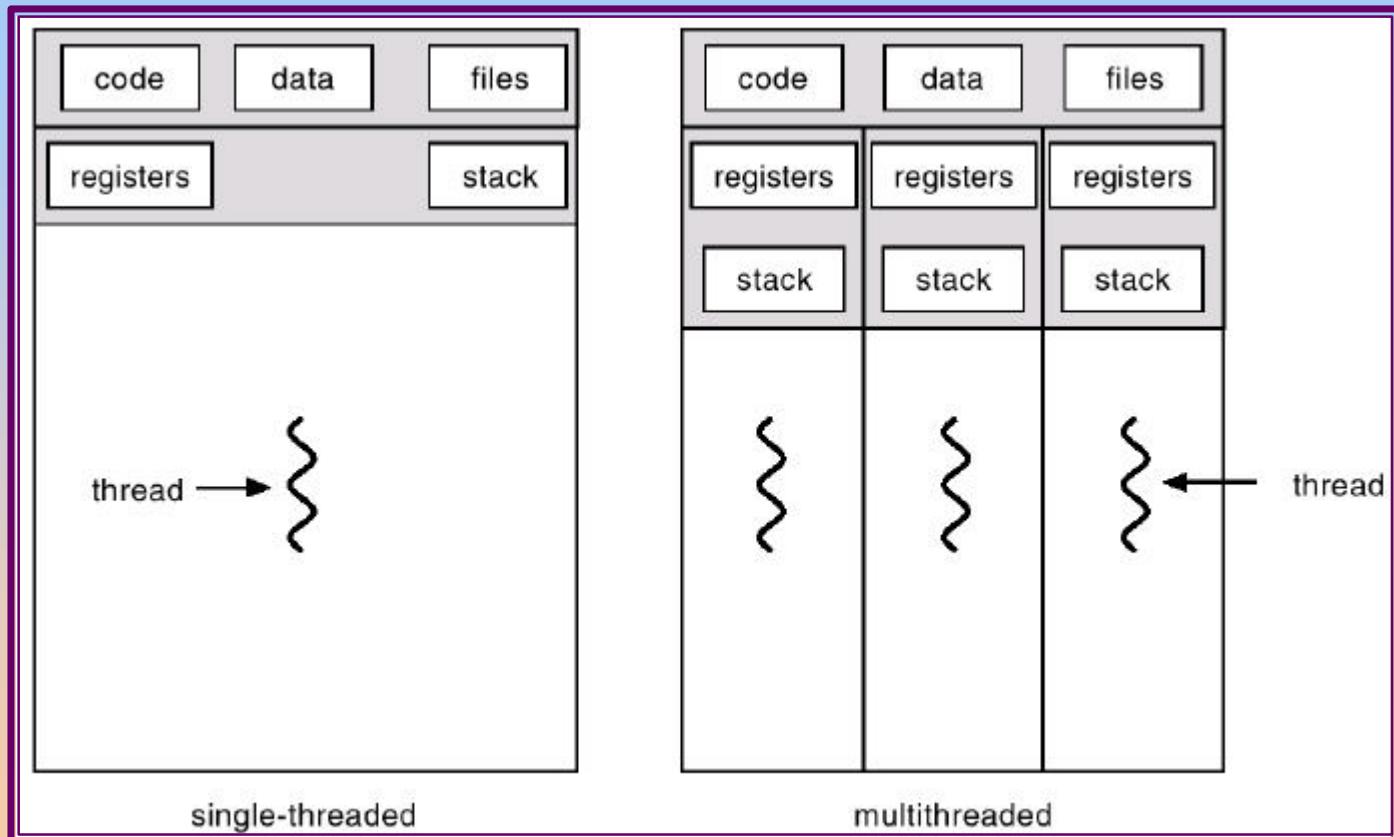
Marshalling Parameters

client

val = server.someMethod(A,B)

stub

remote object

boolean someMethod (Object x, Object y)
{
    implementation of someMethod
    . . .
}

skeleton

A, B, someMethod

boolean return value
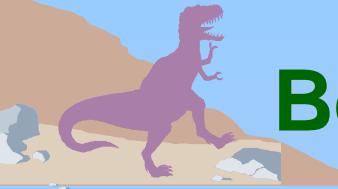
Chapter 5: Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Solaris 2 Threads
- Windows 2000 Threads
- Linux Threads
- Java Threads

Single and Multithreaded Processes

code	data	files

| registers | | stack |

thread →

single-threaded

code	data	files

| registers | registers | registers |
| stack | stack | stack |

← thread

multithreaded

Benefits

- Responsiveness

- Resource Sharing

- Economy

- Utilization of MP Architectures

User Threads

- Thread management done by user-level threads library

- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Kernel Threads

- Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux
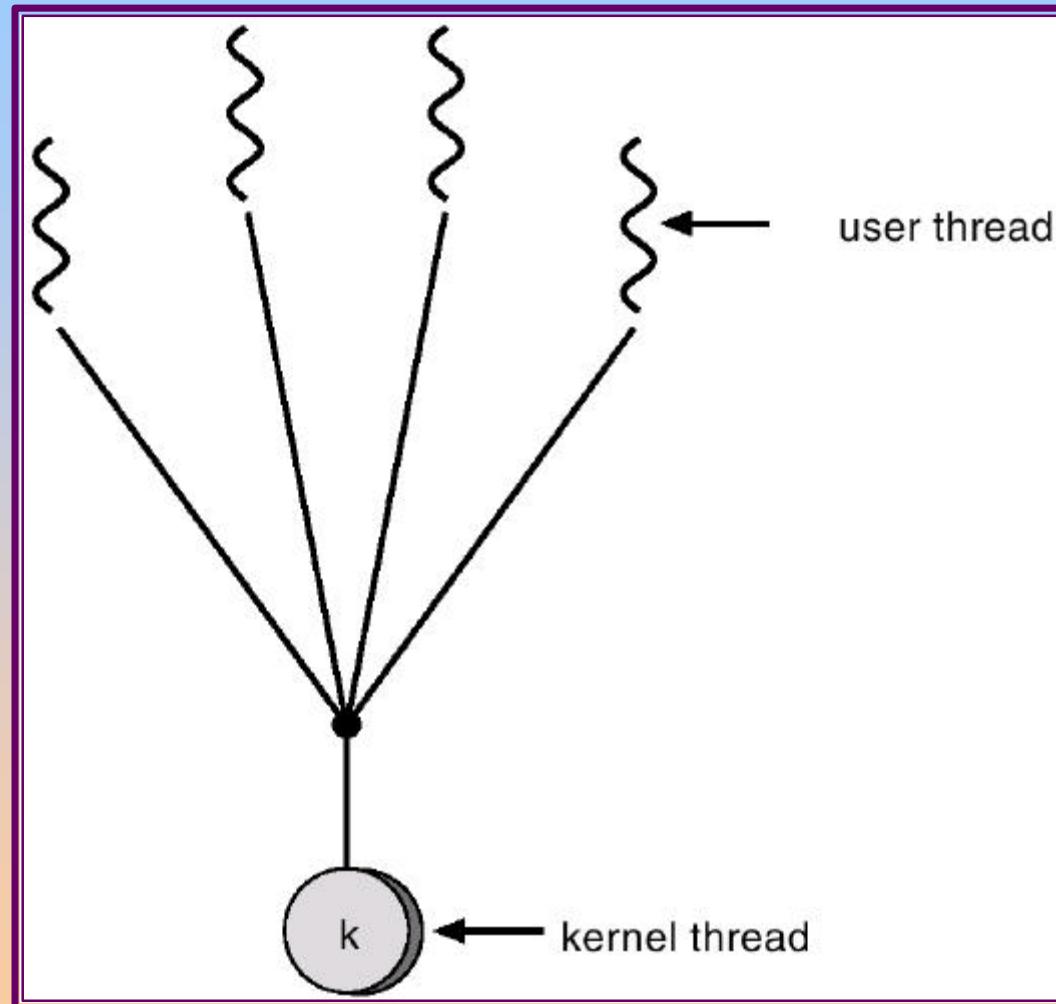
Multithreading Models

- Many-to-One

- One-to-One

- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread.

- Used on systems that do not support kernel threads.
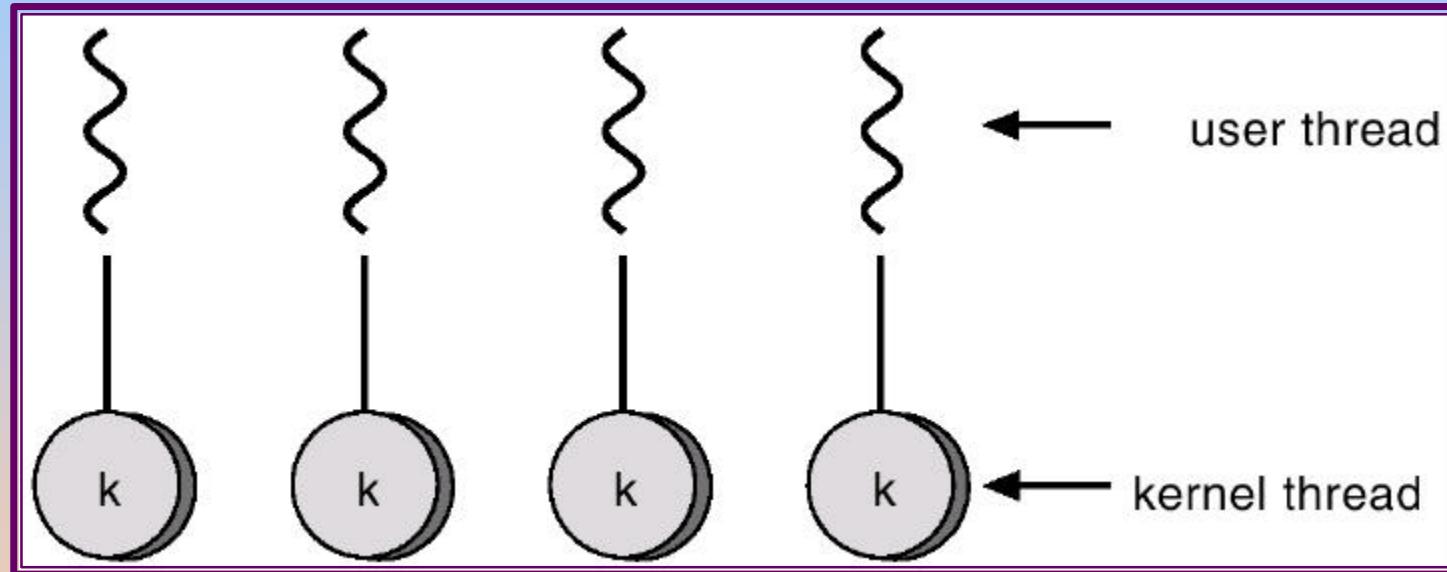
Many-to-One Model

One-to-One

- Each user-level thread maps to kernel thread.

- Examples
  - Windows 95/98/NT/2000
  - OS/2

One-to-one Model

user thread
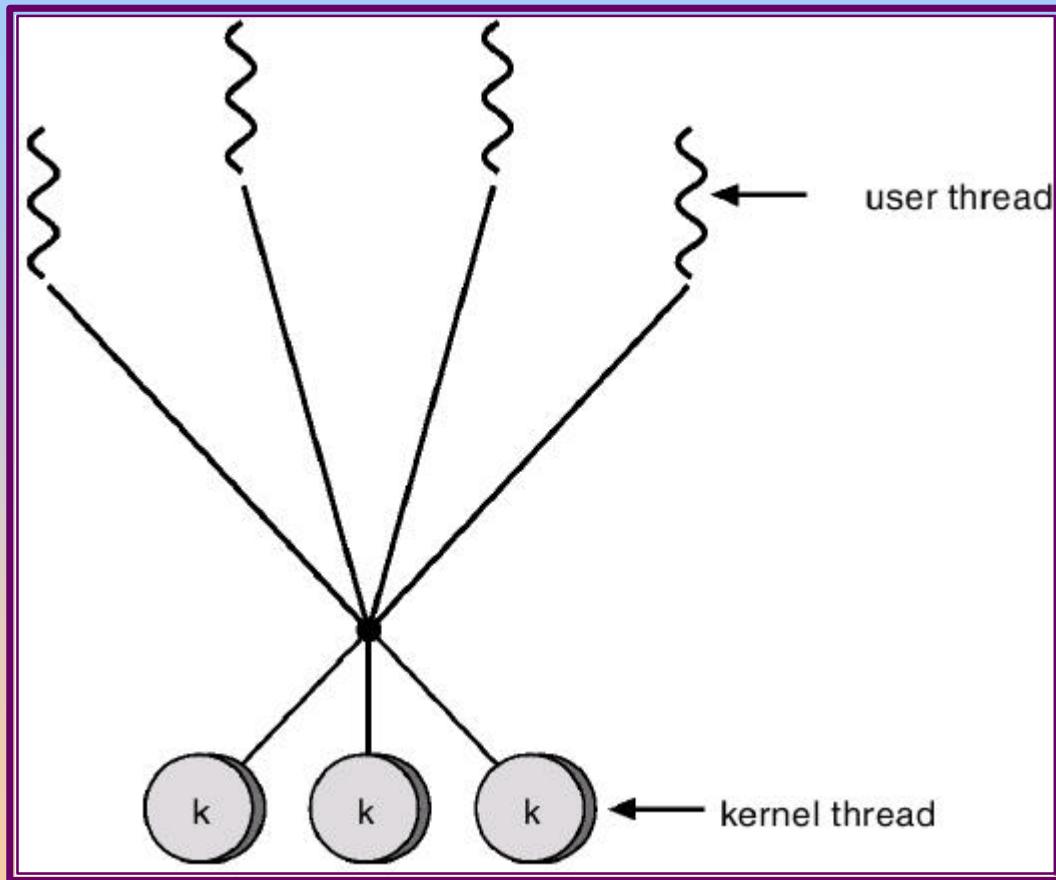
kernel thread

Many-to-Many Model

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

- Allows the operating system to create a sufficient number of kernel threads.

- Solaris 2

- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model
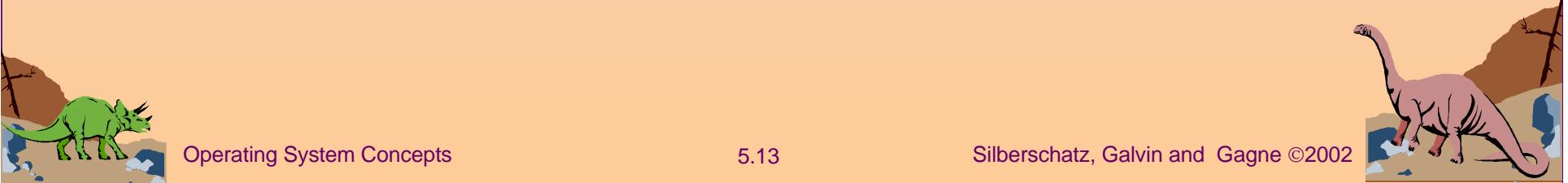
Threading Issues
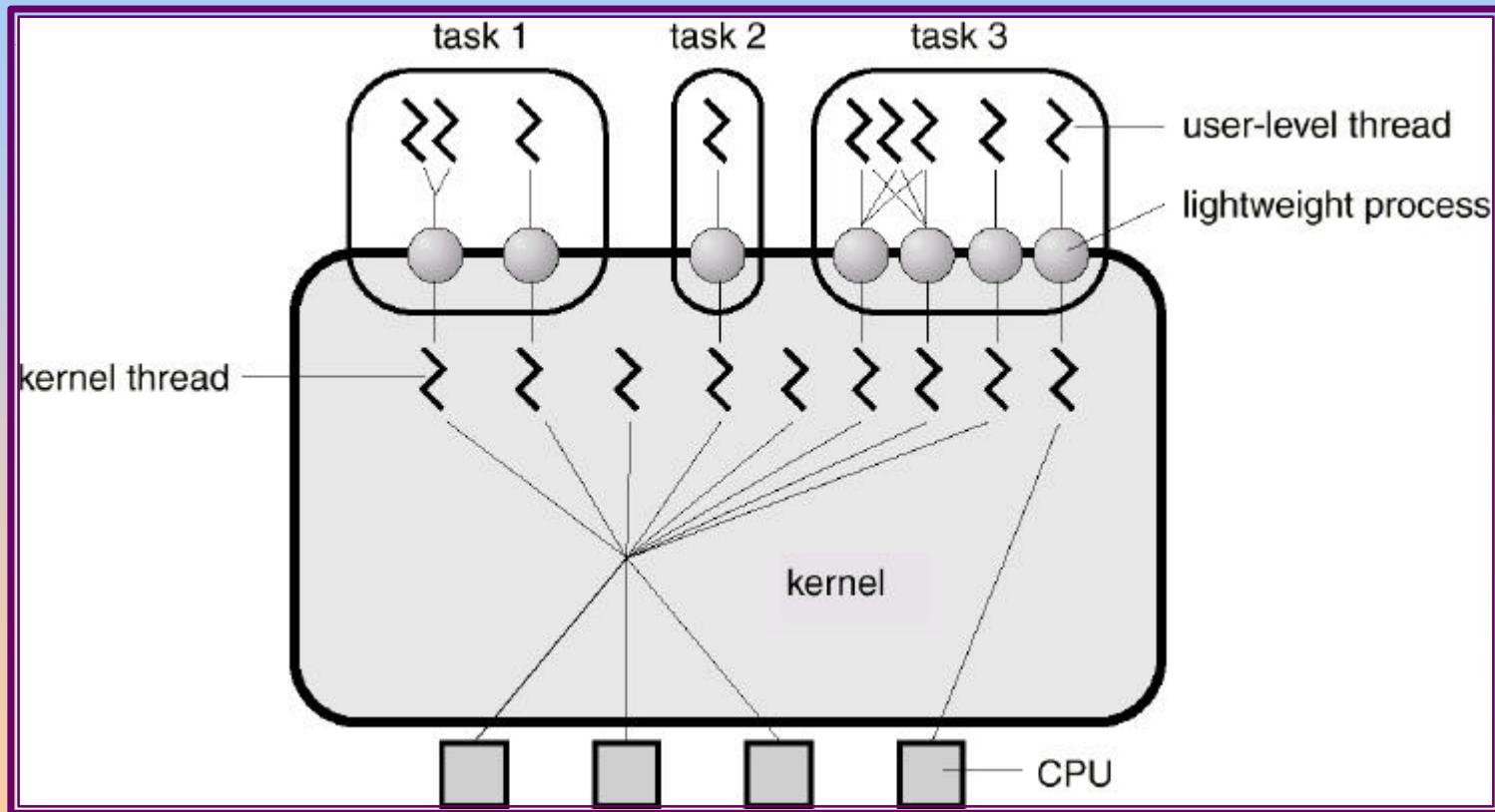
- Semantics of fork() and exec() system calls.
- Thread cancellation.
- Signal handling
- Thread pools
- Thread specific data
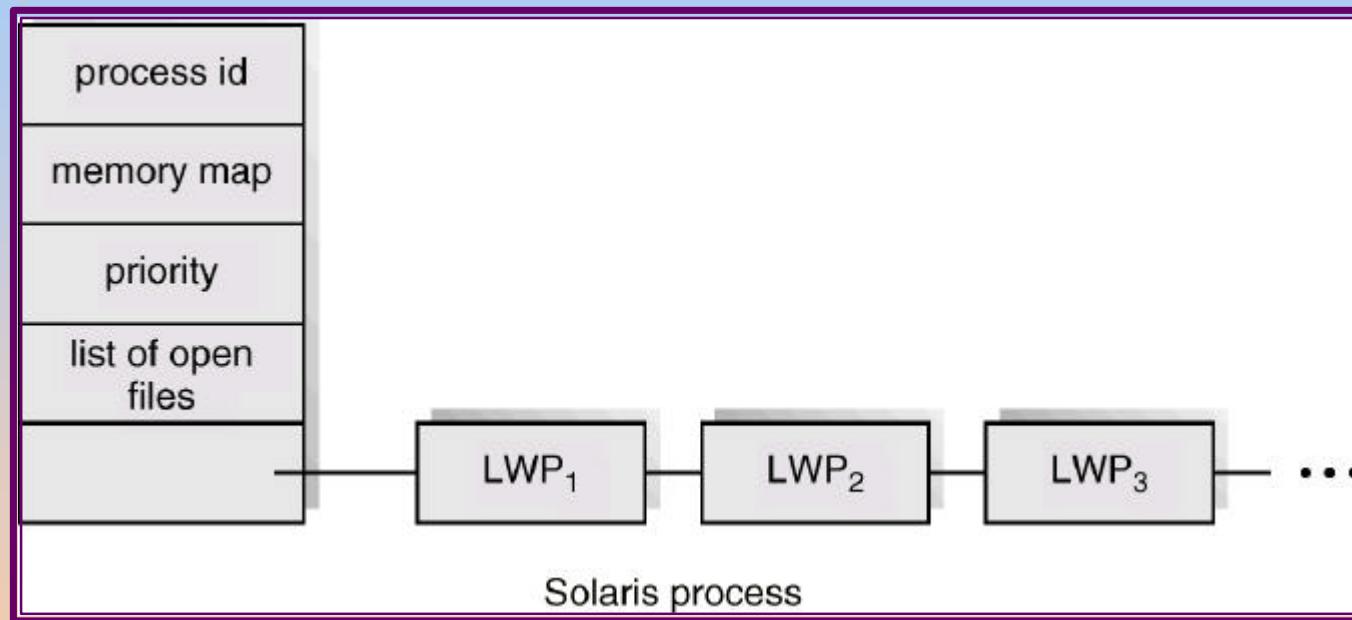
Pthreads

- a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.

- API specifies behavior of the thread library, implementation is up to development of the library.

- Common in UNIX operating systems.

Solaris 2 Threads

Solaris Process

Solaris process

Windows 2000 Threads

- Implements the one-to-one mapping.
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area
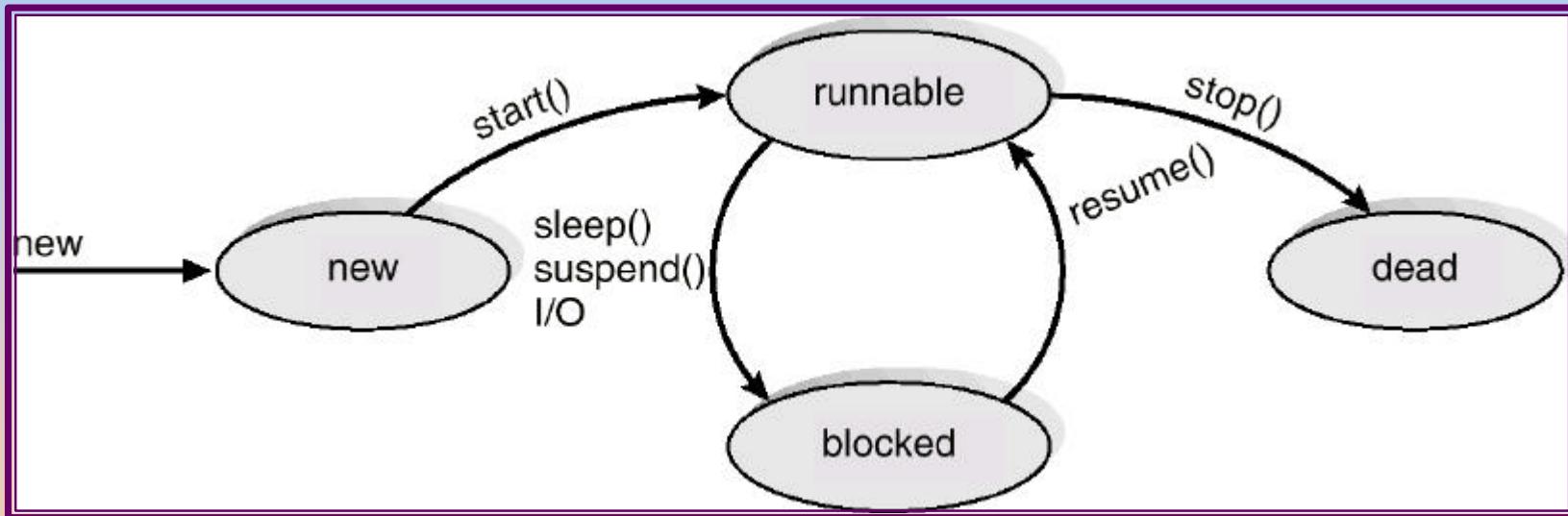
Linux Threads
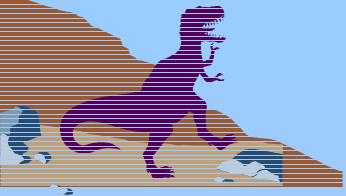
- Linux refers to them as *tasks* rather than *threads*.
- Thread creation is done through clone() system call.
- Clone() allows a child task to share the address space of the parent task (process)

Java Threads

■ Java threads may be created by:

  ✦ Extending Thread class
  ✦ Implementing the Runnable interface

■ Java threads are managed by the JVM.

Java Thread States
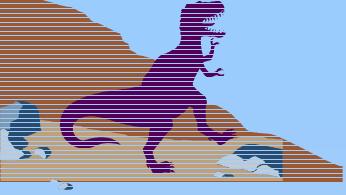
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation
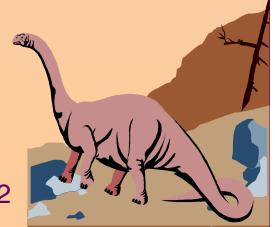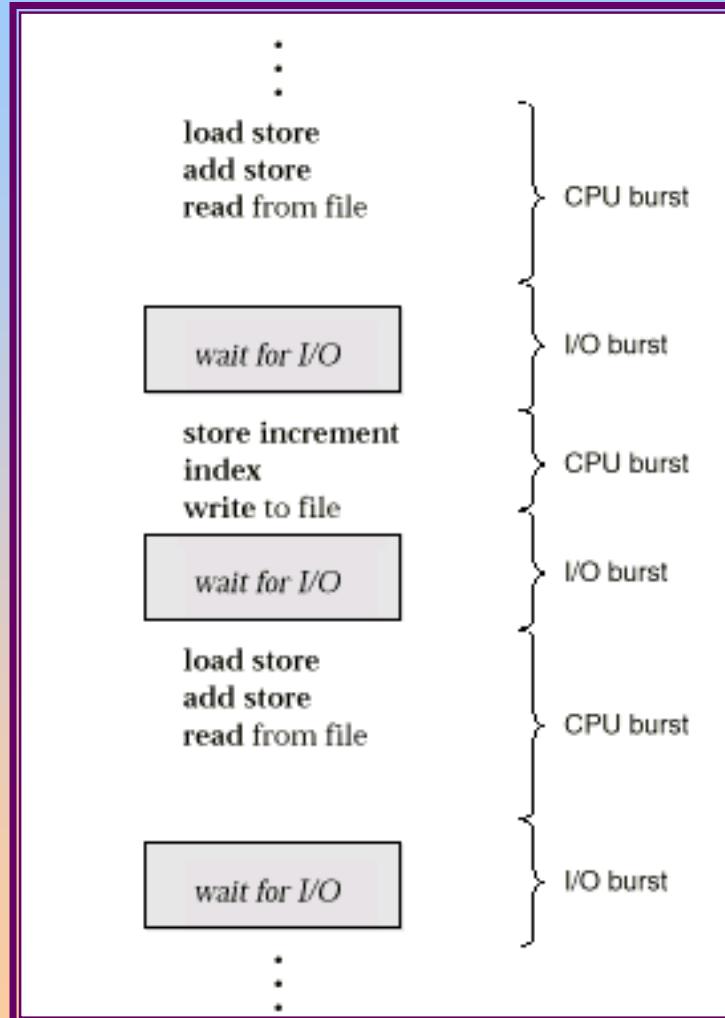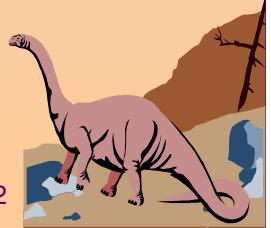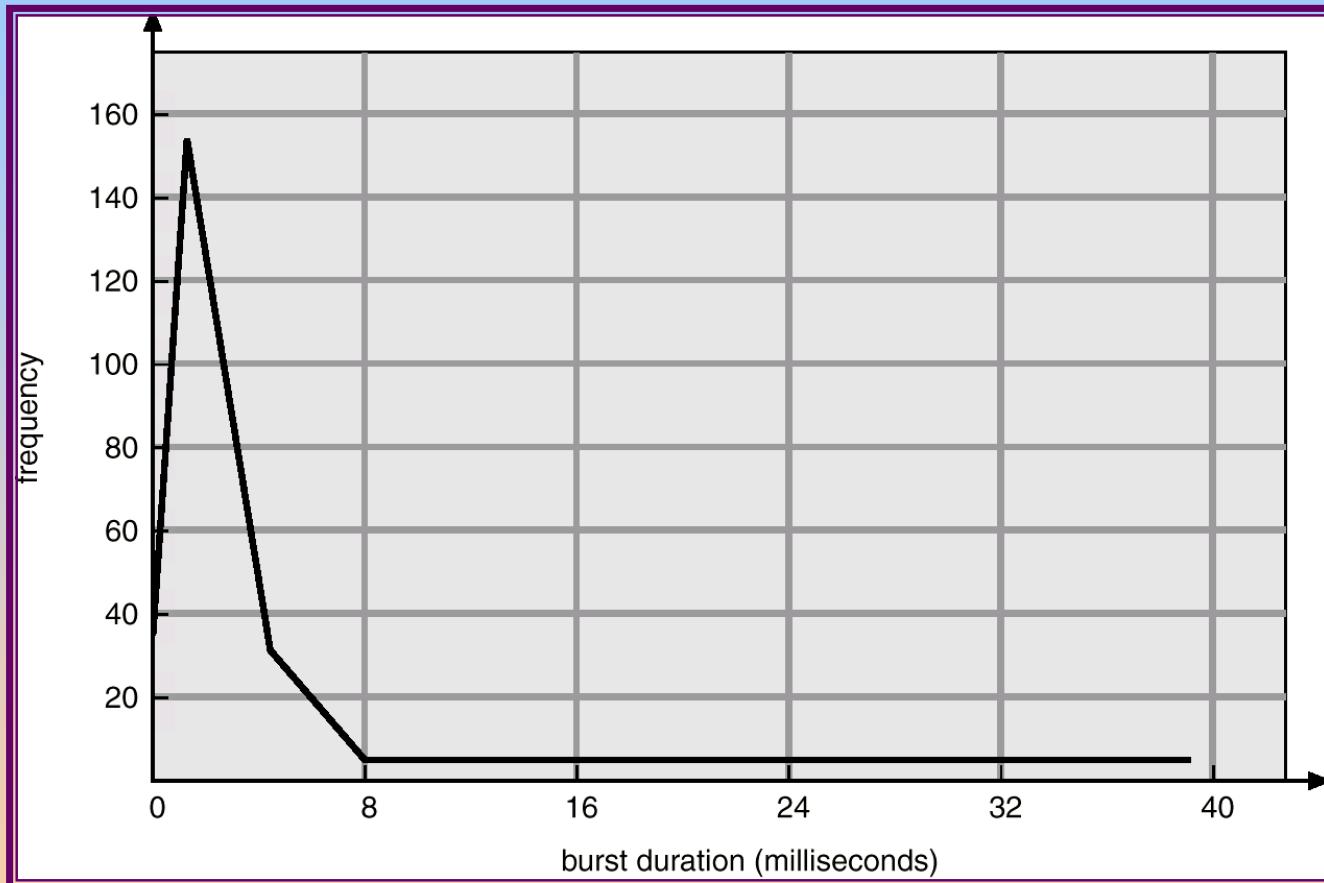
Basic Concepts
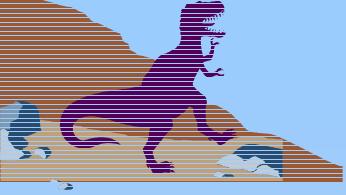
- Maximum CPU utilization obtained with multiprogramming

- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.

- CPU burst distribution

Alternating Sequence of CPU And I/O Bursts
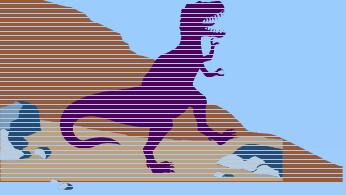
Histogram of CPU-burst Times
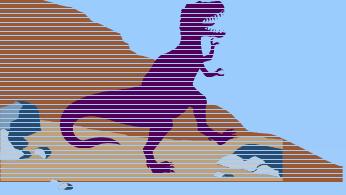
CPU Scheduler

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

6.5   Silberschatz, Galvin and Gagne ©2002
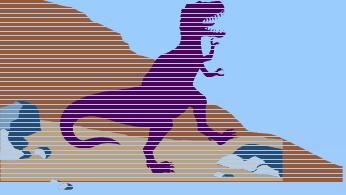
Dispatcher

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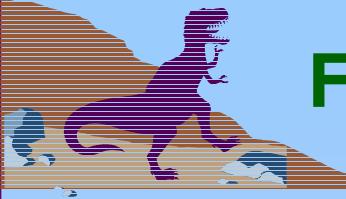
Scheduling Criteria

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

Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling
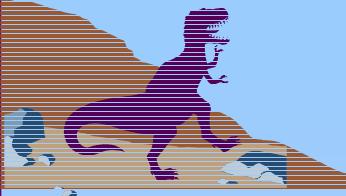
Process	Burst Time
P_1	24
P_2	3
P_3	3

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

P_1		P_2	P_3
0	24	27	30

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

FCFS Scheduling (Cont.)
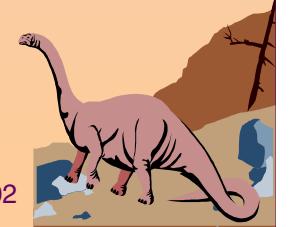
Suppose that the processes arrive in the order
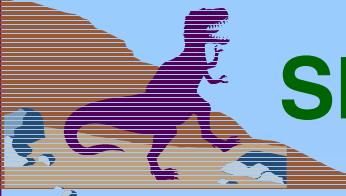
$$P_2, P_3, P_1.$$

■ The Gantt chart for the schedule is:

P$_2$	P$_3$	P$_1$

0　　　　3　　　6　　　　　　　　　　　　　　30

■ Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
■ Average waiting time: $(6 + 0 + 3)/3 = 3$
■ Much better than previous case.
■ *Convoy effect* short process behind long process
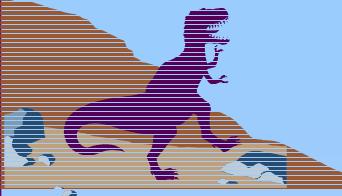
Shortest-Job-First (SJR) Scheduling

■ Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

■ Two schemes:

✦ nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.

✦ preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

■ SJF is optimal – gives minimum average waiting time for a given set of processes.
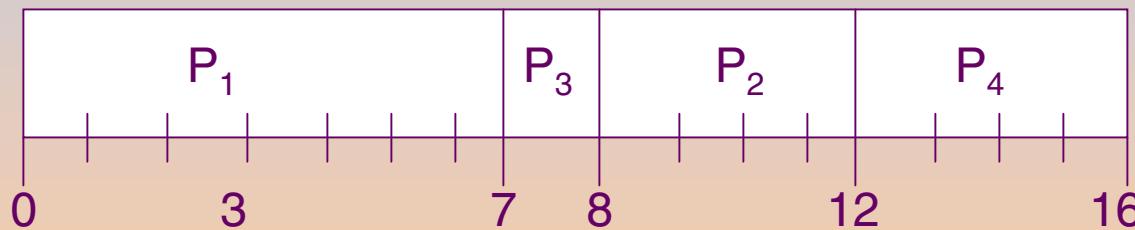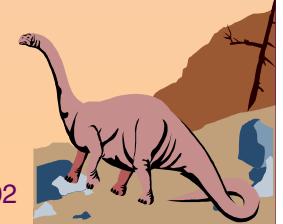
Example of Non-Preemptive SJF
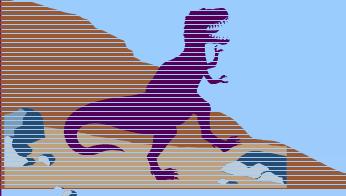
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

■ SJF (non-preemptive)

```
| P1            | P3 | P2       | P4       |
0       3       7  8          12         16
```

■ Average waiting time = (0 + 6 + 3 + 7)/4 - 4
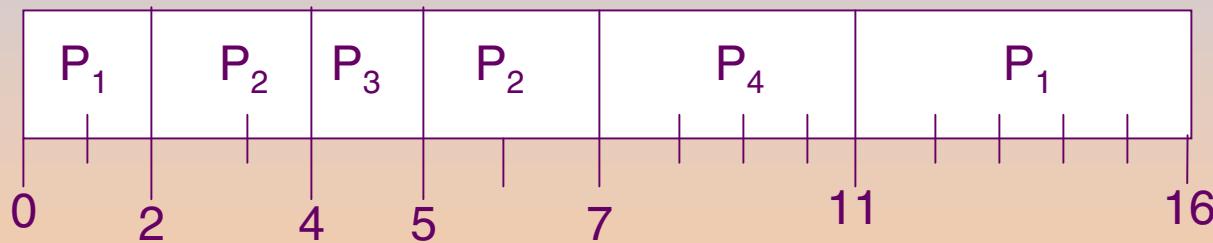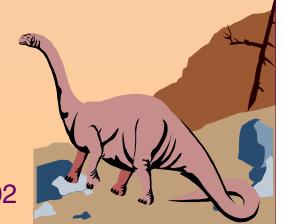
Example of Preemptive SJF
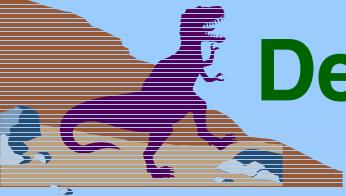
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

P_1	P_2	P_3	P_2	P_4	P_1

0   2   4   5   7   11   16

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

Determining Length of Next CPU Burst
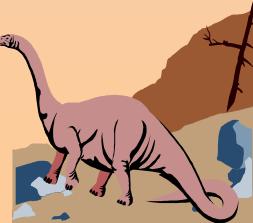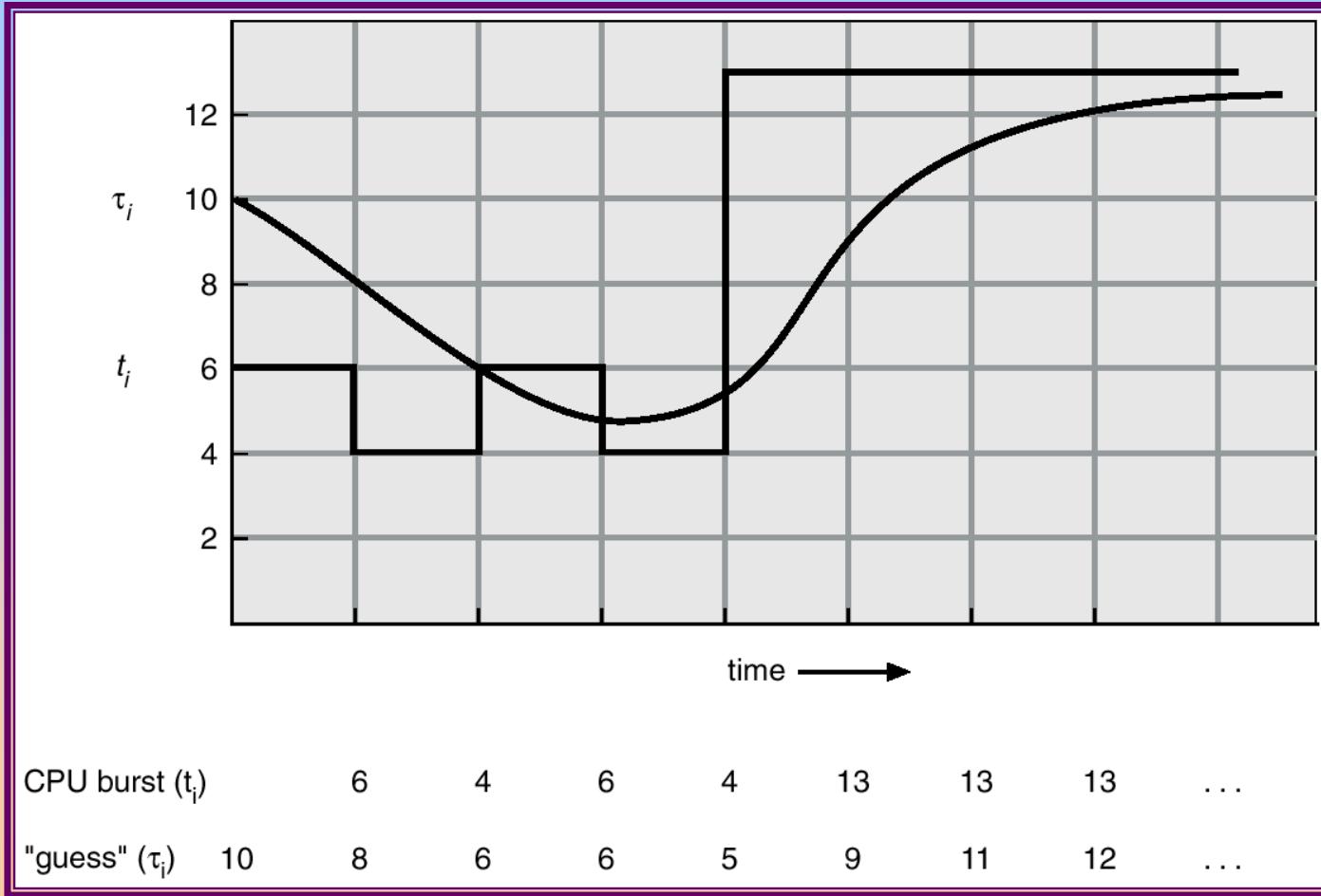
- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.
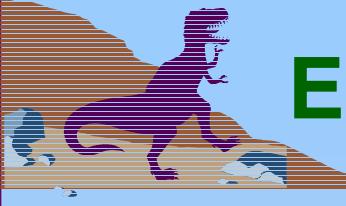
1. t_n = actual lenght of nth CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

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

Prediction of the Length of the Next CPU Burst

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

Examples of Exponential Averaging

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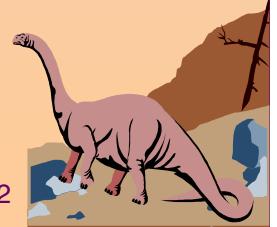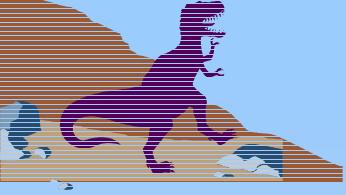
  $$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_n - 1 + \dots$$
  $$+ (1 - \alpha)^j\, \alpha\, t_n - 1 + \dots$$
  $$+ (1 - \alpha)^{n=1}\, t_n\, \tau_0$$
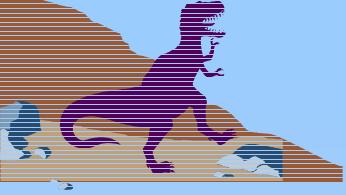
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

Priority Scheduling

- A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority).
  - Preemptive
  - nonpreemptive

- SJF is a priority scheduling where priority is the predicted next CPU burst time.

- Problem $\equiv$ Starvation – low priority processes may never execute.

- Solution $\equiv$ Aging – as time progresses increase the priority of the process.

Round Robin (RR)
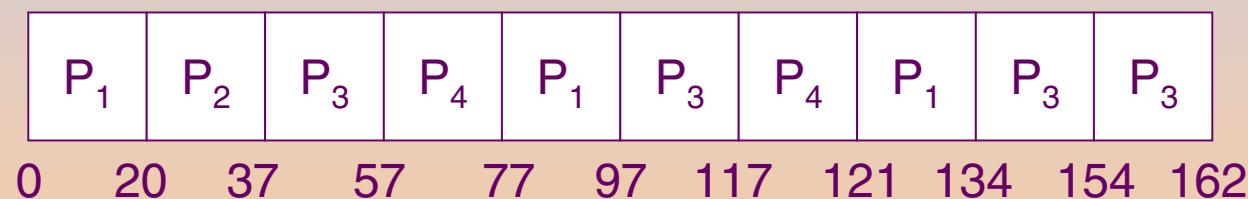
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.

- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.
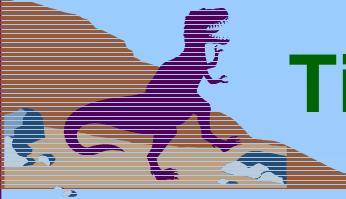
Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- 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
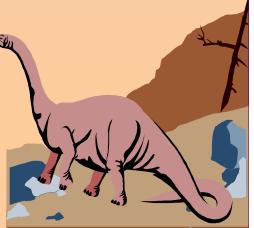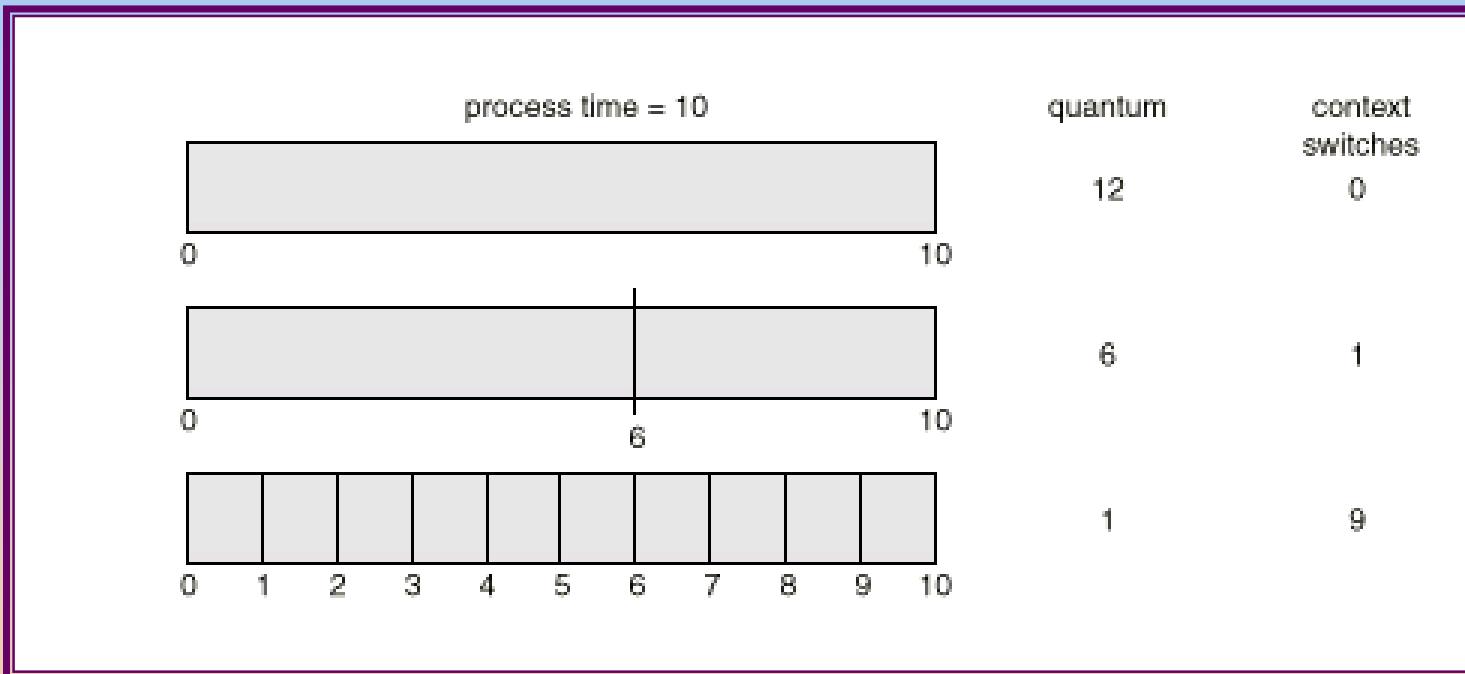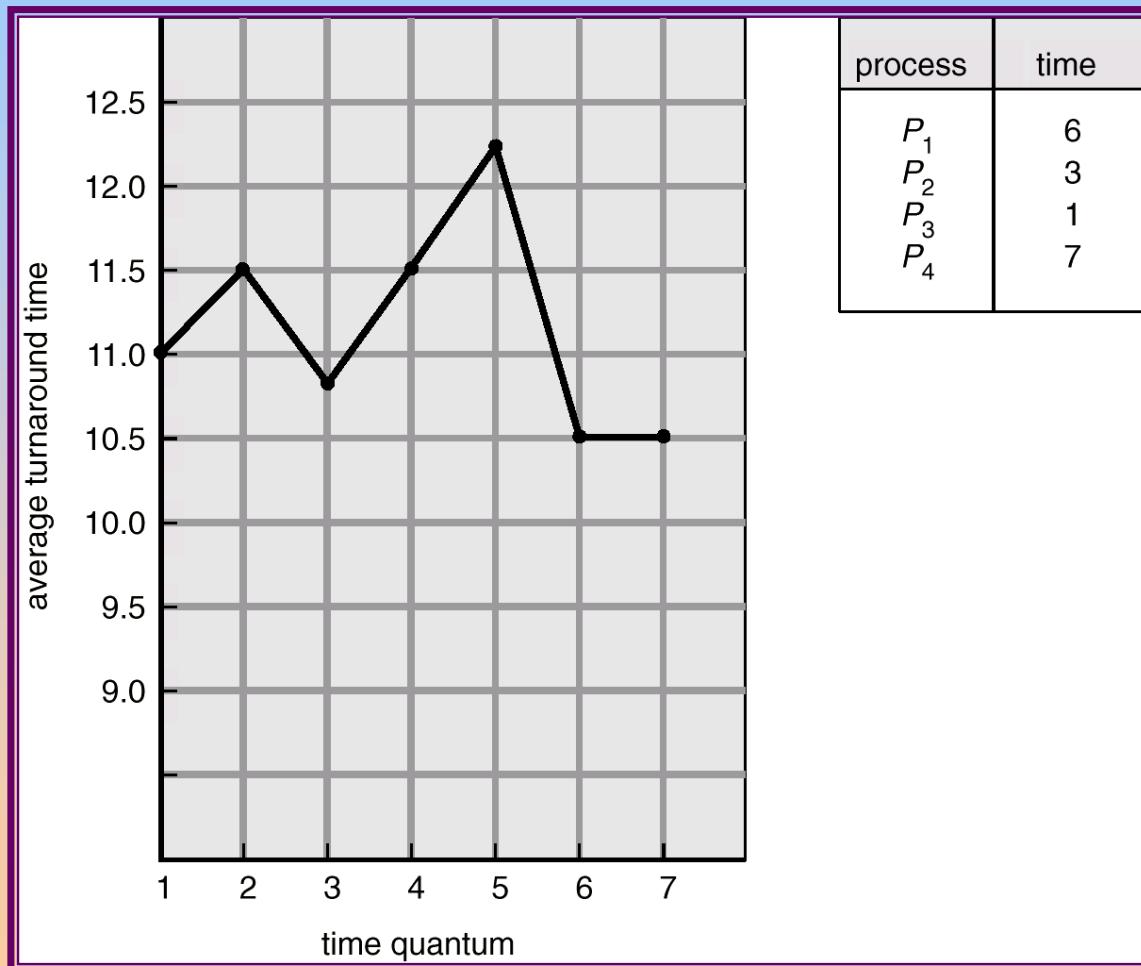
- Typically, higher average turnaround than SJF, but better *response*.
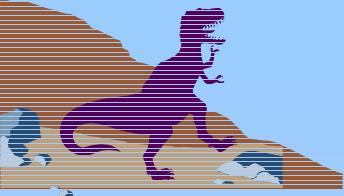
Time Quantum and Context Switch Time

process time = 10	quantum	context switches
0 ———————— 10	12	0
0 ———— 6 ———— 10	6	1
0 1 2 3 4 5 6 7 8 9 10	1	9

Turnaround Time Varies With The Time Quantum

process	time
P_1	6
P_2	3
P_3	1
P_4	7
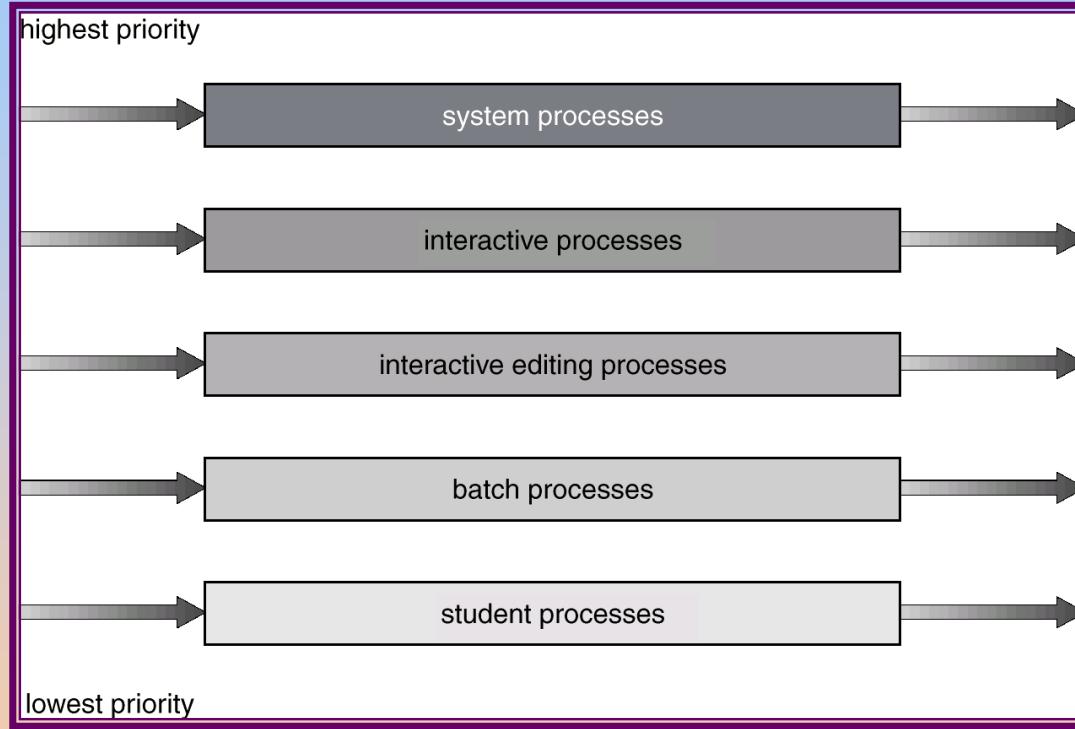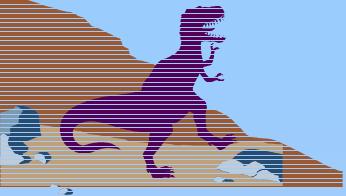
Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)

- Each queue has its own scheduling algorithm,
  foreground – RR
  background – FCFS

- Scheduling must be done between the queues.

  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.

  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR

  - 20% to background in FCFS
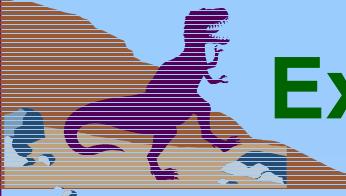
Multilevel Queue Scheduling

highest priority

```
        ┌──────────────────────────────────┐
───────▶│        system processes          │──────▶
        └──────────────────────────────────┘

        ┌──────────────────────────────────┐
───────▶│      interactive processes       │──────▶
        └──────────────────────────────────┘

        ┌──────────────────────────────────┐
───────▶│   interactive editing processes  │──────▶
        └──────────────────────────────────┘

        ┌──────────────────────────────────┐
───────▶│         batch processes          │──────▶
        └──────────────────────────────────┘

        ┌──────────────────────────────────┐
───────▶│        student processes         │──────▶
        └──────────────────────────────────┘
```

lowest priority

Multilevel Feedback Queue

■ A process can move between the various queues; aging can be implemented this way.

■ Multilevel-feedback-queue scheduler defined by the following parameters:

  ✦ number of queues

  ✦ scheduling algorithms for each queue

  ✦ method used to determine when to upgrade a process

  ✦ method used to determine when to demote a process

  ✦ method used to determine which queue a process will enter when that process needs service
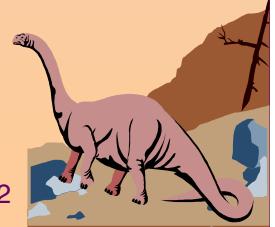
Example of Multilevel Feedback Queue
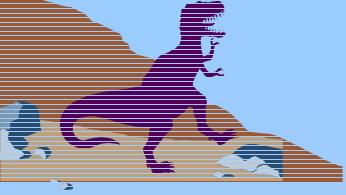
- Three queues:
    - Q_0 – time quantum 8 milliseconds
    - Q_1 – time quantum 16 milliseconds
    - Q_2 – FCFS
- Scheduling
    - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
    - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.
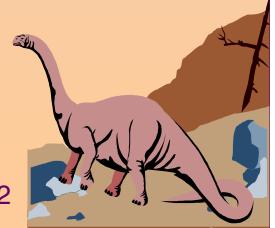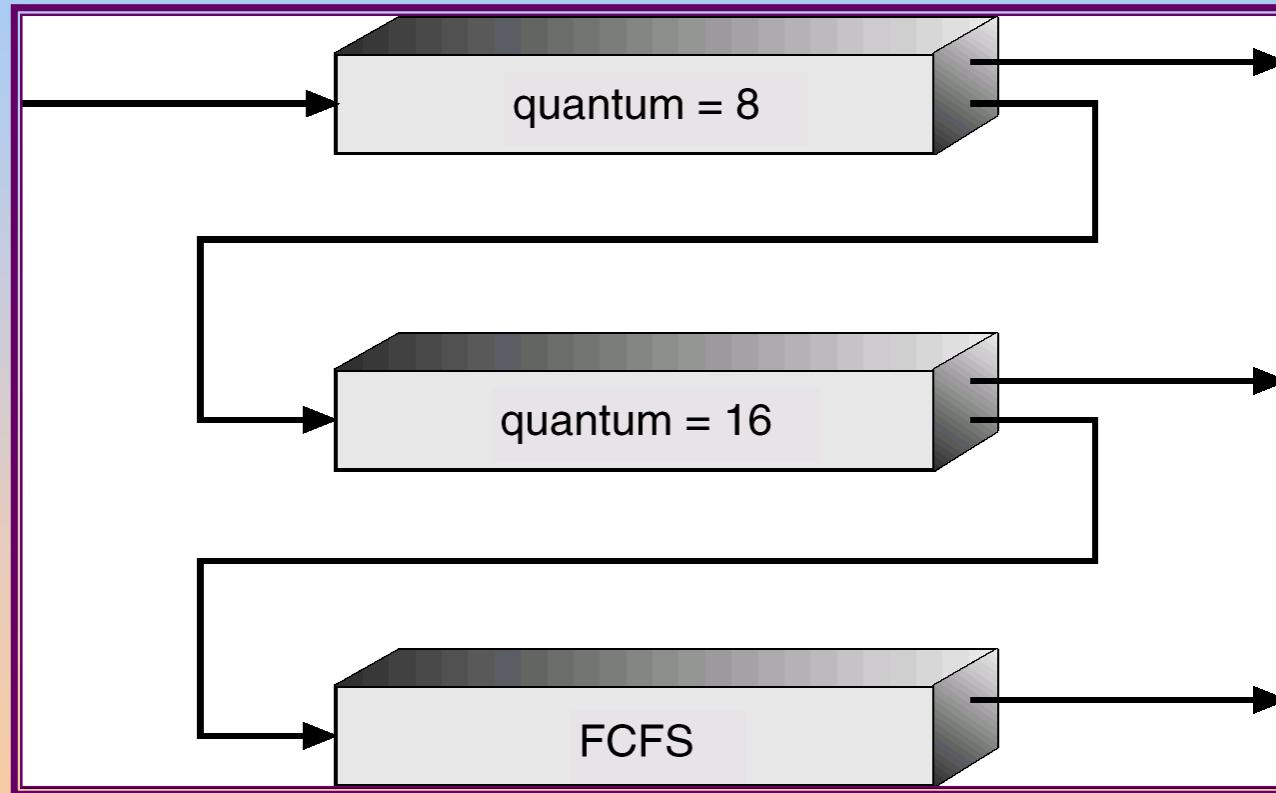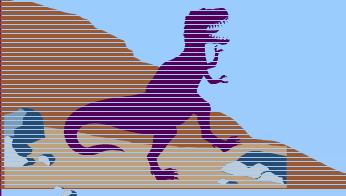
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
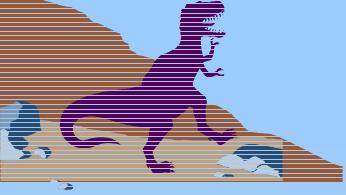
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.

- *Homogeneous processors* within a multiprocessor.

- *Load sharing*

- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.
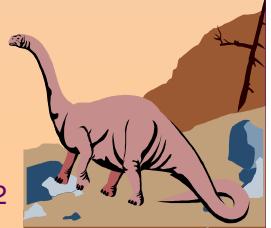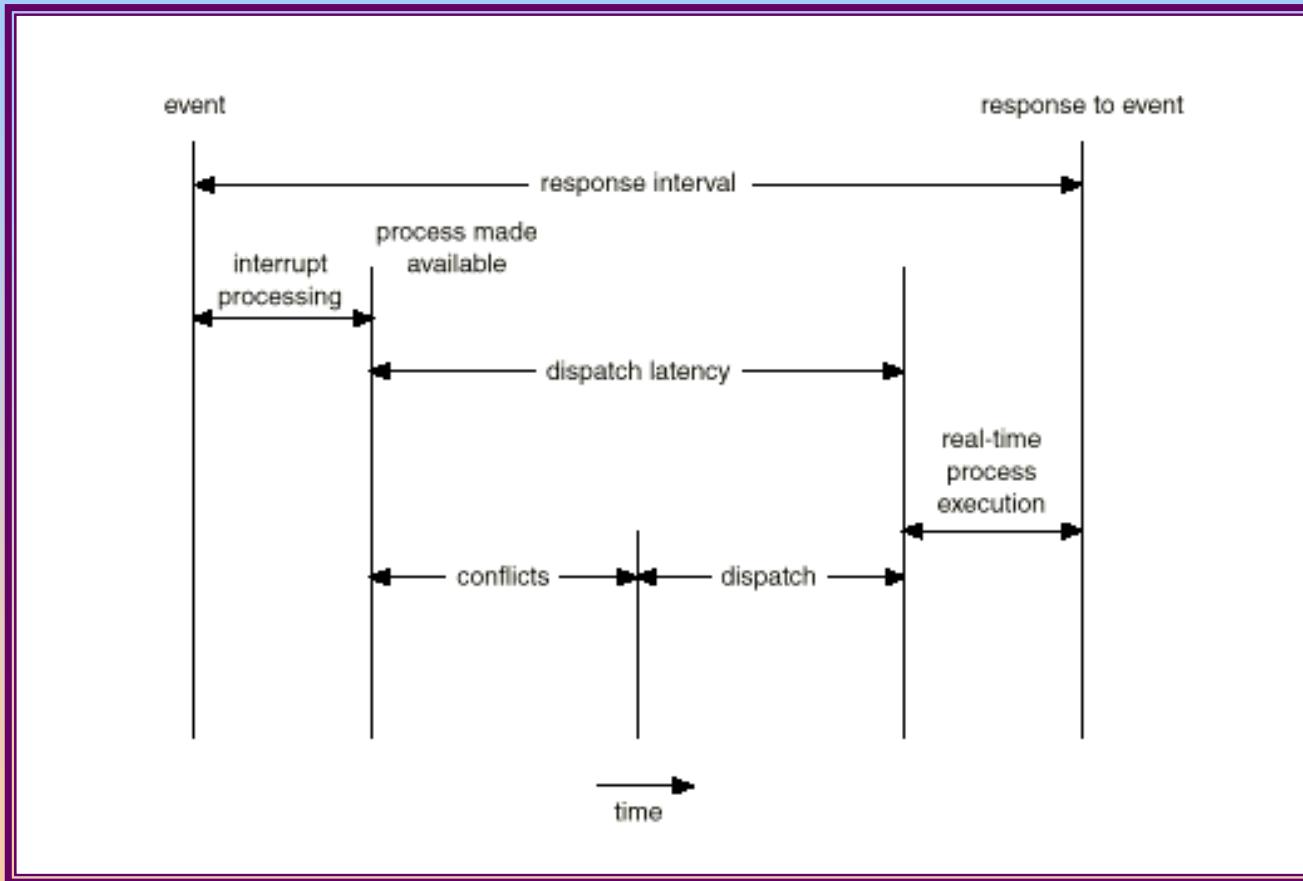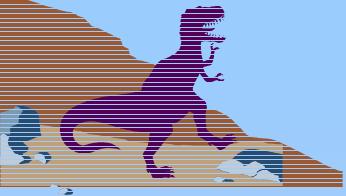
Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.

- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.

Dispatch Latency
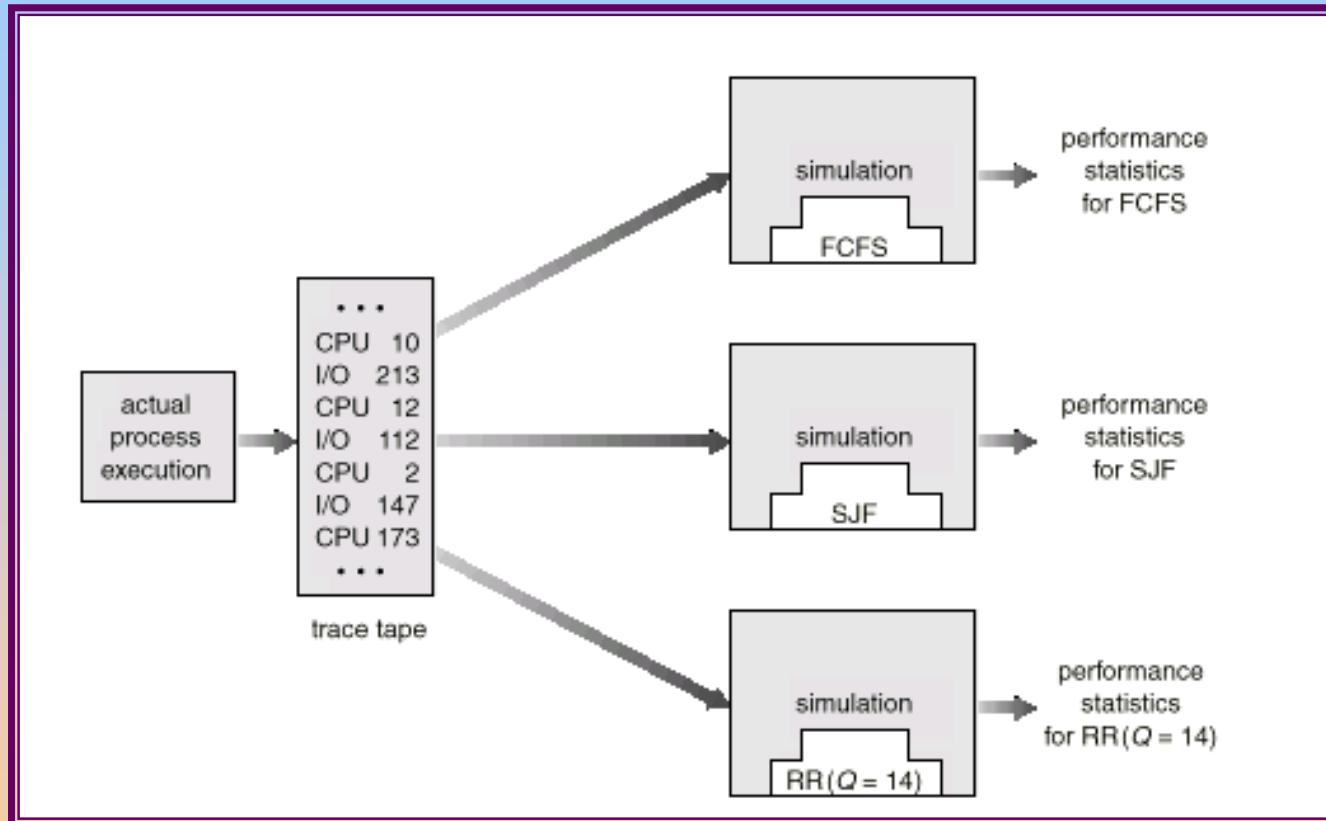
Algorithm Evaluation
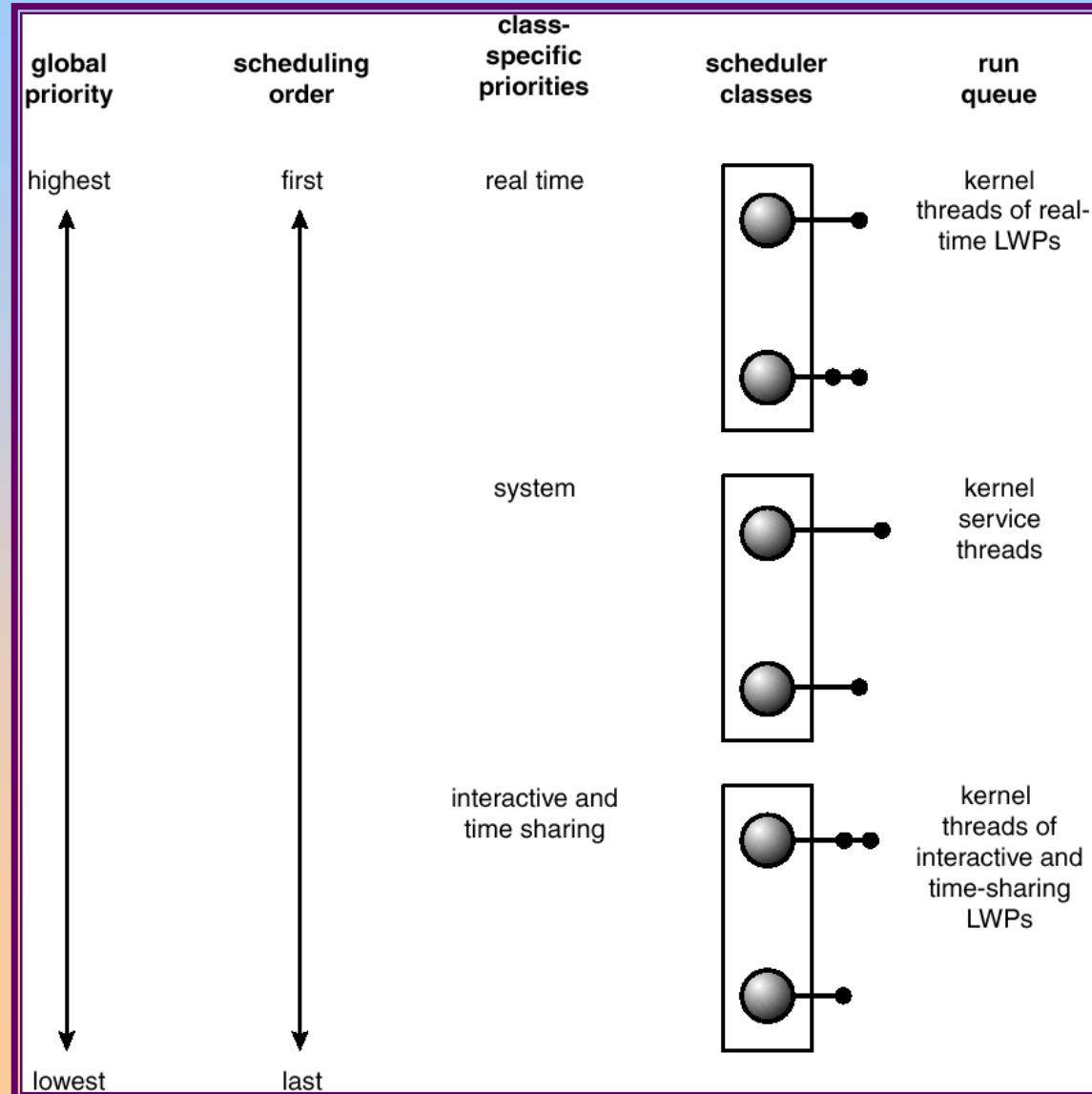
- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.

- Queueing models

- Implementation
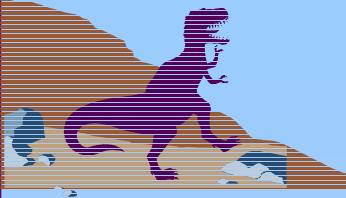
Evaluation of CPU Schedulers by Simulation
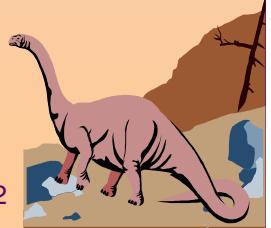
Solaris 2 Scheduling
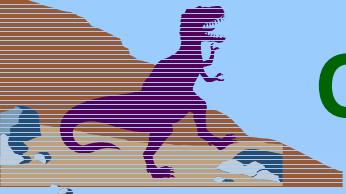
Windows 2000 Priorities
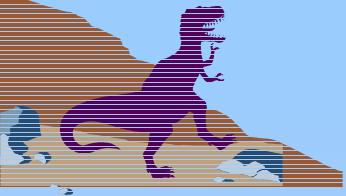
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1
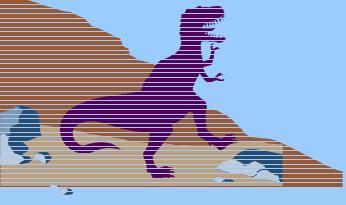
Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000

Background
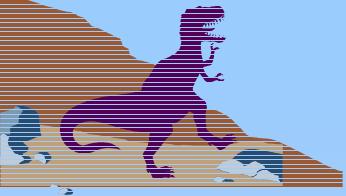
- Concurrent access to shared data may result in data inconsistency.

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

- Shared-memory solution to bounded-butter problem (Chapter 4) 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 variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

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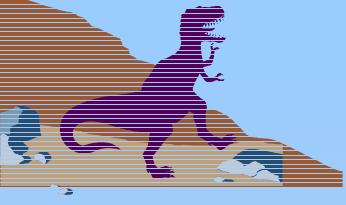
```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process
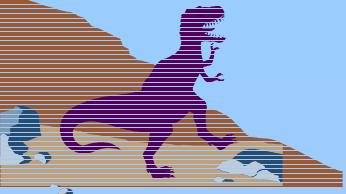
```
item nextProduced;

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

Bounded-Buffer

- Consumer process

```
item nextConsumed;

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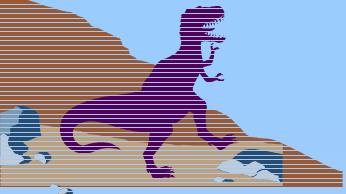
Bounded Buffer

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

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

  register1 = counter
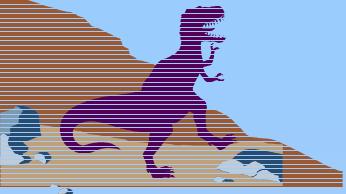
  register1 = register1 + 1
  counter = register1

- The statement "**count—**" may be implemented as:
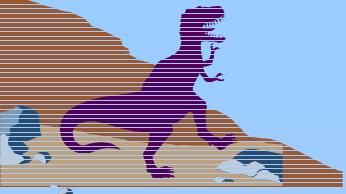
  register2 = counter
  register2 = register2 – 1
  counter = register2

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer

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

producer: **register1 = counter** (*register1 = 5*)
producer: **register1 = register1 + 1** (*register1 = 6*)
consumer: **register2 = counter** (*register2 = 5*)
consumer: **register2 = register2 – 1** (*register2 = 4*)
producer: **counter = register1** (*counter = 6*)
consumer: **counter = register2** (*counter = 4*)
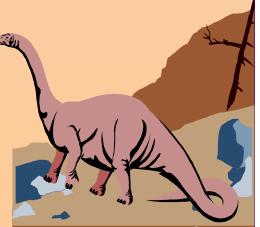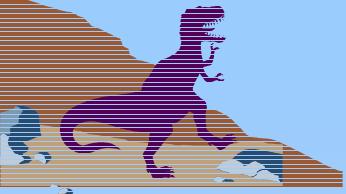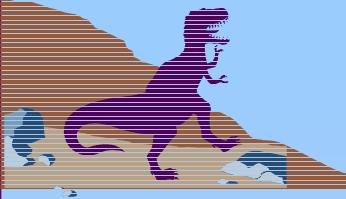
■ The value of **count** may be either 4 or 6, where the correct result should be 5.
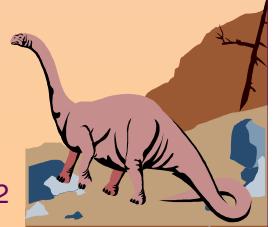
Race Condition
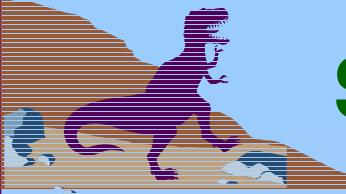
- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

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 allowed to execute in its critical section.
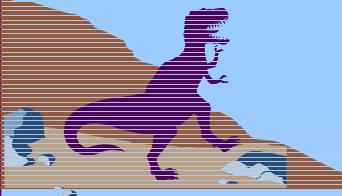
Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the n processes.
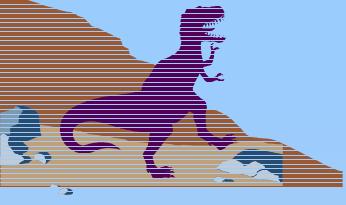
Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

$$\textbf{do } \{$$

$$\boxed{\textit{entry section}}$$

critical section

$$\boxed{\textit{exit section}}$$

reminder section

$$\} \textbf{ while (1)};$$

- Processes may share some common variables to synchronize their actions.

Algorithm 1
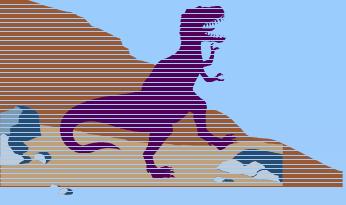
- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn - i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

$$\textbf{do } \{$$
$$\textbf{while (turn != i)} ;$$
$$\text{critical section}$$
$$\textbf{turn = j};$$
$$\text{reminder section}$$
$$\} \textbf{ while (1)};$$

- Satisfies mutual exclusion, but not progress

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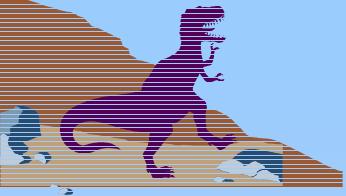
- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow P_i$ ready to enter its critical section
- Process P_i

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

- Satisfies mutual exclusion, but not progress requirement.
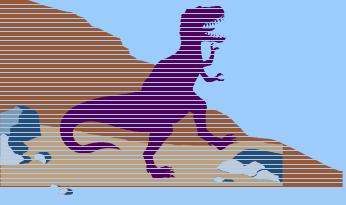
Algorithm 3

- 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);**
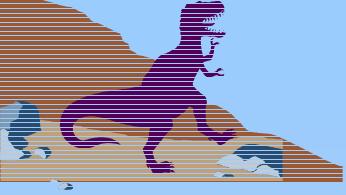
- Meets all three requirements; solves the critical-section problem for two processes.

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., 1,2,3,3,3,3,4,5...
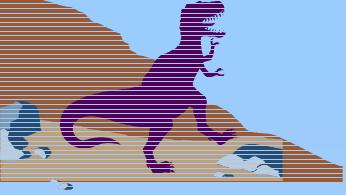
Bakery Algorithm

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

- Shared data
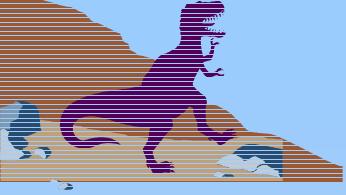
  boolean choosing[n];

  int number[n];

  Data structures are initialized to **false** and **0** respectively

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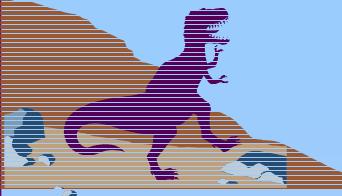
```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
    }
      critical section
    number[i] =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

Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    tqrget = true;

    return rv;
}
```
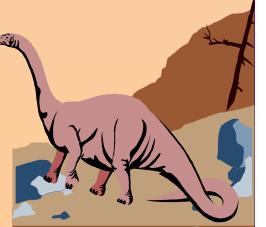
Mutual Exclusion with Test-and-Set
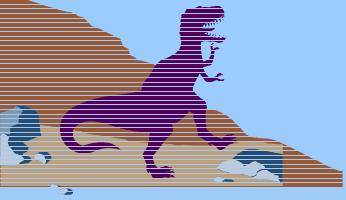
- Shared data:

  boolean lock = false;

- Process P_i

  do {

  while (TestAndSet(lock)) ;

  critical section

  lock = false;

  remainder section

  }
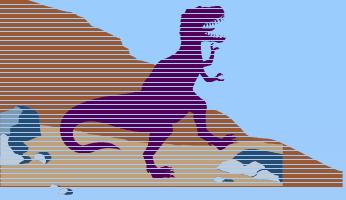
Synchronization Hardware

■ 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;
  boolean waiting[n];

- Process P_i
  do {
  　　key = true;
  　　while (key == true)
  　　　　Swap(lock,key);
  　　　critical section
  　　lock = false;
  　　　remainder section
  }
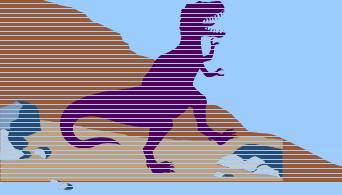
Semaphores
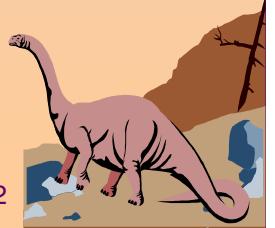
■ Synchronization tool that does not require busy waiting.

■ Semaphore S – integer variable

■ can only be accessed via two indivisible (atomic) operations
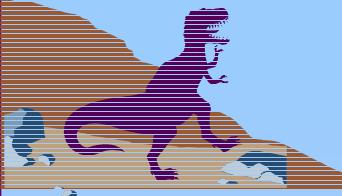
    wait (S):

        while $S \leq 0$ do *no-op*;
        S--;

    signal (S):

        S++;

Critical Section of *n* Processes
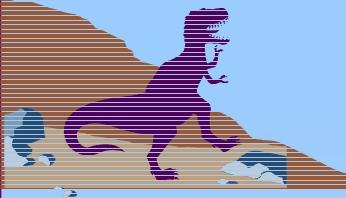
- Shared data:

  semaphore mutex; //initially *mutex* = 1

- Process *Pi:*

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

Semaphore Implementation
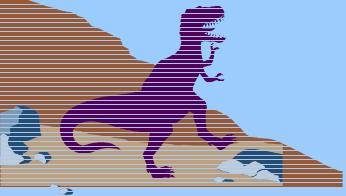
■ Define a semaphore as a record

typedef struct {

int value;
struct process *L;
} semaphore;

■ Assume two simple operations:

✦ **block** suspends the process that invokes it.

✦ **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation
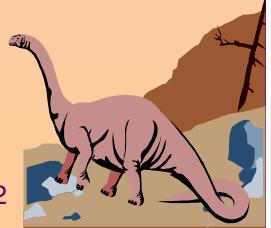
- Semaphore operations now defined as

  wait(*S*):

  S.value--;
  if (S.value < 0) {

  add this process to **S.L;**
  block;

  }

  signal(*S*):

  S.value++;
  if (S.value <= 0) {

  remove a process **P** from **S.L;**
  wakeup(P);

  }
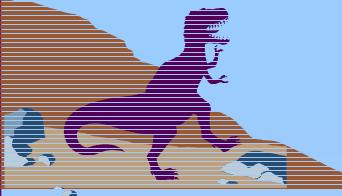
Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

$$P_i \qquad\qquad P_j$$
$$\vdots \qquad\qquad \vdots$$

$$A \qquad\qquad wait(flag)$$
$$signal(flag) \qquad\qquad B$$
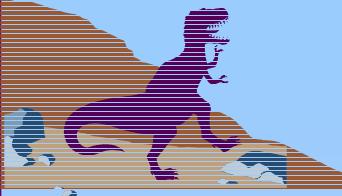
Deadlock and Starvation

- **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);
$\vdots$	$\vdots$
signal(S);	signal(Q);
signal(Q)	signal(S);
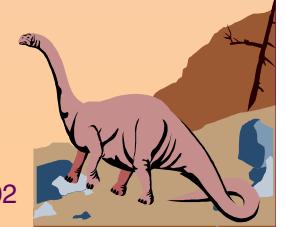
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
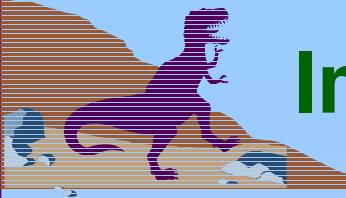
Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.

- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.

- Can implement a counting semaphore *S* as a binary semaphore.

Implementing *S* as a Binary Semaphore

- Data structures:

  binary-semaphore S1, S2;
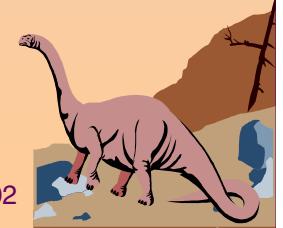
  int C:
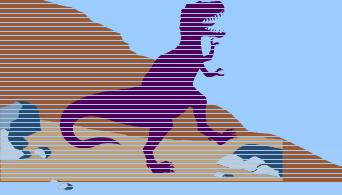
- Initialization:

  S1 = 1

  S2 = 0

  C = initial value of semaphore **S**
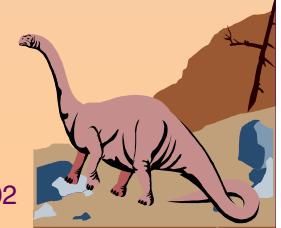
Implementing *S*
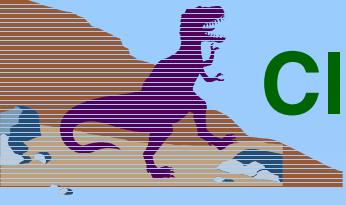
- *wait* operation

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

- *signal* operation

```
wait(S1);
C ++;
if (C <= 0)
        signal(S2);
else
        signal(S1);
```
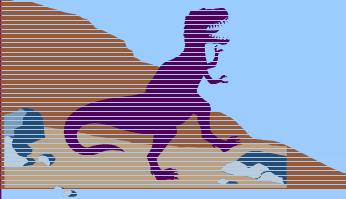
Classical Problems of Synchronization

- Bounded-Buffer Problem

- Readers and Writers Problem

- Dining-Philosophers Problem

Bounded-Buffer Problem
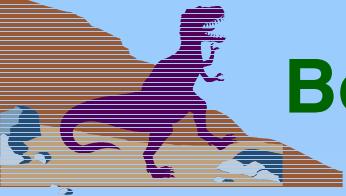
■ Shared data

semaphore full, empty, mutex;

Initially:
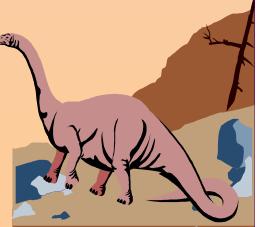
full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process
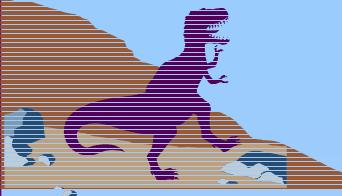
```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (1);
```

Bounded-Buffer Problem 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);
```
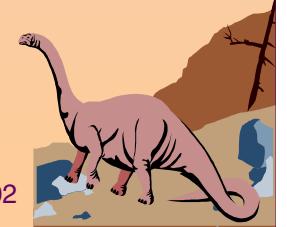
Readers-Writers Problem
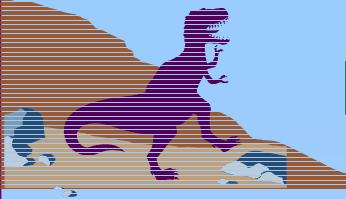
- Shared data

   semaphore mutex, wrt;

   Initially

   mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

wait(wrt);

...
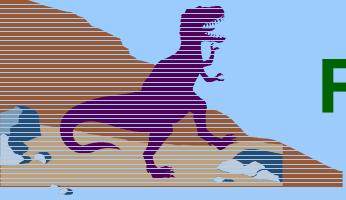
writing is performed

...

signal(wrt);
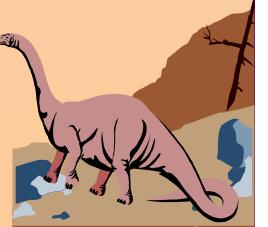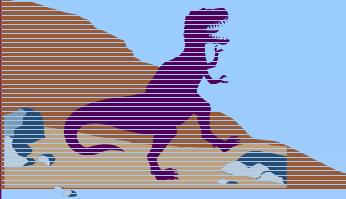
Readers-Writers Problem Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(rt);
signal(mutex);

        …
    reading is performed
        …
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex):
```
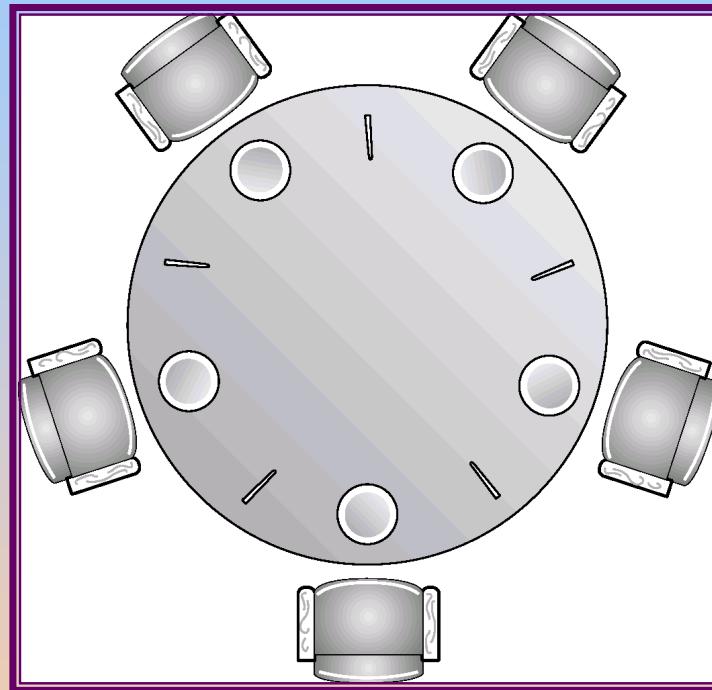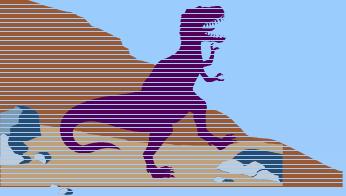
Dining-Philosophers Problem

- Shared data

   semaphore chopstick[5];

   Initially all values are 1
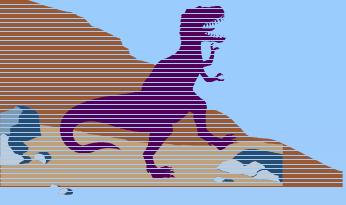
Dining-Philosophers Problem

- Philosopher *i*:

```
do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])

      …
    eat

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

      …
    think

      …
} while (1);
```

Critical Regions
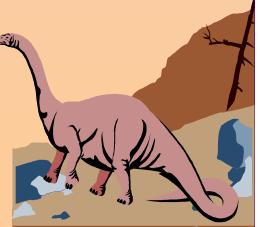
- High-level synchronization construct
- A shared variable v of type T, is declared as:
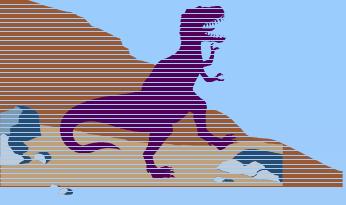
   v: shared T

- Variable v accessed only inside statement

   region v when B do S

   where B is a boolean expression.
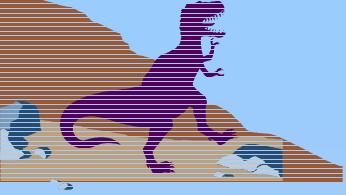
- While statement S is being executed, no other process can access variable v.

Critical Regions

■ Regions referring to the same shared variable exclude each other in time.

■ When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v.
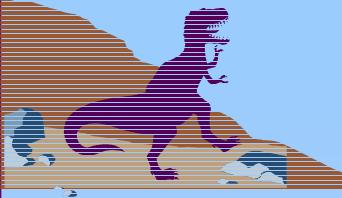
Example – Bounded Buffer

- Shared data:

```
struct buffer {
        int pool[n];
        int count, in, out;
}
```
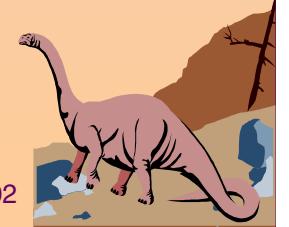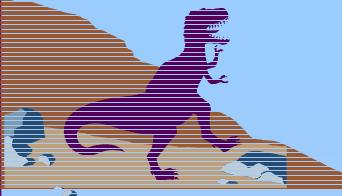
Bounded Buffer Producer Process

■ Producer process inserts **nextp** into the shared buffer

```
region buffer when( count < n) {
        pool[in] = nextp;
        in:= (in+1) % n;
        count++;
}
```
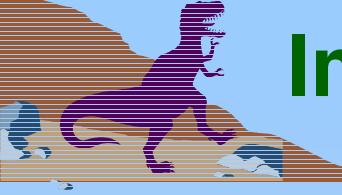
Bounded Buffer Consumer Process

■ Consumer process removes an item from the shared buffer and puts it in **nextc**

```
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```
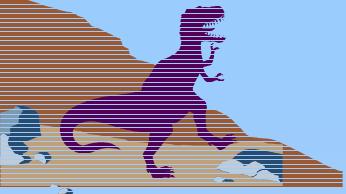
Implementation region *x* when *B* do *S*
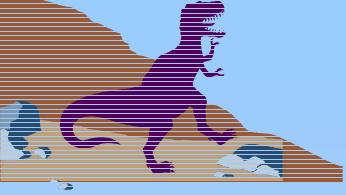
■ Associate with the shared variable *x*, the following variables:

semaphore mutex, first-delay, second-delay;
int first-count, second-count;

■ Mutually exclusive access to the critical section is provided by **mutex**.

■ If a process cannot enter the critical section because the Boolean expression **B** is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate *B*.

Implementation

- Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively.

- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.

- For an arbitrary queuing discipline, a more complicated implementation is required.
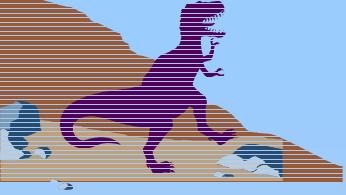
Monitors

■ High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
        shared variable declarations
        procedure body P1 (...) {
          . . .
        }
        procedure body P2 (...) {
          . . .
        }
        procedure body Pn (...) {
           . . .
        }
        {
            initialization code
        }
}
```
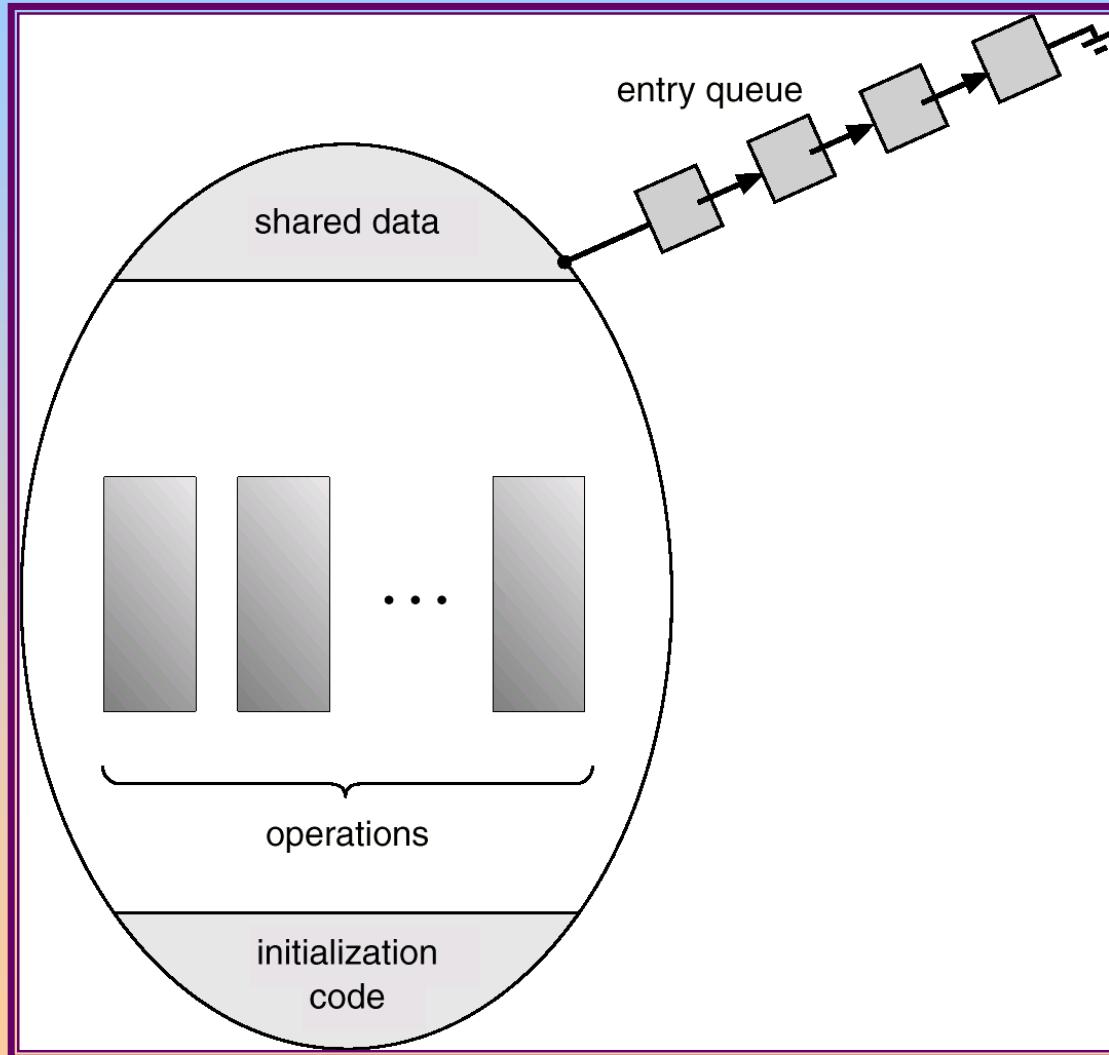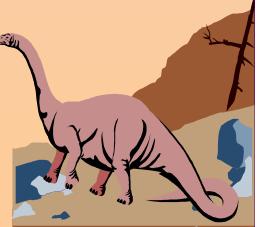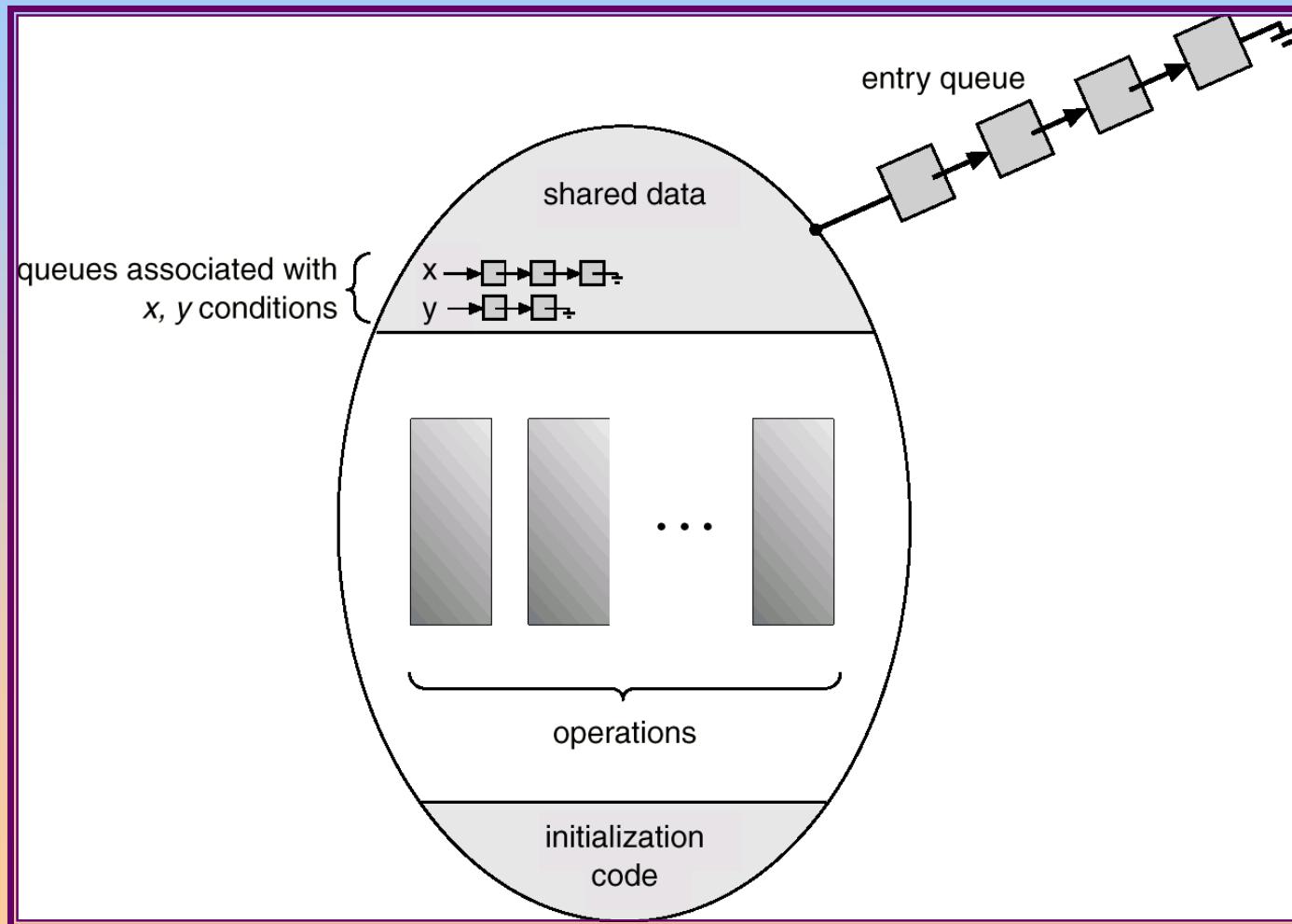
Monitors
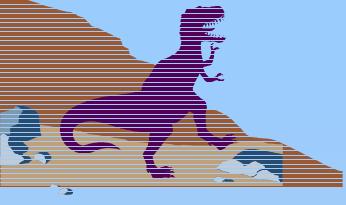
■ To allow a process to wait within the monitor, a **condition** variable must be declared, as

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.

Schematic View of a Monitor

shared data

entry queue

...

operations

initialization
code
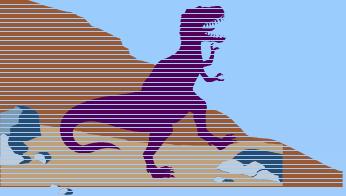
Monitor With Condition Variables

Dining Philosophers Example
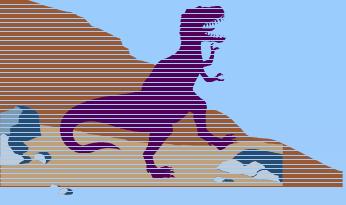
```
monitor dp
{

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

Dining Philosophers

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

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

Dining Philosophers

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

Monitor Implementation Using Semaphores

- Variables

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

- Each external procedure *F* will be replaced by

  wait(mutex);

  …
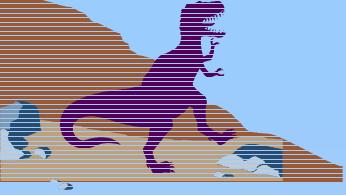
  body of *F*;

  …

  if (next-count > 0)
  signal(next)
  else
  signal(mutex);

- Mutual exclusion within a monitor is ensured.
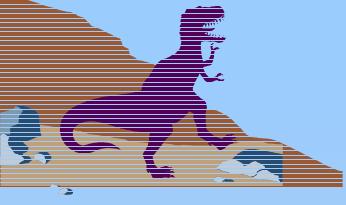
Monitor Implementation

■ For each condition variable x, we have:

semaphore x-sem; // (initially = 0)
int x-count = 0;

■ The operation **x.wait** can be implemented as:
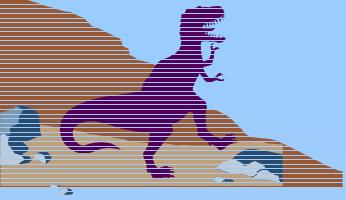
```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

Monitor Implementation
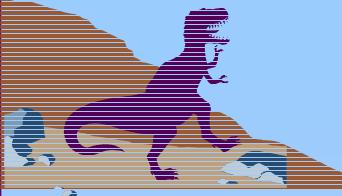
■ The operation **x.signal** can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```
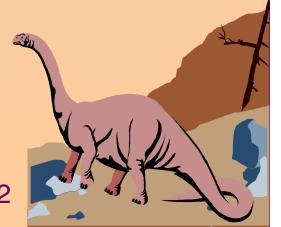
Monitor Implementation
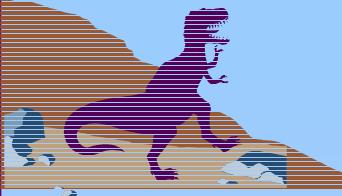
- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.

- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.

- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.

- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.
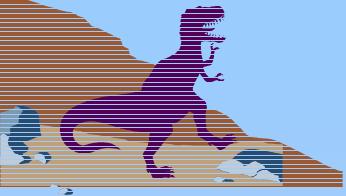
Windows 2000 Synchronization
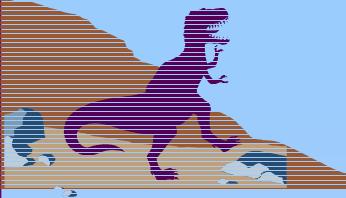
- Uses interrupt masks to protect access to global resources on uniprocessor systems.

- Uses *spinlocks* on multiprocessor systems.

- Also provides *dispatcher objects* which may act as wither mutexes and semaphores.

- Dispatcher objects may also provide *events*. An event acts much like a condition variable.

Chapter 8: Deadlocks
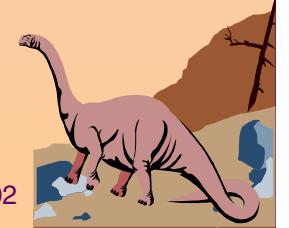
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling
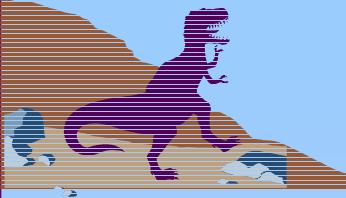
The Deadlock Problem
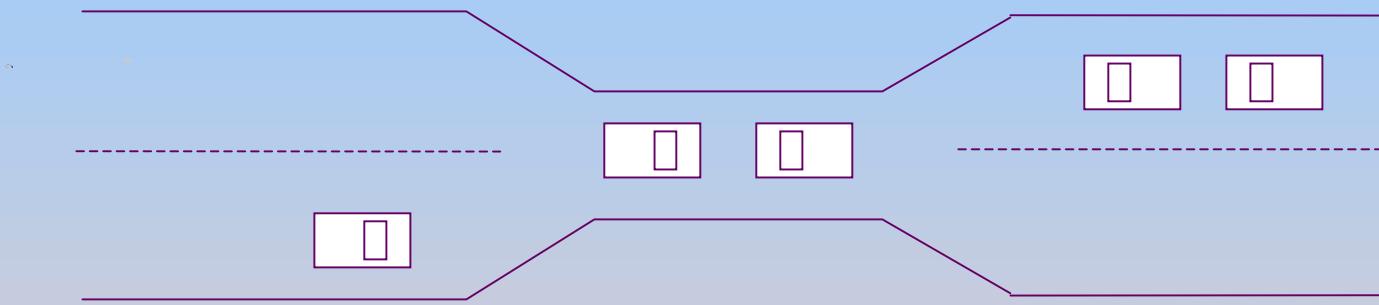
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.

- Example
  - semaphores A and B, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*
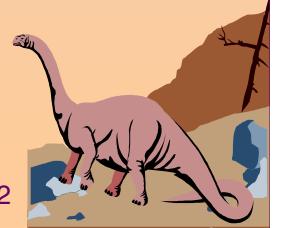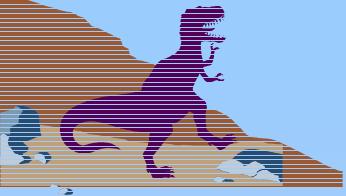
Bridge Crossing Example
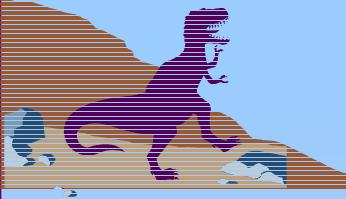
- Traffic only in one direction.

- Each section of a bridge can be viewed as a resource.

- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).

- Several cars may have to be backed up if a deadlock occurs.

- Starvation is possible.

System Model

- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:

  - request
  - use
  - release
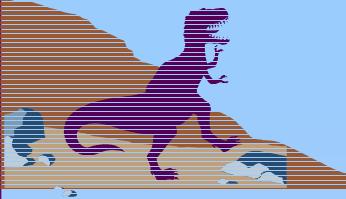
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

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

Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.
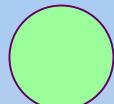
- V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)
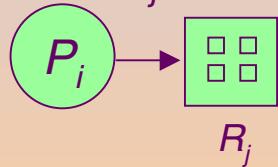
- Process
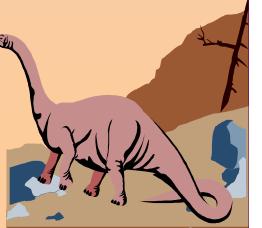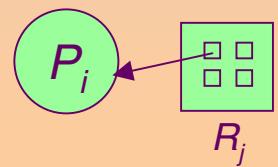
- Resource Type with 4 instances
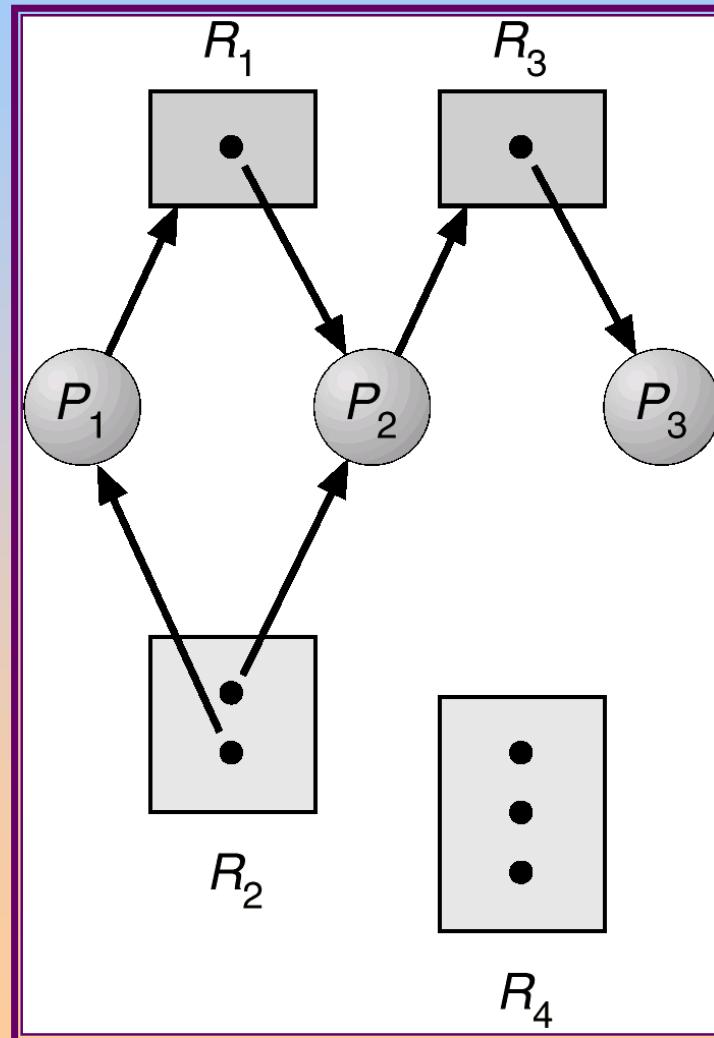
- P_i requests instance of R_j

  $P_i \longrightarrow R_j$
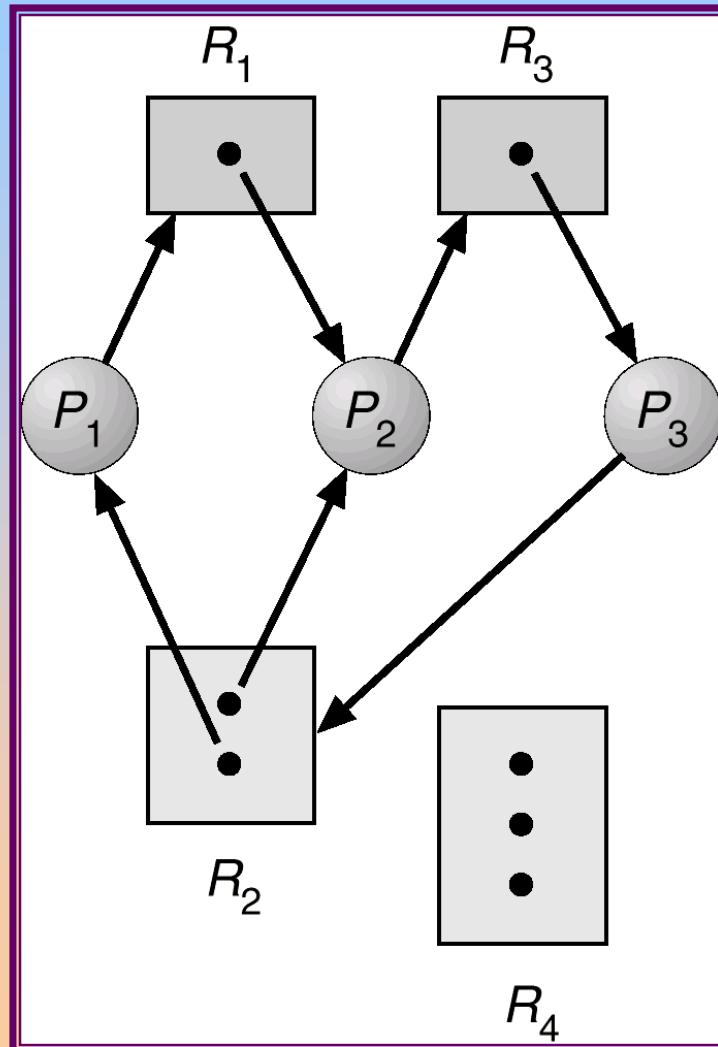
- P_i is holding an instance of R_j

  $P_i \longleftarrow R_j$
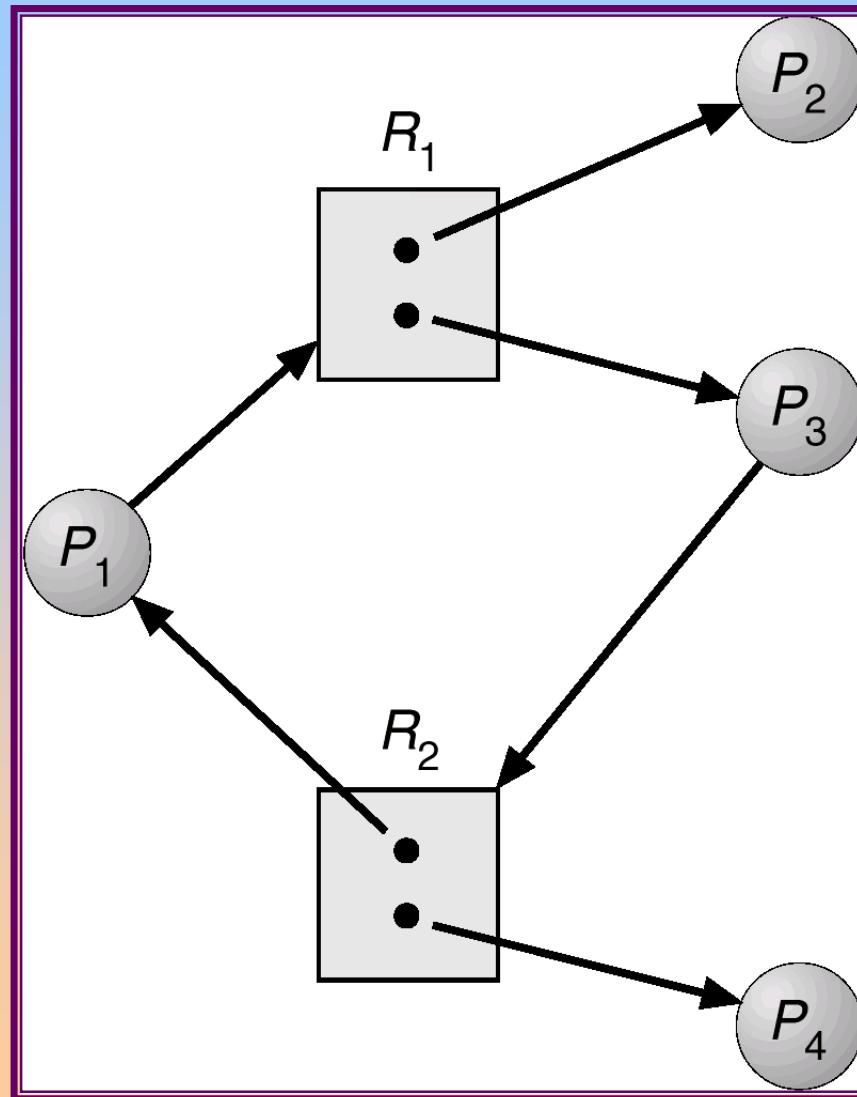
Example of a Resource Allocation Graph
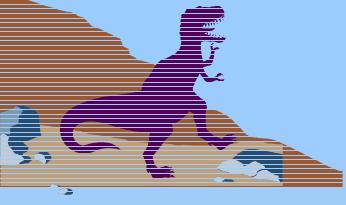
Resource Allocation Graph With A Deadlock
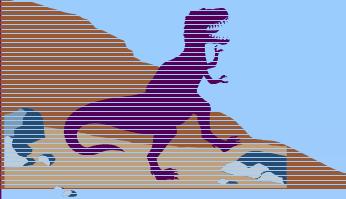
Resource Allocation Graph With A Cycle But No Deadlock
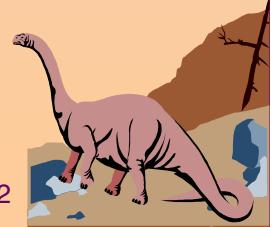
Basic Facts
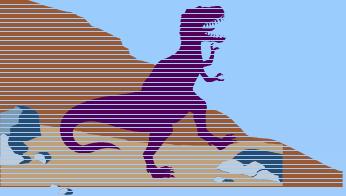
- If graph contains no cycles $\Rightarrow$ no deadlock.

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

Methods for Handling Deadlocks

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

- Allow the system to enter a deadlock state and then recover.

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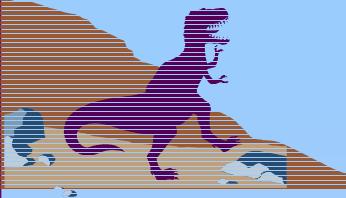
Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
    - ✦ Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
    - ✦ Low resource utilization; starvation possible.
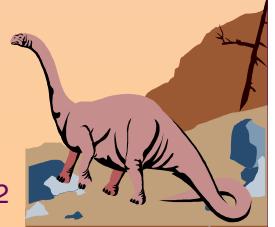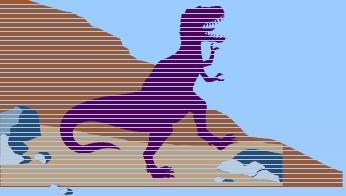
Deadlock Prevention (Cont.)

- **No Preemption** –
    - ✦ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
    - ✦ Preempted resources are added to the list of resources for which the process is waiting.
    - ✦ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
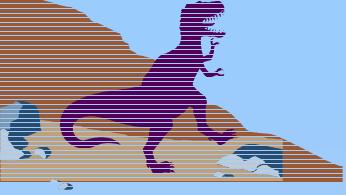
Deadlock Avoidance
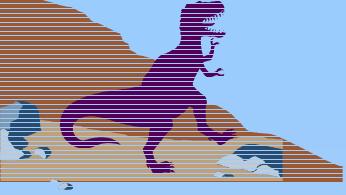
Requires that the system has some additional *a priori* information available.

- ■ Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

- ■ The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

- ■ Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.
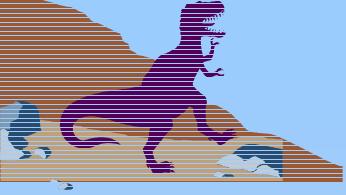
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.

- System is in safe state if there exists a safe sequence of all processes.

- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<I$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
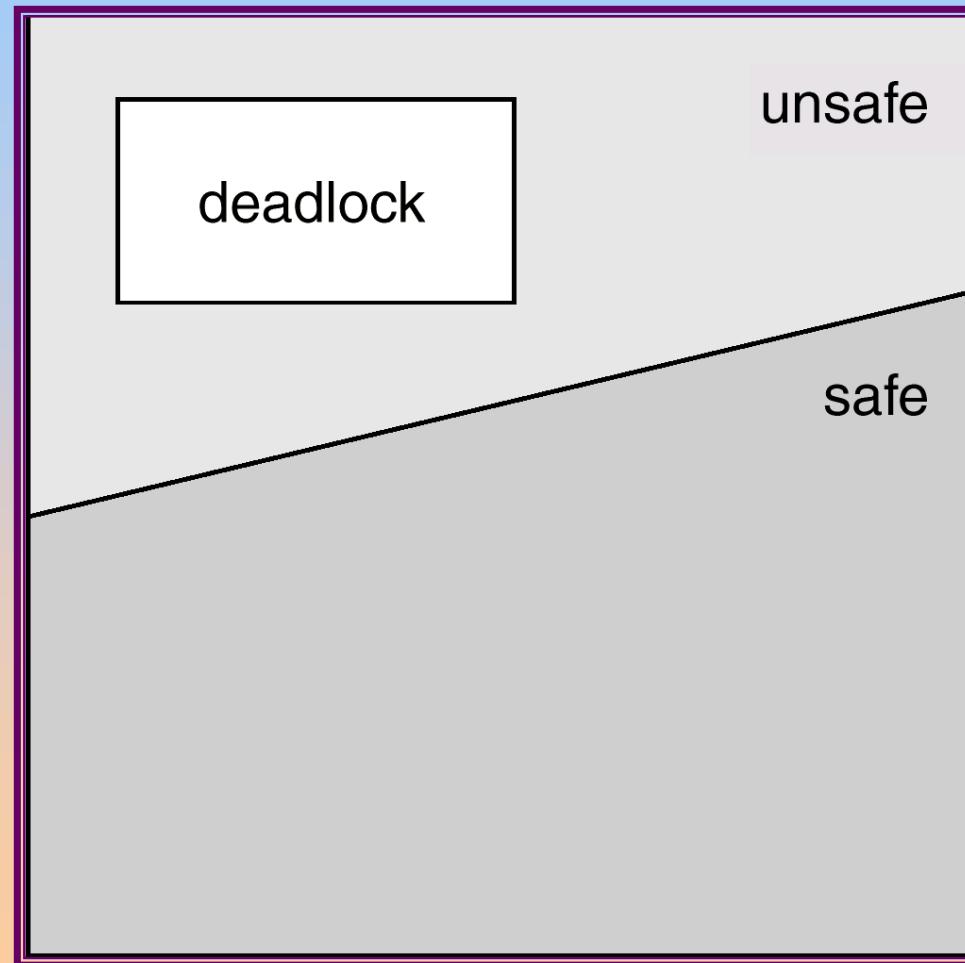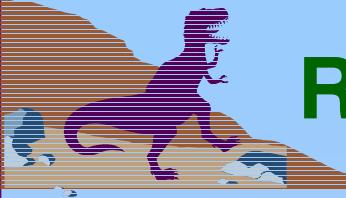
Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.

- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.

- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
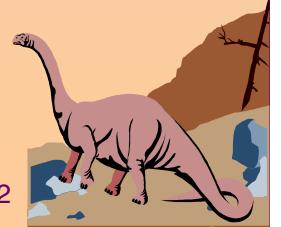
Safe, Unsafe , Deadlock State
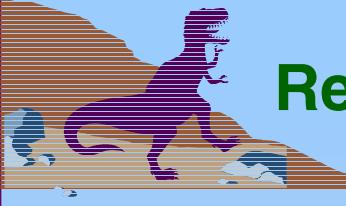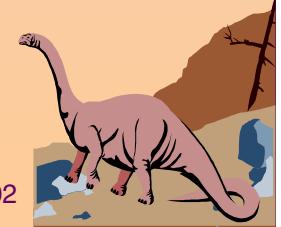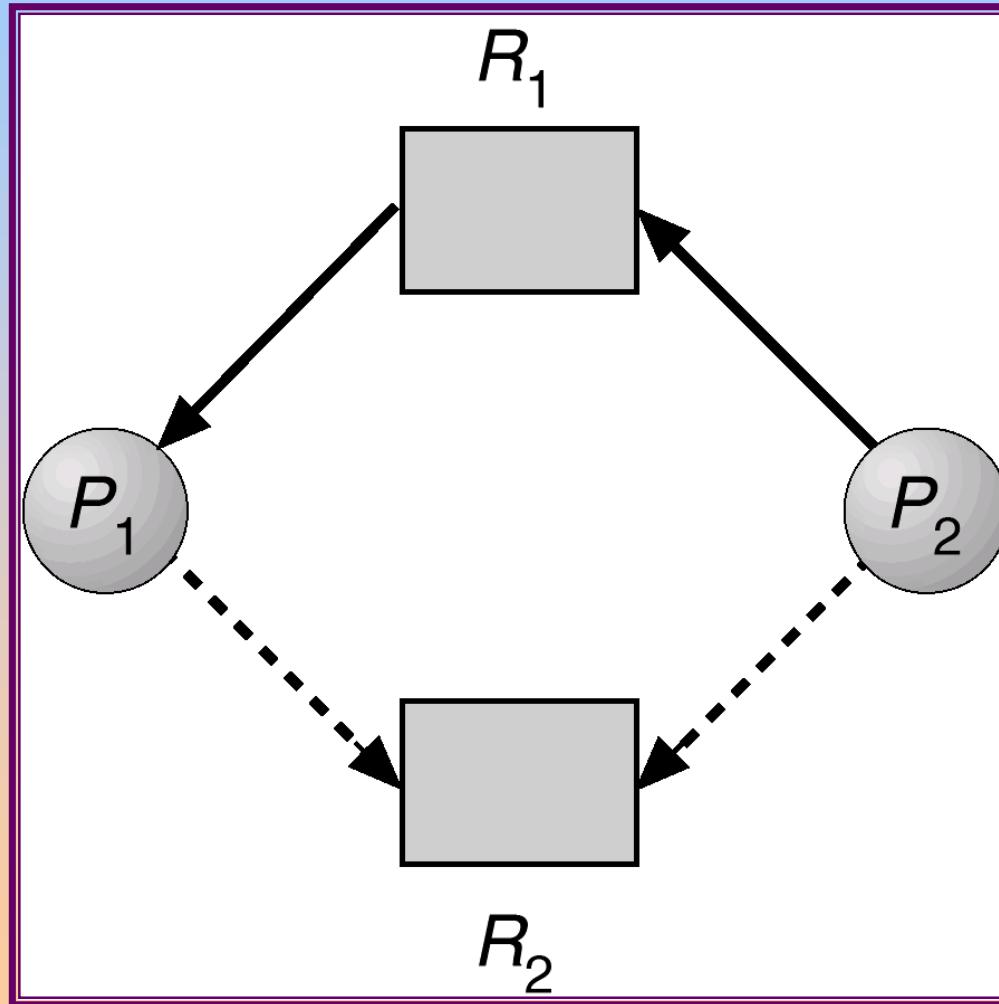
Resource-Allocation Graph Algorithm
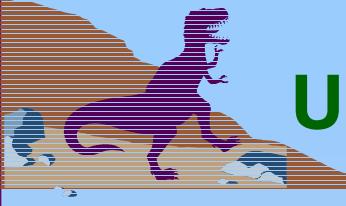
- *Claim edge $P_i \rightarrow R_j$* indicated that process P_j may request resource R_j; represented by a dashed line.

- Claim edge converts to request edge when a process requests a resource.

- When a resource is released by a process, assignment edge reconverts to a claim edge.

- Resources must be claimed *a priori* in the system.
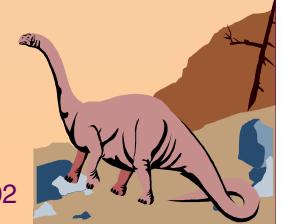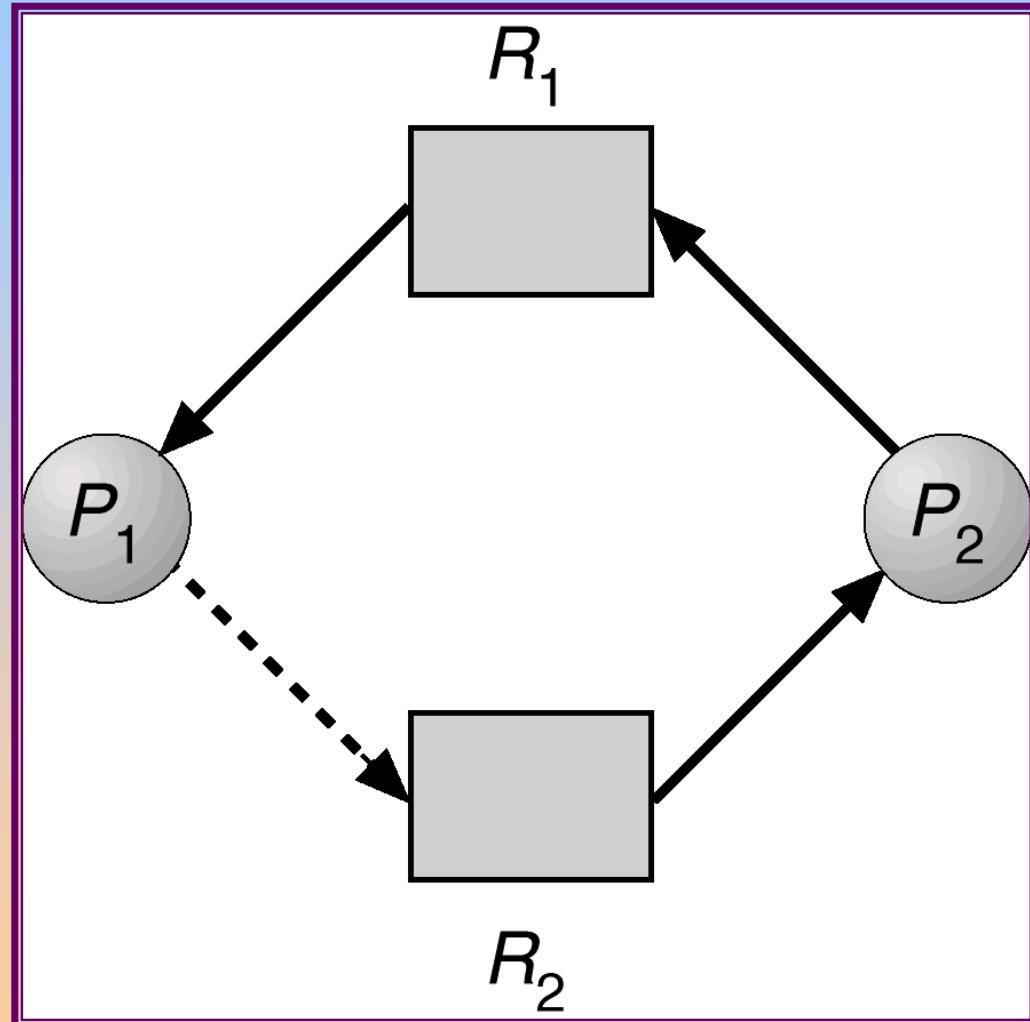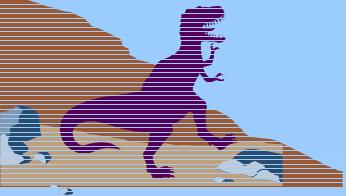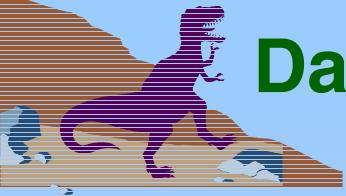
Banker's Algorithm

- Multiple instances.

- Each process must a priori claim maximum use.

- When a process requests a resource it may have to wait.

- When a process gets all its resources it must return them in a finite amount of time.
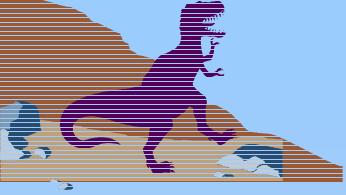
Data Structures for the Banker's Algorithm

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

- *Available:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.

- *Max: n x m* matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.

- *Allocation: n x m* matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.

- *Need: n x m* matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

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

Safety Algorithm

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

   > *Work = Available*
   >
   > *Finish* [*i*] = *false* for *i* - 1,3, …, *n.*

2. Find and *i* such that both:

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

   (b) *Need$_i$ $\leq$ Work*
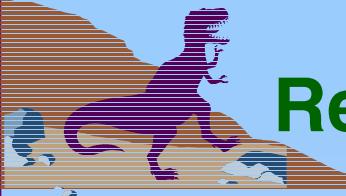
   If no such *i* exists, go to step 4.

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

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

Resource-Request Algorithm for Process P_i

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

1. If *Request*$_i$ ≤ *Need*$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If *Request*$_i$ ≤ *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:
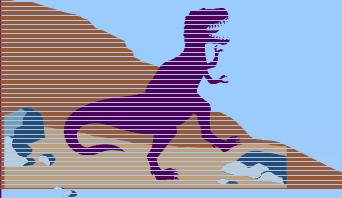
$$Available = Available = Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_{i;};$$

- *If safe* ⇒ *the resources are allocated to* P_i.
- *If unsafe* ⇒ P_i *must wait, and the old resource-allocation state is restored*
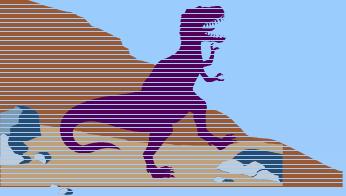
Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5instances, and C (7 instances).

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	
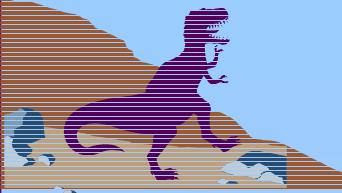
Example (Cont.)

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

$$\underline{Need}$$

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

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.
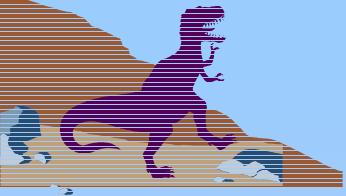
Example P_1 Request (1,0,2) (Cont.)

■ Check that Request ≤ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ *true.*

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

■ Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.

■ Can request for (3,3,0) by P_4 be granted?

■ Can request for (0,2,0) by P_0 be granted?

Deadlock Detection
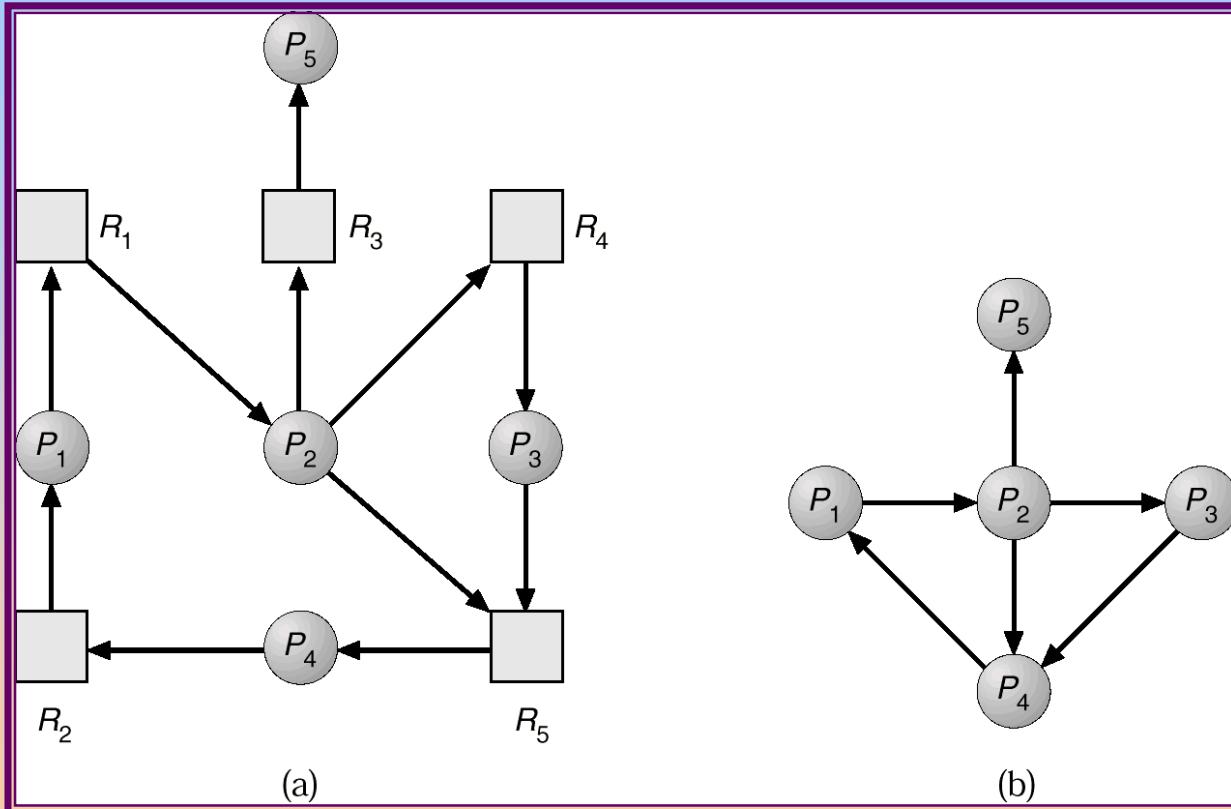
- Allow system to enter deadlock state

- Detection algorithm

- Recovery scheme

Single Instance of Each Resource Type

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

- Periodically invoke an algorithm that searches for a cycle in the graph.

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.
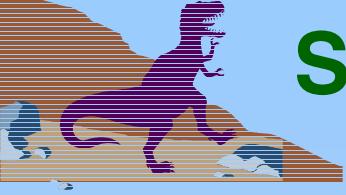
Resource-Allocation Graph and Wait-for Graph

(a)

Resource-Allocation Graph

(b)

Corresponding wait-for graph
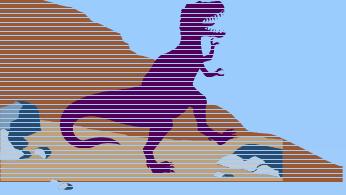
Several Instances of a Resource Type

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

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

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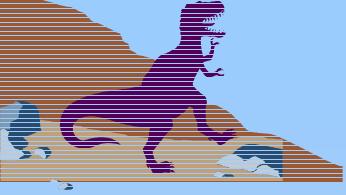
Detection Algorithm

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

   (a) *Work = Available*

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

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

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

   (b) $Request_i \leq Work$
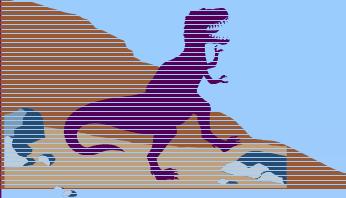
   If no such *i* exists, go to step 4.

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

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

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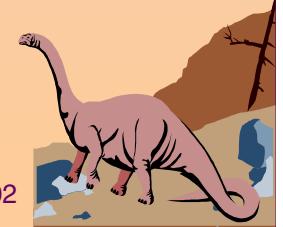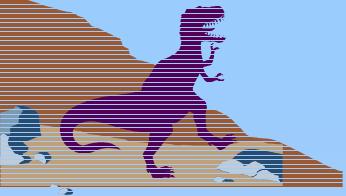
Example of Detection Algorithm

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

- Snapshot at time T_0:

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

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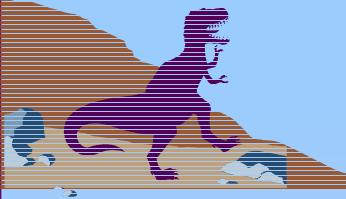
Example (Cont.)

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

<div align="center">

Request

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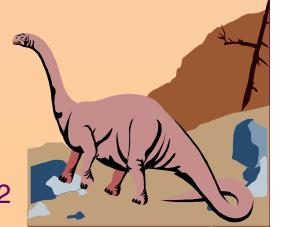
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

</div>
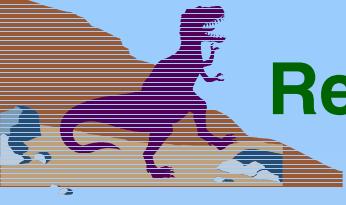
■ State of system?

   ✦ Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.

   ✦ Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage
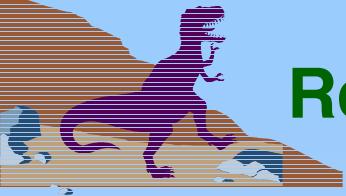
■ When, and how often, to invoke depends on:

✦ How often a deadlock is likely to occur?

✦ How many processes will need to be rolled back?

✔ one for each disjoint cycle

■ If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.
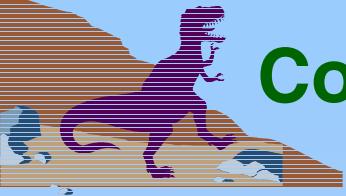
Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes.**

- **Abort one process at a time until the deadlock cycle is eliminated.**

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

8.38

Recovery from Deadlock: Resource Preemption
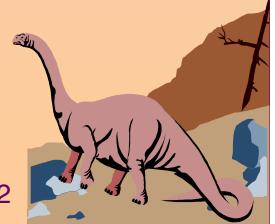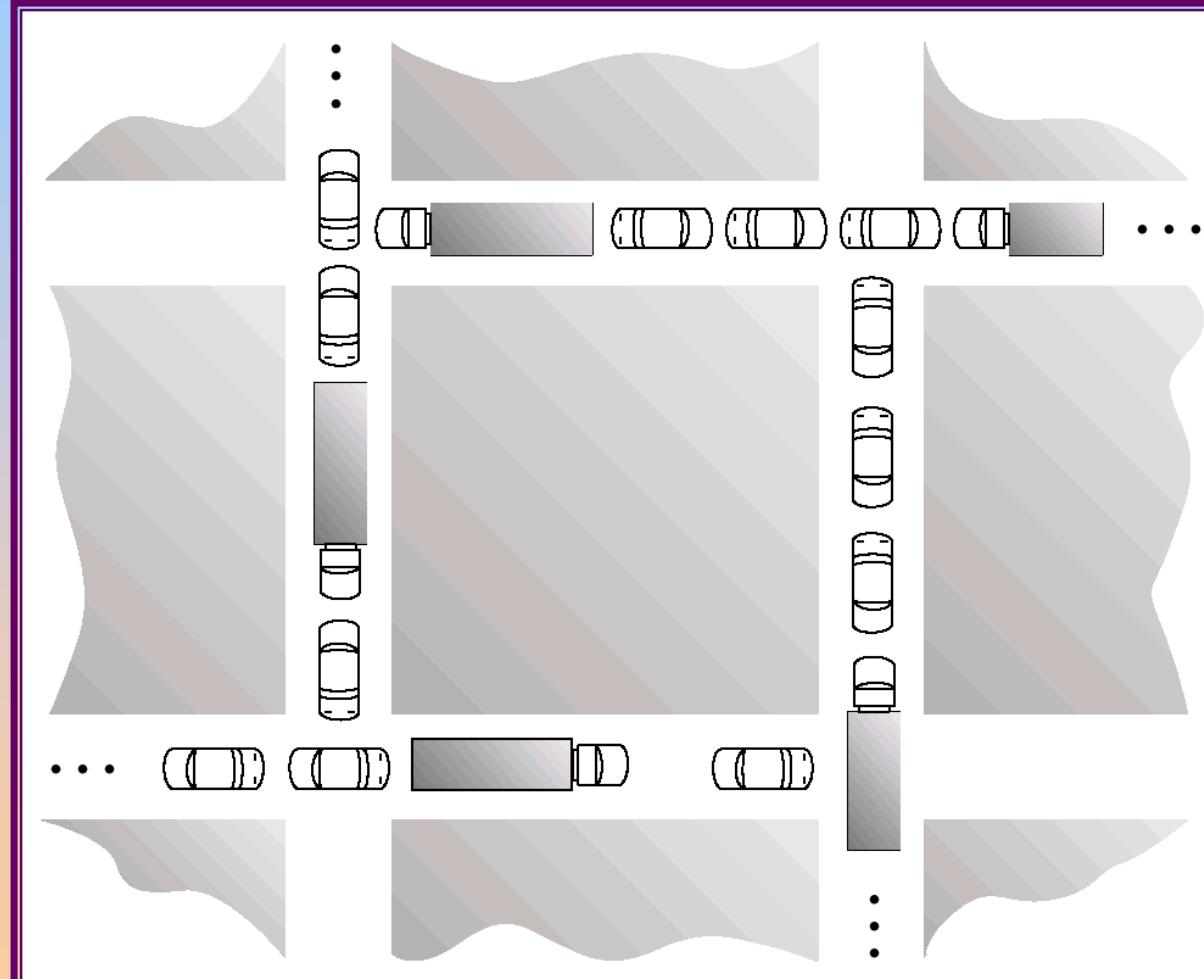
- Selecting a victim – minimize cost.

- Rollback – return to some safe state, restart process for that state.

- Starvation – same process may always be picked as victim, include number of rollback in cost factor.
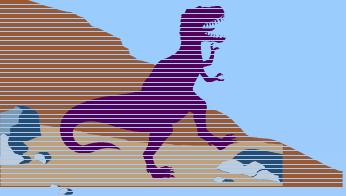
Combined Approach to Deadlock Handling

- 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.
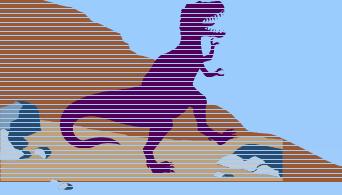
Traffic Deadlock for Exercise 8.4
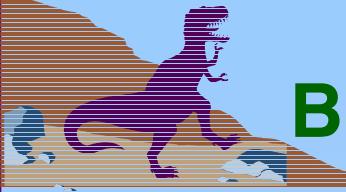
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Program must be brought into memory and placed within a process for it to be run.

- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.

- User programs go through several steps before being run.
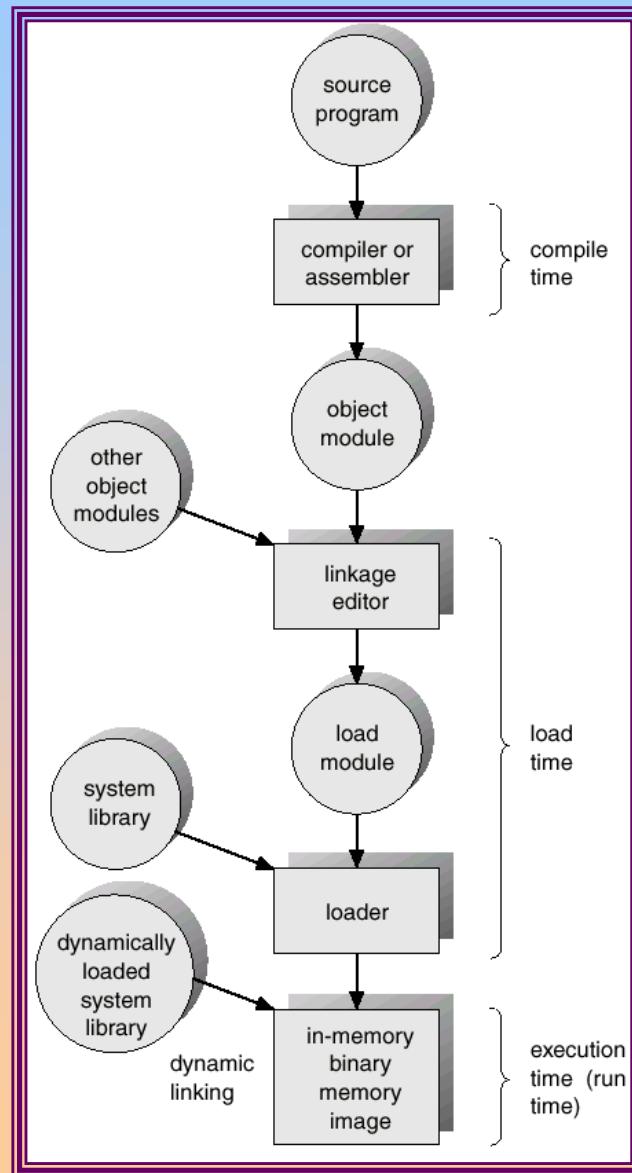
Binding of Instructions and Data to Memory
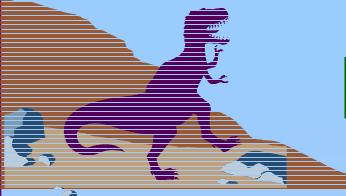
Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

- **Load time**: Must generate *relocatable* code if memory location is not known at compile time.

- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).
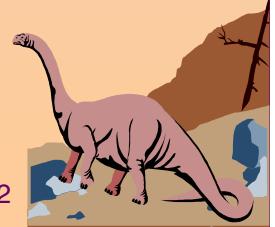
Multistep Processing of a User Program
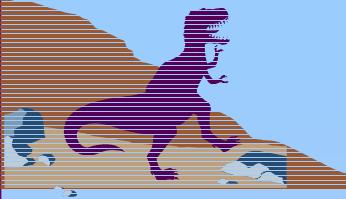
Logical vs. Physical Address Space
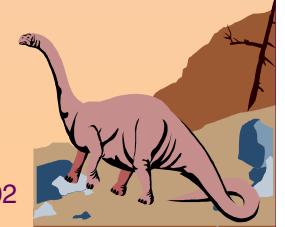
- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
    - *Logical address* – generated by the CPU; also referred to as *virtual address*.
    - *Physical address* – address seen by the memory unit.

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
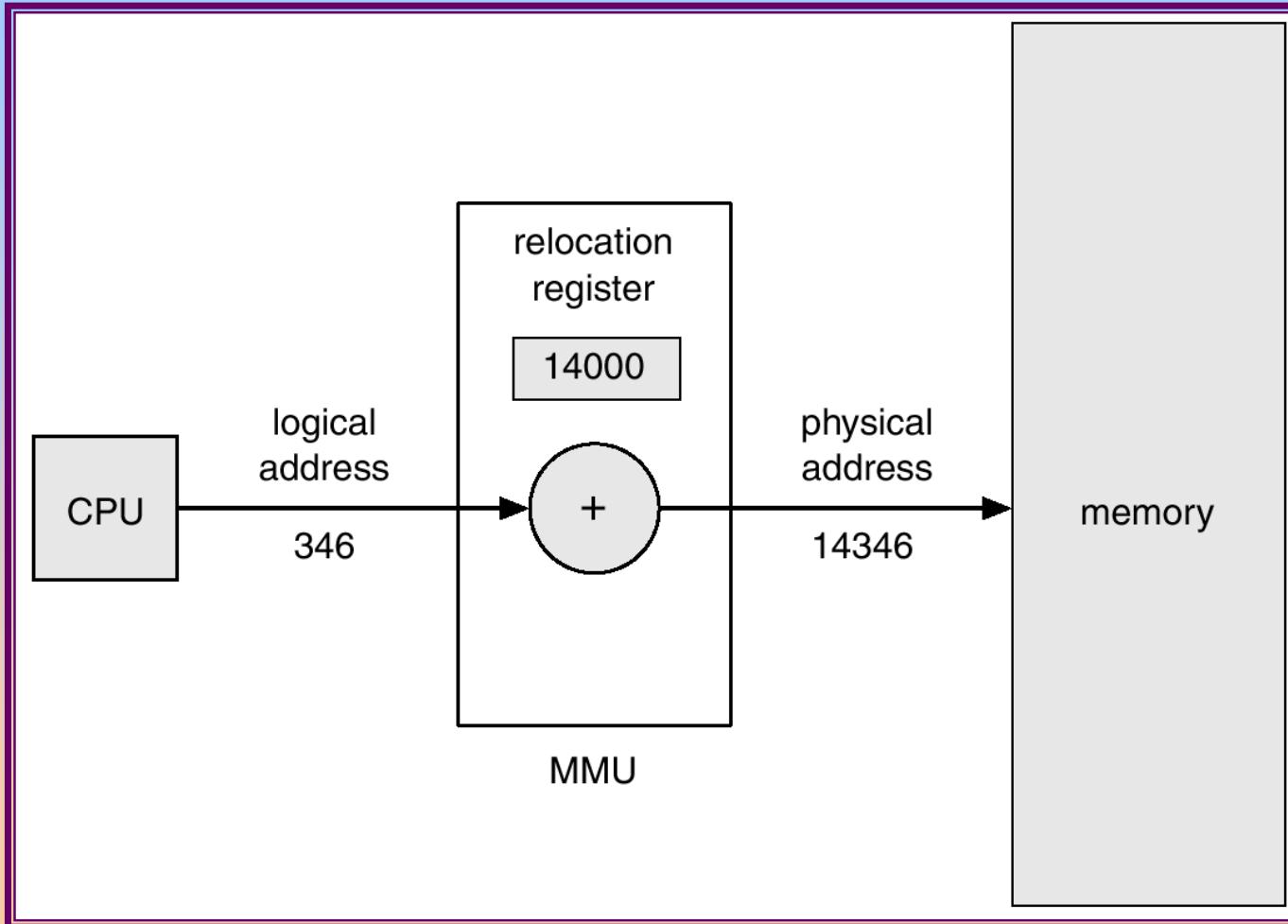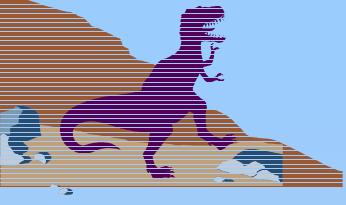
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.

- 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.
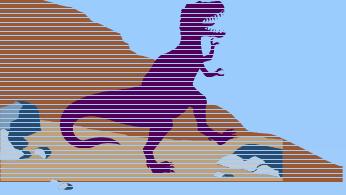
Dynamic relocation using a relocation register

relocation register

14000

CPU

logical address

346

+

MMU

physical address

14346
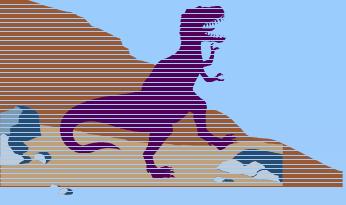
memory

Dynamic Loading

- Routine is not loaded until it is called

- Better memory-space utilization; unused routine is never loaded.

- Useful when large amounts of code are needed to handle infrequently occurring cases.

- No special support from the operating system is required implemented through program design.
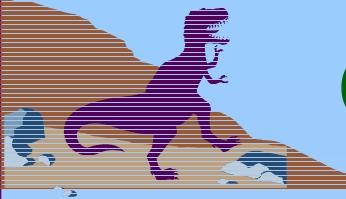
Dynamic Linking

- Linking postponed until execution time.

- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.

- Stub replaces itself with the address of the routine, and executes the routine.

- Operating system needed to check if routine is in processes' memory address.

- Dynamic linking is particularly useful for libraries.
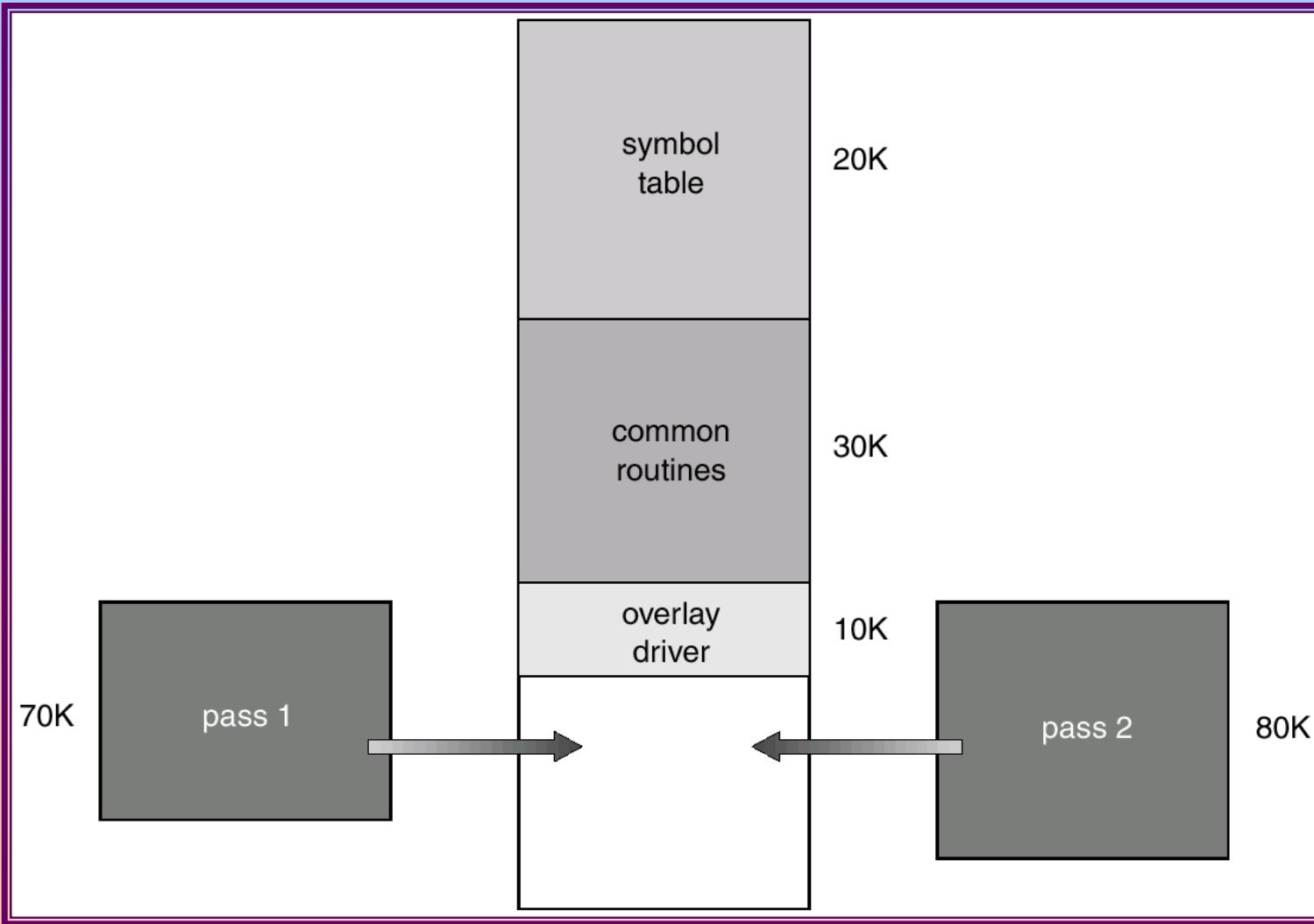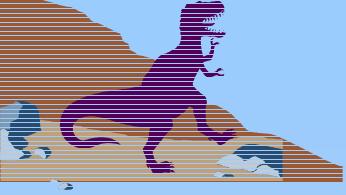
Overlays

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

- 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

Overlays for a Two-Pass Assembler
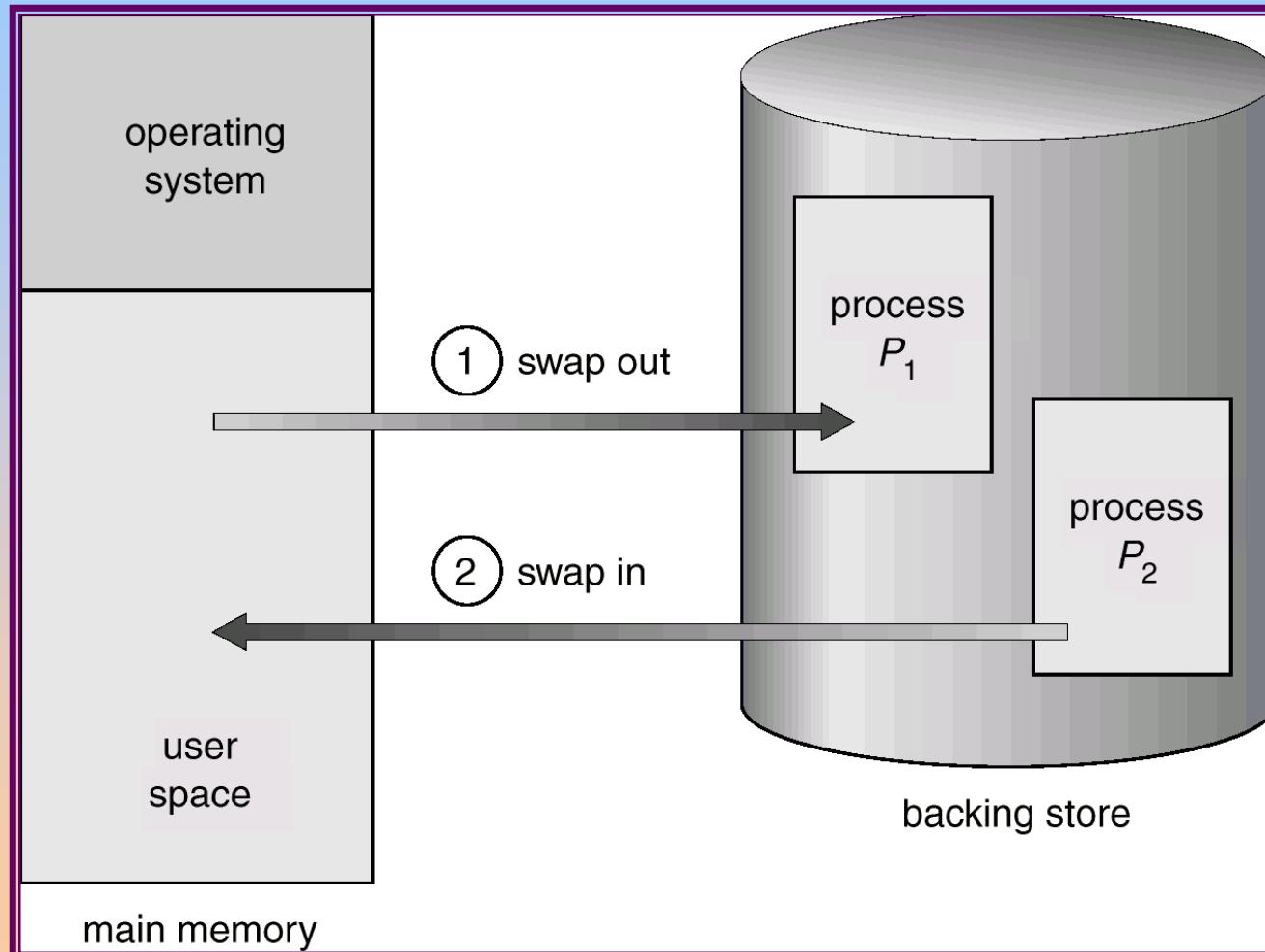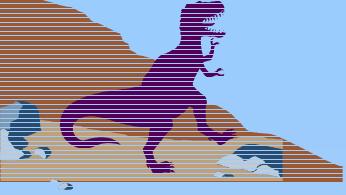
Swapping

- 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.

- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.

- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

Schematic View of Swapping

operating
system

① swap out

process
P_1

② swap in

process
P_2

user
space

backing store
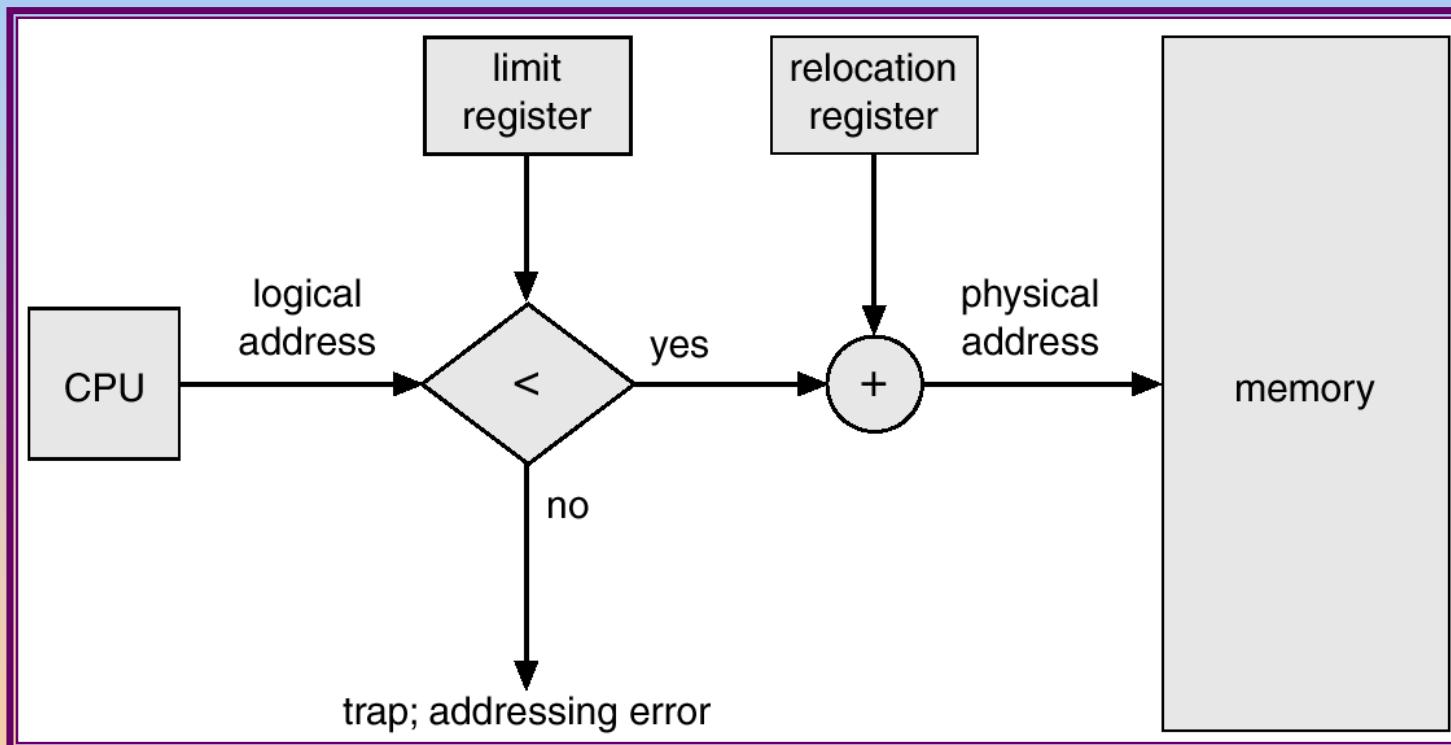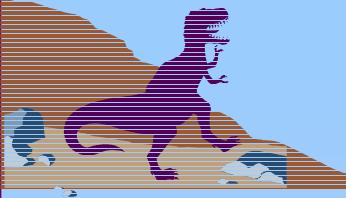
main memory

Contiguous Allocation

- Main memory usually into two partitions:
  - ✦ Resident operating system, usually held in low memory with interrupt vector.
  - ✦ User processes then held in high memory.

- Single-partition allocation
  - ✦ Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - ✦ Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

Hardware Support for Relocation and Limit Registers
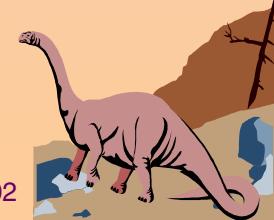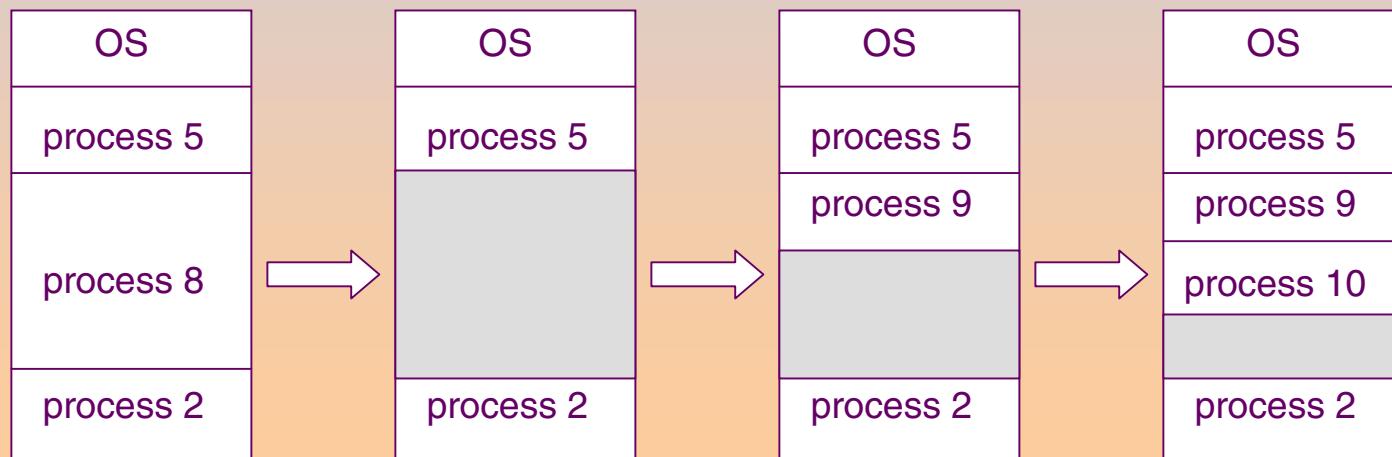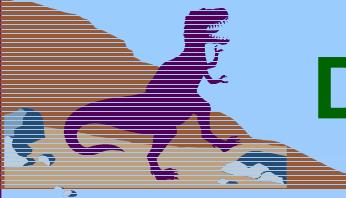
Contiguous Allocation (Cont.)

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

OS
process 5
process 8
process 2

⟹

OS
process 5
process 2

⟹

OS
process 5
process 9
process 2

⟹

OS
process 5
process 9
process 10
process 2
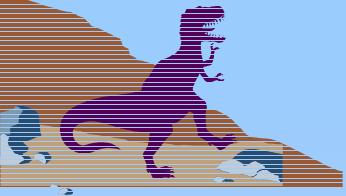
Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the *first* hole that is big enough.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
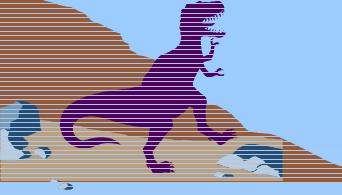
First-fit and best-fit better than worst-fit in terms of speed and storage utilization.
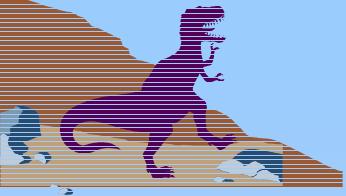
Fragmentation

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

Paging
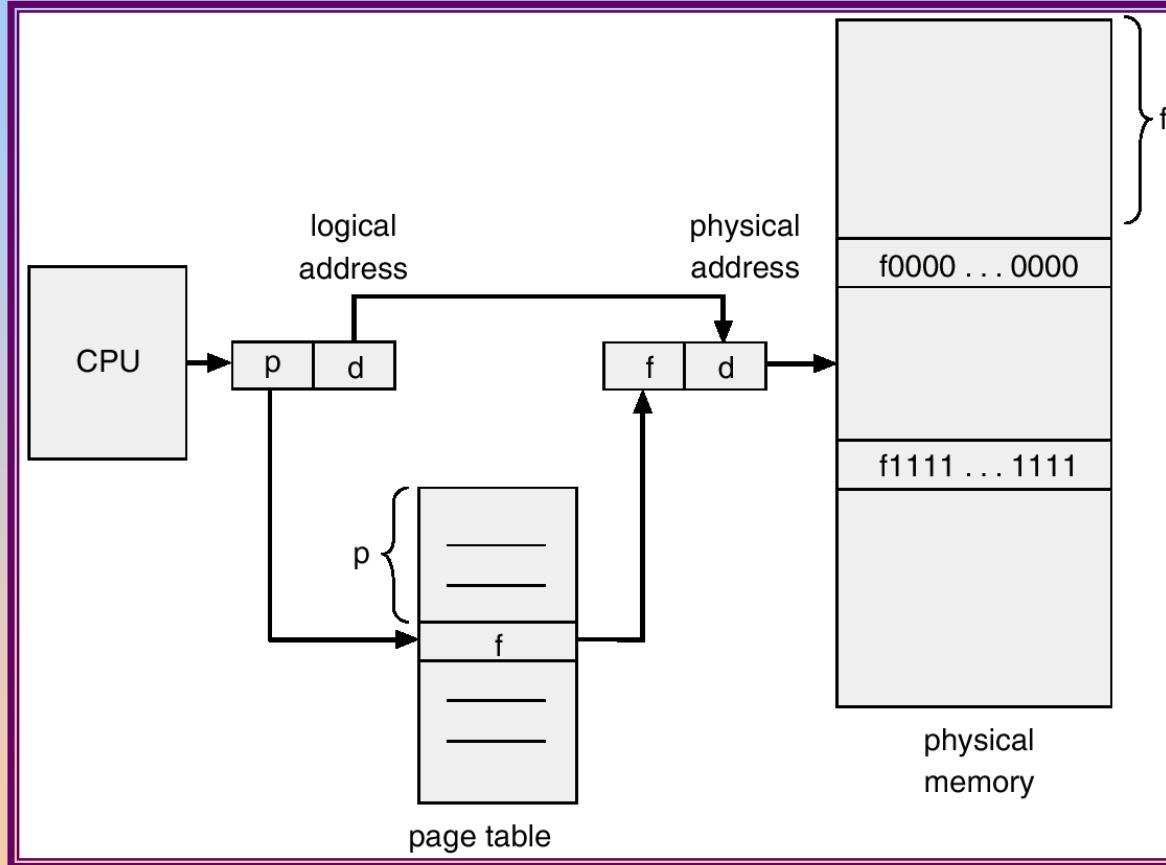
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).

- Divide logical memory into blocks of same size called **pages**.

- Keep track of all free frames.

- To run a program of size n pages, need to find n free frames and load program.

- Set up a page table to translate logical to physical addresses.

- Internal fragmentation.
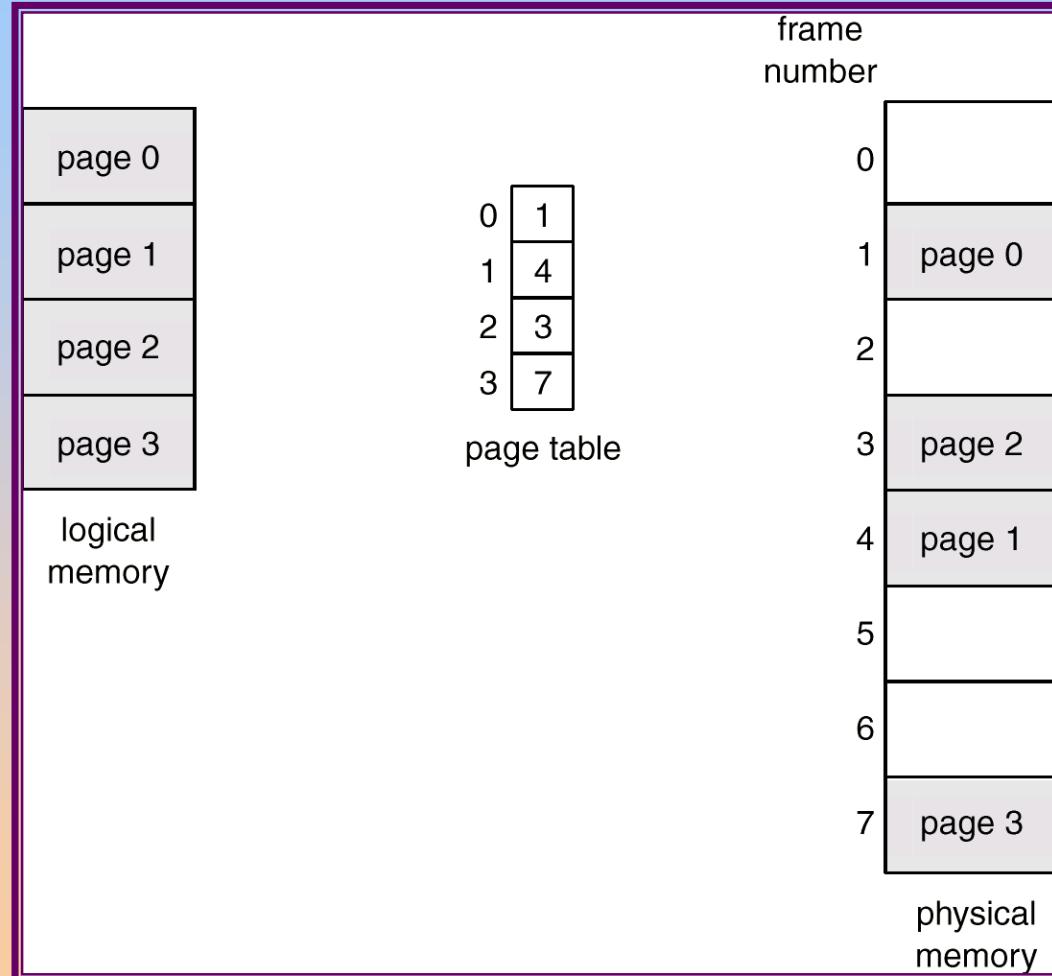
Address Translation Scheme

- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.

  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.
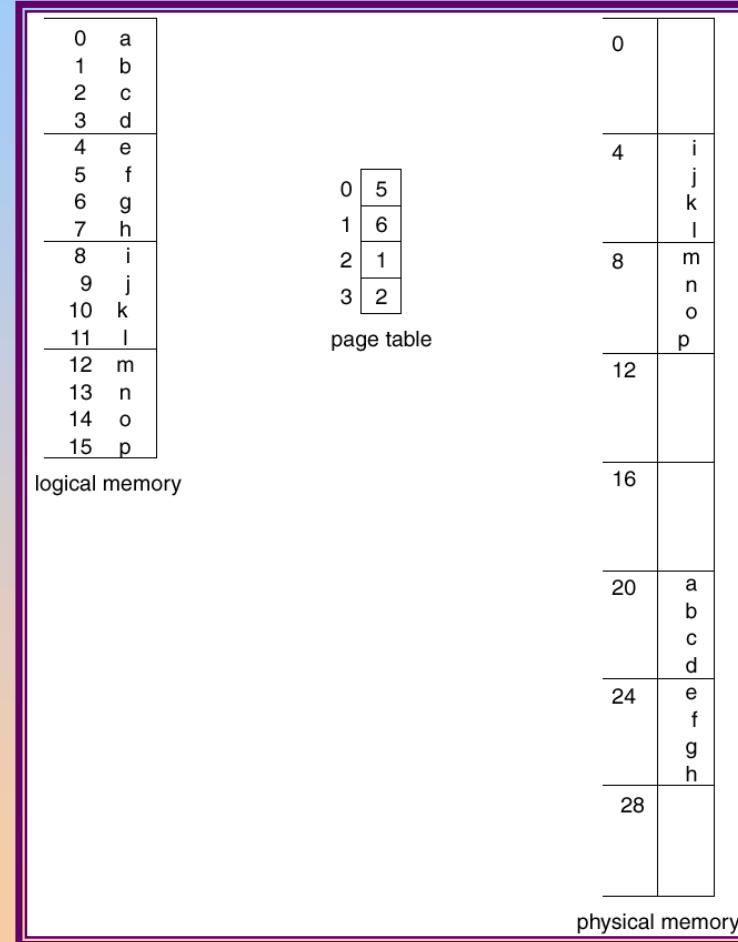
Address Translation Architecture

logical
address

physical
address

CPU → | p | d | | f | d | → physical memory

f0000 . . . 0000

f1111 . . . 1111

p {

| f |

page table

physical
memory

Paging Example

frame
number

page 0
page 1
page 2
page 3

logical
memory

0	1
1	4
2	3
3	7

page table

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

physical
memory

Paging Example

logical memory

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

page table

0	5
1	6
2	1
3	2

physical memory

Free Frames

free-frame list
14
13
18
20
15

page 0
page 1
page 2
page 3
new process

13
14
15
16
17
18
19
20
21

(a)

free-frame list
15

page 0
page 1
page 2
page 3
new process

0 | 14
1 | 13
2 | 18
3 | 20

new-process page table

13 | page 1
14 | page 0
15 |
16 |
17 |
18 | page 2
19 |
20 | page 3
21 |

(b)

Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory.

- *Page-table base register (*PTBR) points to the page table.

- *Page-table length register* (PRLR) indicates size of the page table.

- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*

Associative Memory
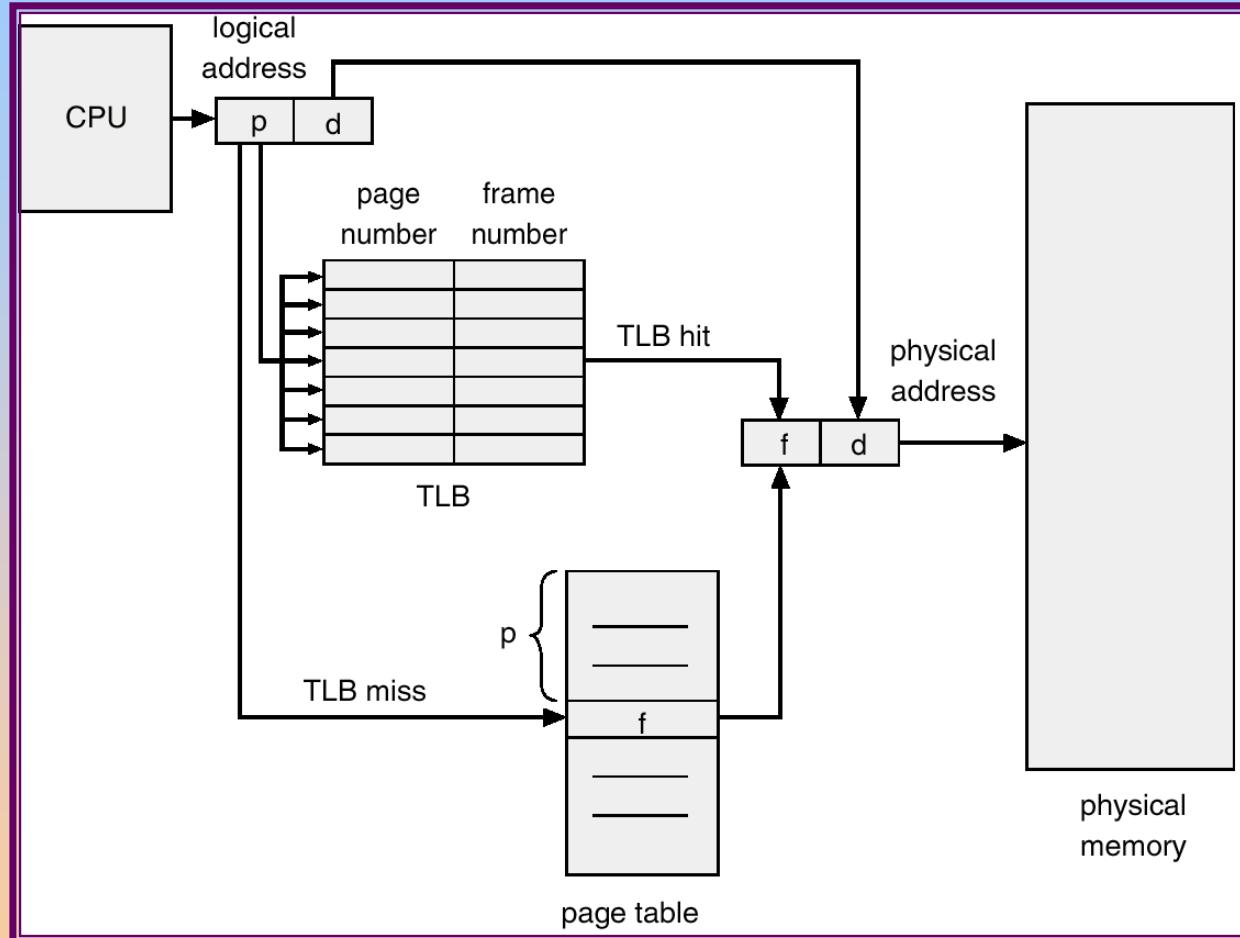
- Associative memory – parallel search
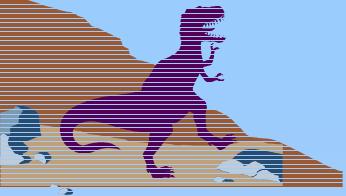
Page #	Frame #

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
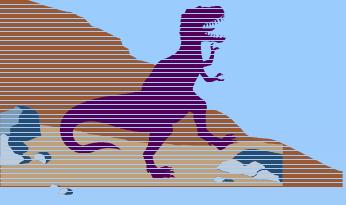
Effective Access Time

- Associative Lookup = ε time unit

- Assume memory cycle time is 1 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)

$$EAT = (1 + \varepsilon)\, \alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$
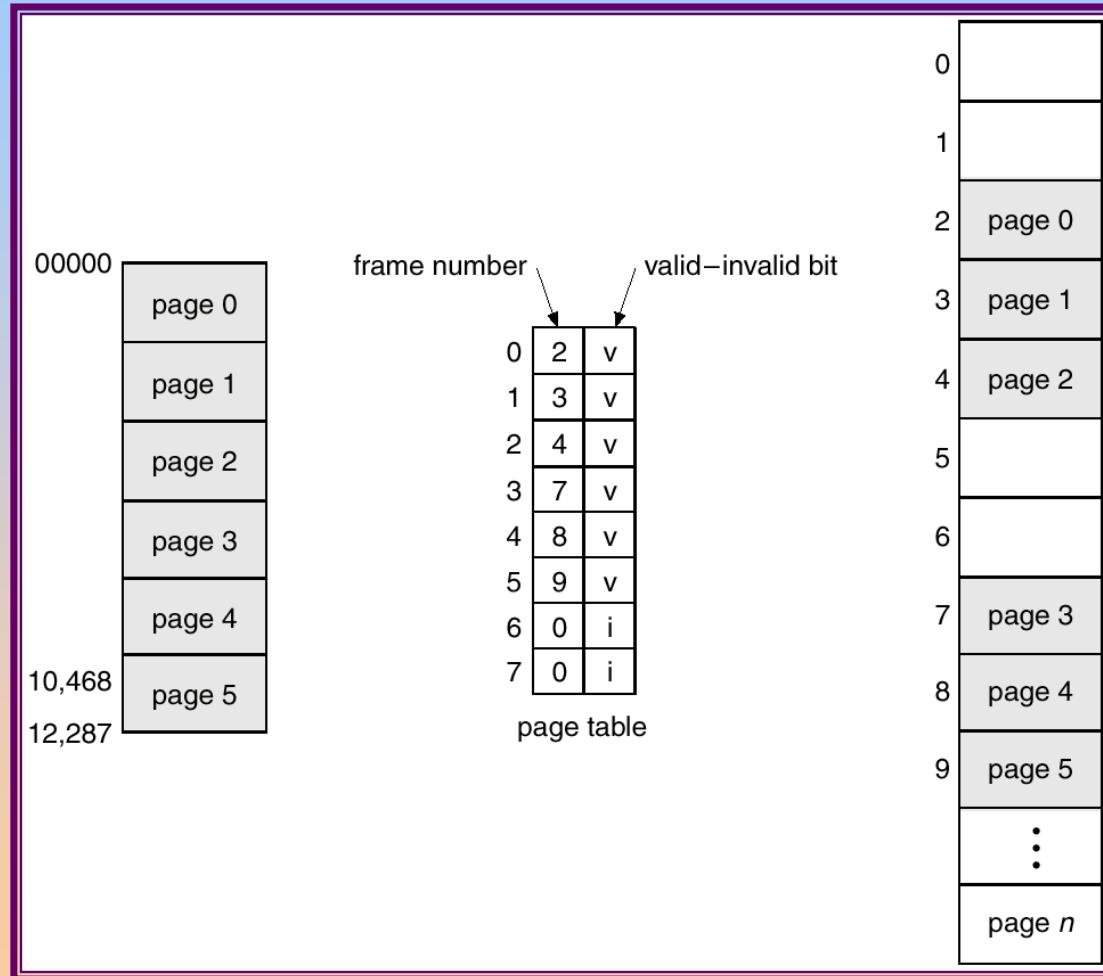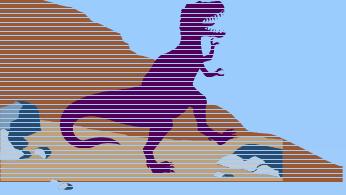
Memory Protection

- Memory protection implemented by associating protection bit with each frame.

- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.
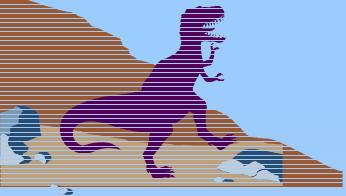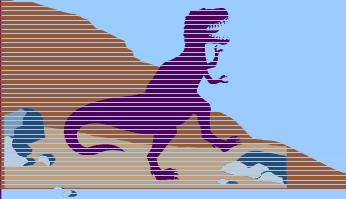
Page Table Structure

- Hierarchical Paging

- Hashed Page Tables

- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables.

- A simple technique is a two-level page table.
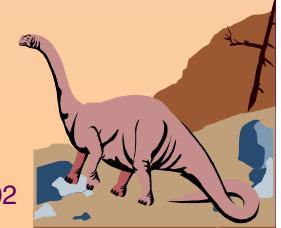
Two-Level Paging Example
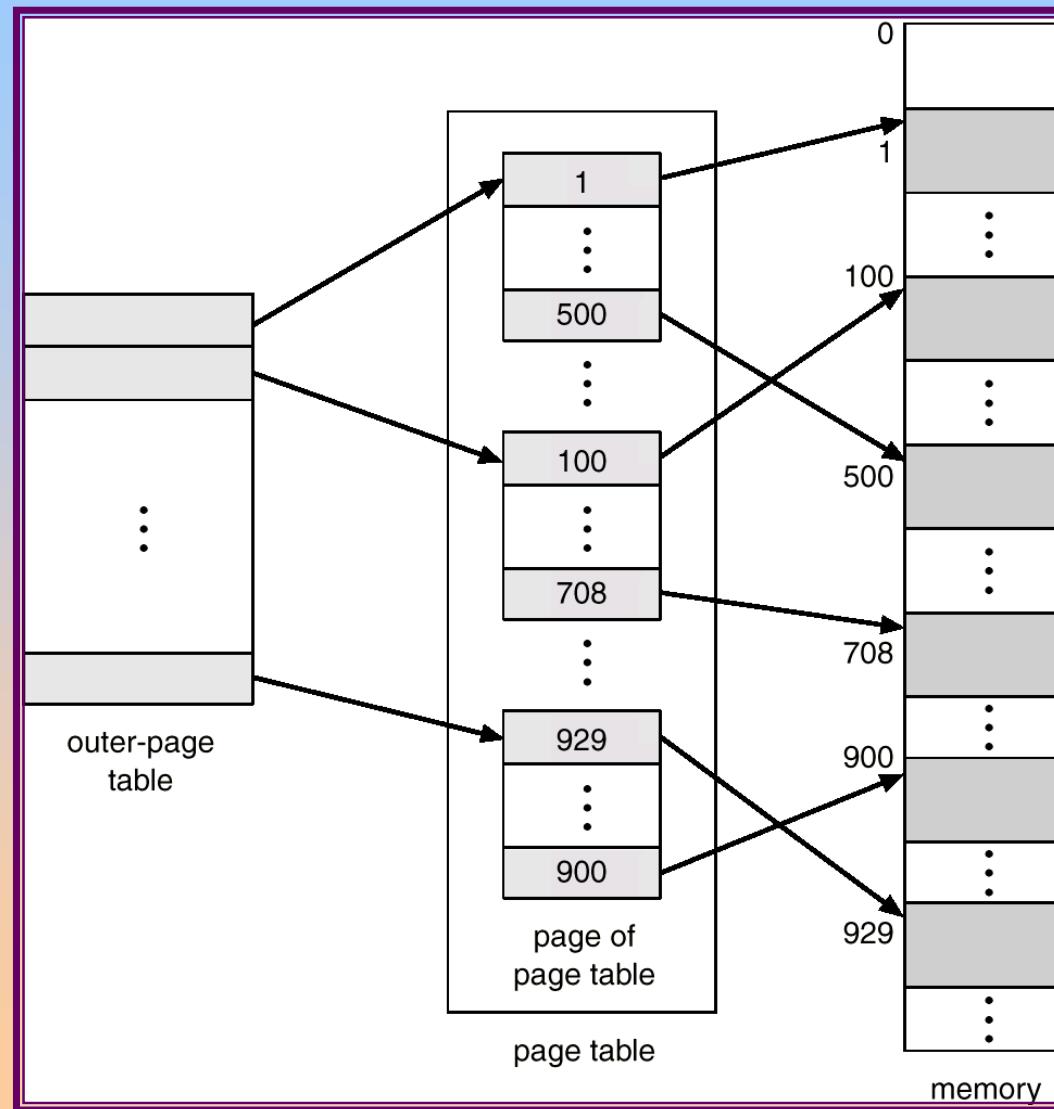
■ A logical address (on 32-bit machine with 4K page size) is divided into:

✦ a page number consisting of 20 bits.

✦ a page offset consisting of 12 bits.

■ Since the page table is paged, the page number is further divided into:

✦ a 10-bit page number.

✦ a 10-bit page offset.

■ Thus, a logical address is as follows:
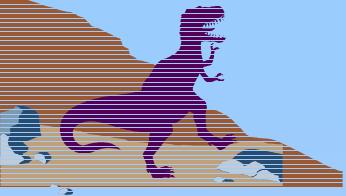
page number		page offset
p_i	p_2	d
10	10	12

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.
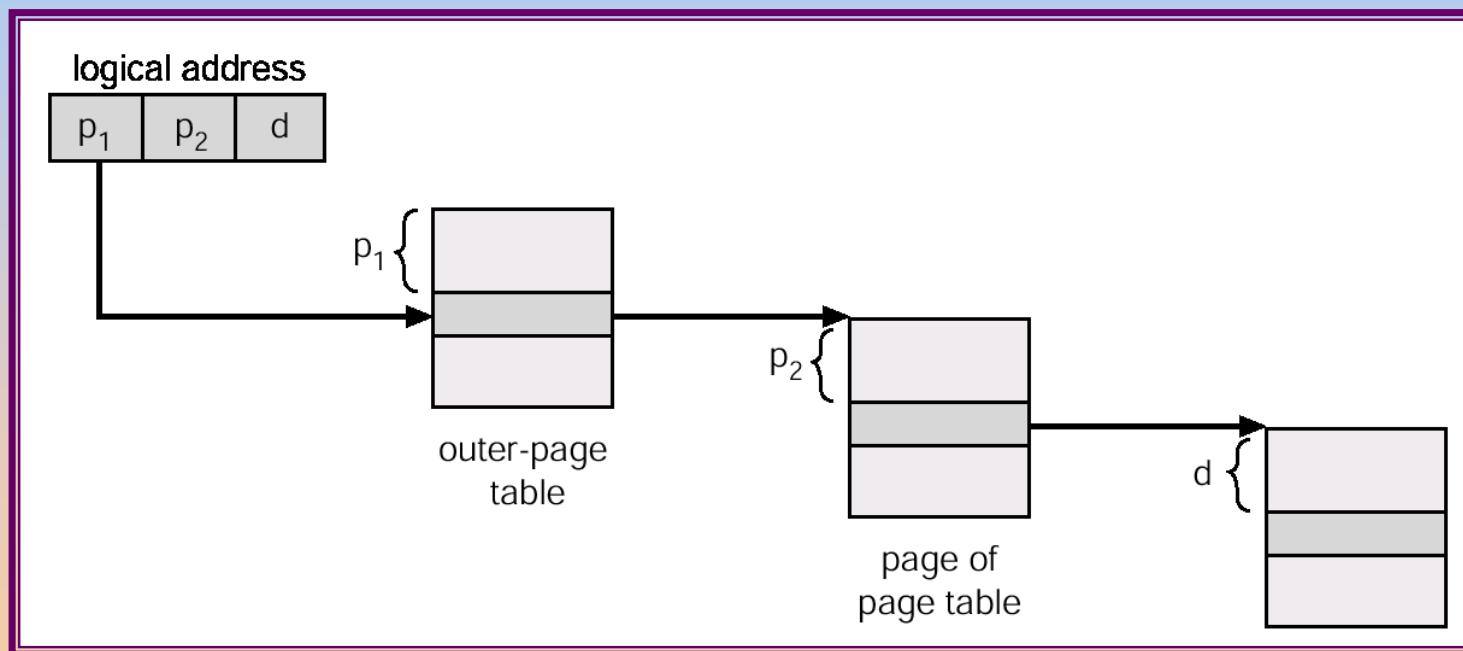
Two-Level Page-Table Scheme
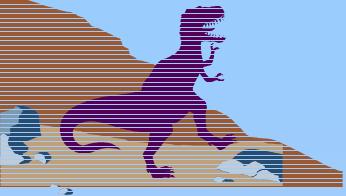
Address-Translation Scheme

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

logical address

| p_1 | p_2 | d |

p_1 { outer-page table

p_2 { page of page table

d {
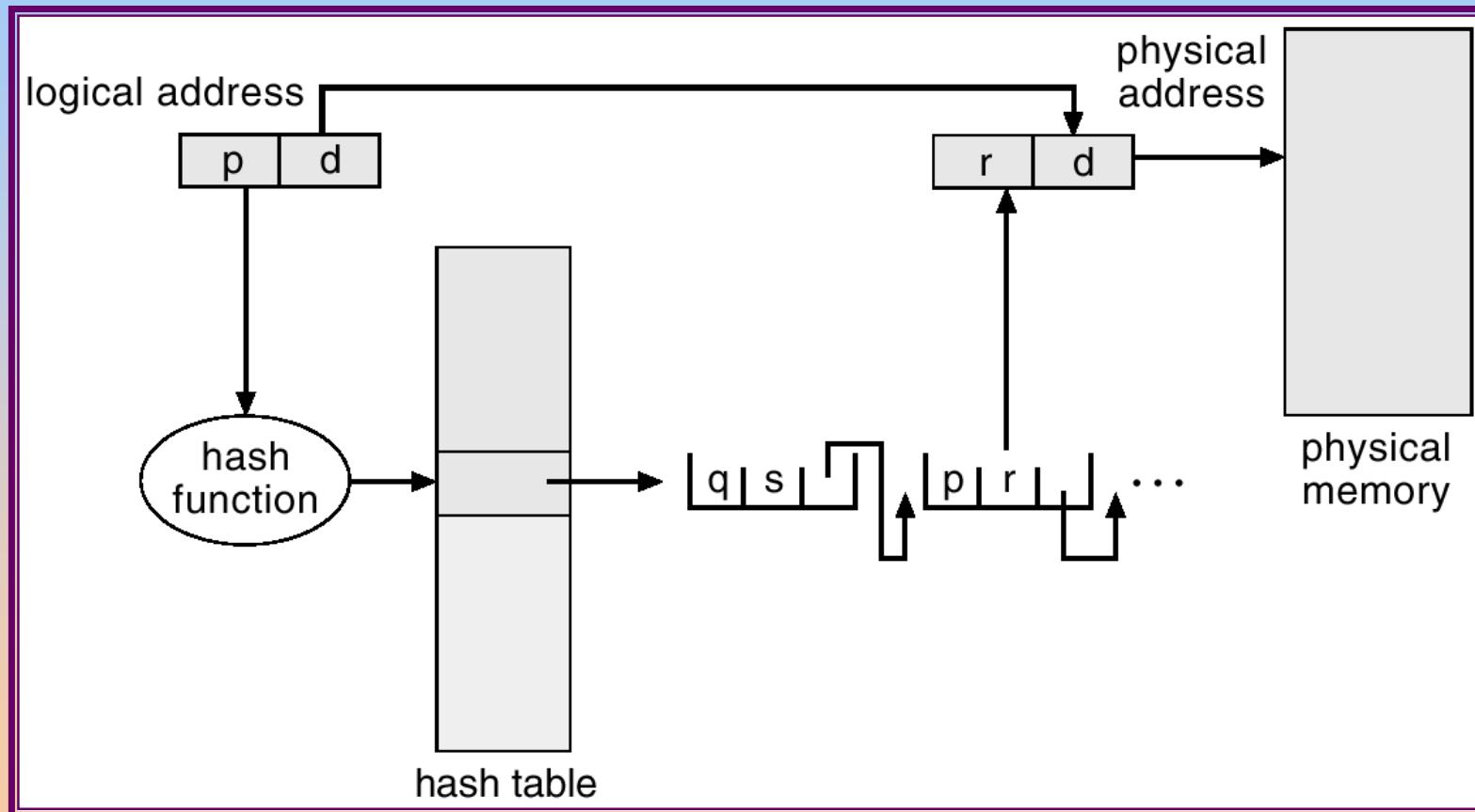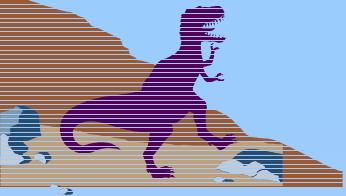
Hashed Page Tables

- Common in address spaces > 32 bits.

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.

- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
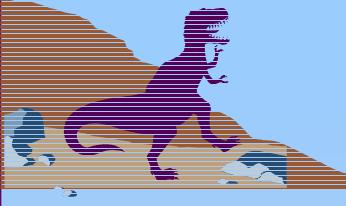
Hashed Page Table
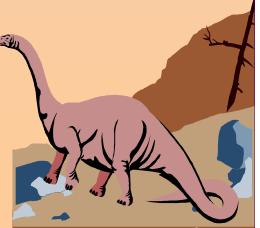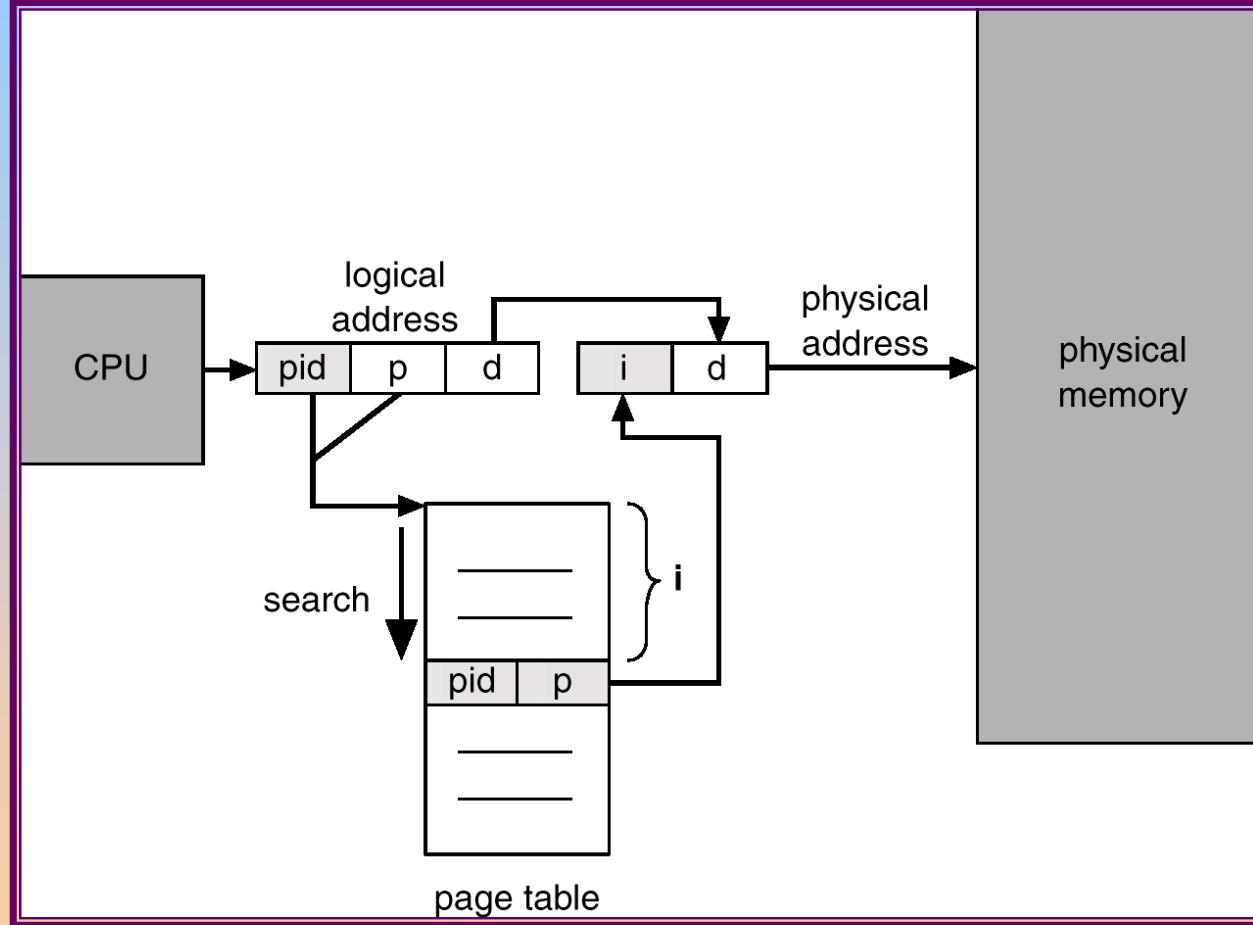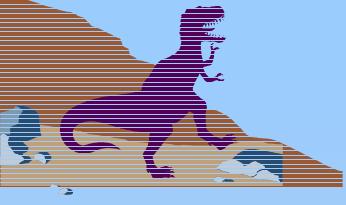
Inverted Page Table

- One entry for each real page of memory.

- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

- Use hash table to limit the search to one — or at most a few — page-table entries.

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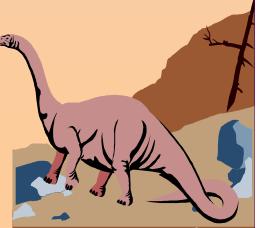
Shared Pages
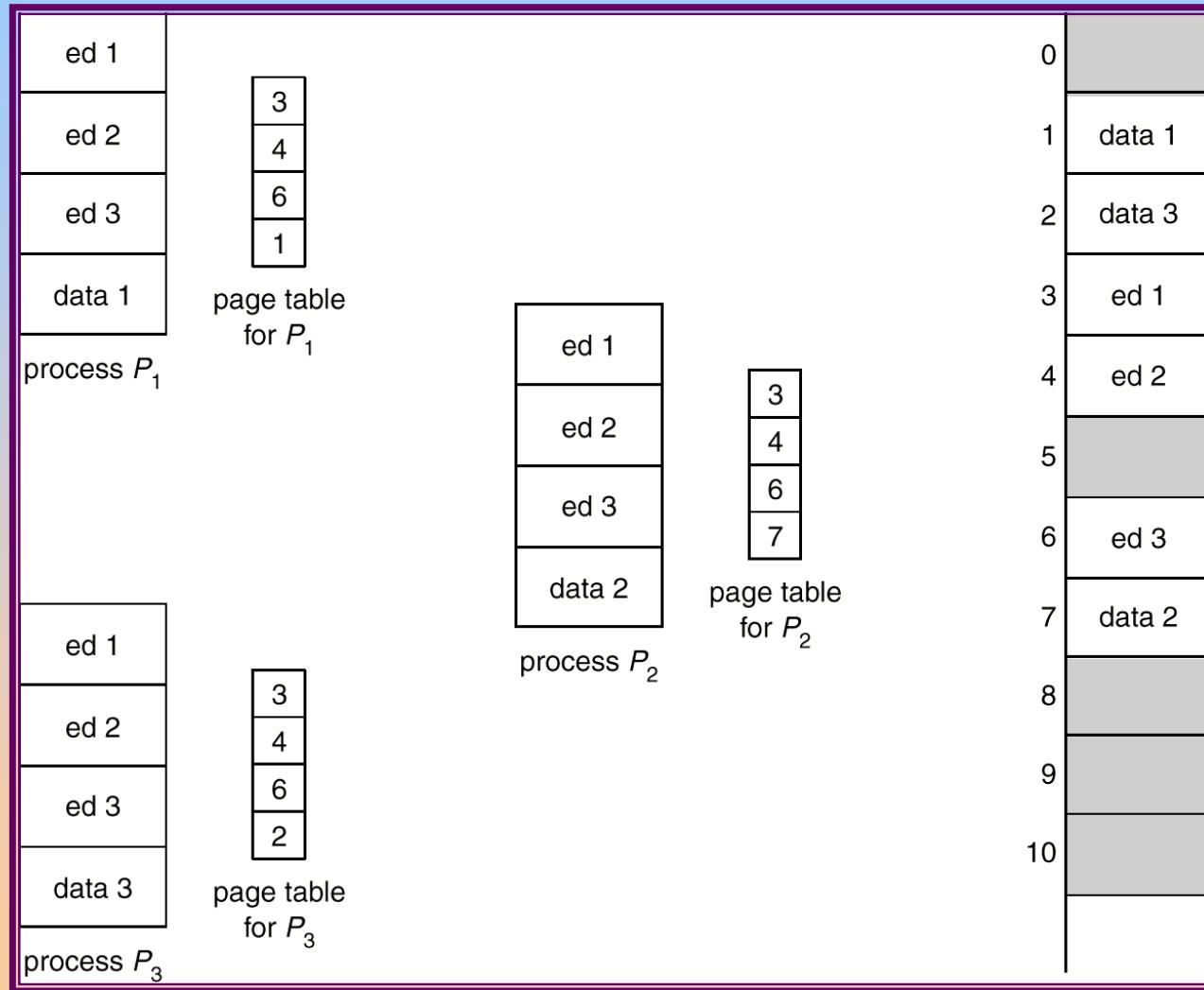
- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
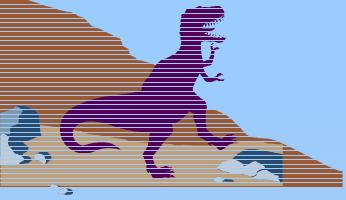
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example

ed 1		0	
ed 2	3	1	data 1
ed 3	4	2	data 3
data 1	6	3	ed 1
	1	4	ed 2

page table for P_1

process P_1

ed 1	3
ed 2	4
ed 3	6
data 2	7

page table for P_2

process P_2

ed 1	3
ed 2	4
ed 3	6
data 3	2

page table for P_3

process P_3

0	
1	data 1
2	data 3
3	ed 1
4	ed 2
5	
6	ed 3
7	data 2
8	
9	
10	
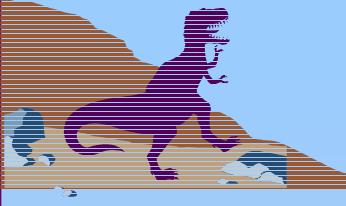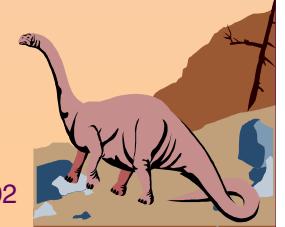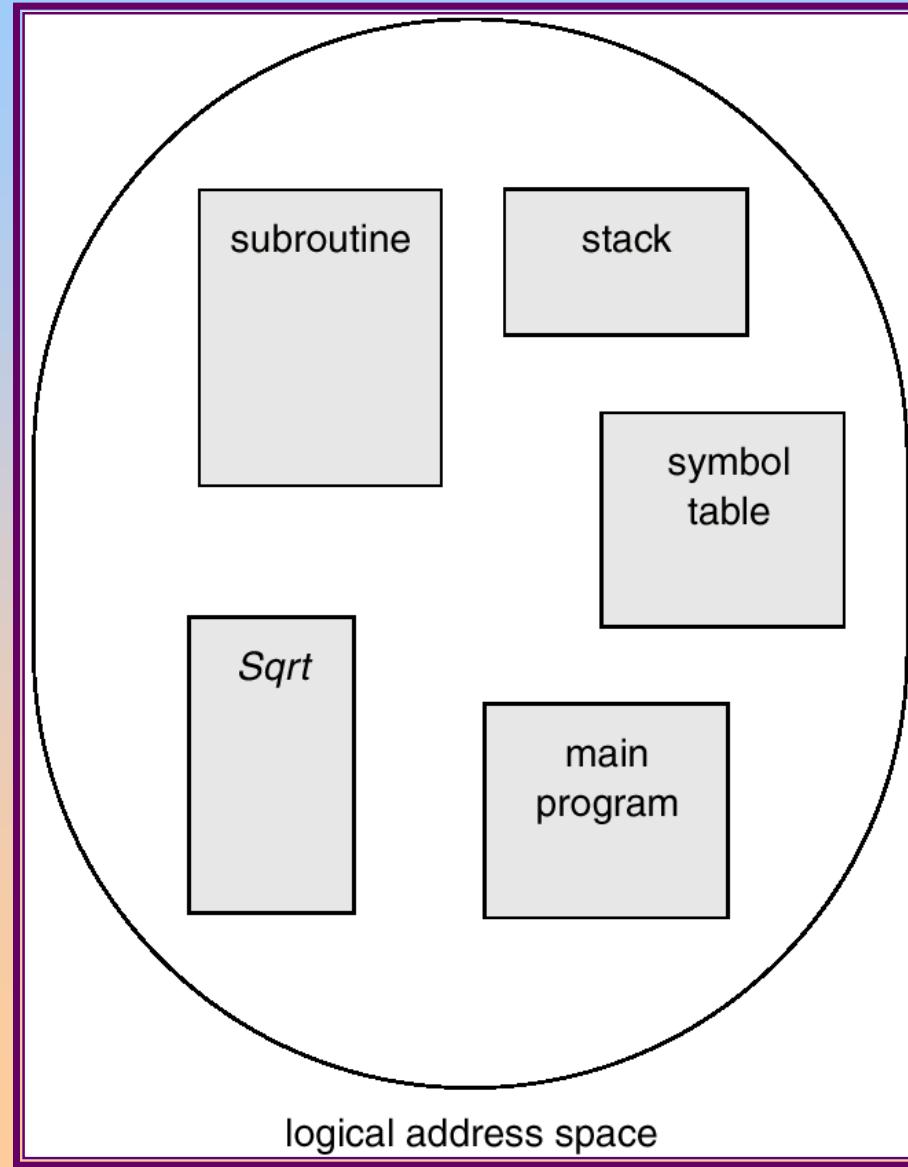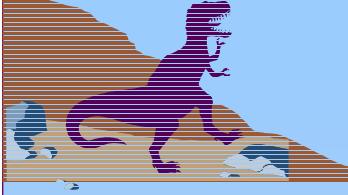
Segmentation

- 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

subroutine

stack

symbol
table

Sqrt

main
program
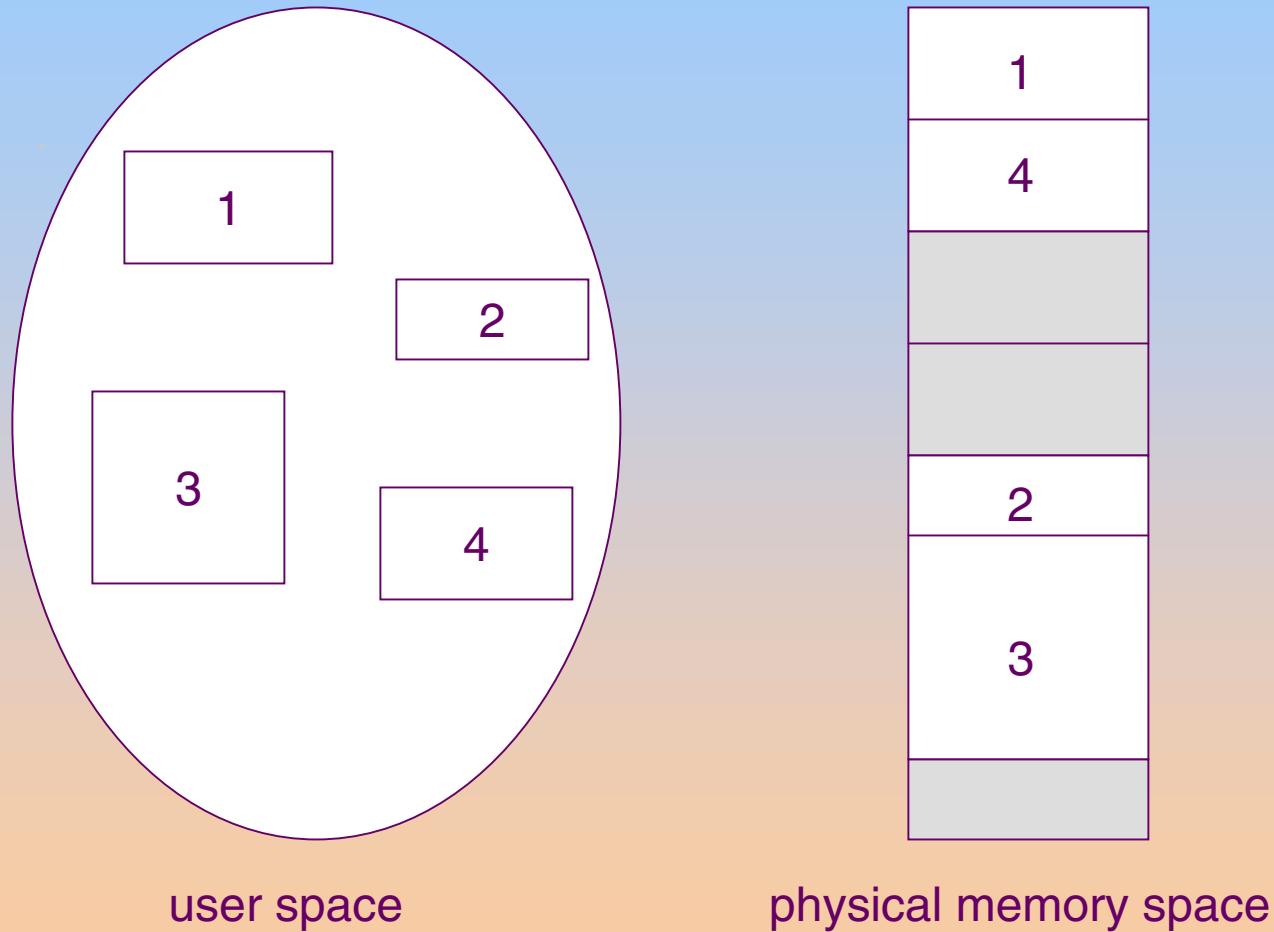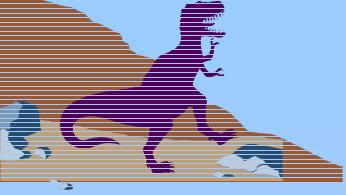
logical address space
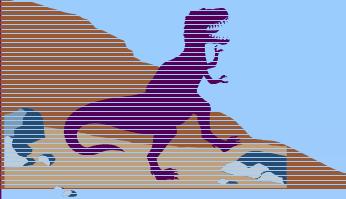
Logical View of Segmentation

user space   physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- *Segment table* – maps two-dimensional physical addresses; each table entry has:

  - base – contains the starting physical address where the segments reside in memory.

  - *limit* – specifies the length of the segment.

- *Segment-table base register (STBR)* points to the segment table's location in memory.

- *Segment-table length register (STLR)* indicates number of segments used by a program;

  segment number s is legal if $s <$ STLR.

Segmentation Architecture (Cont.)
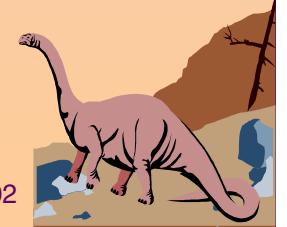
- Relocation.
  - dynamic
  - by segment table
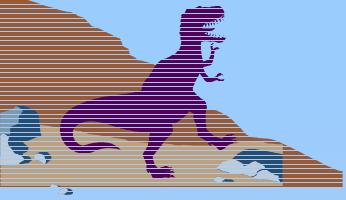
- Sharing.
  - shared segments
  - same segment number

- Allocation.
  - first fit/best fit
  - external fragmentation
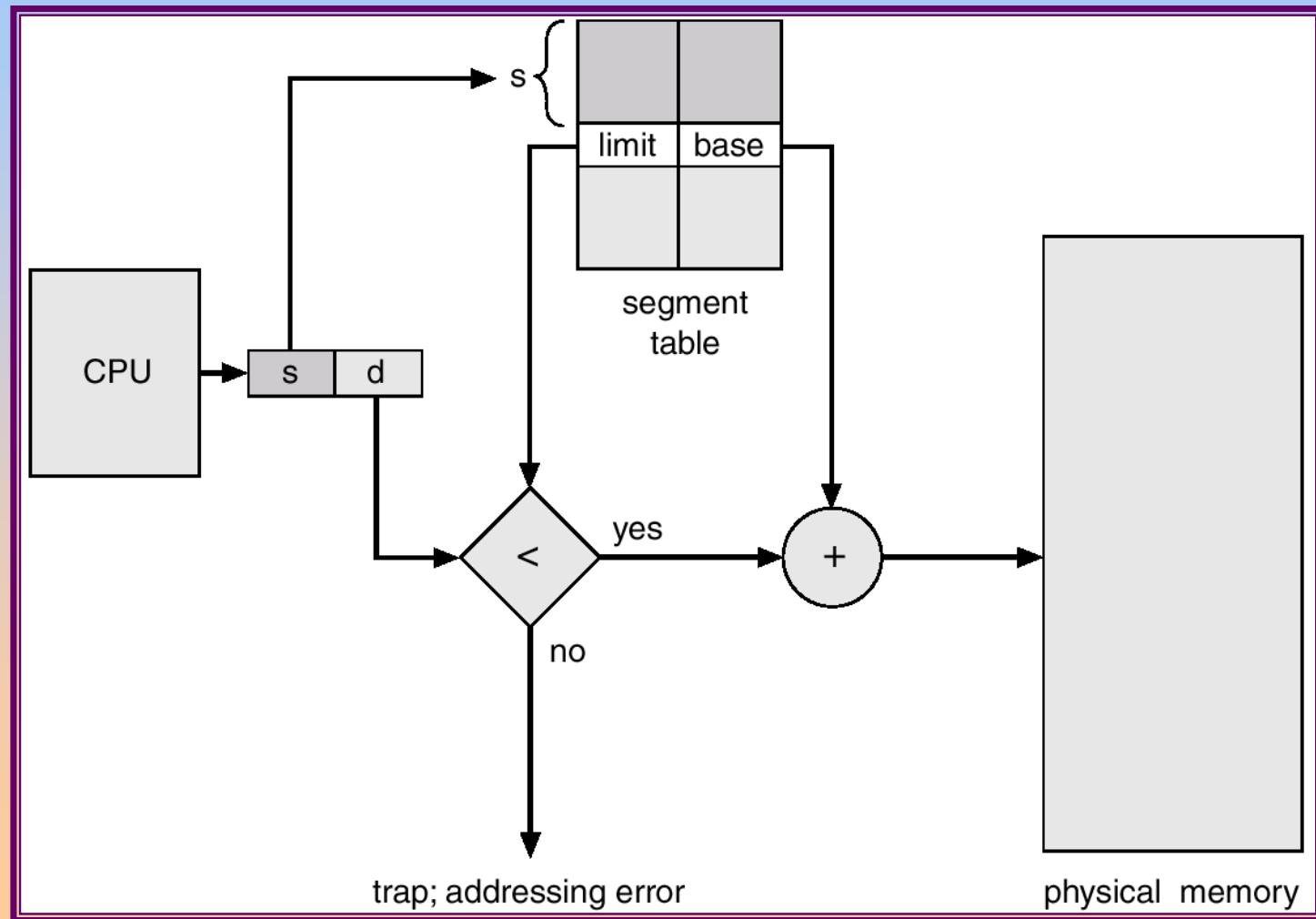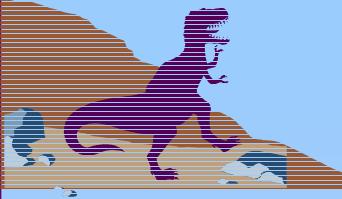
Segmentation Architecture (Cont.)
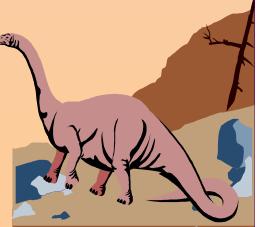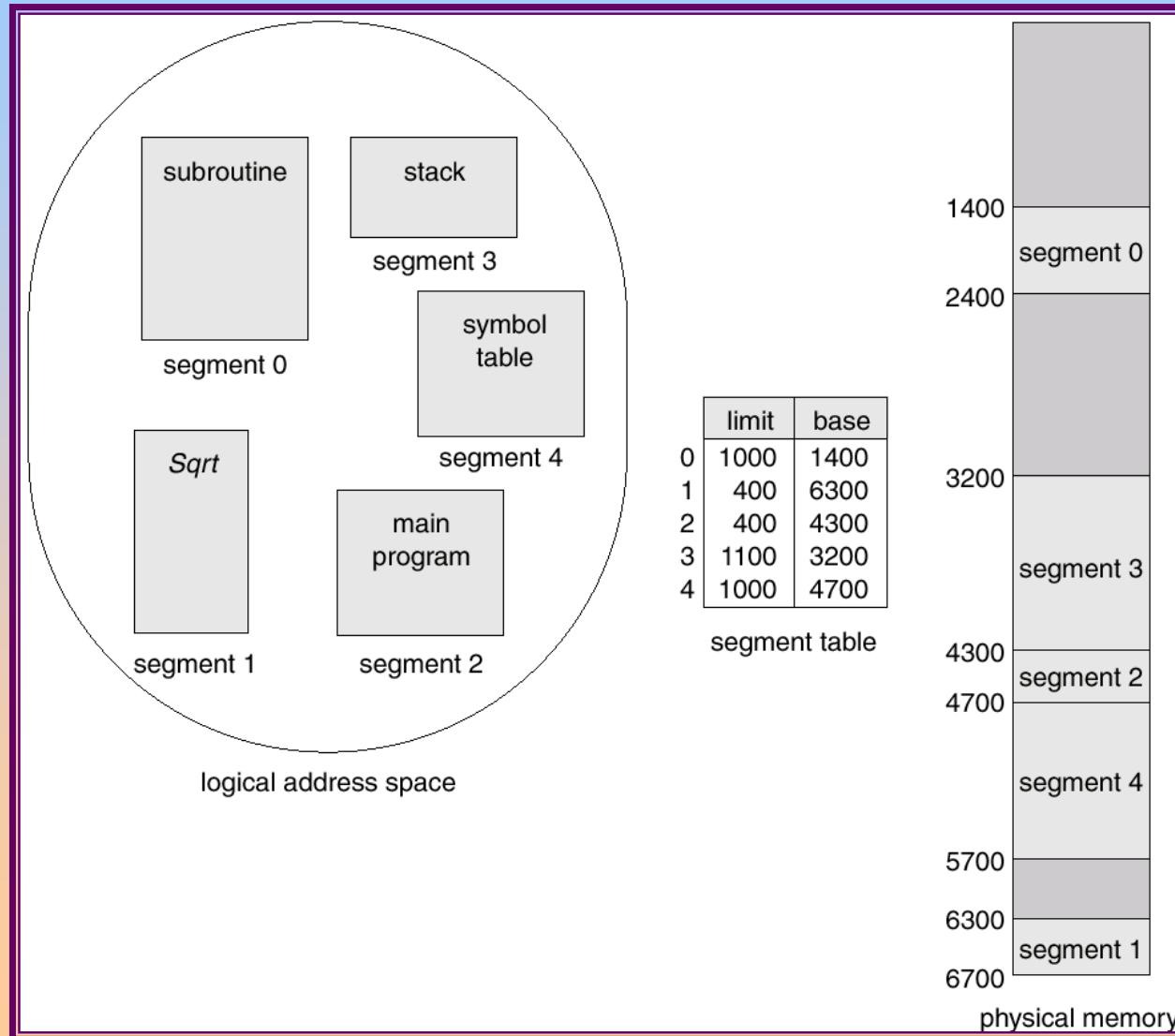
- Protection. With each entry in segment table associate:
  - validation bit = 0 $\Rightarrow$ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.
- A segmentation example is shown in the following diagram

Segmentation Hardware
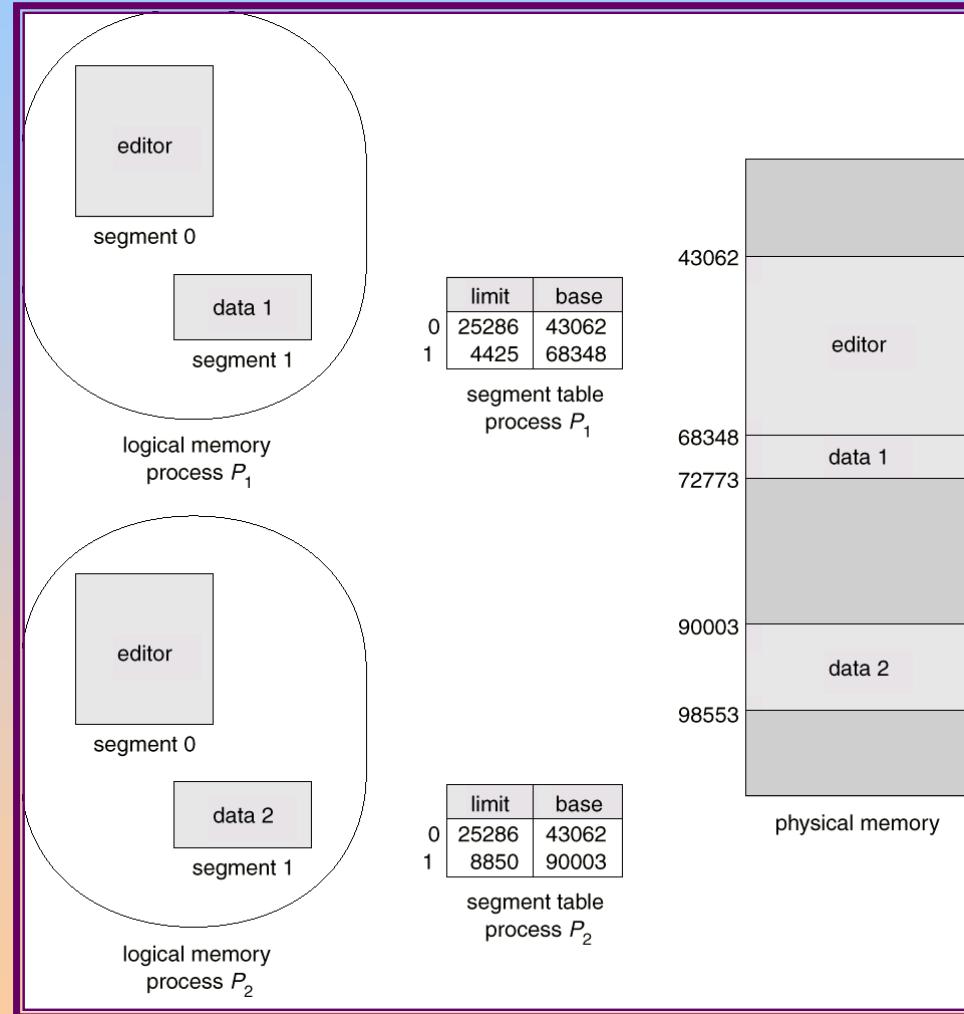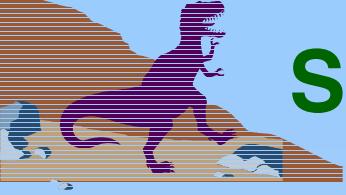
Example of Segmentation
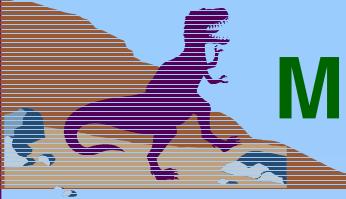
Sharing of Segments
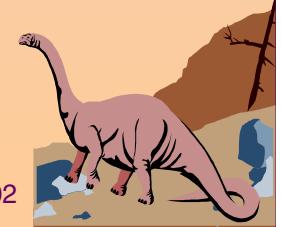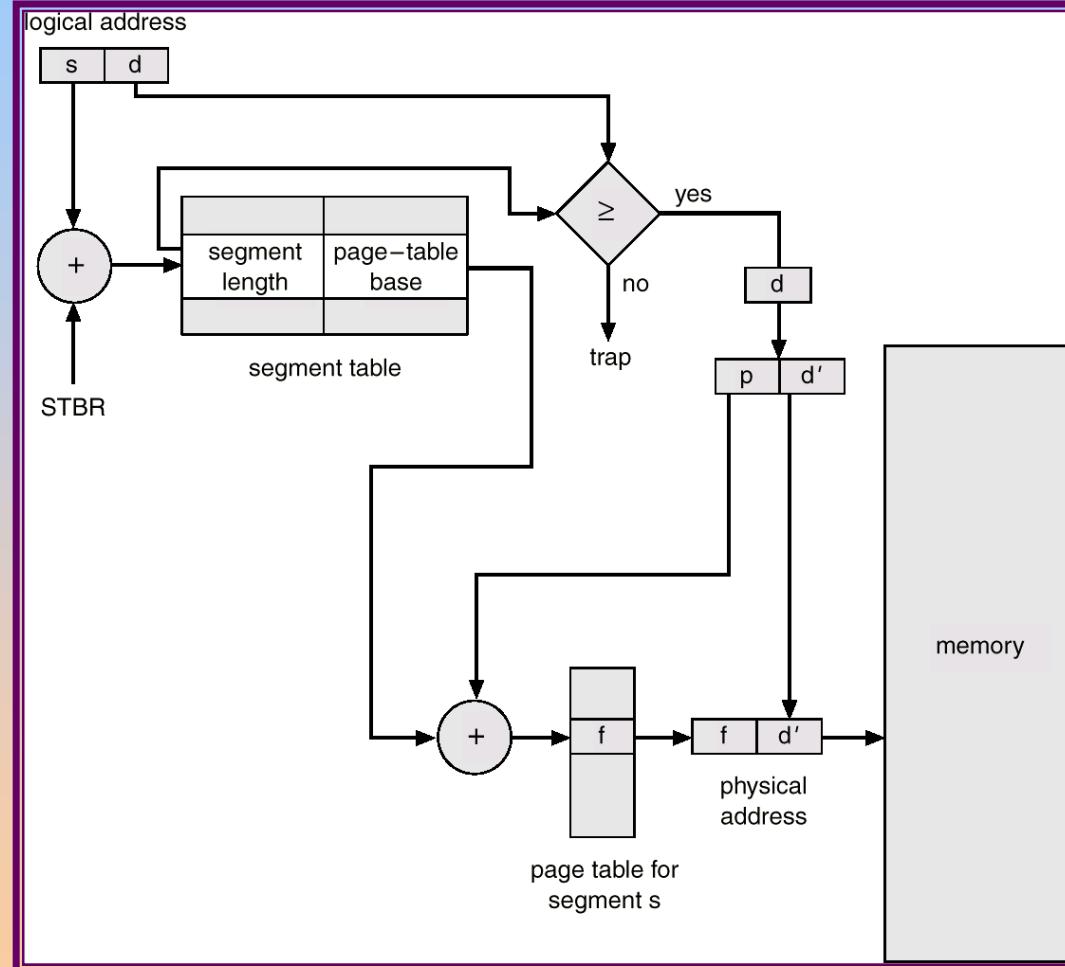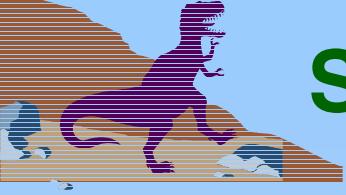
Segmentation with Paging – MULTICS

- 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.
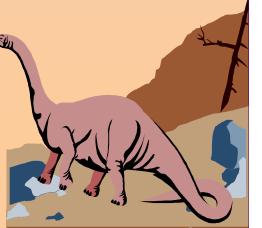
MULTICS Address Translation Scheme
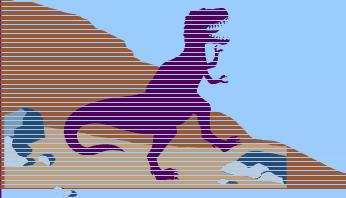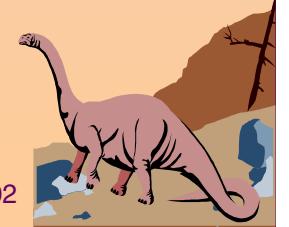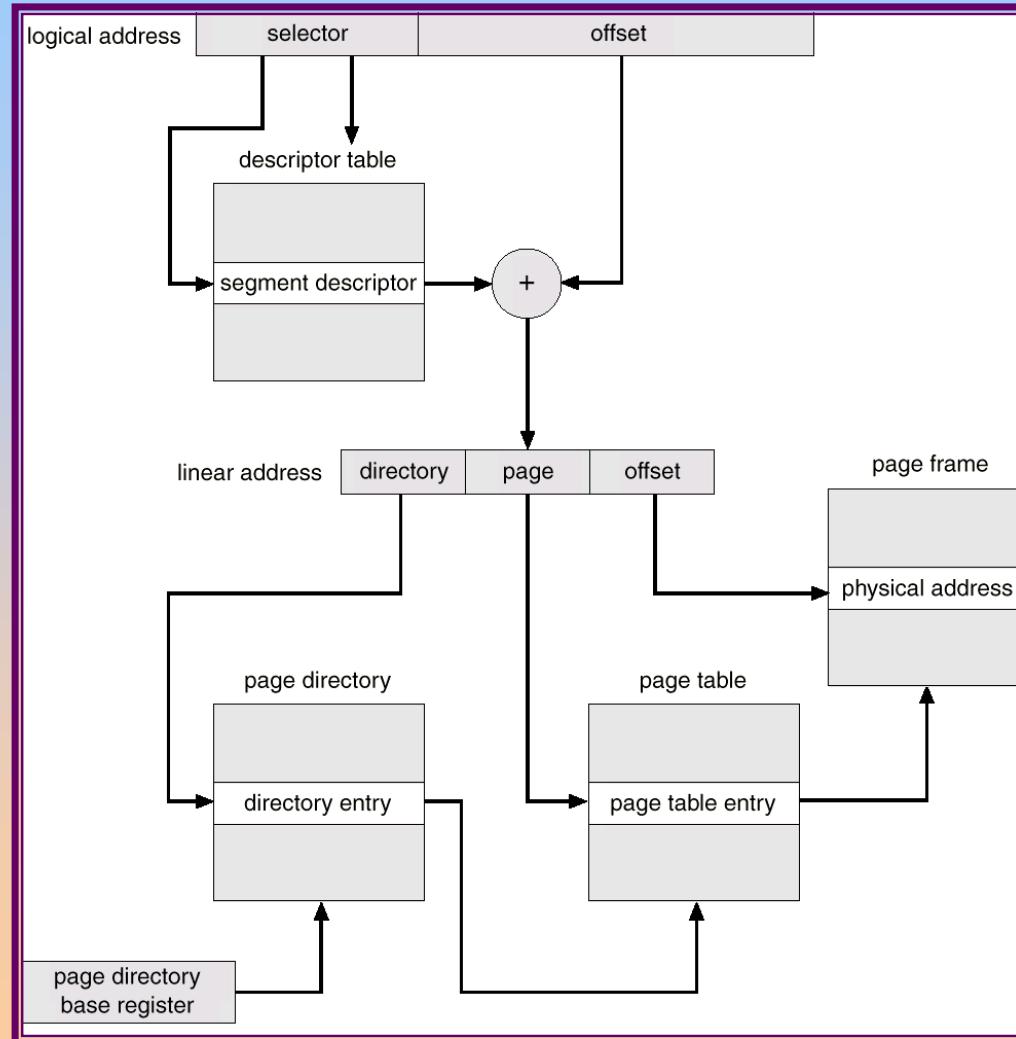
Segmentation with Paging – Intel 386
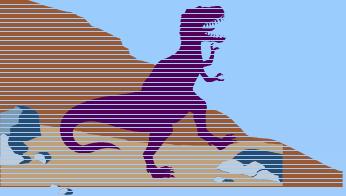
- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.
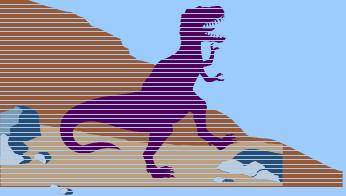
Intel 30386 Address Translation

Chapter 10: Virtual Memory
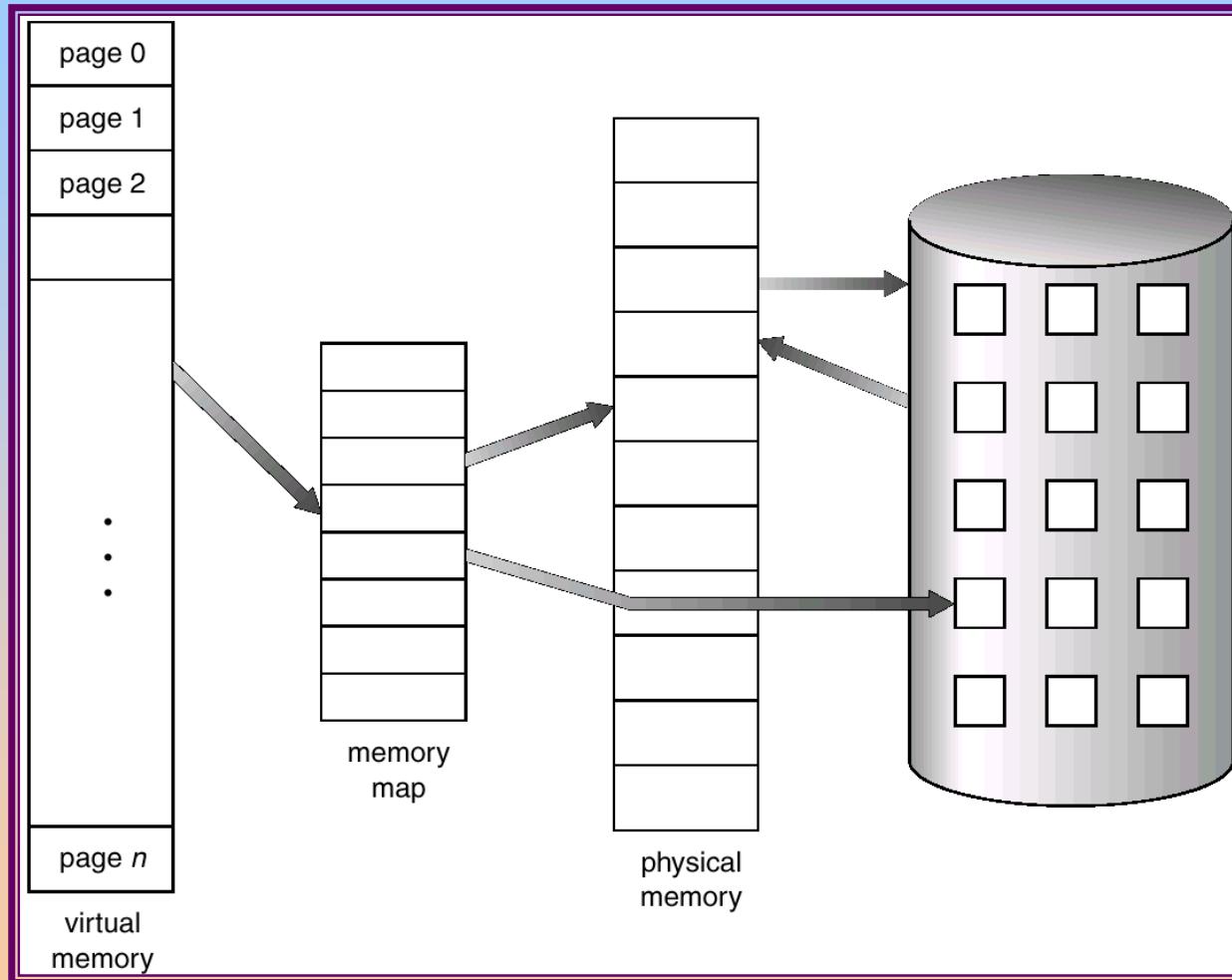
- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Operating System Examples
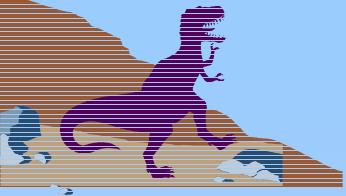
Background

- **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 That is Larger Than Physical Memory

page 0

page 1

page 2

. . .

page *n*

virtual memory

memory map

physical memory
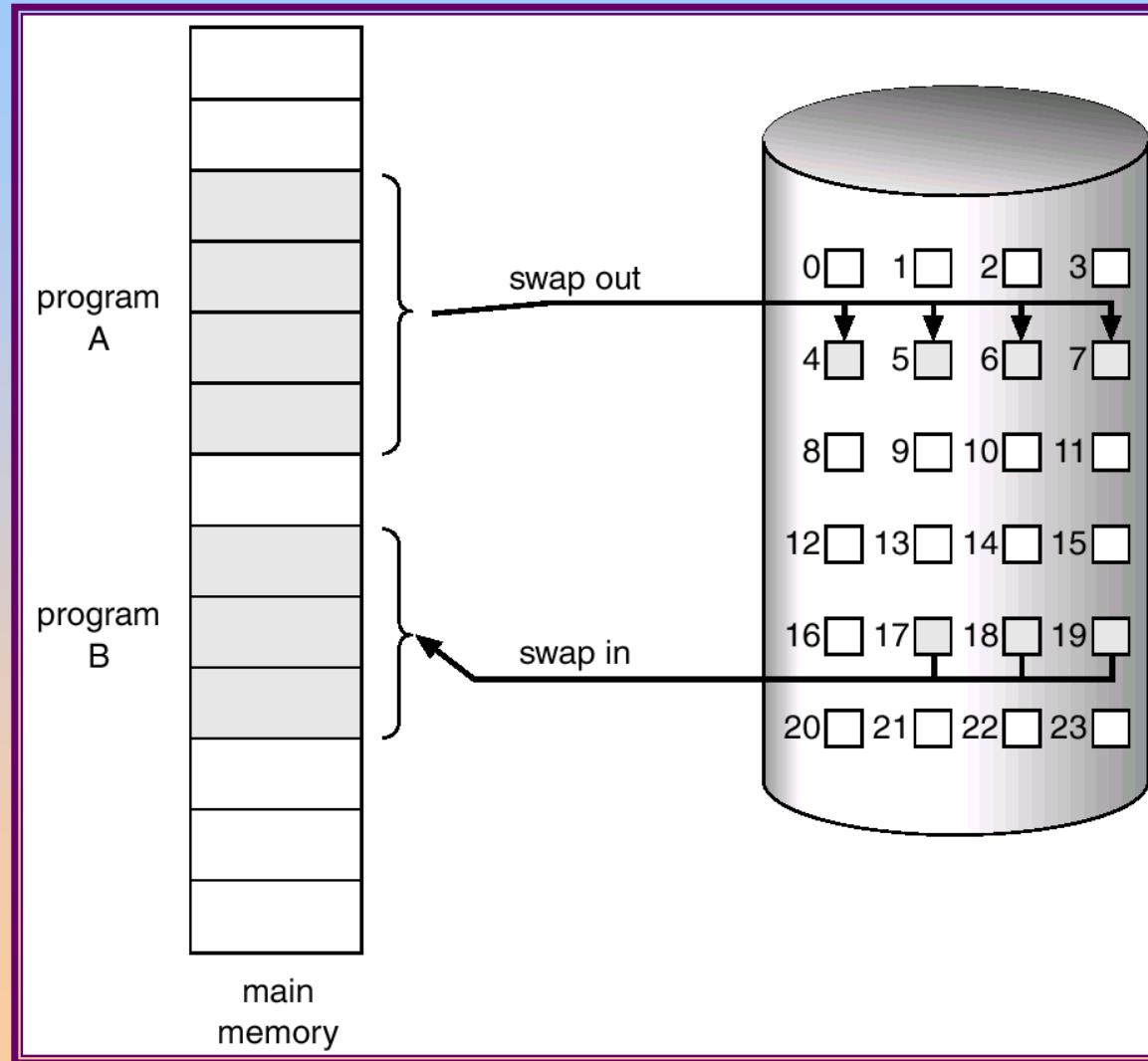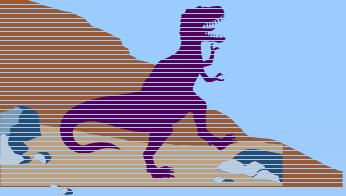
Demand Paging

■ Bring a page into memory only when it is needed.

  ✦ 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 to memory
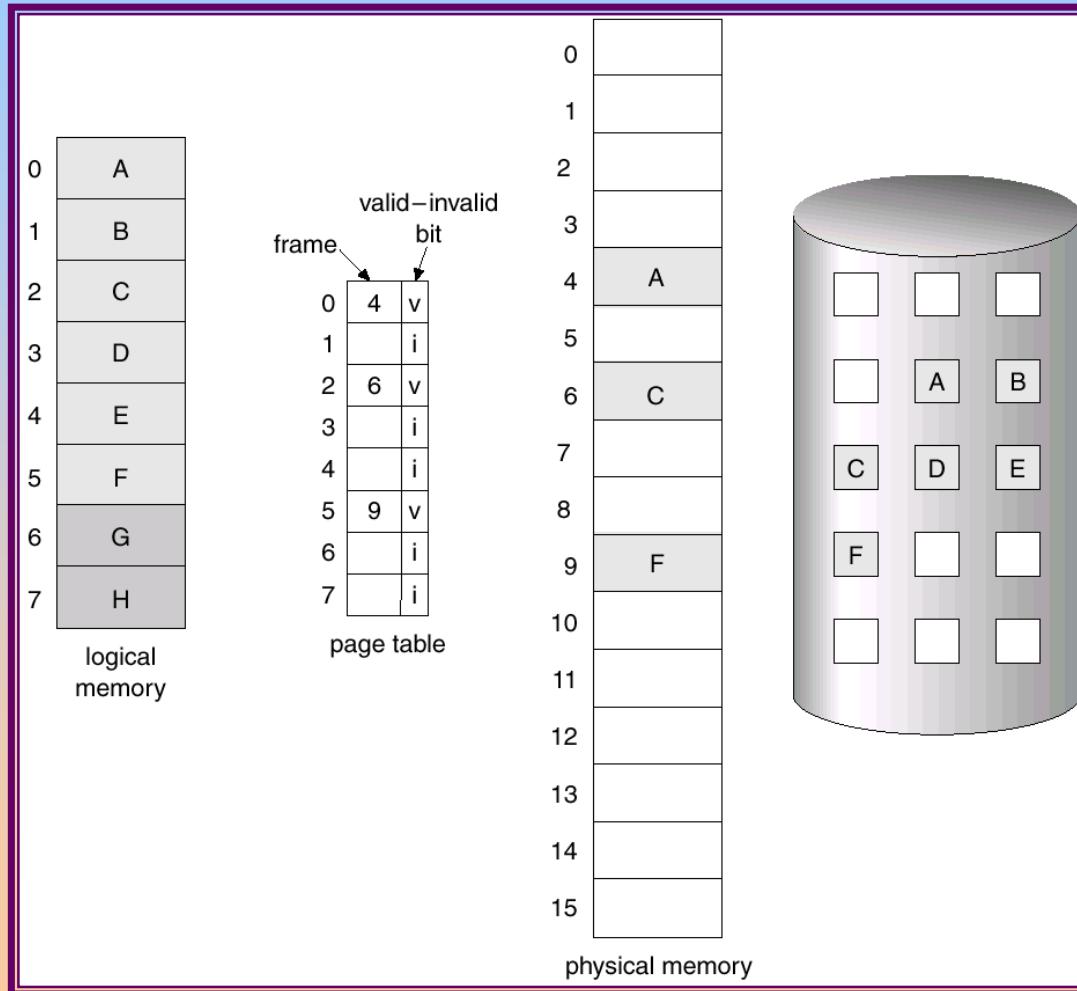
Valid-Invalid Bit

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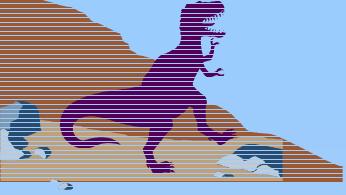
Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

- During address translation, if valid–invalid bit in page table entry is 0 $\Rightarrow$ page fault.
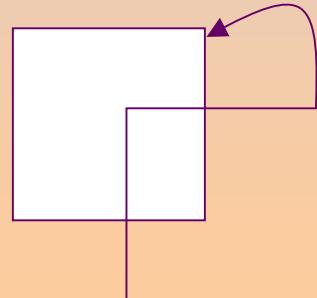
Page Fault
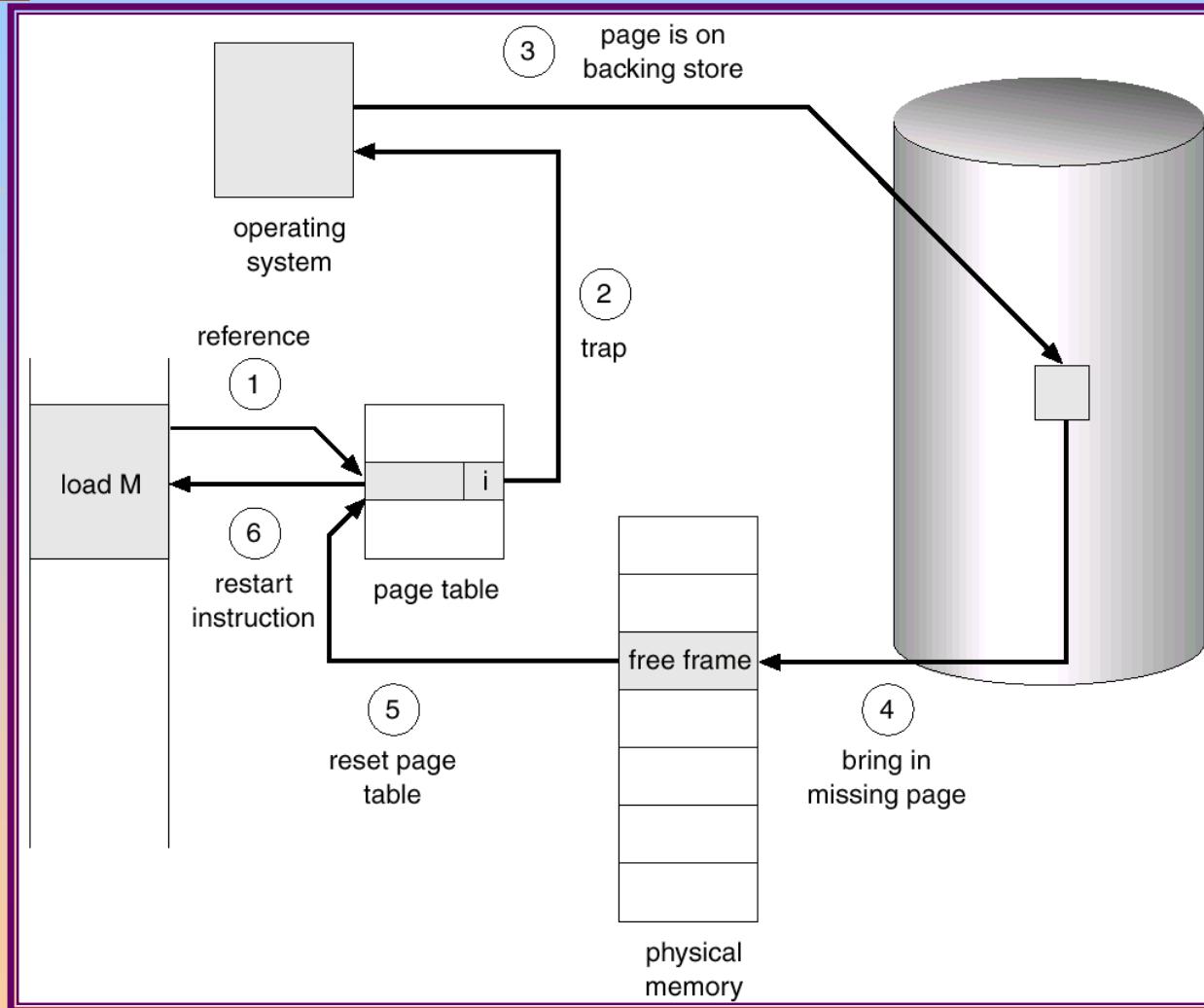
- If there is ever a reference to a page, first reference will trap to
  OS ⇒ page fault

- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.

- Get empty frame.

- Swap page into frame.

- Reset tables, validation bit = 1.

- Restart instruction: Least Recently Used
  - block move

  - auto increment/decrement location
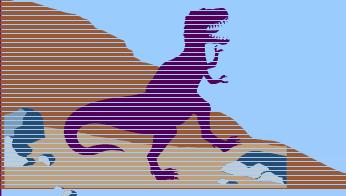
Steps in Handling a Page Fault

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 page may be brought into memory several times.
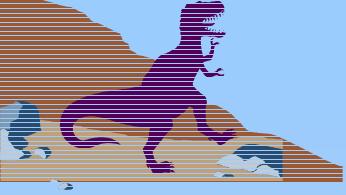
Performance of Demand Paging

■ Page Fault Rate $0 \le p \le 1.0$

  ✦ if $p = 0$ no page faults

  ✦ if $p = 1$, every reference is a fault

■ Effective Access Time (EAT)
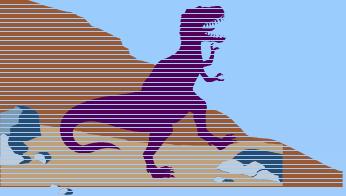
$$EAT = (1 - p) \times \text{memory access}$$
$$+ p \text{ (page fault overhead}$$
$$+ [\text{swap page out }]$$
$$+ \text{swap page in}$$
$$+ \text{restart overhead)}$$
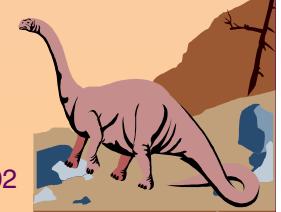
Demand Paging Example
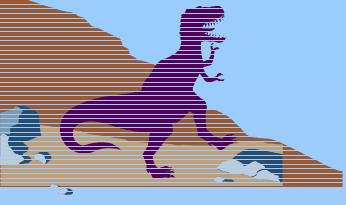
- Memory access time = 1 microsecond

- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.

- Swap Page Time = 10 msec = 10,000 msec

$$EAT = (1 - p) \times 1 + p (15000)$$

$$1 + 15000P \qquad (in\ msec)$$

Process Creation
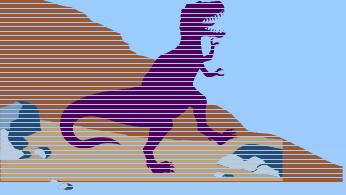
■ Virtual memory allows other benefits during process creation:

- Copy-on-Write

- Memory-Mapped Files

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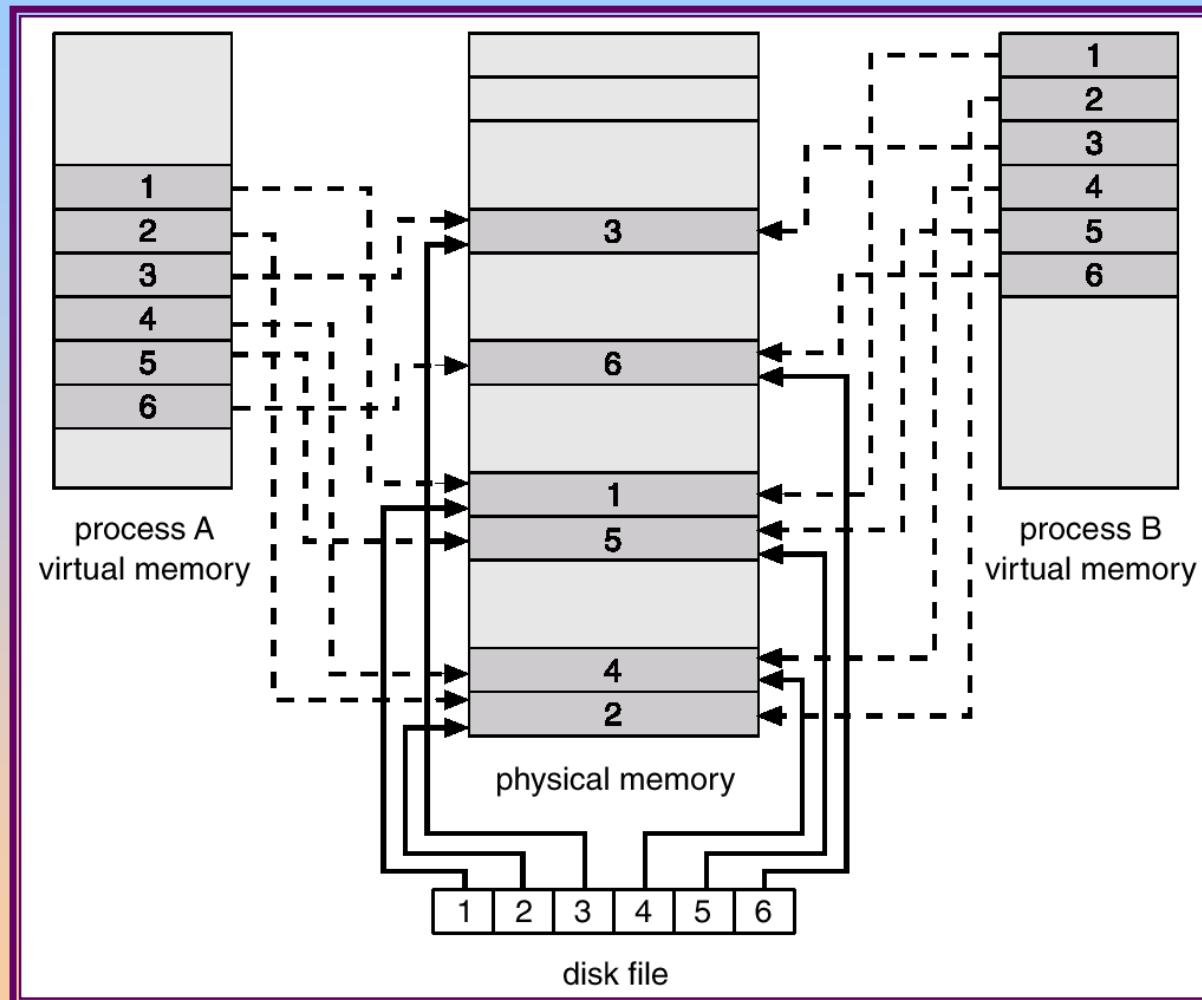
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory.

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

- COW allows more efficient process creation as only modified pages are copied.

- Free pages are allocated from a *pool* of zeroed-out pages.
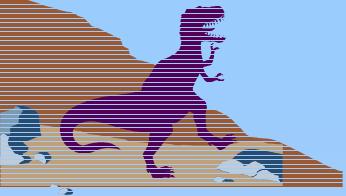
Memory-Mapped Files

- 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() write()** system calls.

- Also allows several processes to map the same file allowing the pages in memory to be shared.

Memory Mapped Files
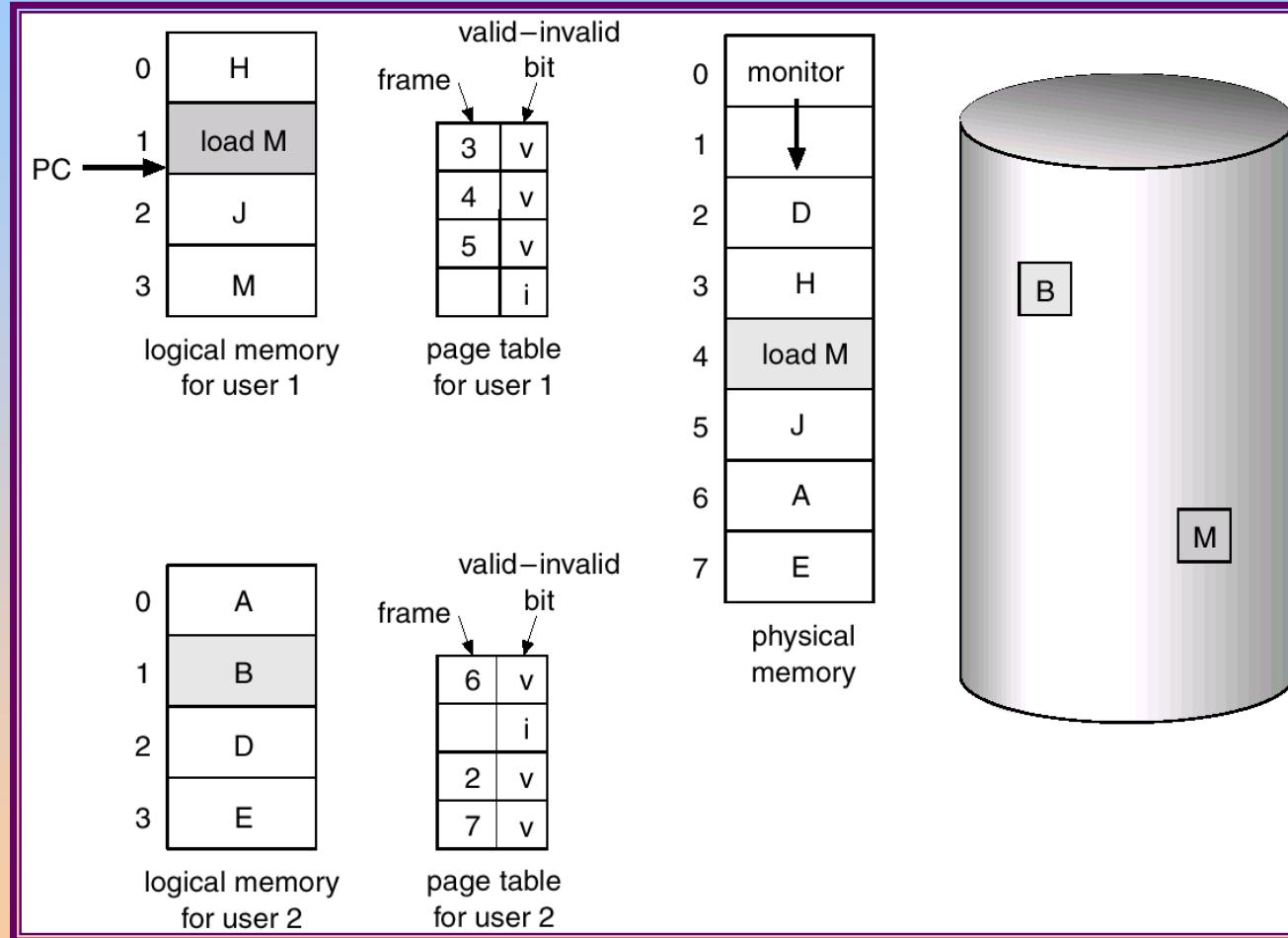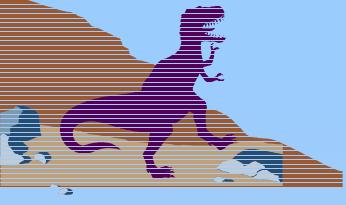
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

- Use *modify* (*dirty*) *bit* to reduce overhead of page transfers – only modified pages are written to disk.

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

Need For Page Replacement

logical memory
for user 1

page table
for user 1

logical memory
for user 2

page table
for user 2

physical
memory
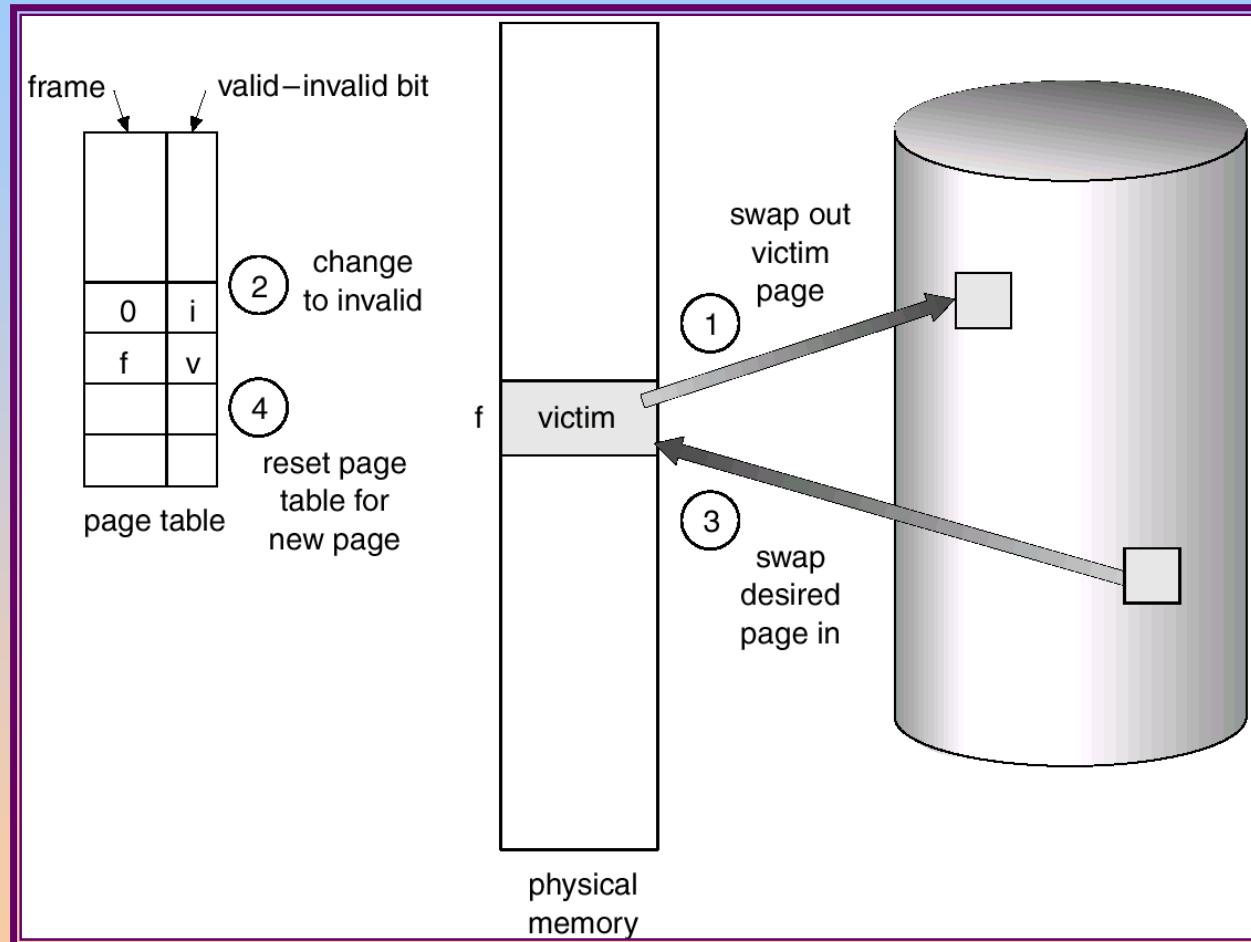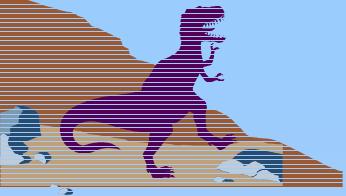
Basic Page Replacement

- 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.

- Read the desired page into the (newly) free frame. Update the page and frame tables.

- Restart the process.

Page Replacement

frame

valid–invalid bit

page table

0	i
f	v

② change to invalid

④ reset page table for new page

physical memory

f victim

① swap out victim page

③ swap desired page in
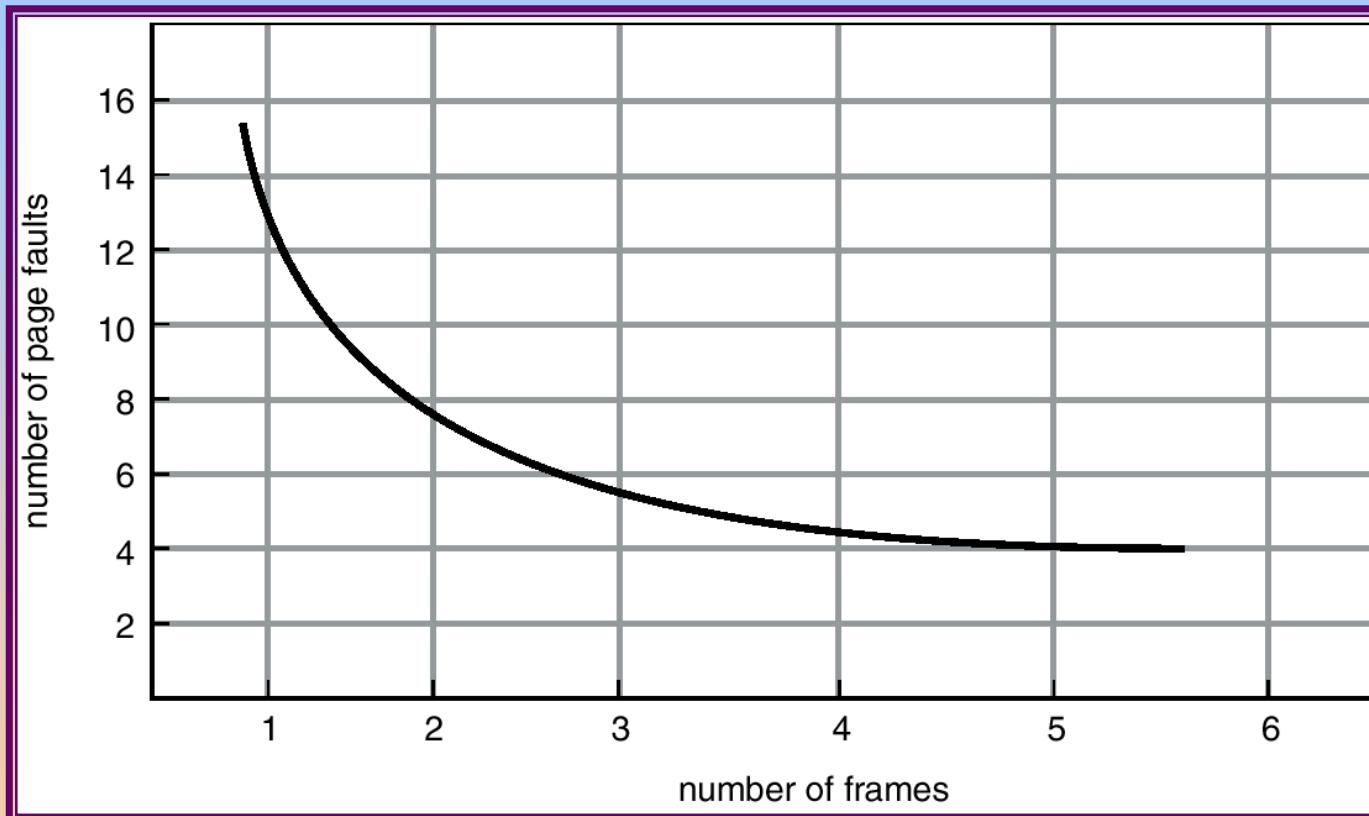
Page Replacement Algorithms

- 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.

- In all our examples, the reference string is

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

Graph of Page Faults Versus The Number of Frames

First-In-First-Out (FIFO) Algorithm

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

1	**1**	4	5	
2	**2**	1	3	9 page faults
3	**3**	2	4	

- 4 frames

1	**1**	5	4	
2	**2**	1	5	10 page faults
3	**3**	2		
4	**4**	3		
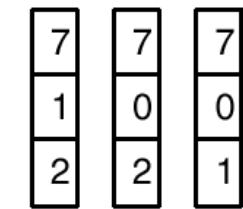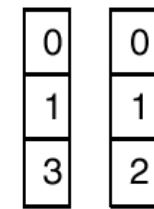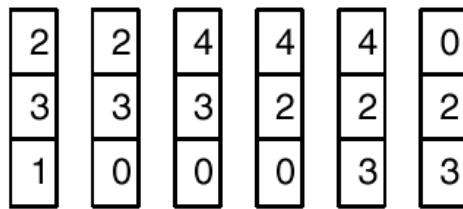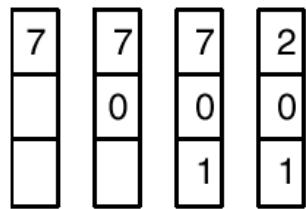
- FIFO Replacement – Belady's Anomaly
  - ✦ more frames ⇒ less page faults

FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
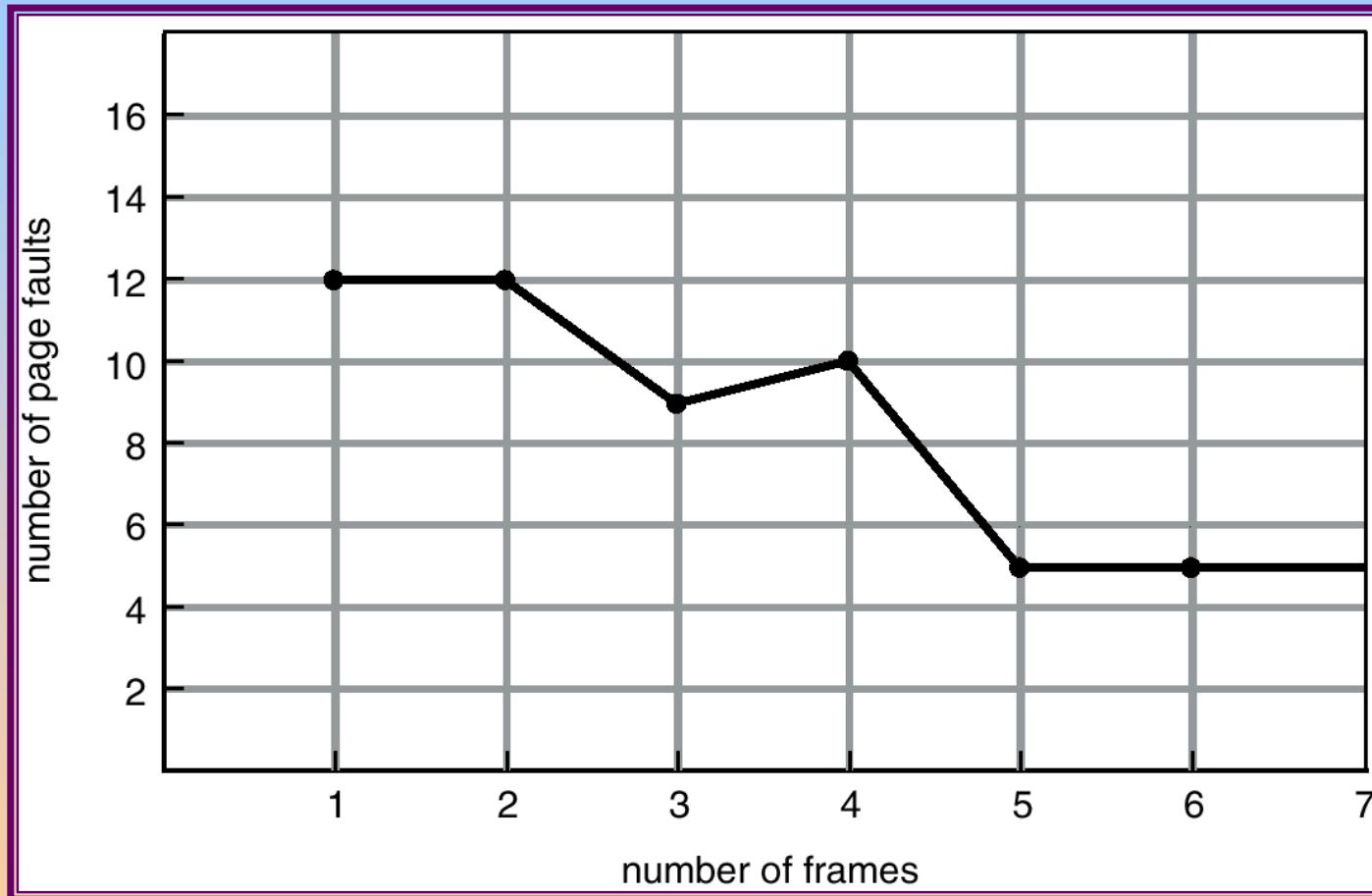
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1
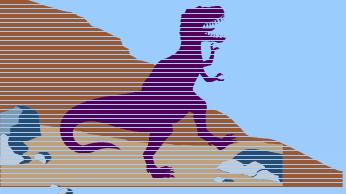
page frames

FIFO Illustrating Belady's Anamoly
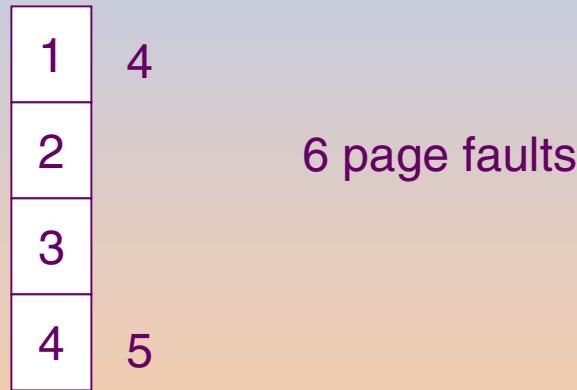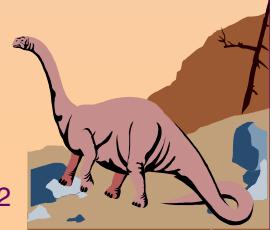
Optimal Algorithm
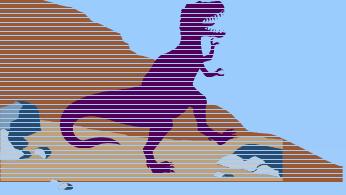
- Replace page that will not be used for longest period of time.

- 4 frames example

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

1	4
2	
3	
4	5

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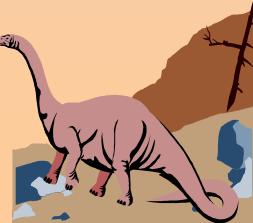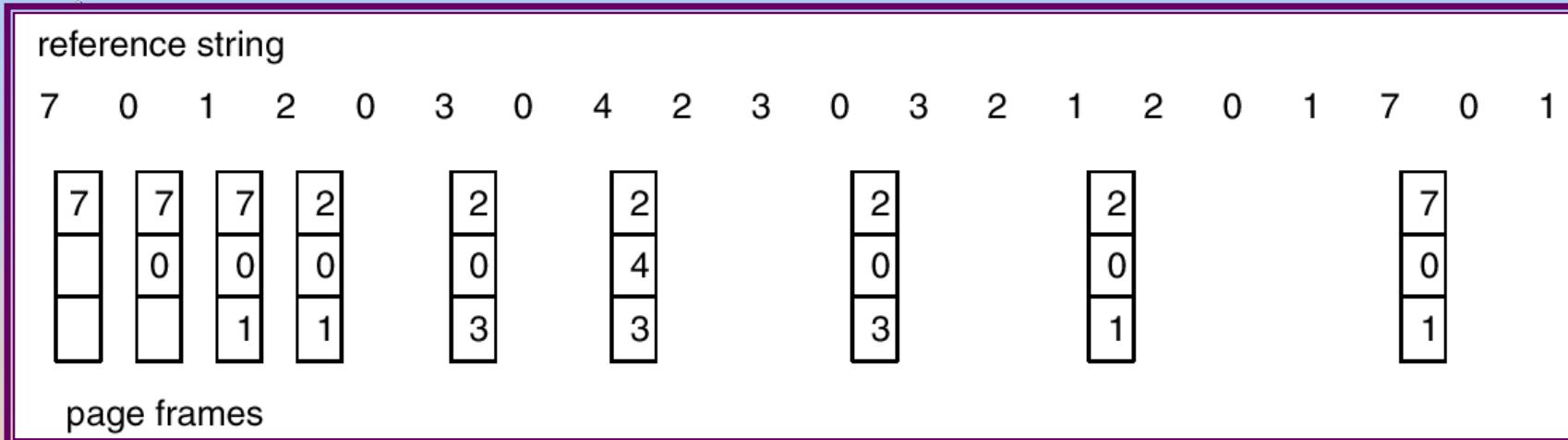
- How do you know this?
- Used for measuring how well your algorithm performs.

Optimal Page Replacement

reference string

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

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

page frames
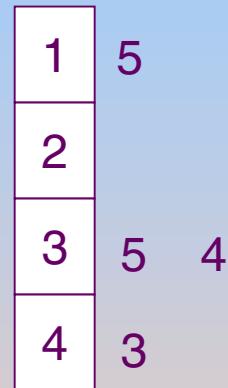
Least Recently Used (LRU) Algorithm

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

1	5	
2		
3	5	4
4	3	
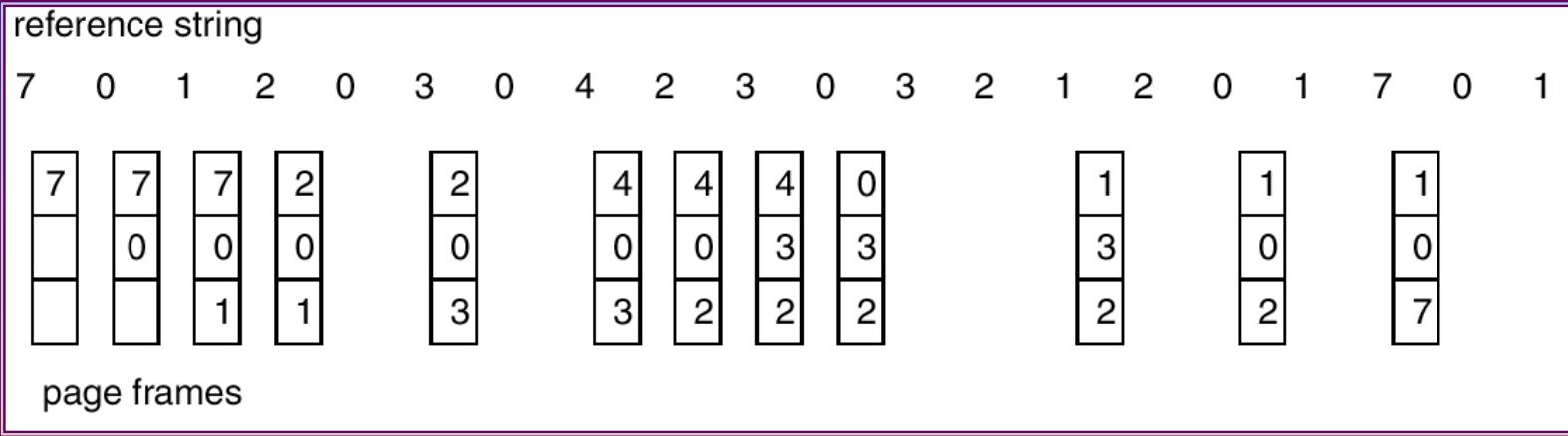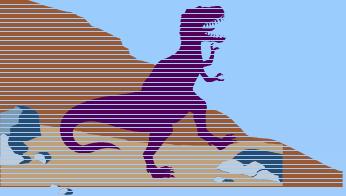
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.

LRU Page Replacement

reference string

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

page frames
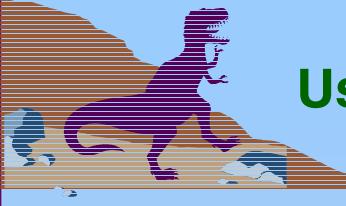
LRU Algorithm (Cont.)
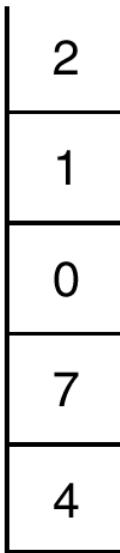
■ Stack implementation – keep a stack of page numbers in a double link form:

  ✦ Page referenced:

    ✔ move it to the top

    ✔ requires 6 pointers to be changed

  ✦ No search for replacement

reference string

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

  a   b
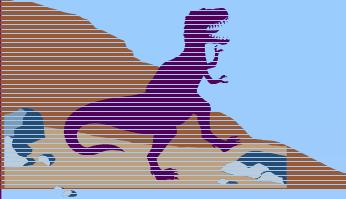
stack before a
2
1
0
7
4

stack after b
7
2
1
0
4
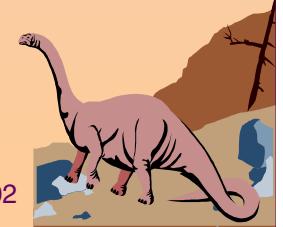
LRU Approximation Algorithms
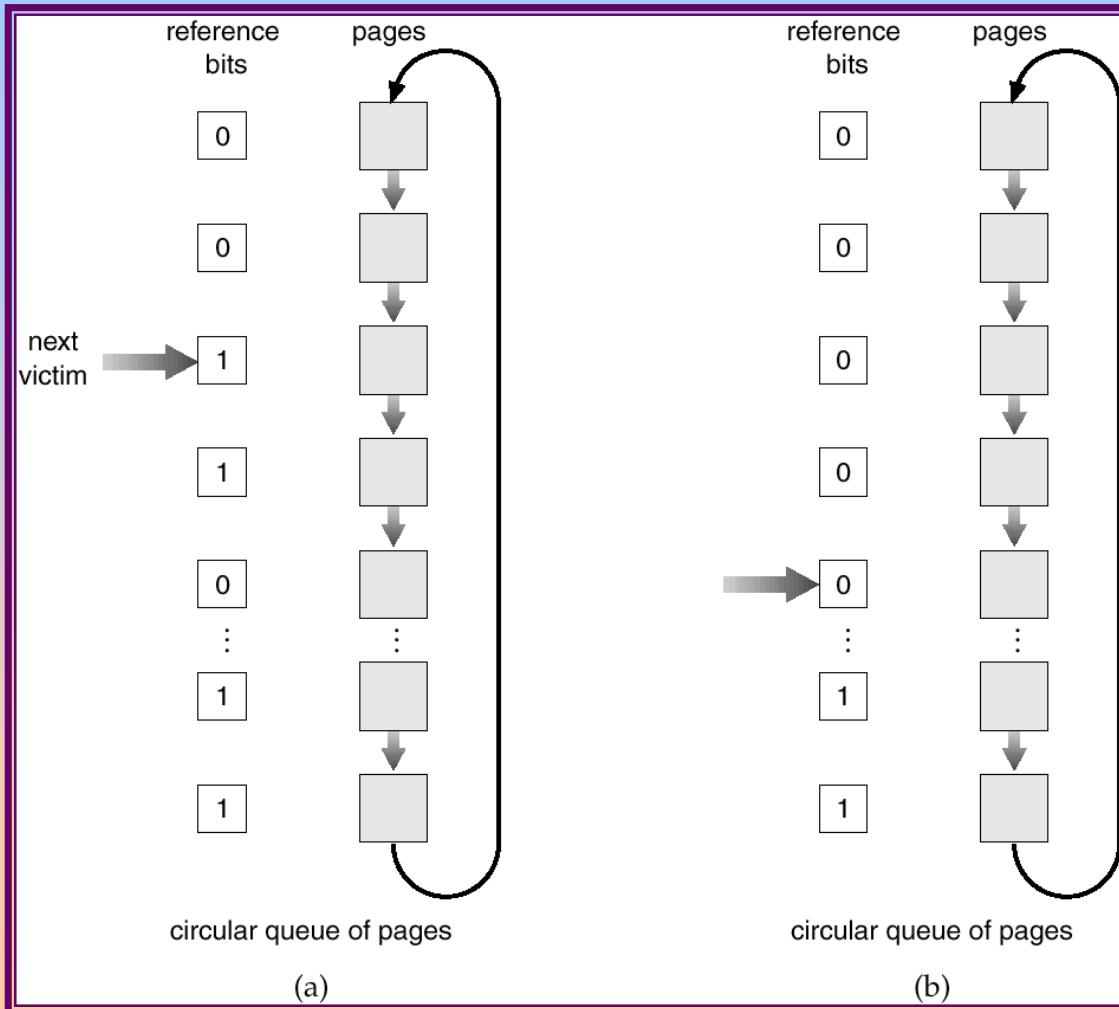
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists). We do not know the order, however.
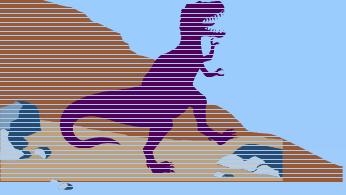
- **Second chance**
  - Need reference bit.
  - Clock replacement.
  - If page to be replaced (in clock order) has reference bit = 1. then:
    - ✔ set reference bit 0.
    - ✔ leave page in memory.
    - ✔ replace next page (in clock order), subject to same rules.
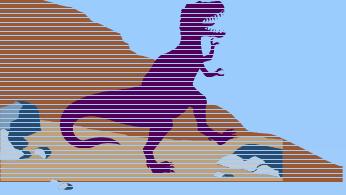
Second-Chance (clock) Page-Replacement Algorithm

reference bits | pages

0
0
next victim → 1
1
0
⋮
1
1

circular queue of pages

(a)

reference bits | pages

0
0
0
0
→ 0
⋮
1
1

circular queue of pages

(b)
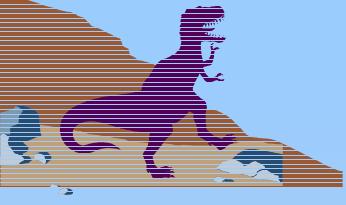
Counting Algorithms

- Keep a counter of the number of references that have been made to each page.

- LFU Algorithm: replaces page with smallest count.

- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
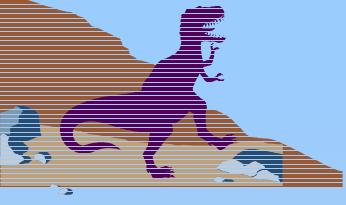
Allocation of Frames

- 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**.
- Two major allocation schemes.
    - fixed allocation
    - priority allocation
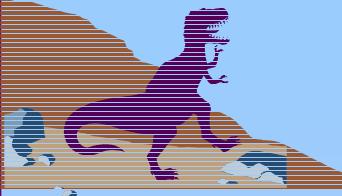
Fixed Allocation

- 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 process p_i

  - $S = \sum s_i$

  - m = total number of frames

  - a_i = allocation for $p_i = \dfrac{s_i}{S} \times m$

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

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

$$a_2 = \frac{127}{137} \times 64 \approx 59$$
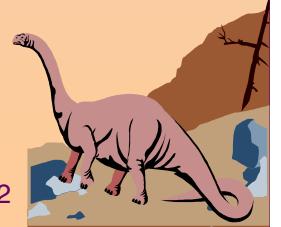
Priority Allocation
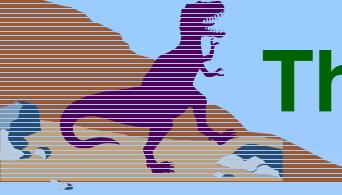
- Use a proportional allocation scheme using priorities rather than size.

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

Global vs. Local Allocation
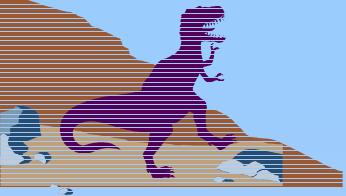
- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.

- **Local** replacement – each process selects from only its own set of allocated frames.
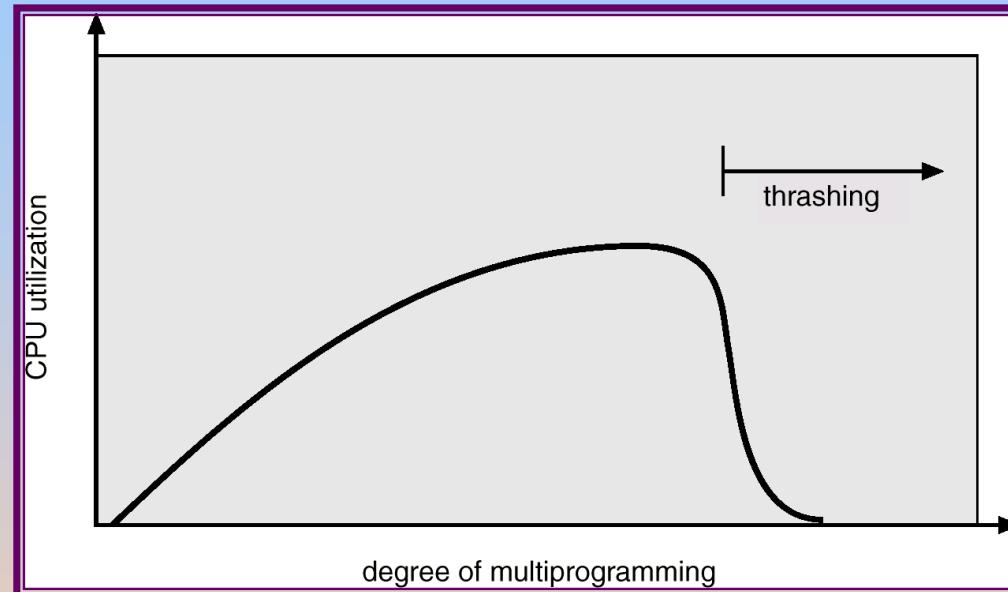
Thrashing

■ If a process does not have "enough" pages, the page-fault rate is very high. This leads to:

❖ low CPU utilization.

❖ operating system thinks that it needs to increase the degree of multiprogramming.

❖ another process added to the system.

■ **Thrashing** $\equiv$ a process is busy swapping pages in and out.
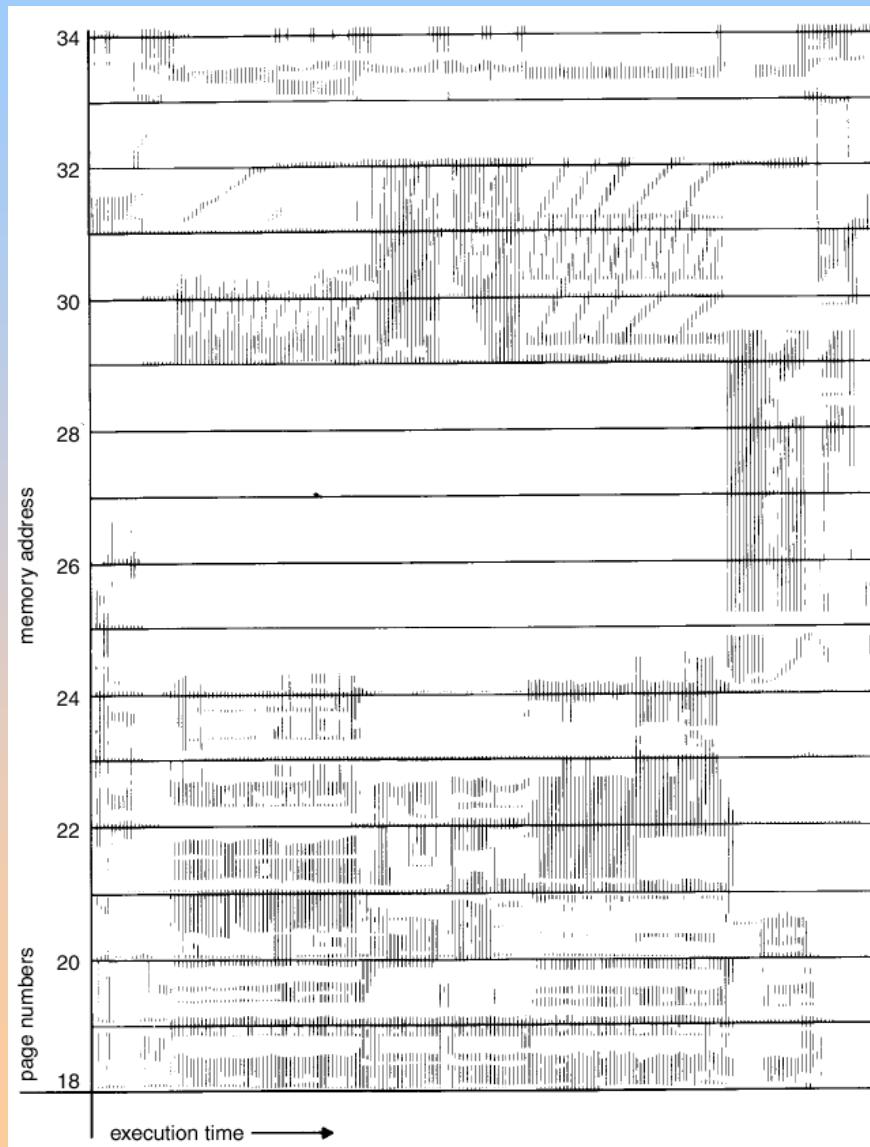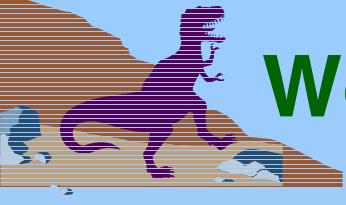
Thrashing

- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size

Locality In A Memory-Reference Pattern
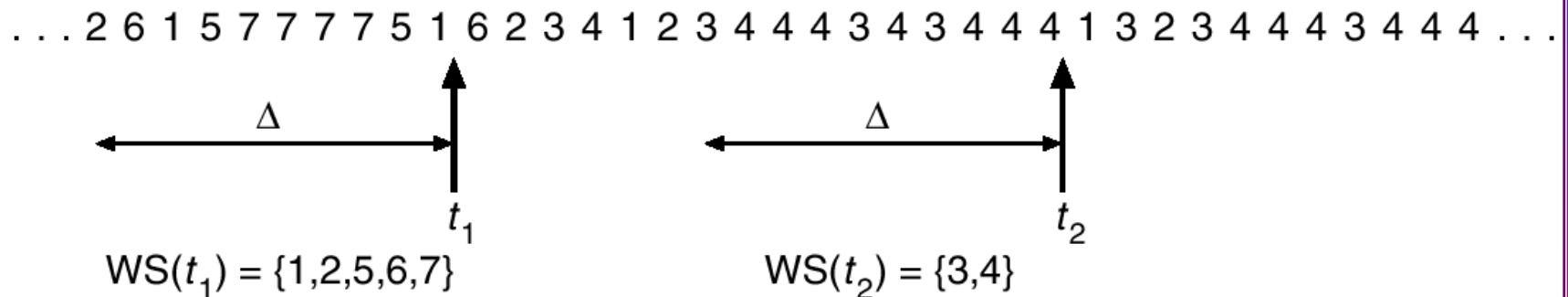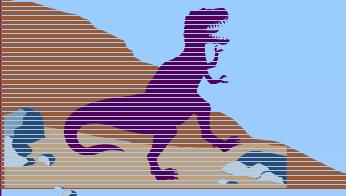
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction

- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)

  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.

- $D = \Sigma\ WSS_i \equiv$ total demand frames

- if $D > m \Rightarrow$ Thrashing

- Policy if $D > m$, then suspend one of the processes.

Working-set model

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 . . .

Δ t_1 Δ t_2

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$
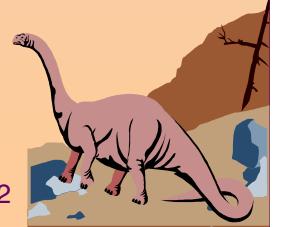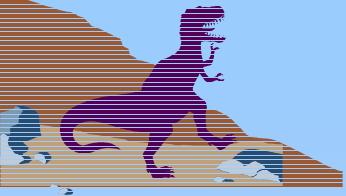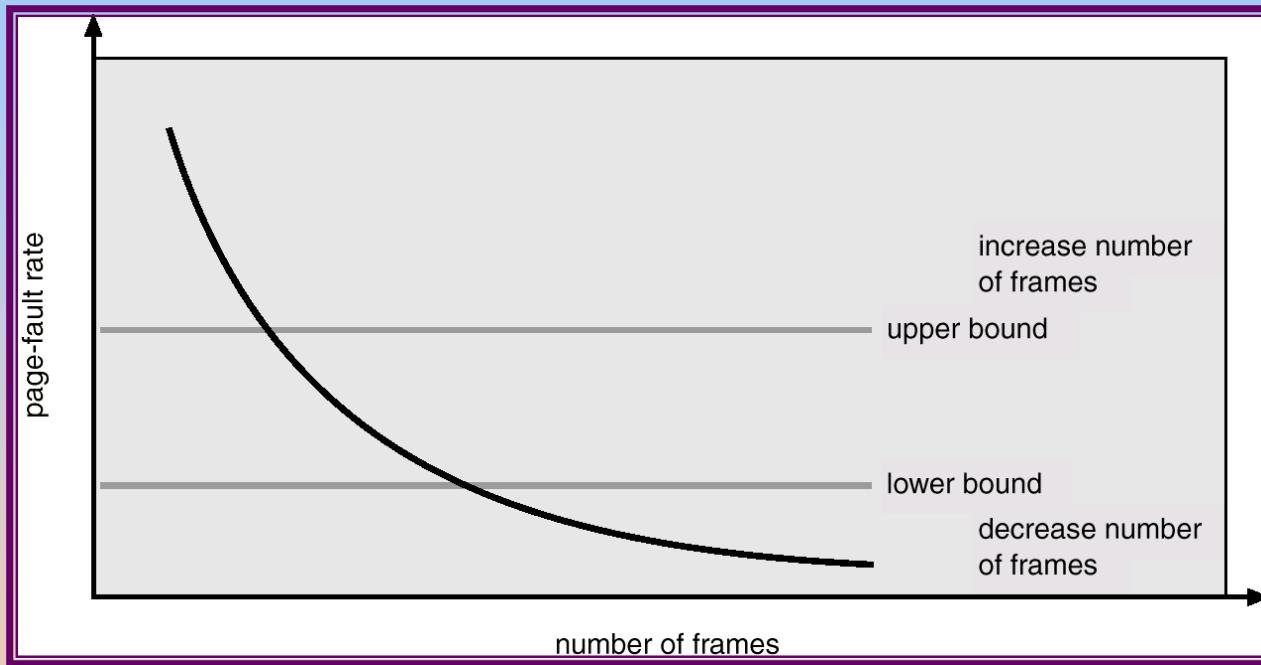
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
    - Timer interrupts after every 5000 time units.
    - Keep in memory 2 bits for each page.
    - Whenever a timer interrupts copy and sets the values of all reference bits to 0.
    - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.
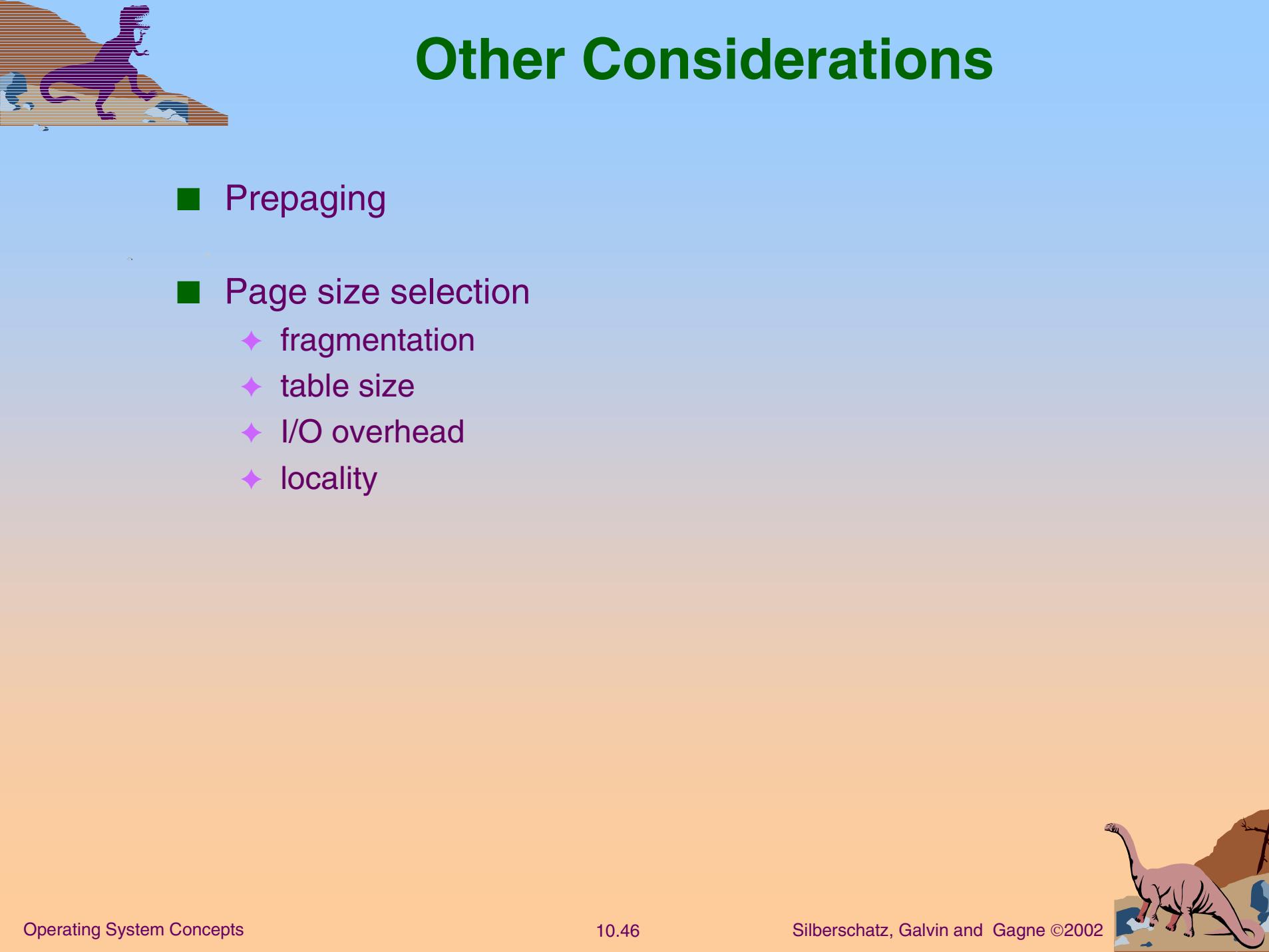
Page-Fault Frequency Scheme
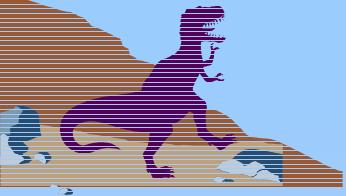
- Establish "acceptable" page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.

Other Considerations
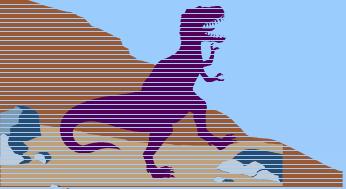
- Prepaging

- Page size selection
  - fragmentation
  - table size
  - I/O overhead
  - locality
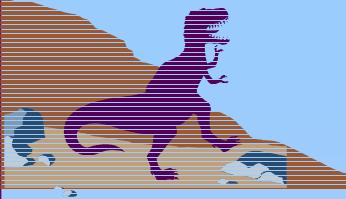
Other Considerations (Cont.)

- **TLB Reach** - The amount of memory accessible from the TLB.

- TLB Reach = (TLB Size) X (Page Size)

- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.

Increasing the Size of the TLB

- **Increase the Page Size**. This may lead to an increase in fragmentation as not all applications require a large page size.

- **Provide Multiple Page Sizes**. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Considerations (Cont.)

■ Program structure

✦ **int A[][] = new int[1024][1024];**

✦ Each row is stored in one page

✦ Program 1
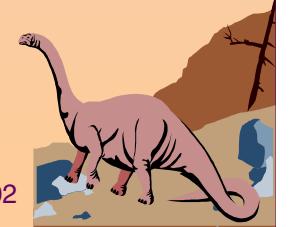
```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```
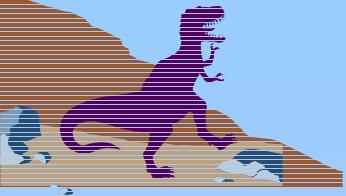
1024 x 1024 page faults

✦ Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

1024 page faults
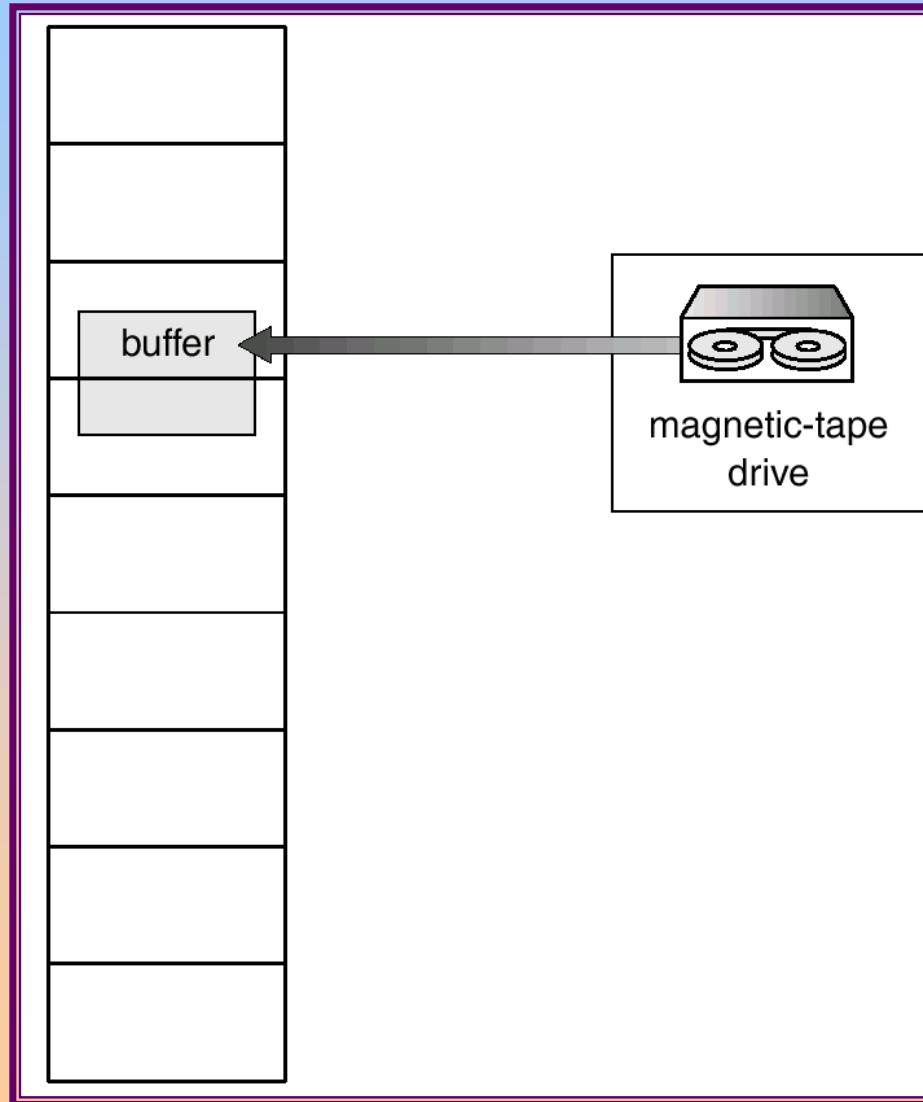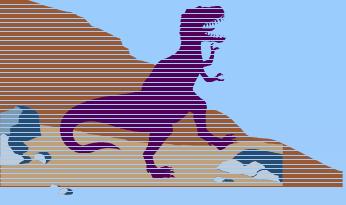
Other Considerations (Cont.)

- **I/O Interlock** – Pages must sometimes be locked into memory.

- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

buffer

magnetic-tape
drive
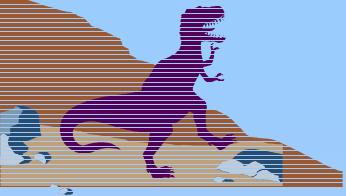
Operating System Examples
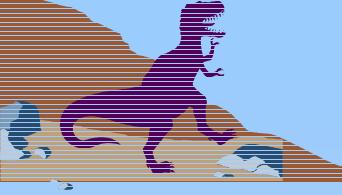
- Windows NT

- Solaris 2

Windows NT
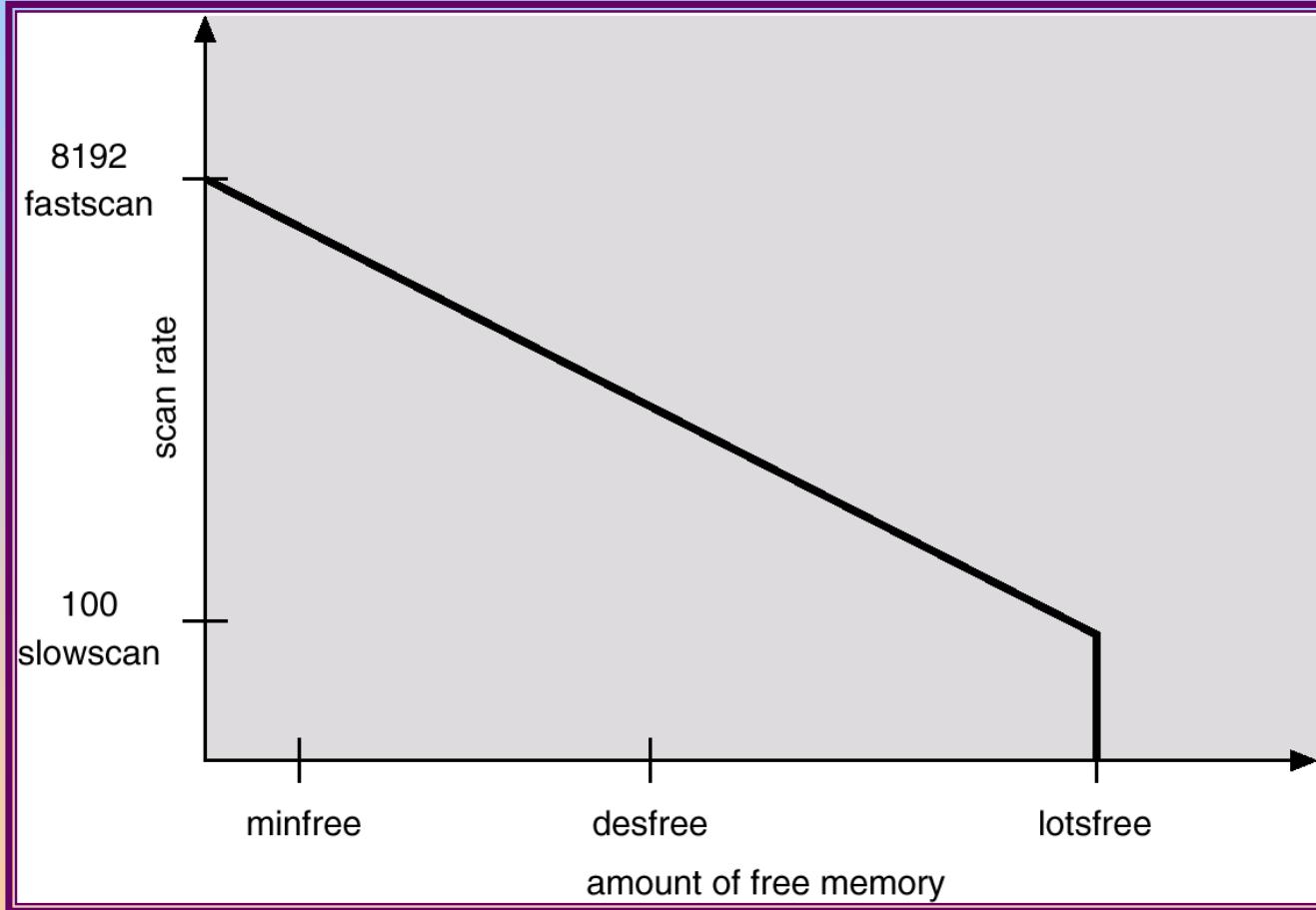
- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.

- Processes are assigned **working set minimum** and **working set maximum**.

- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.

- A process may be assigned as many pages up to its working set maximum.

- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory.

- Working set trimming removes pages from processes that have pages in excess of their working set minimum.
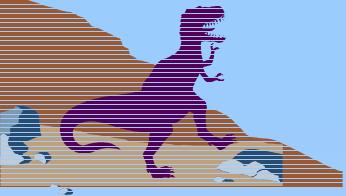
Solaris 2

- Maintains a list of free pages to assign faulting processes.

- **Lotsfree** – threshold parameter to begin paging.

- Paging is peformed by *pageout* process.

- Pageout scans pages using modified clock algorithm.

- **Scanrate** is the rate at which pages are scanned. This ranged from **slowscan** to **fastscan**.

- Pageout is called more frequently depending upon the amount of free memory available.
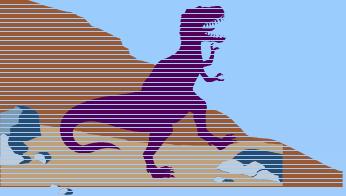
Solar Page Scanner

Chapter 11: File-System Interface
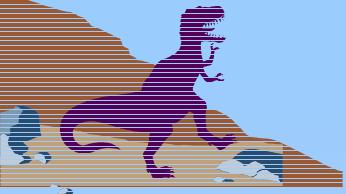
- File Concept

- Access Methods

- Directory Structure

- File System Mounting

- File Sharing

- Protection

File Concept
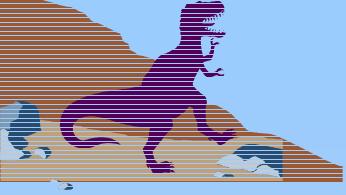
- Contiguous logical address space

- Types:
  - Data
    - ✔ numeric
    - ✔ character
    - ✔ binary
  - Program
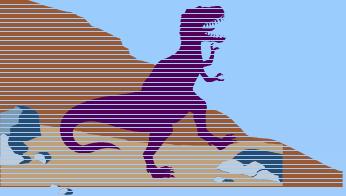
File Structure

- 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
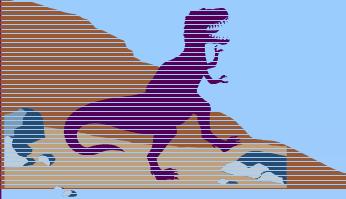
File Attributes

- **Name** – only information kept in human-readable form.

- **Type** – needed for systems that support different types.

- **Location** – pointer to file location on device.

- **Size** – current file size.

- **Protection** – controls who can do reading, writing, executing.

- **Time**, **date**, **and user identification** – data for protection, security, and usage monitoring.

- Information about files are kept in the directory structure, which is maintained on the disk.

File Operations
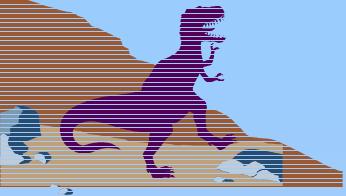
- Create
- Write
- Read
- Reposition within file – file seek
- Delete
- Truncate
- Open(F_i) – search the directory structure on disk for entry F_i, and move the content of entry to memory.
- Close (F_i) – move the content of entry F_i in memory to directory structure on disk.

11.5

File Types – Name, Extension

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, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, 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	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Access Methods
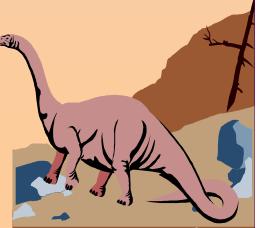
- **Sequential Access**

  read next
  write next
  reset
  no *read after last write*
  (rewrite)
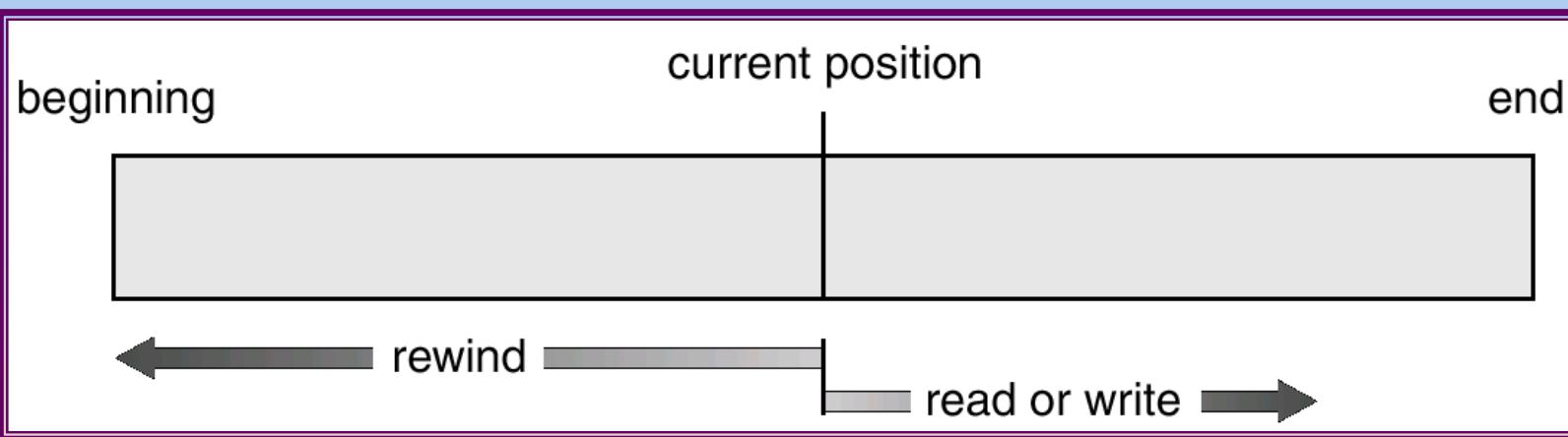
- **Direct Access**
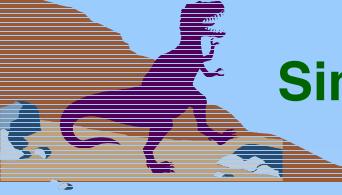
  read n
  write n
  position to n
  read next
  write next
  rewrite n

  n = relative block number

Sequential-access File
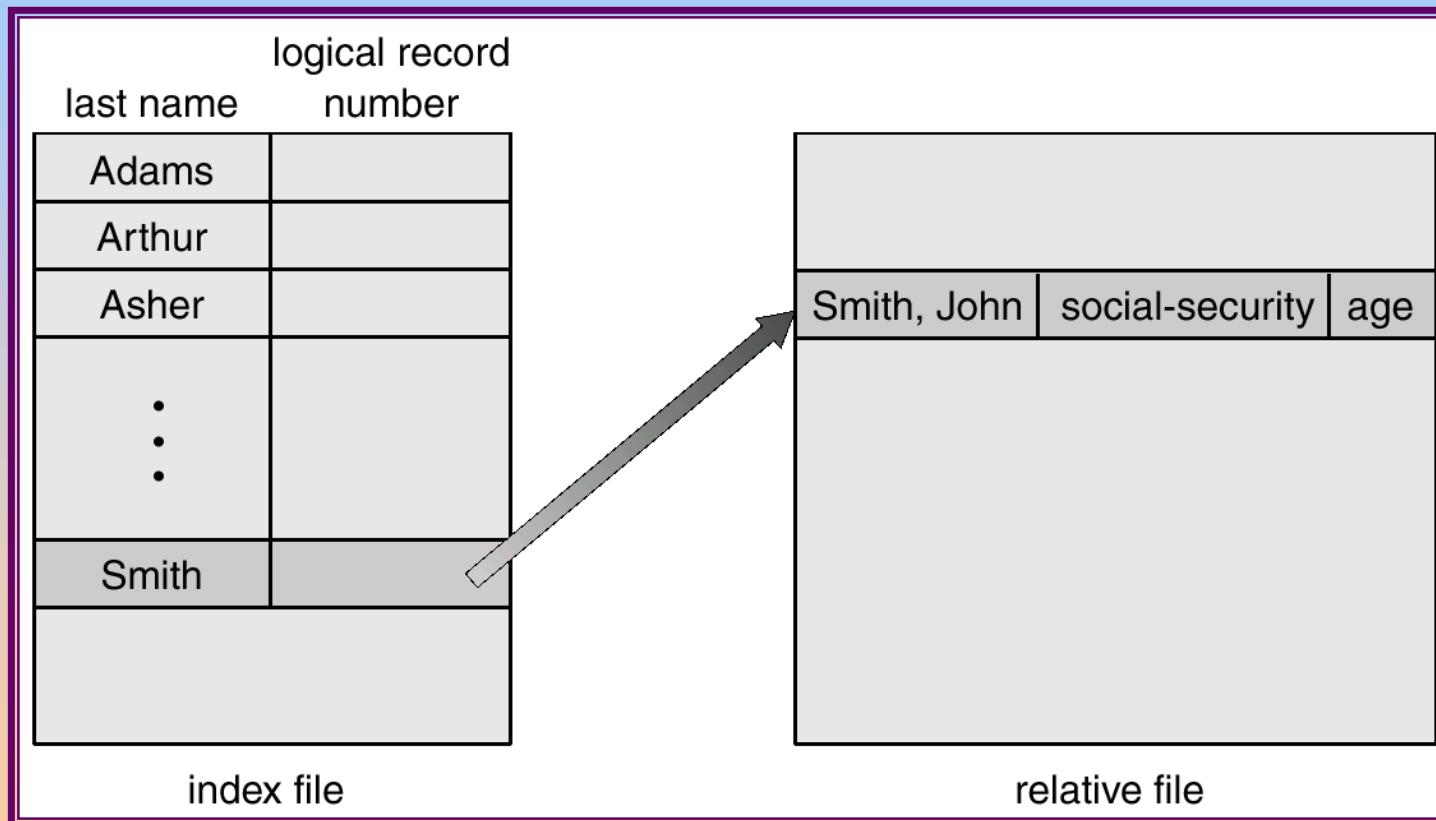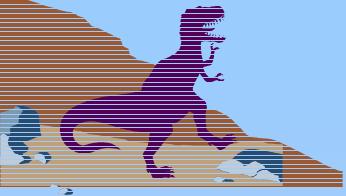
Simulation of Sequential Access on a Direct-access File

sequential access	implementation for direct access
reset	$cp = 0$;
read next	*read* cp; $cp = cp+1$;
write next	*write* cp; $cp = cp+1$;
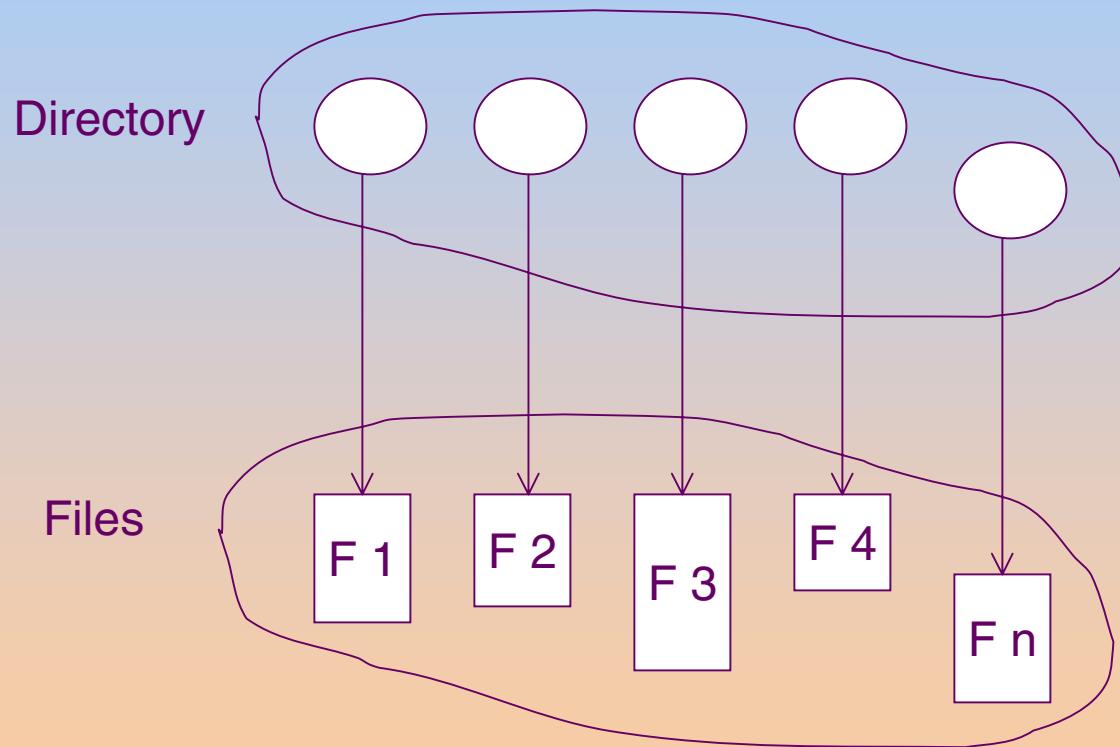
Example of Index and Relative Files
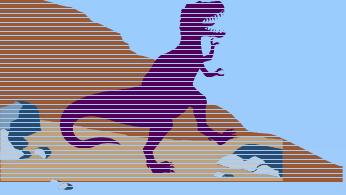
Directory Structure
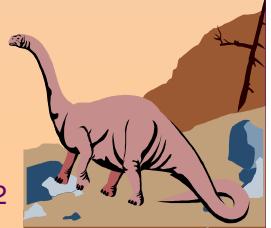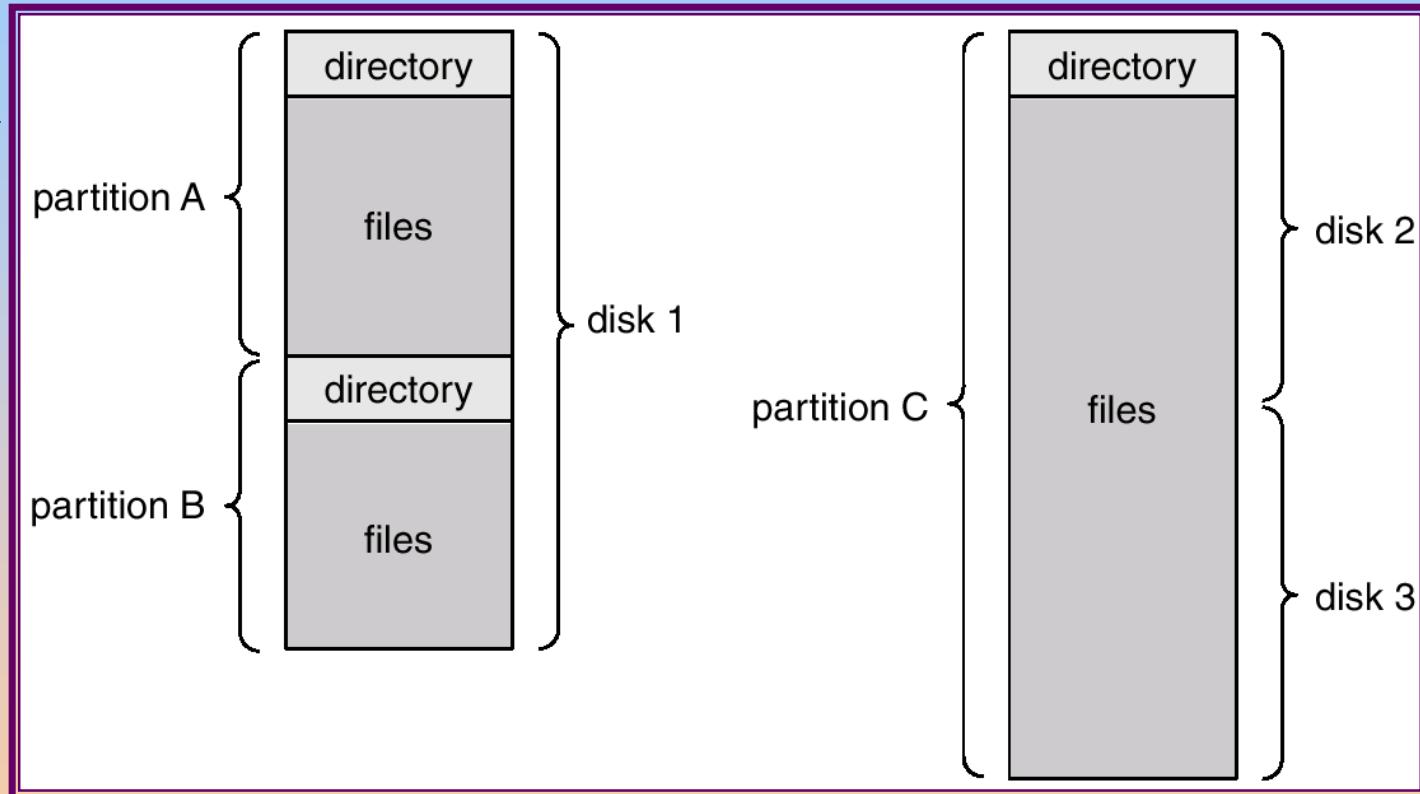
- A collection of nodes containing information about all files.
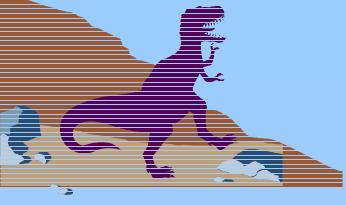
Directory

Files

Both the directory structure and the files reside on disk.
Backups of these two structures are kept on tapes.

F 1 F 2 F 3 F 4 F n
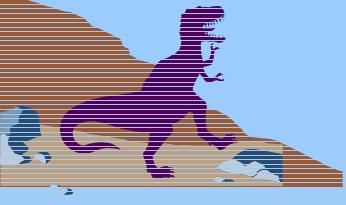
A Typical File-system Organization

Information in a Device Directory

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information (discuss later)

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system
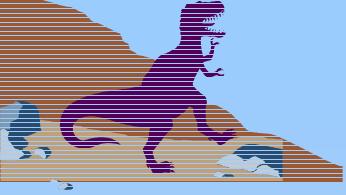
Organize the Directory (Logically) to Obtain
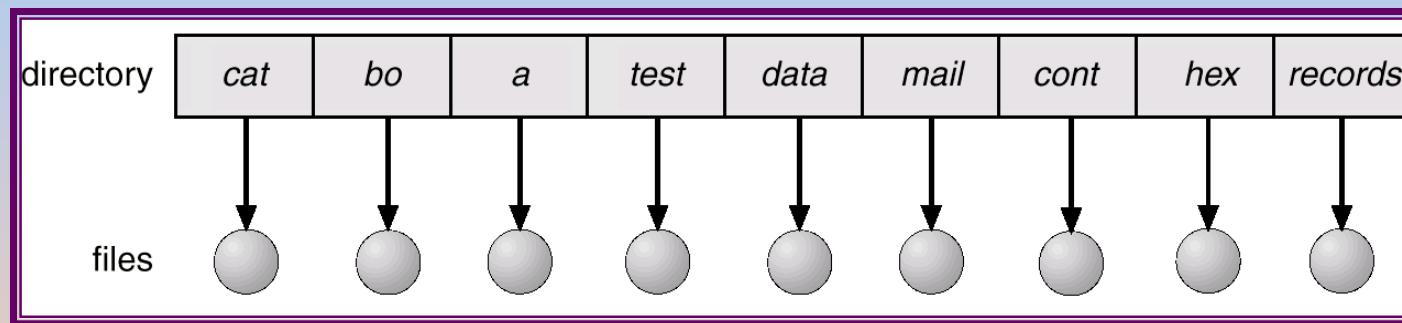
- **Efficiency** – locating a file quickly.
- **Naming** – convenient to users.
  - ✦ Two users can have same name for different files.
  - ✦ The same file can have several different names.
- **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, …)
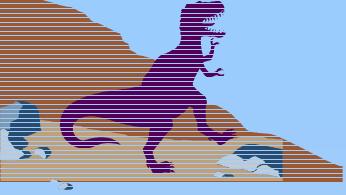
Single-Level Directory

■ A single directory for all users.

directory	cat	bo	a	test	data	mail	cont	hex	records

files ○ ○ ○ ○ ○ ○ ○ ○ ○
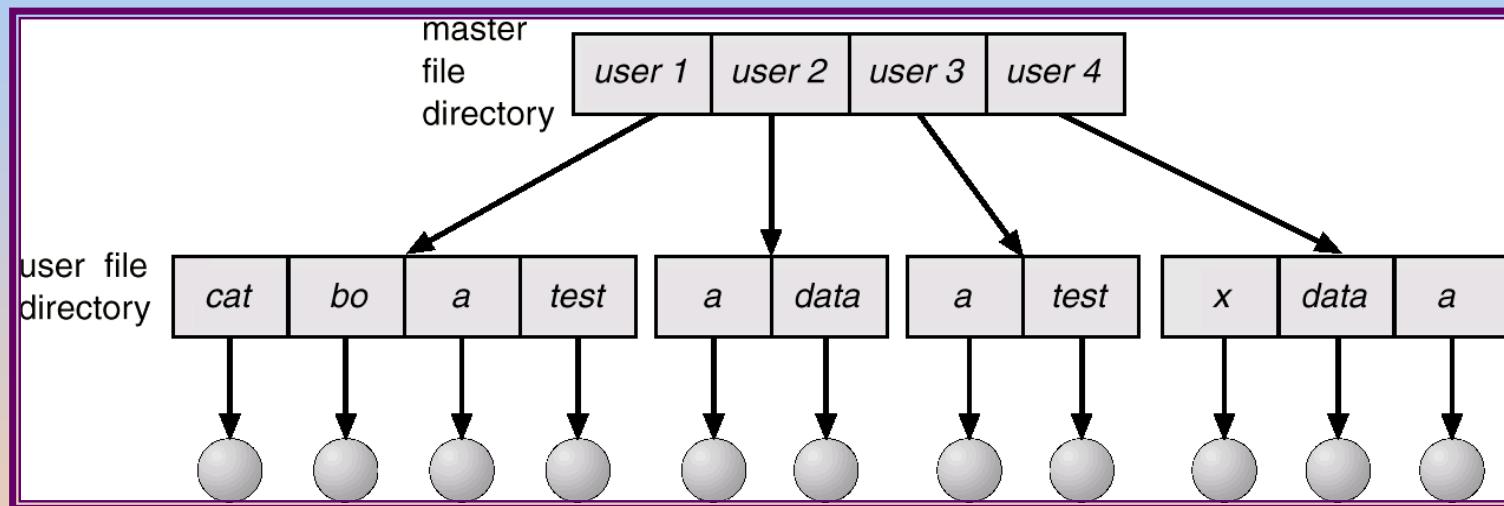
Naming problem

Grouping problem
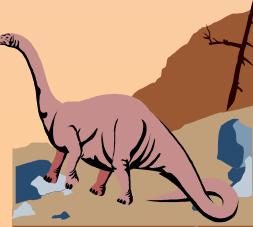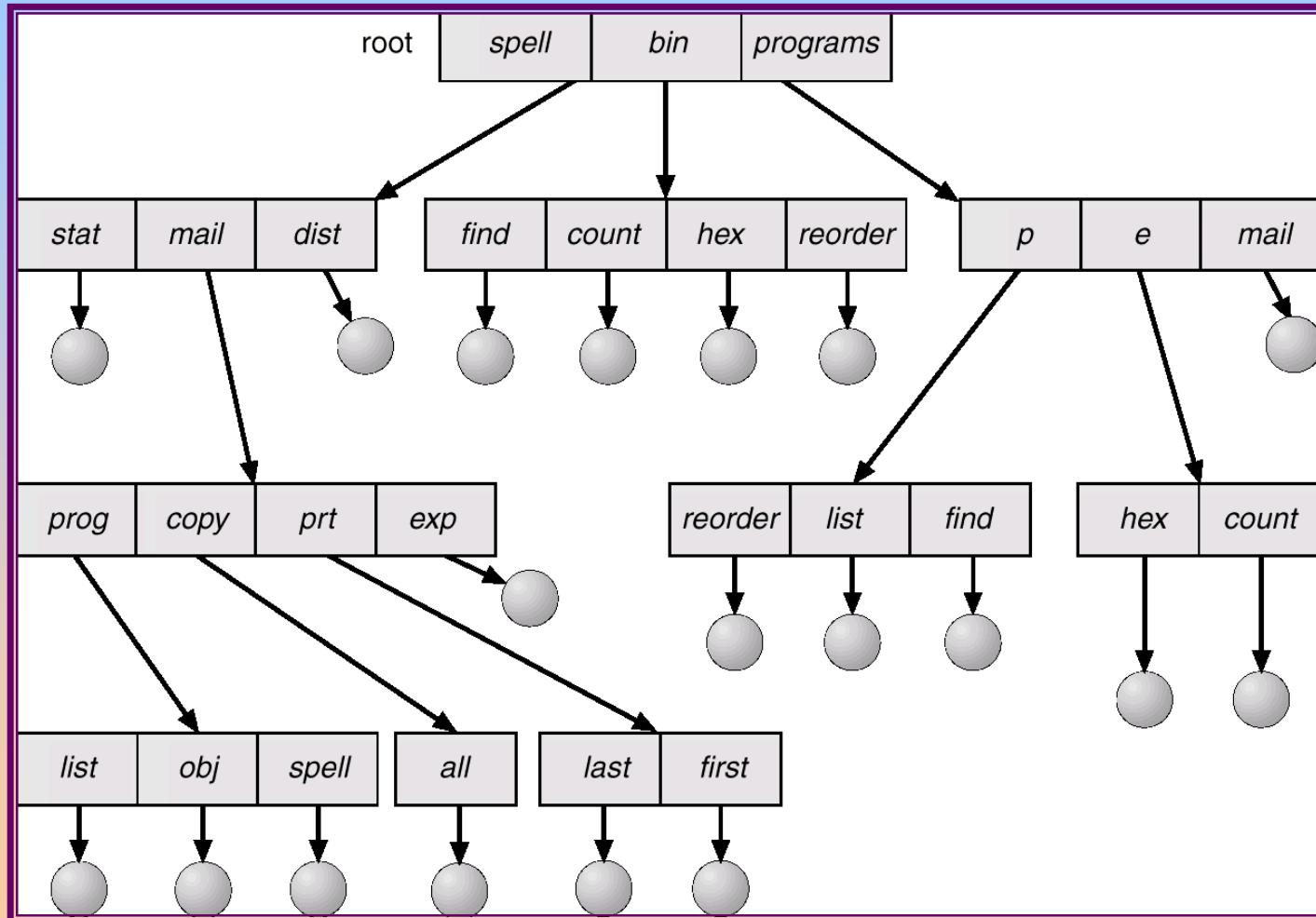
Two-Level Directory

■ Separate directory for each user.

master file directory	user 1	user 2	user 3	user 4

user file directory	cat	bo	a	test	a	data	a	test	x	data	a
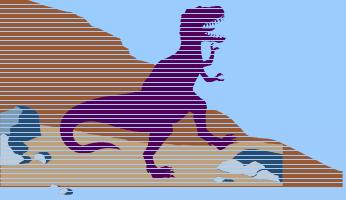
•Path name
•Can have the same file name for different user
•Efficient searching
•No grouping capability

Tree-Structured Directories
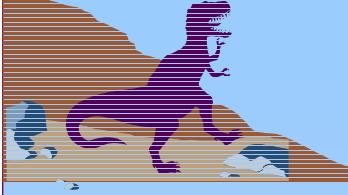
Tree-Structured Directories (Cont.)

- Efficient searching

- Grouping Capability

- Current directory (working directory)
  - **cd** /spell/mail/prog
  - **type** list

Tree-Structured Directories (Cont.)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory.
- Delete a file
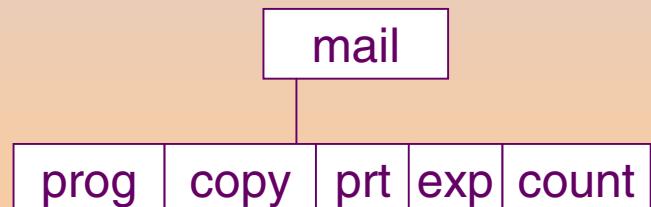
  rm <file-name>
- Creating a new subdirectory is done in current directory.

  mkdir <dir-name>
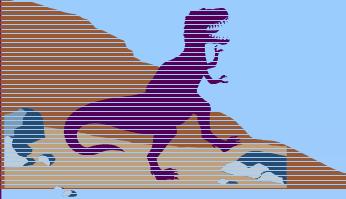
  Example: if in current directory   **/mail**

  mkdir count

```
          ┌──────┐
          │ mail │
          └──┬───┘
             │
┌──────┬──────┬─────┬─────┬───────┐
│ prog │ copy │ prt │ exp │ count │
└──────┴──────┴─────┴─────┴───────┘
```
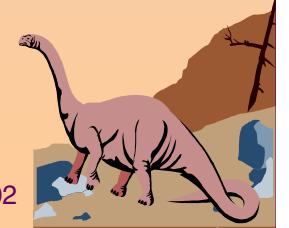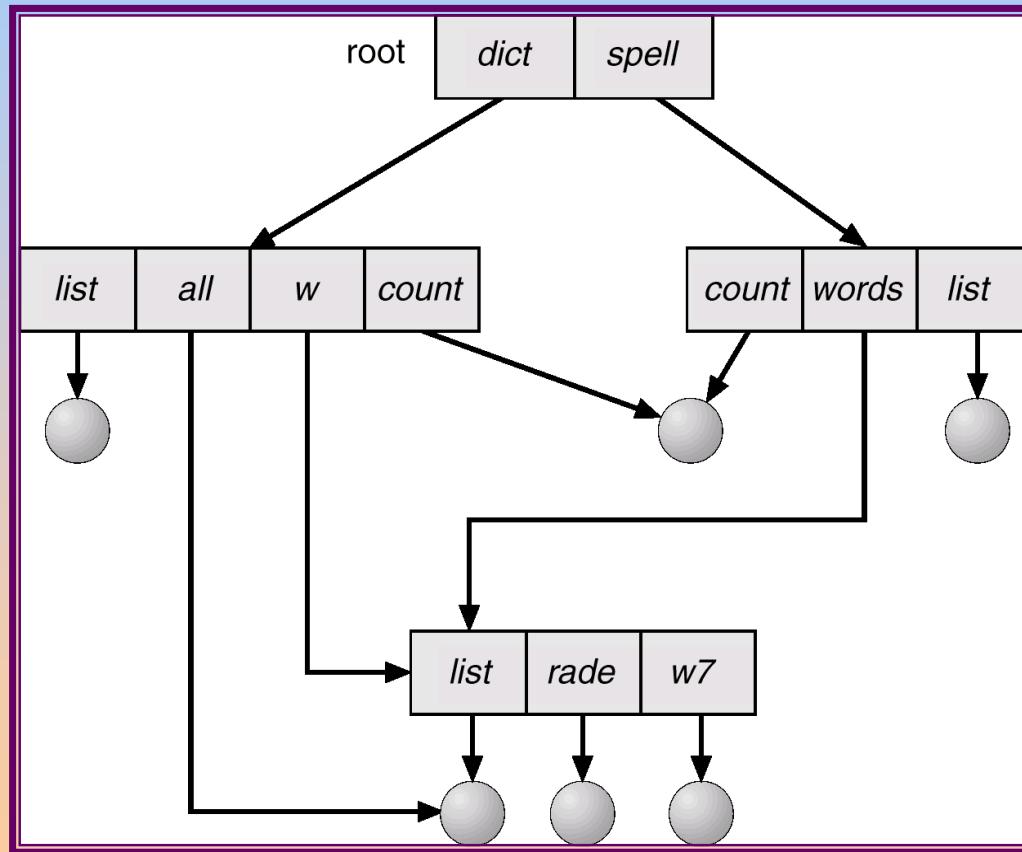
Deleting "mail" ⇒ deleting the entire subtree rooted by "mail".
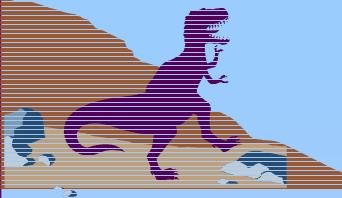
Acyclic-Graph Directories

■ Have shared subdirectories and files.
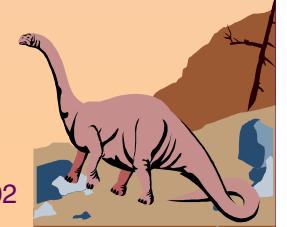
Acyclic-Graph Directories (Cont.)
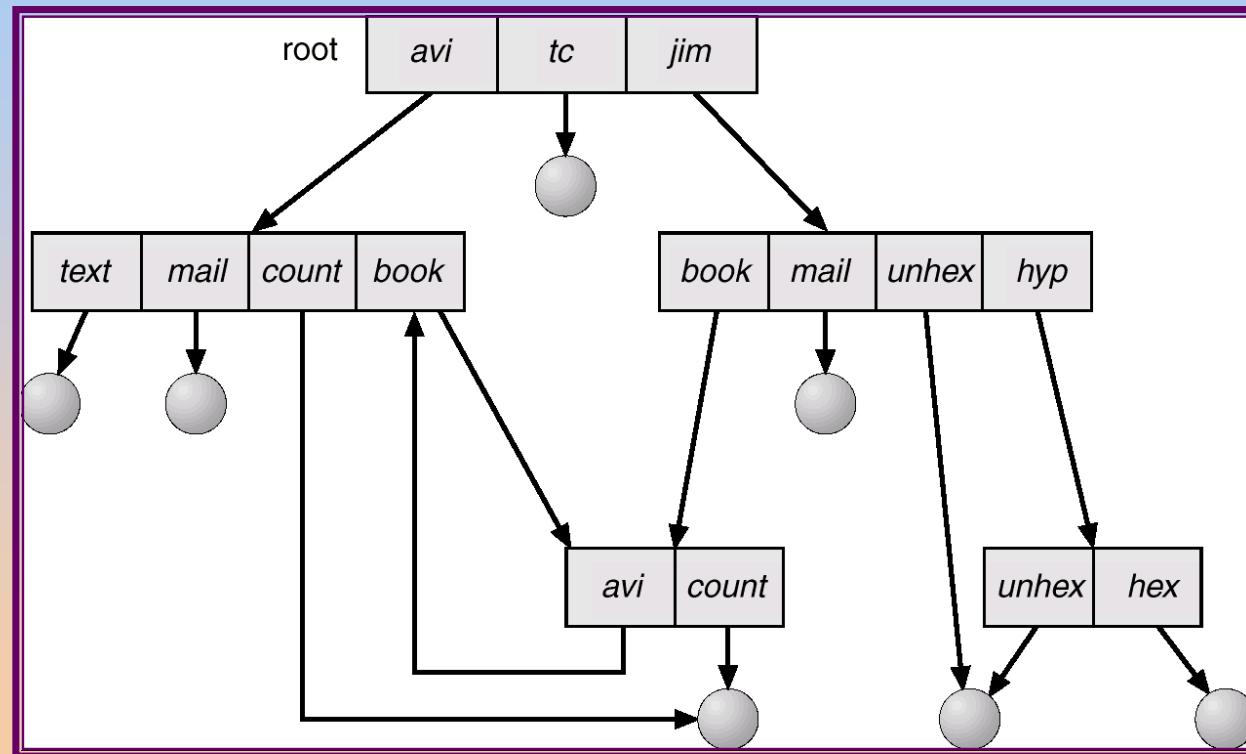
■ Two different names (aliasing)

■ If *dict* deletes *list* $\Rightarrow$ dangling pointer.
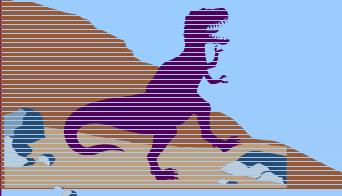
Solutions:

✦ Backpointers, so we can delete all pointers.
Variable size records a problem.

✦ Backpointers using a daisy chain organization.

✦ Entry-hold-count solution.

General Graph Directory
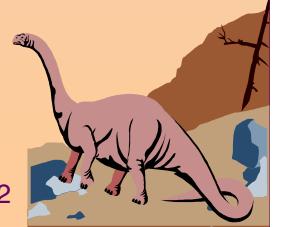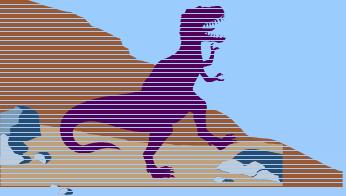
General Graph Directory (Cont.)

■ How do we guarantee no cycles?

✦ Allow only links to file not subdirectories.

✦ Garbage collection.

✦ Every time a new link is added use a cycle detection algorithm to determine whether it is OK.
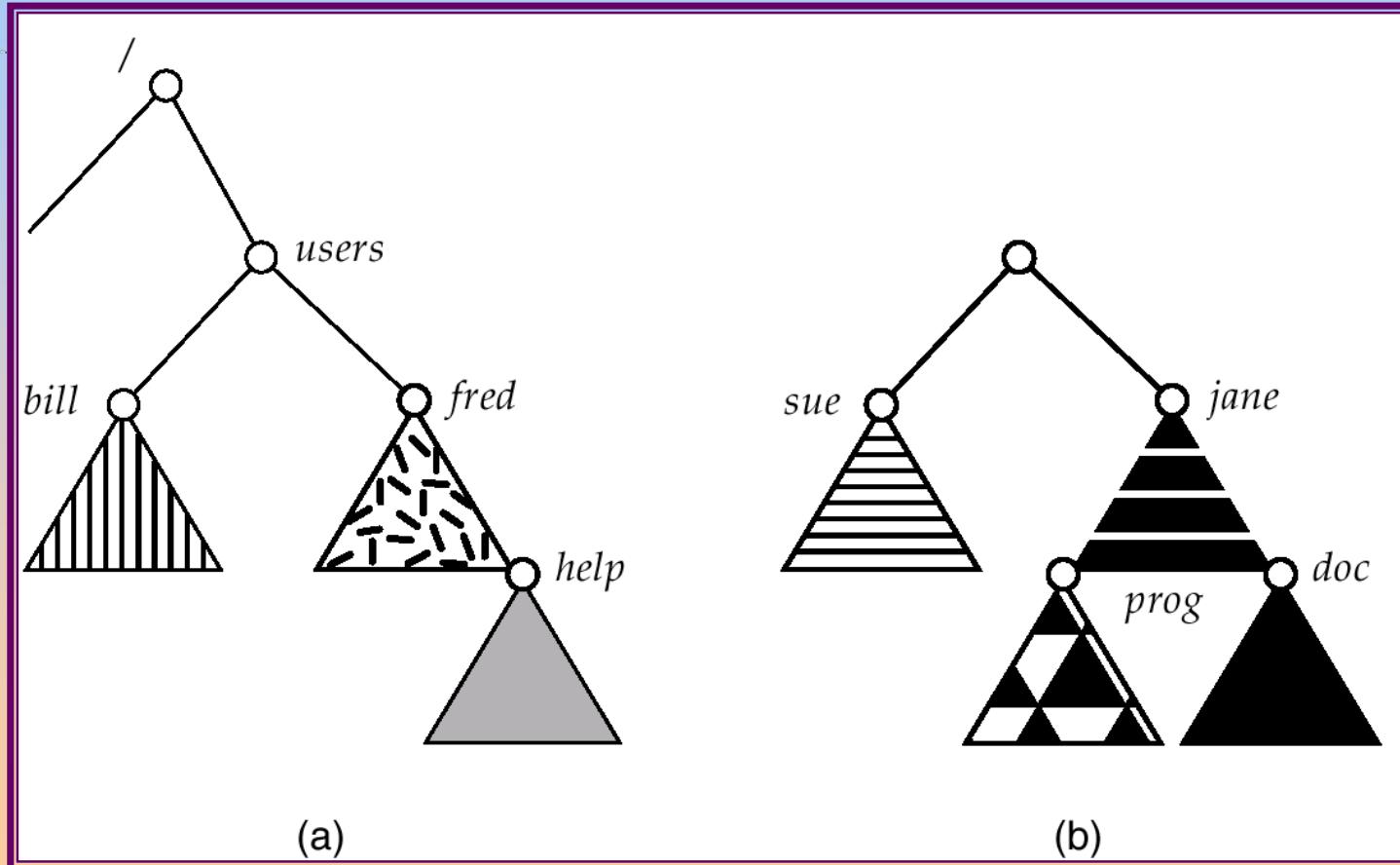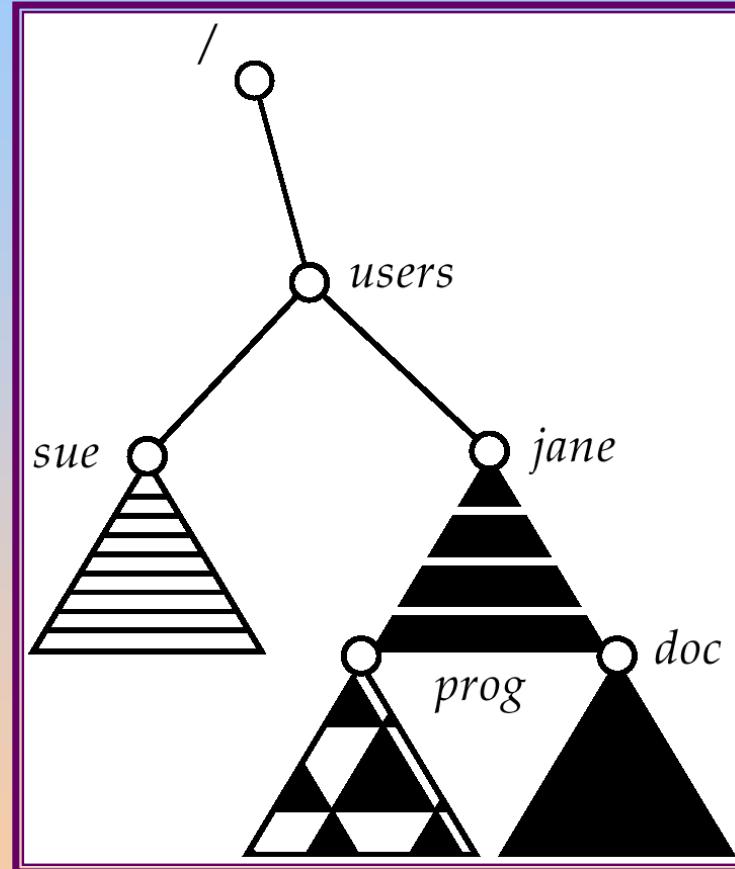
File System Mounting
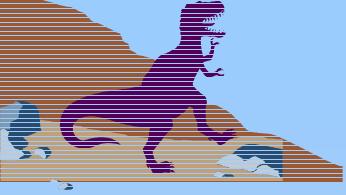
- A file system must be **mounted** before it can be accessed.

- A unmounted file system (I.e. Fig. 11-11(b)) is mounted at a **mount point**.

(a)

(b)
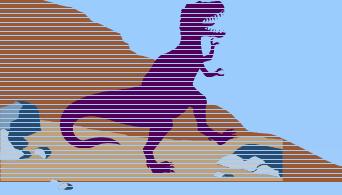
Mount Point

File Sharing

- Sharing of files on multi-user systems is desirable.

- Sharing may be done through a *protection* scheme.

- On distributed systems, files may be shared across a network.

- Network File System (NFS) is a common distributed file-sharing method.
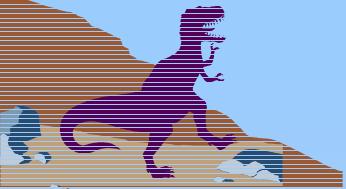
Protection

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

- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List
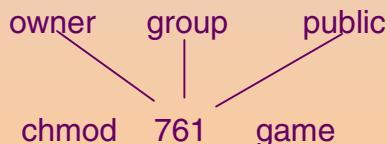
Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users
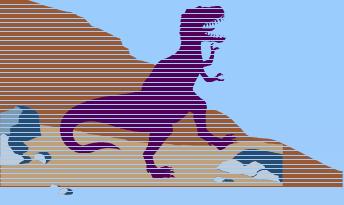
				RWX
a)	**owner access**	7	$\Rightarrow$	1 1 1
				RWX
b)	**group access**	6	$\Rightarrow$	1 1 0
				RWX
c)	**public access**	1	$\Rightarrow$	0 0 1

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

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

owner   group   public

chmod   761   game
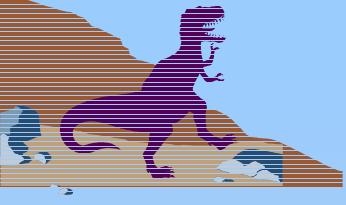
Attach a group to a file

chgrp   G   game
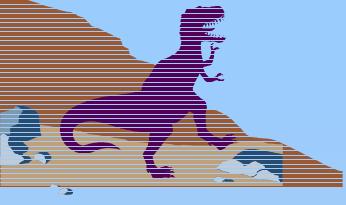
Chapter 12: File System Implementation

- File System Structure
- File System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS
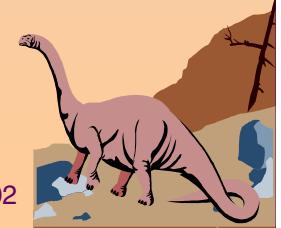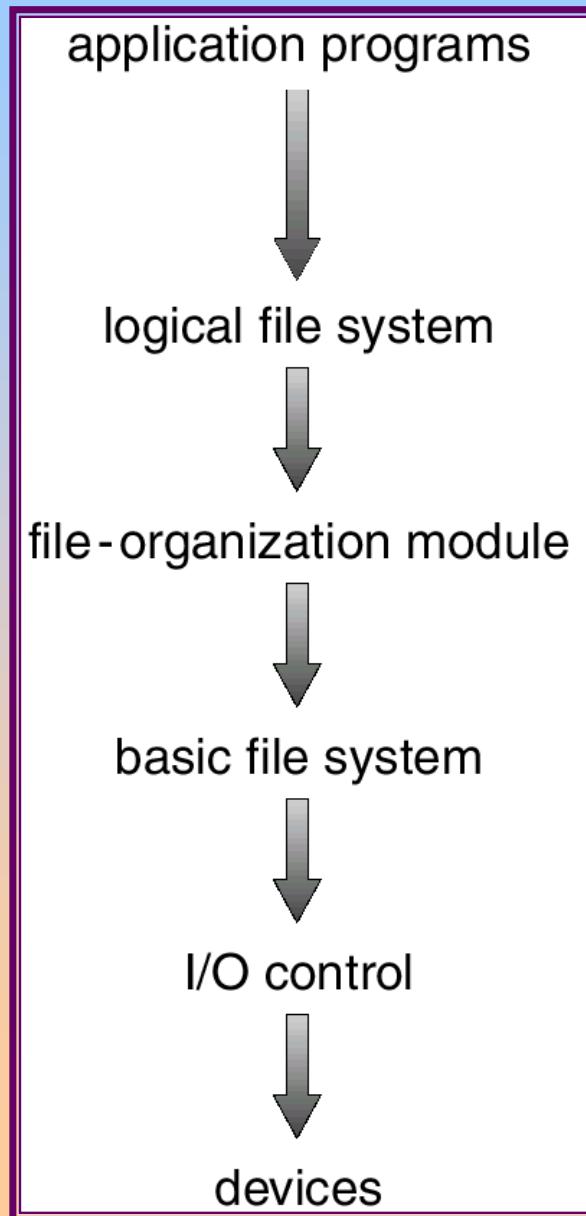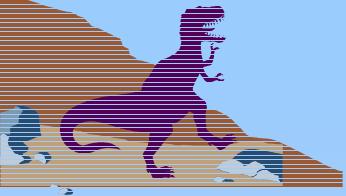
File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks).
- File system organized into layers.
- *File control block* – storage structure consisting of information about a file.

Layered File System

application programs

↓

logical file system

↓

file-organization module

↓

basic file system

↓

I/O control

↓

devices

A Typical File Control Block
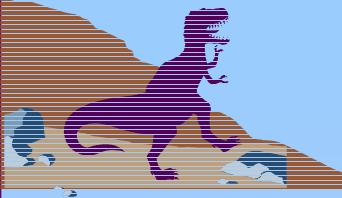
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks
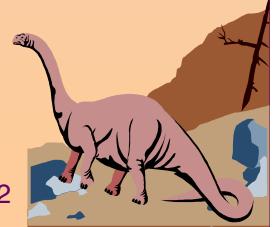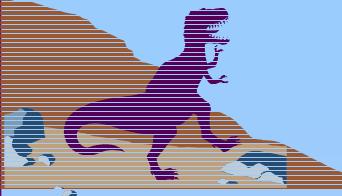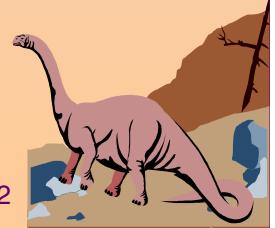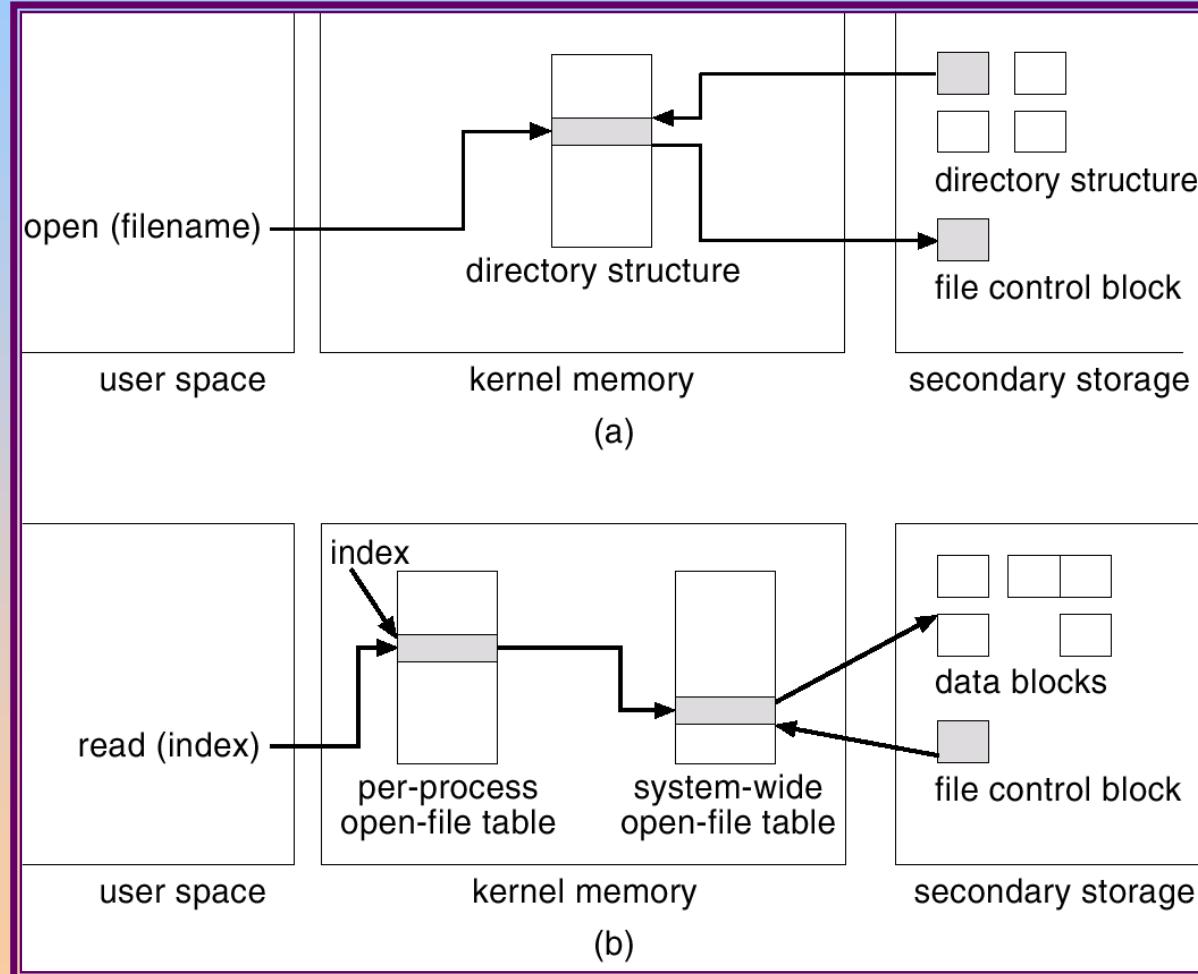
In-Memory File System Structures
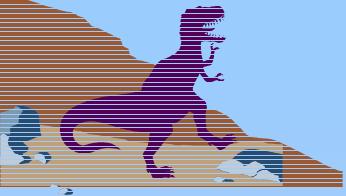
- The following figure illustrates the necessary file system structures provided by the operating systems.

- Figure 12-3(a) refers to opening a file.

- Figure 12-3(b) refers to reading a file.

In-Memory File System Structures

open (filename)

directory structure

directory structure

file control block

user space — kernel memory — secondary storage

(a)

index

read (index)

per-process open-file table

system-wide open-file table

data blocks

file control block

user space — kernel memory — secondary storage

(b)
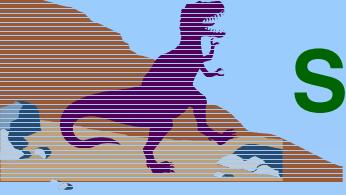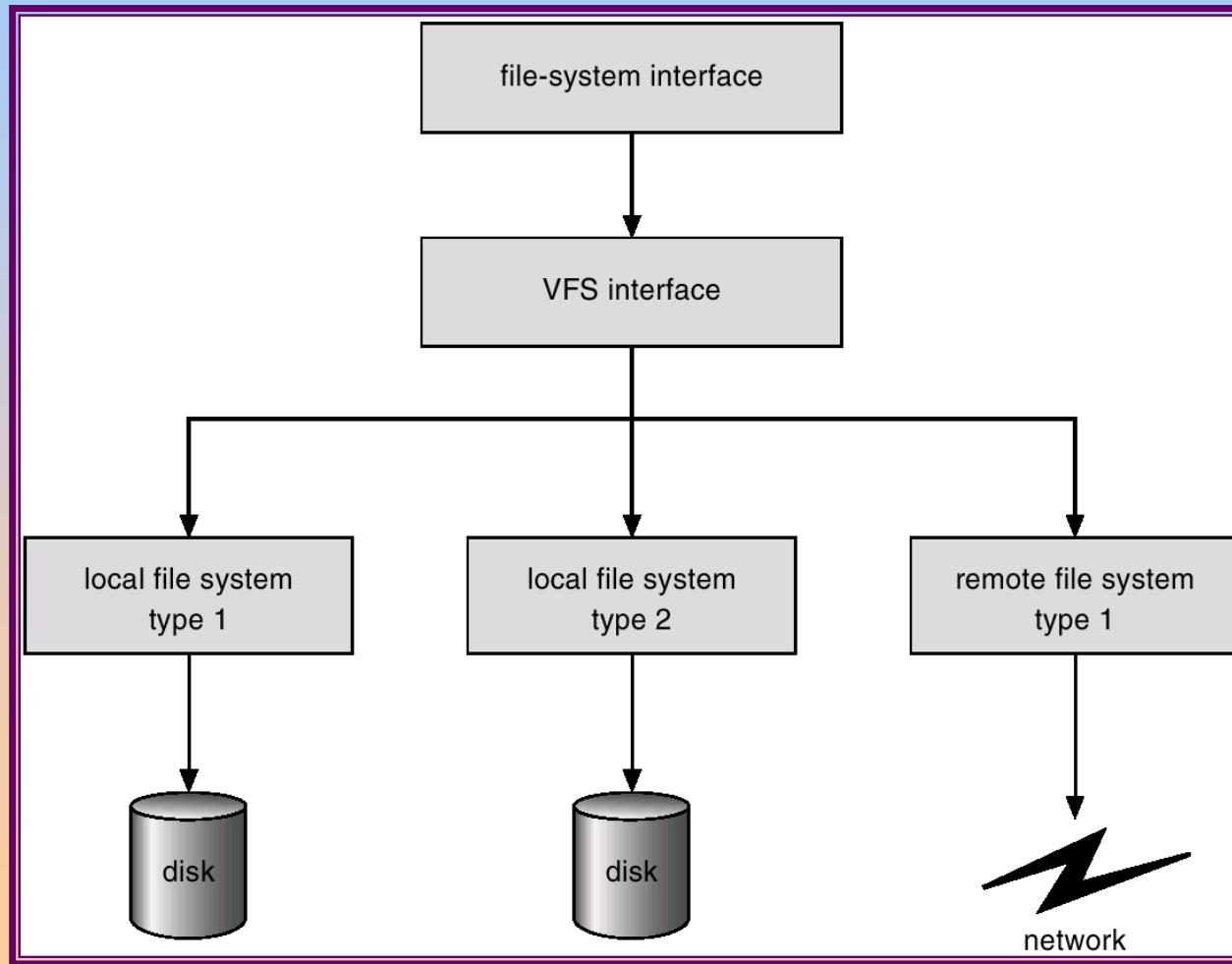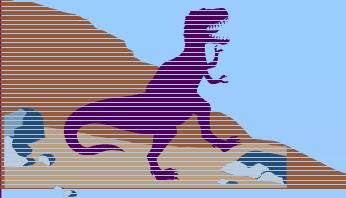
Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.

- VFS allows the same system call interface (the API) to be used for different types of file systems.

- The API is to the VFS interface, rather than any specific type of file system.
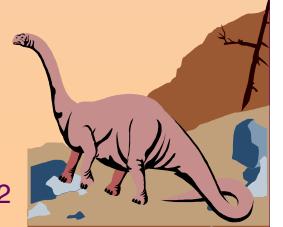
Schematic View of Virtual File System


```
                    +------------------------+
                    |  file-system interface |
                    +------------------------+
                                |
                                v
                    +------------------------+
                    |     VFS interface      |
                    +------------------------+
                                |
           +--------------------+--------------------+
           |                    |                    |
           v                    v                    v
  +-----------------+  +-----------------+  +------------------+
  | local file system| | local file system| | remote file system|
  |     type 1      |  |     type 2      |  |      type 1      |
  +-----------------+  +-----------------+  +------------------+
           |                    |                    |
           v                    v                    v
        (disk)               (disk)             (network)
```
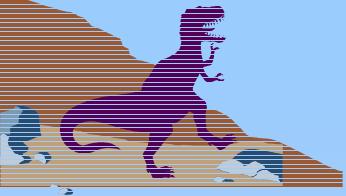
Directory Implementation
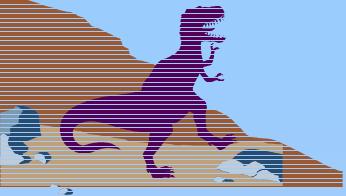
- Linear list of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute

- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
  - fixed size

Allocation Methods
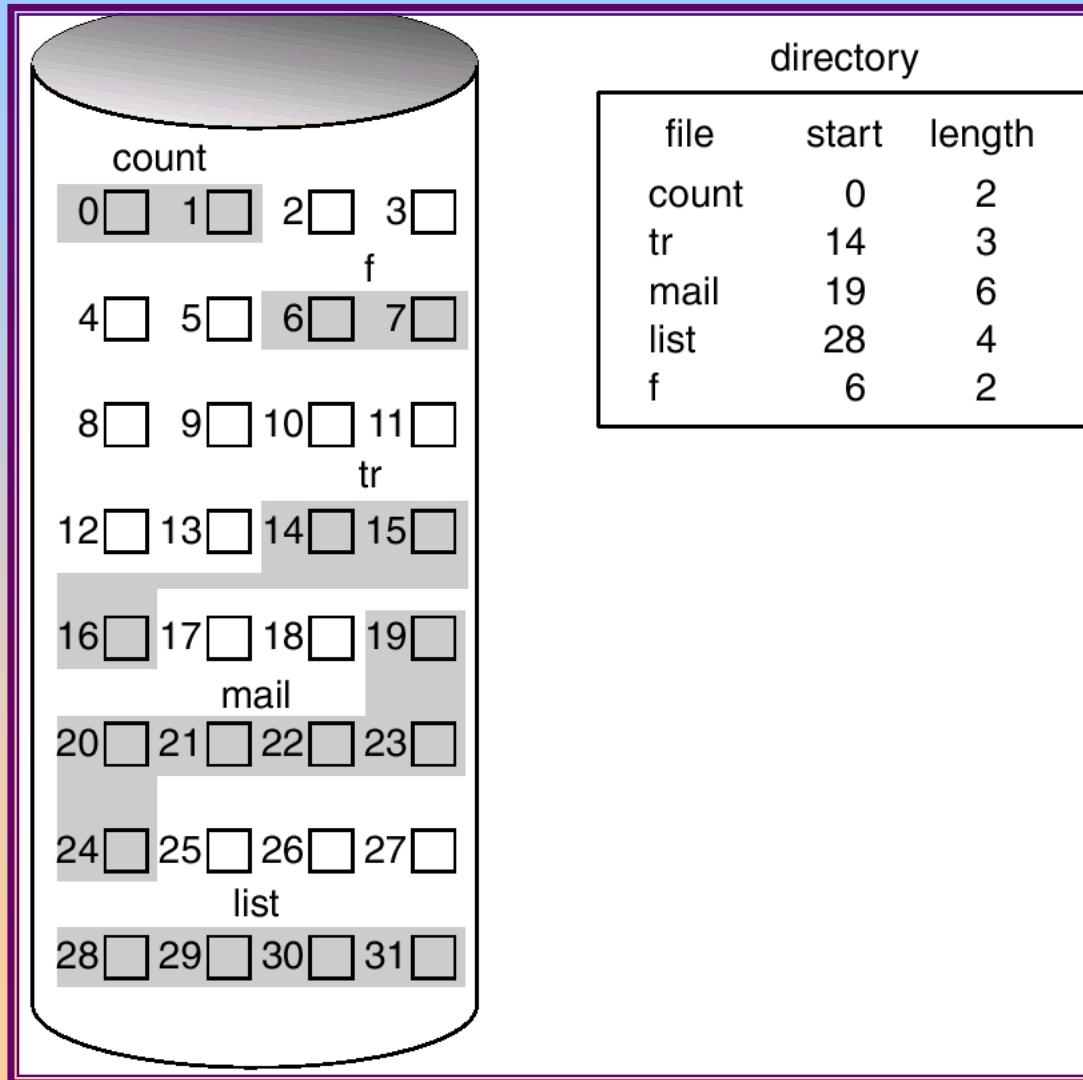
- An allocation method refers to how disk blocks are allocated for files:

- Contiguous allocation

- Linked allocation

- Indexed allocation
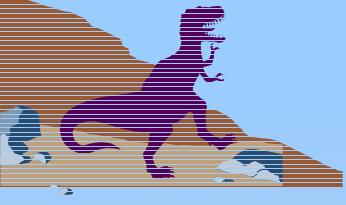
Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.

- Simple – only starting location (block #) and length (number of blocks) are required.

- Random access.

- Wasteful of space (dynamic storage-allocation problem).

- Files cannot grow.
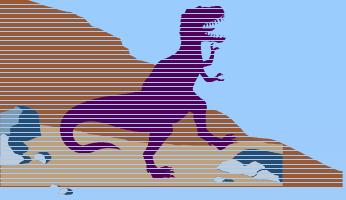
Contiguous Allocation of Disk Space

count

| 0 | 1 | 2 | 3 |

f

| 4 | 5 | 6 | 7 |

| 8 | 9 | 10 | 11 |

tr

| 12 | 13 | 14 | 15 |

| 16 | 17 | 18 | 19 |

mail

| 20 | 21 | 22 | 23 |

| 24 | 25 | 26 | 27 |

list

| 28 | 29 | 30 | 31 |

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2
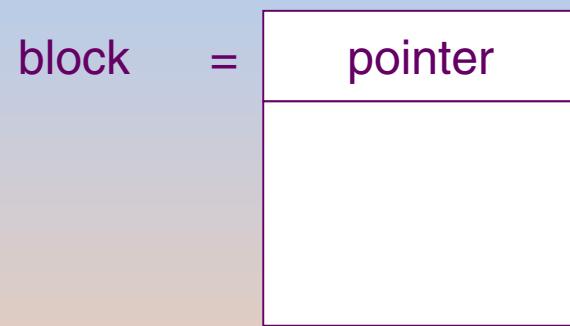
Extent-Based Systems
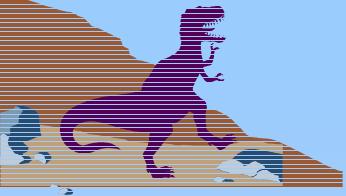
- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme.

- Extent-based file systems allocate disk blocks in **extents**.

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

Linked Allocation

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

block   = | pointer |

Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping

$$LA/511 \begin{cases} Q \\ R \end{cases}$$
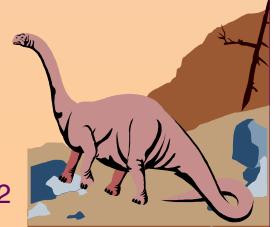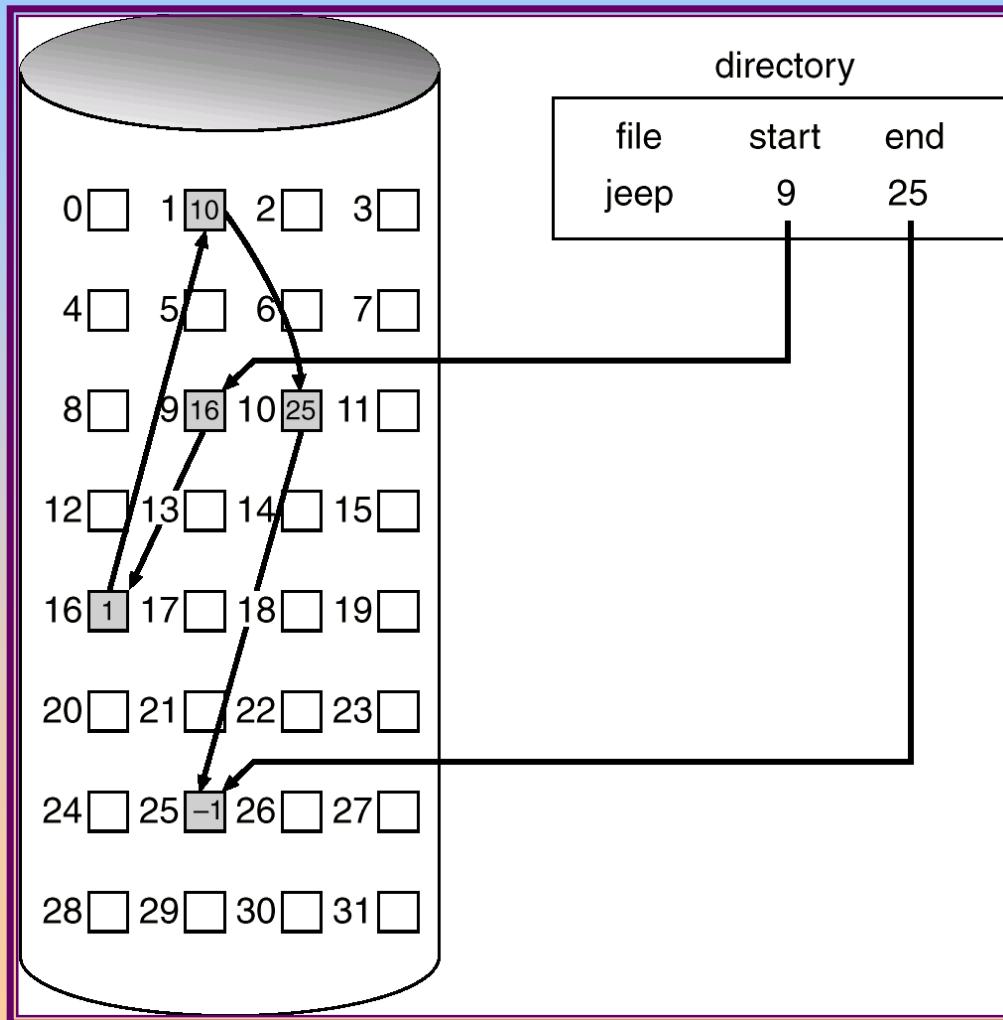
Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1
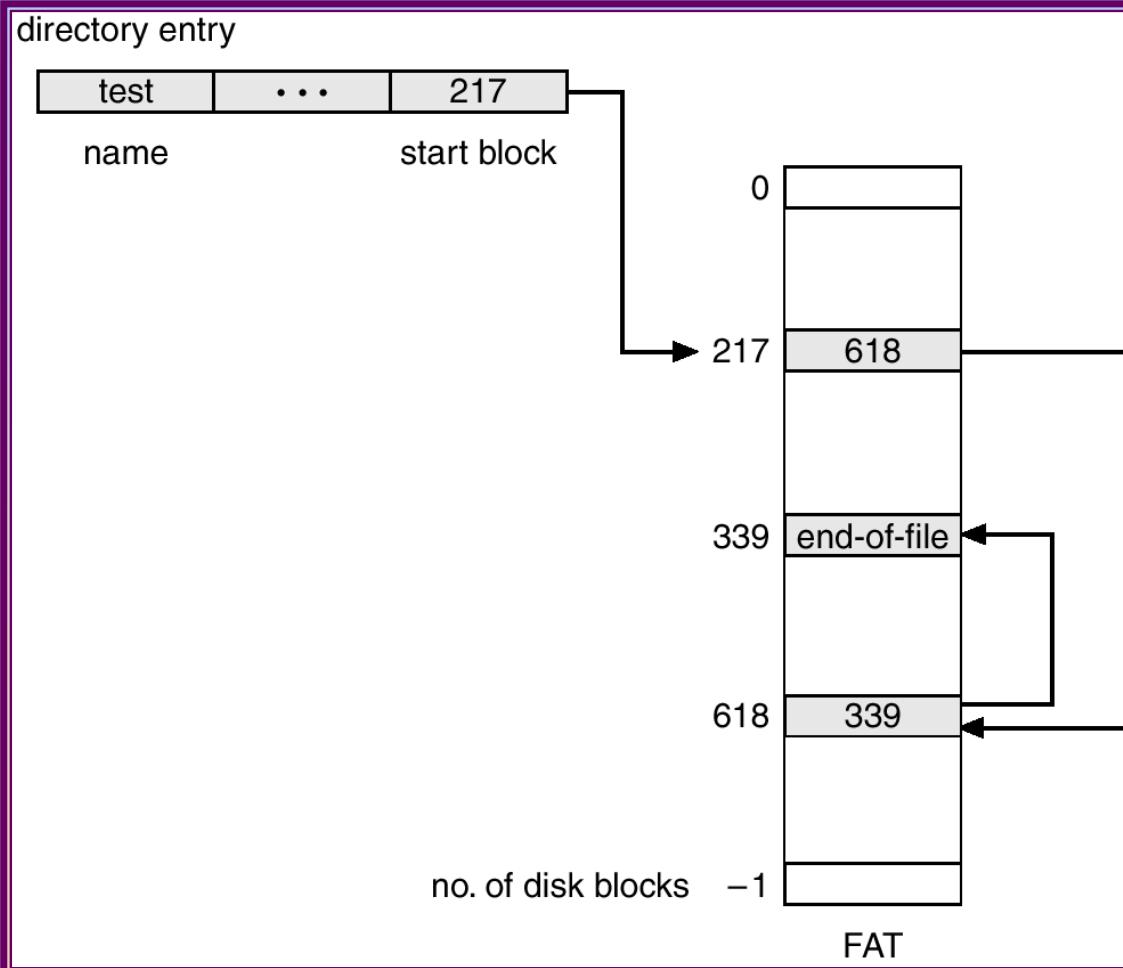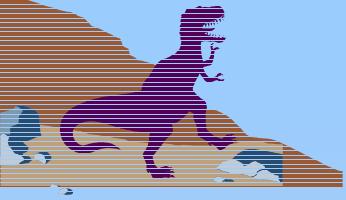
File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.

Linked Allocation

File-Allocation Table

directory entry

test	• • •	217

name   start block

0	
217	618
339	end-of-file
618	339
no. of disk blocks −1	

FAT
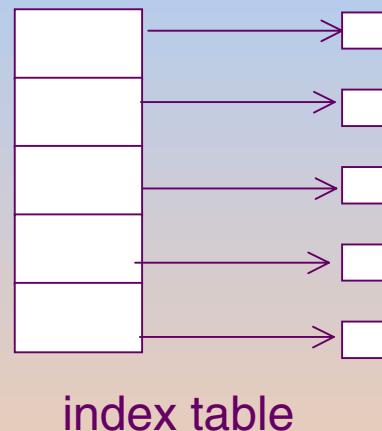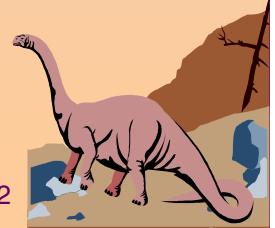
Indexed Allocation
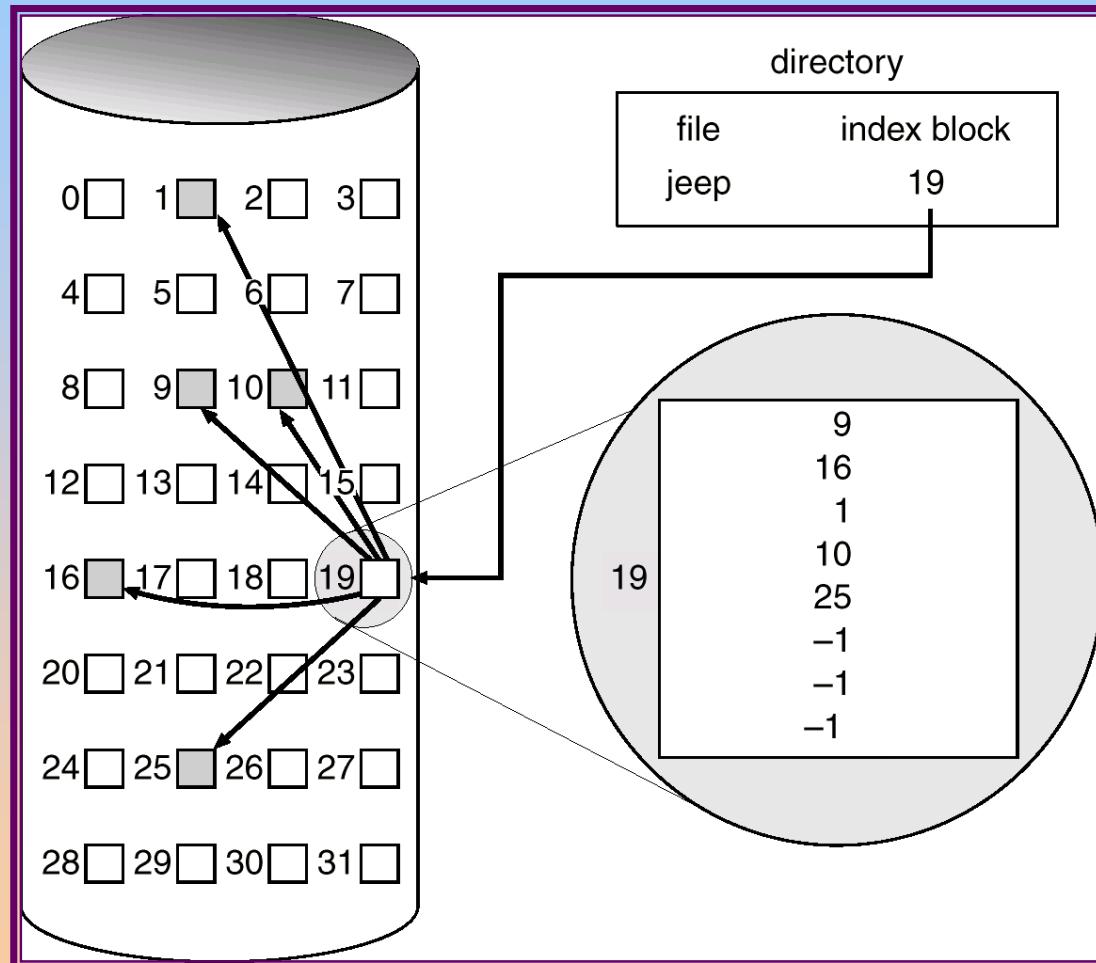
- Brings all pointers together into the *index block.*
- Logical view.
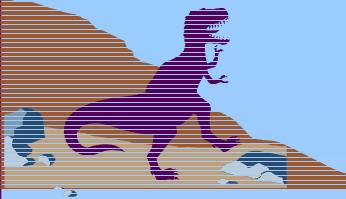
index table

Example of Indexed Allocation
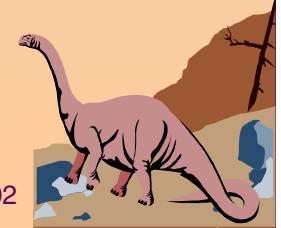
Indexed Allocation (Cont.)
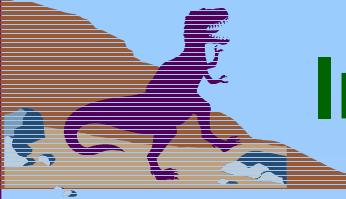
- Need index table

- 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.

$$LA/512 \begin{array}{c} Q \\ \\ R \end{array}$$

Q = displacement into index table
R = displacement into block
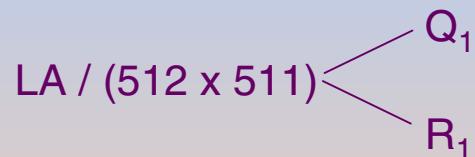
Indexed Allocation – Mapping (Cont.)

■ 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).
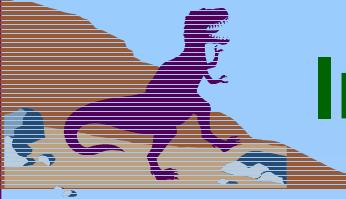
$$LA / (512 \times 511) \begin{cases} Q_1 \\ R_1 \end{cases}$$

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:
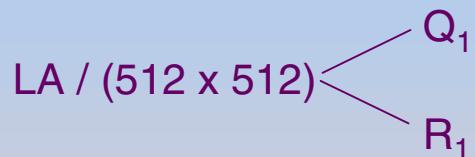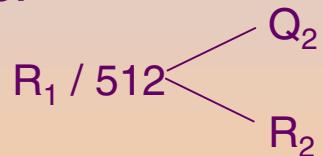
Indexed Allocation – Mapping (Cont.)

- 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:
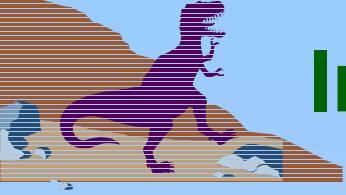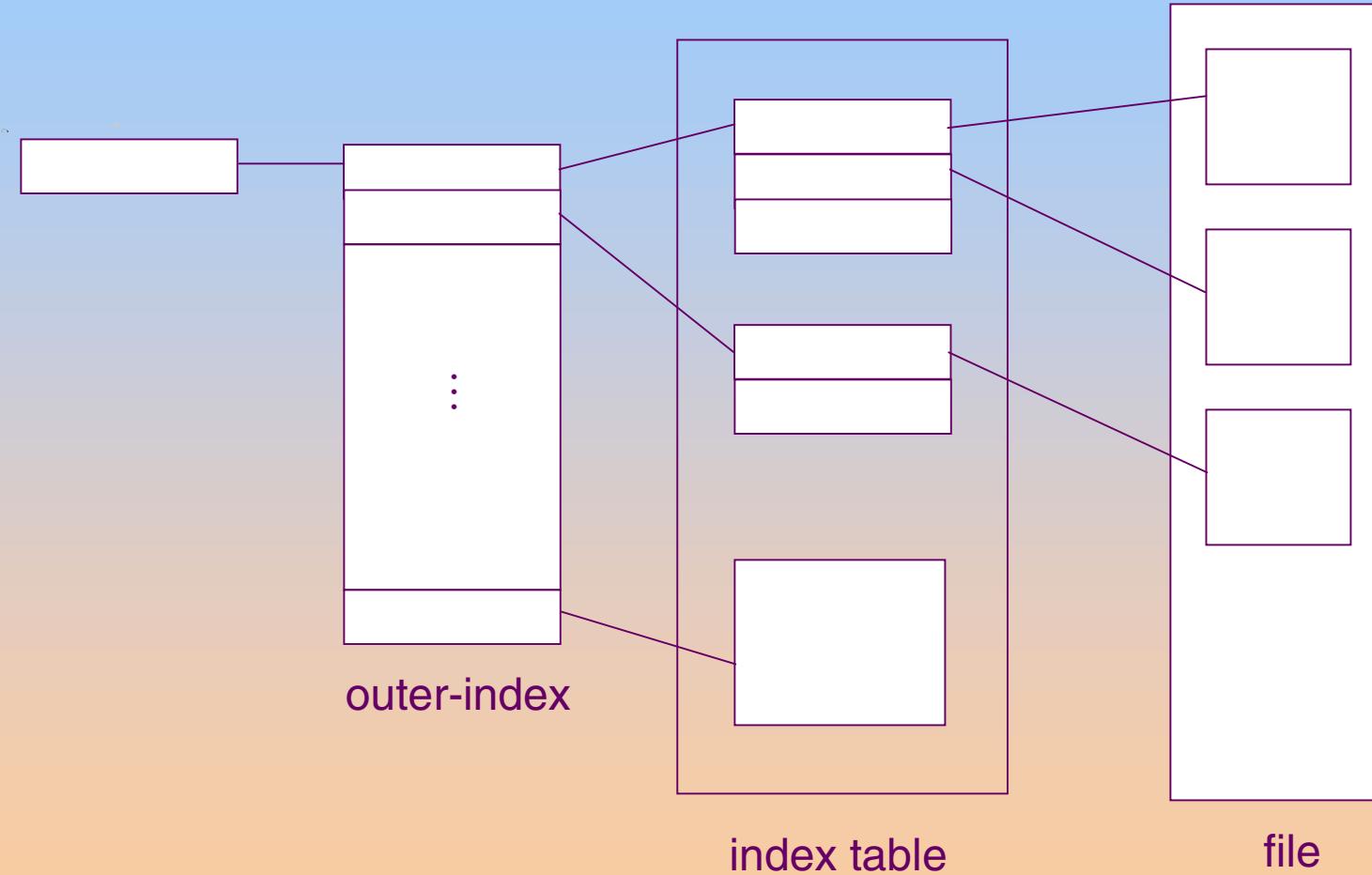
$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

outer-index

index table

file
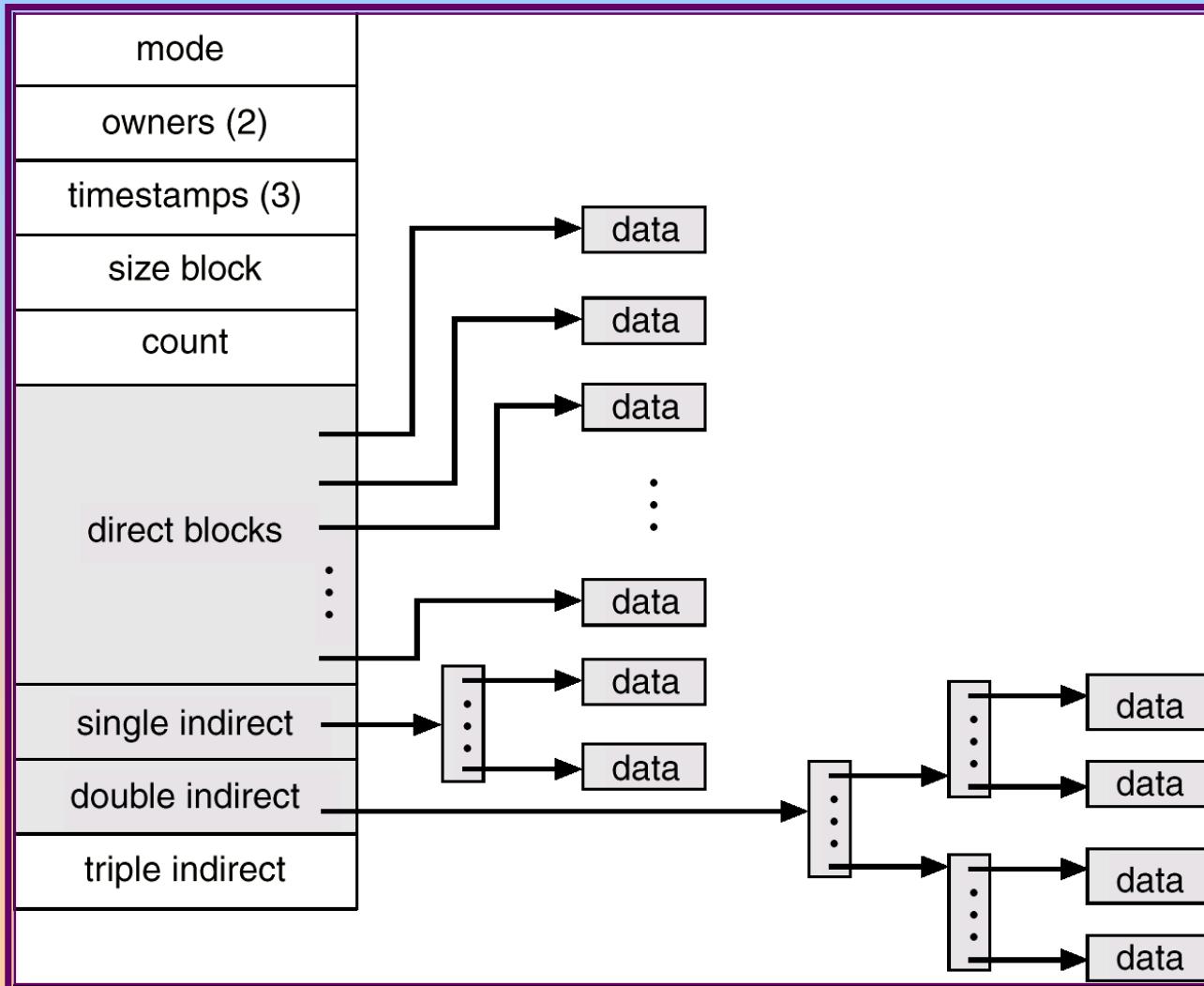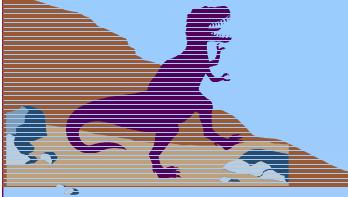
Combined Scheme: UNIX (4K bytes per block)

mode	
owners (2)	
timestamps (3)	→ data
size block	→ data
count	→ data
direct blocks	→ data
	→ data
single indirect	→ data
double indirect	→ data
triple indirect	→ data

Free-Space Management

- Bit vector (*n* blocks)
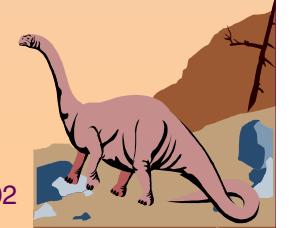
```
  0  1  2                    n-1
┌──┬──┬──┬──┬──┬──┬──────┬──┐
│  │  │  │  │  │  │  …   │  │
└──┴──┴──┴──┴──┴──┴──────┴──┘
```
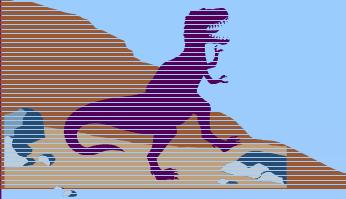
$$
bit[i] = \begin{cases} 0 \Rightarrow block[i] \ free \\ 1 \Rightarrow block[i] \ occupied \end{cases}
$$

Block number calculation
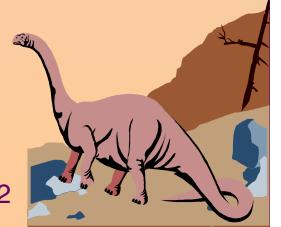
(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit
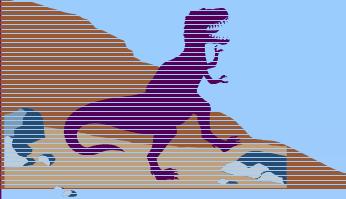
Free-Space Management (Cont.)

- 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)

- Easy to get contiguous files
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping
- Counting
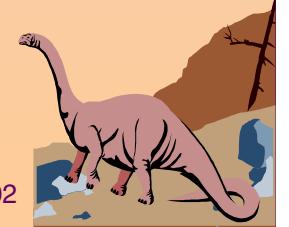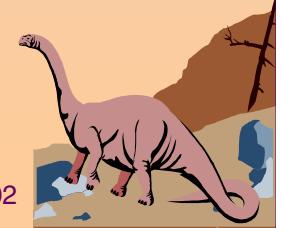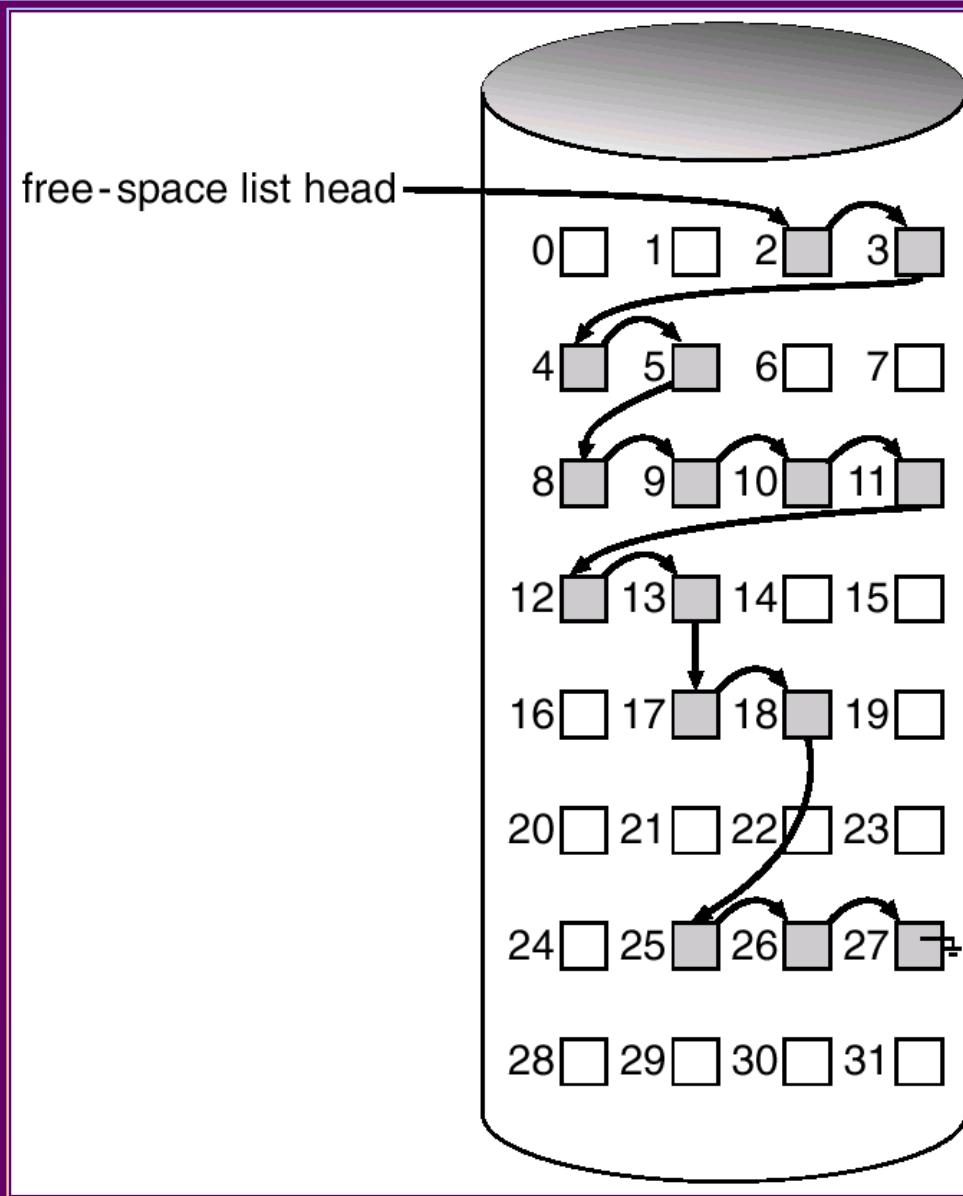
Free-Space Management (Cont.)
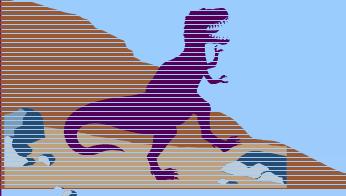
- Need to protect:
    - Pointer to free list
    - Bit map
        - Must be kept on disk
        - Copy in memory and disk may differ.
        - Cannot allow for block[i] to have a situation where bit[i] = 1 in memory and bit[i] = 0 on disk.
    - Solution:
        - Set bit[i] = 1 in disk.
        - Allocate block[i]
        - Set bit[i] = 1 in memory

Linked Free Space List on Disk
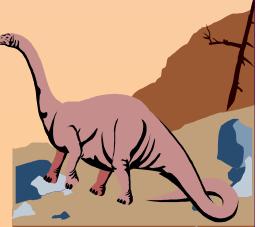
Efficiency and Performance
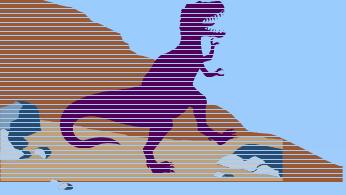
- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
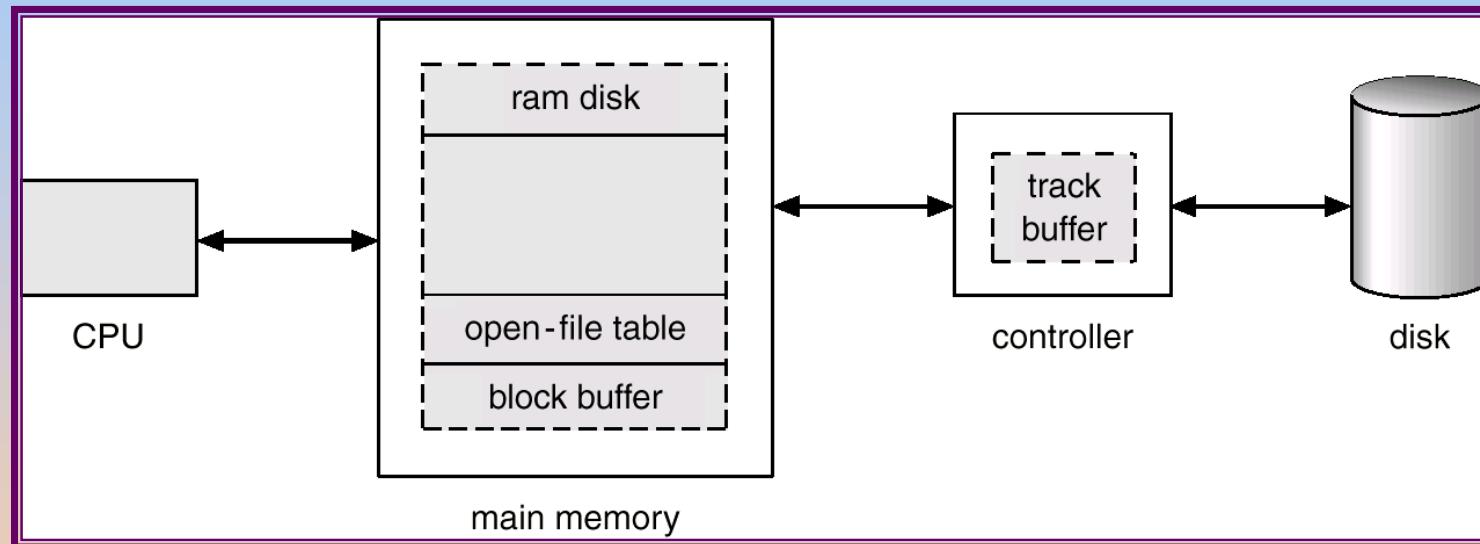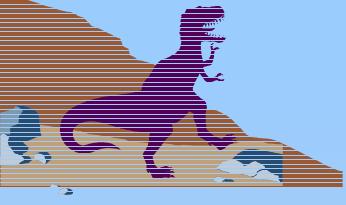
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.
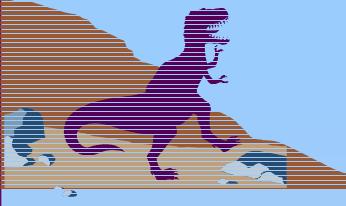
Various Disk-Caching Locations

CPU

ram disk

open-file table

block buffer

main memory

track buffer

controller
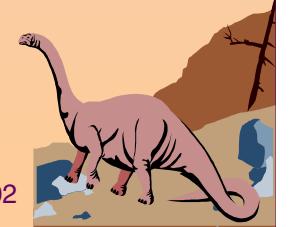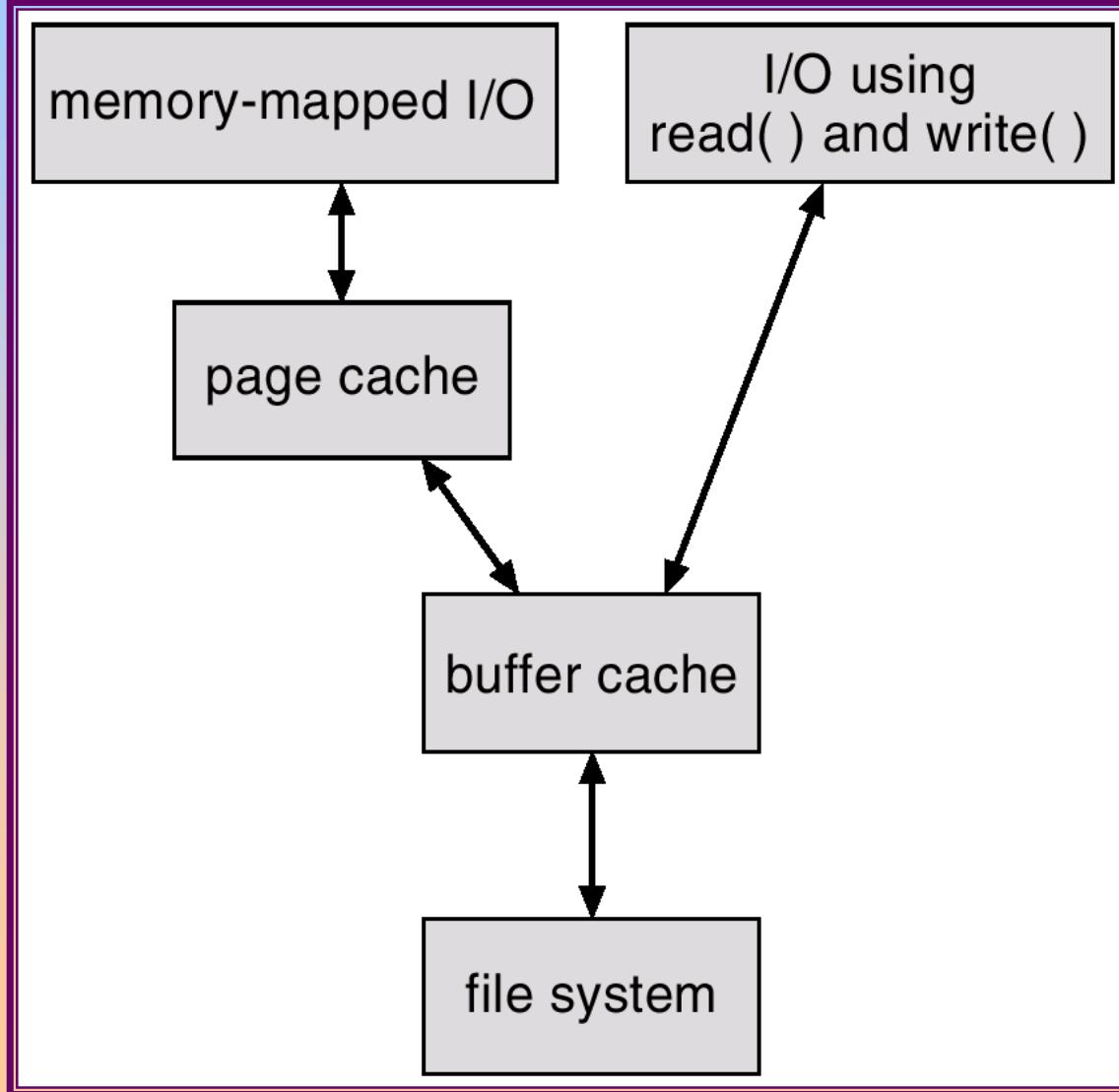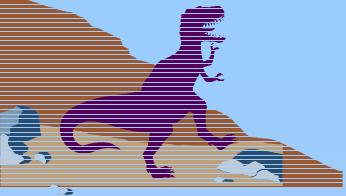
disk

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.

- This leads to the following figure.

I/O Without a Unified Buffer Cache
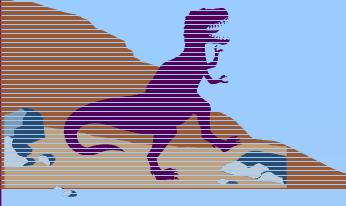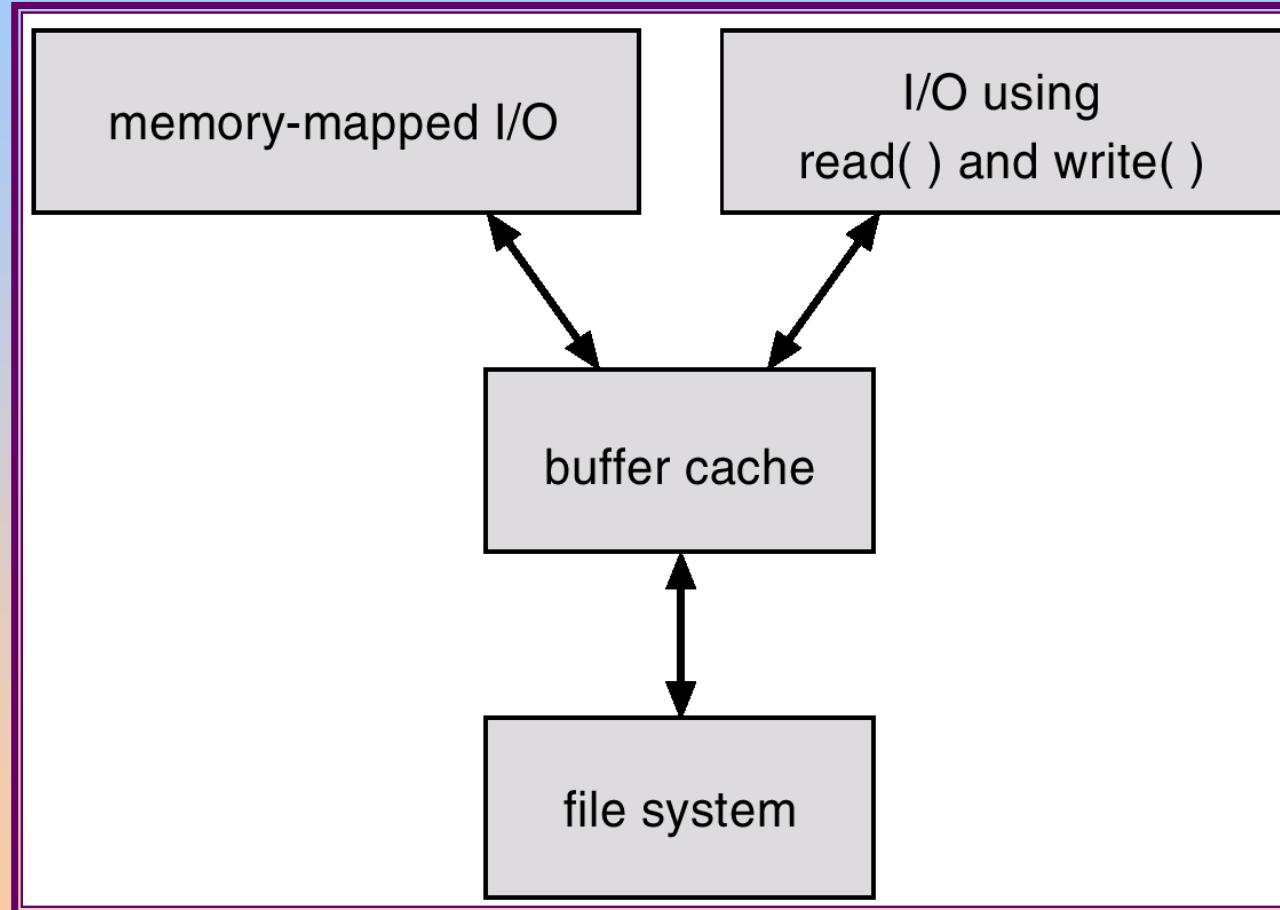
Unified Buffer Cache
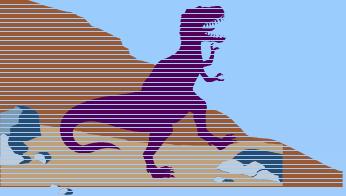
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.
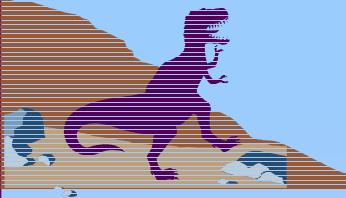
I/O Using a Unified Buffer Cache
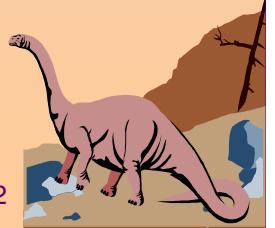
Recovery
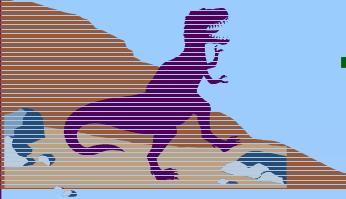
■ Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.

■ 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.

Log Structured File Systems
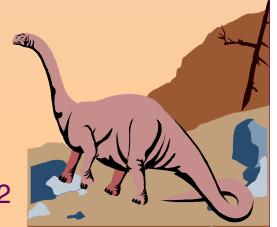
- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**.

- 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.
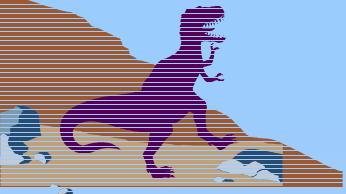
The Sun Network File System (NFS)
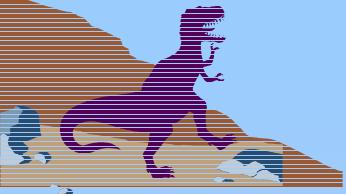
■ An implementation and a specification of a software system for accessing remote files across LANs (or WANs).

■ The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet.
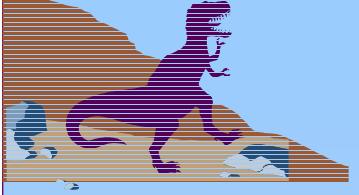
NFS (Cont.)

■ Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner.

  ✦ A remote directory is mounted over a local file system directory. The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory.

  ✦ Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided. Files in the remote directory can then be accessed in a transparent manner.

  ✦ Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory.
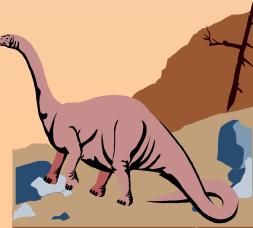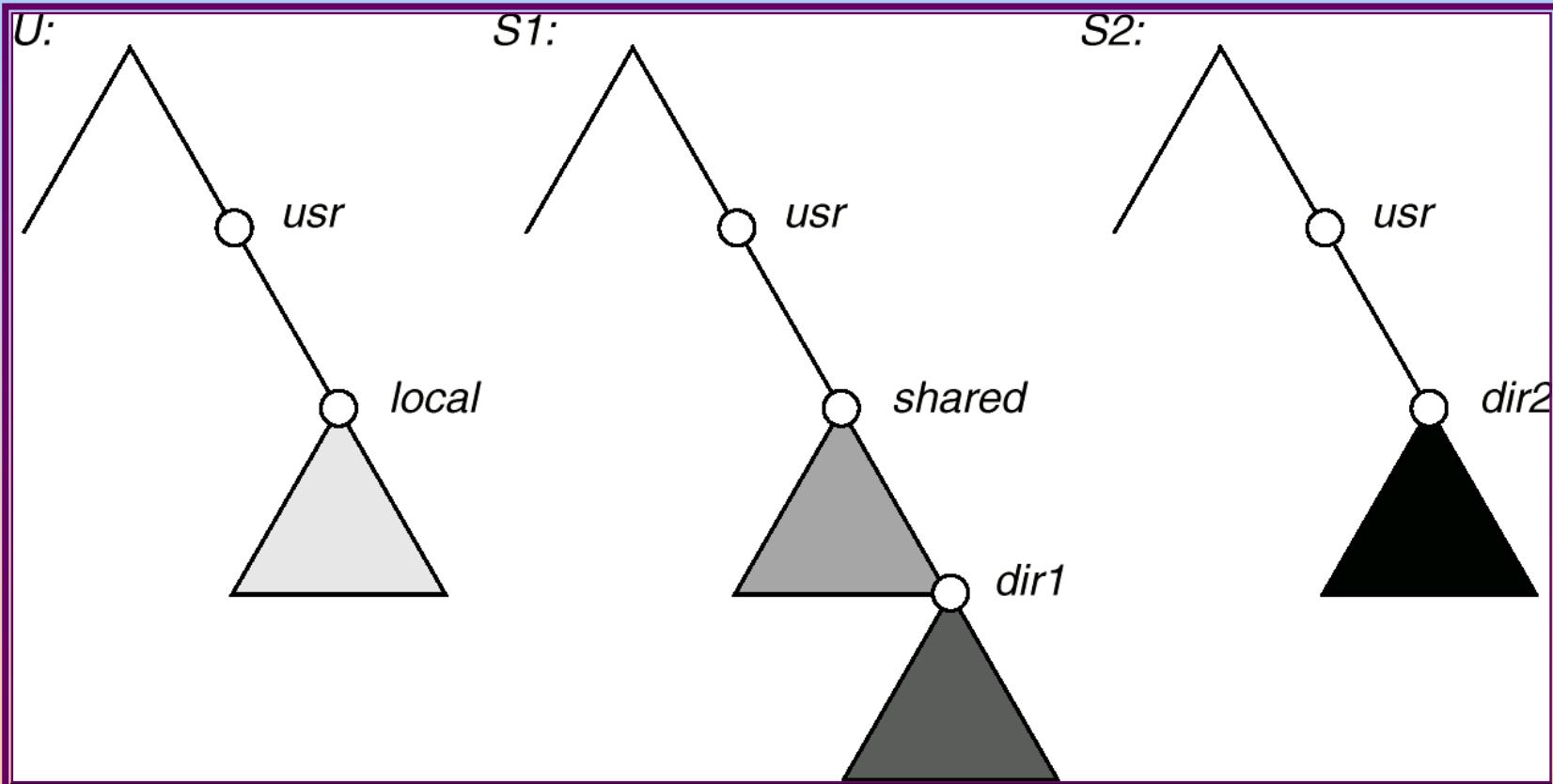
NFS (Cont.)
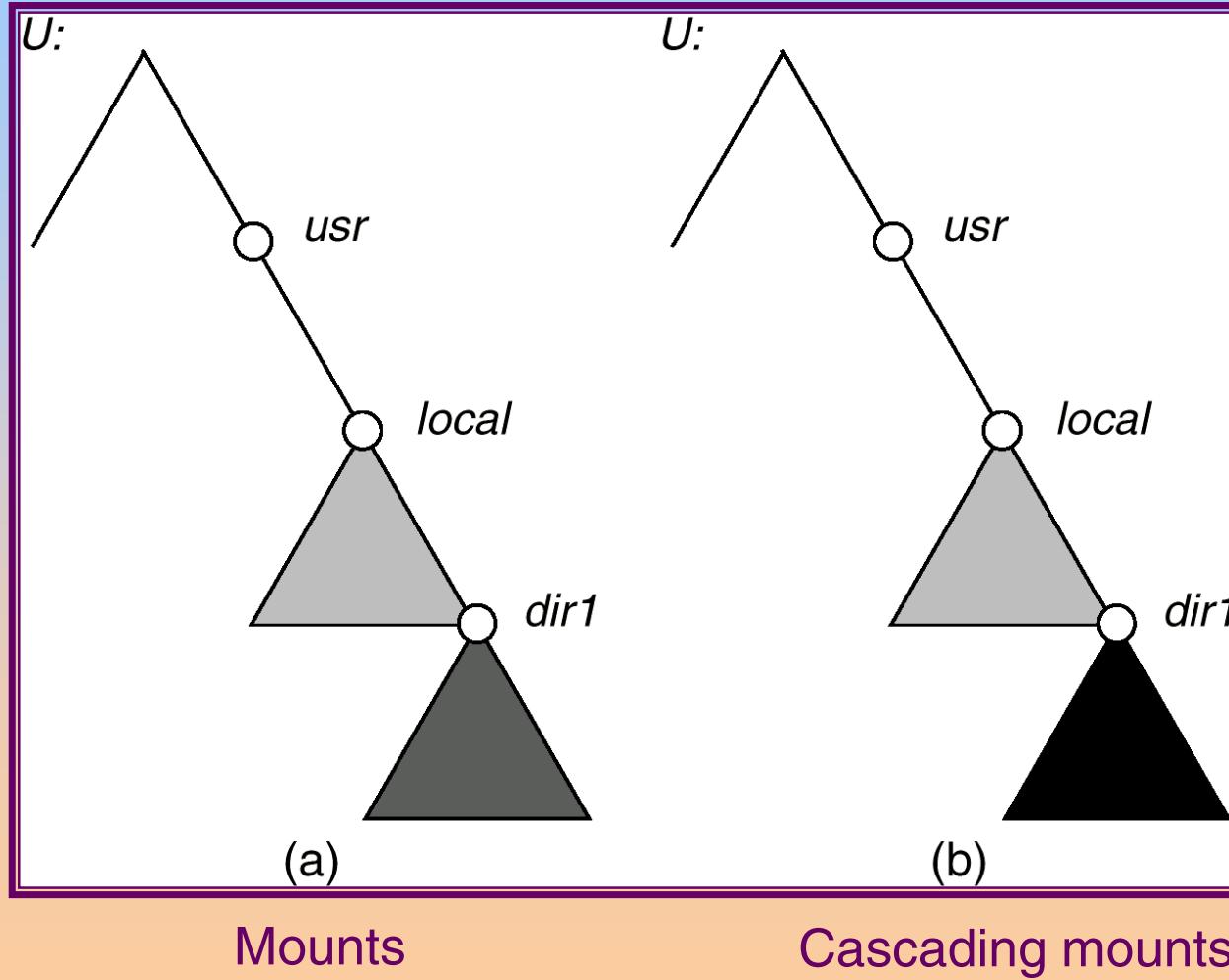
■ NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media.

■ This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces.

■ The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services.
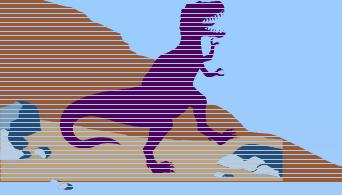
Three Independent File Systems

Mounting in NFS
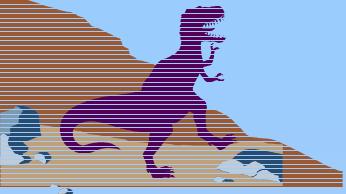
(a) Mounts (b) Cascading mounts
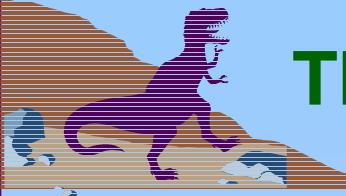
NFS Mount Protocol

- Establishes initial logical connection between server and client.

- Mount operation includes name of remote directory to be mounted and name of server machine storing it.
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine.
  - *Export list* – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them.

- Following a mount request that conforms to its export list, the server returns a *file handle*—a key for further accesses.

- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system.

- The mount operation changes only the user's view and does not affect the server side.
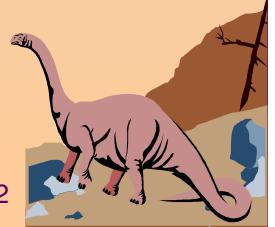
NFS Protocol

■ Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:

  ✦ searching for a file within a directory
  ✦ reading a set of directory entries
  ✦ manipulating links and directories
  ✦ accessing file attributes
  ✦ reading and writing files

■ NFS servers are *stateless*; each request has to provide a full set of arguments.

■ Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching).

■ The NFS protocol does not provide concurrency-control mechanisms.
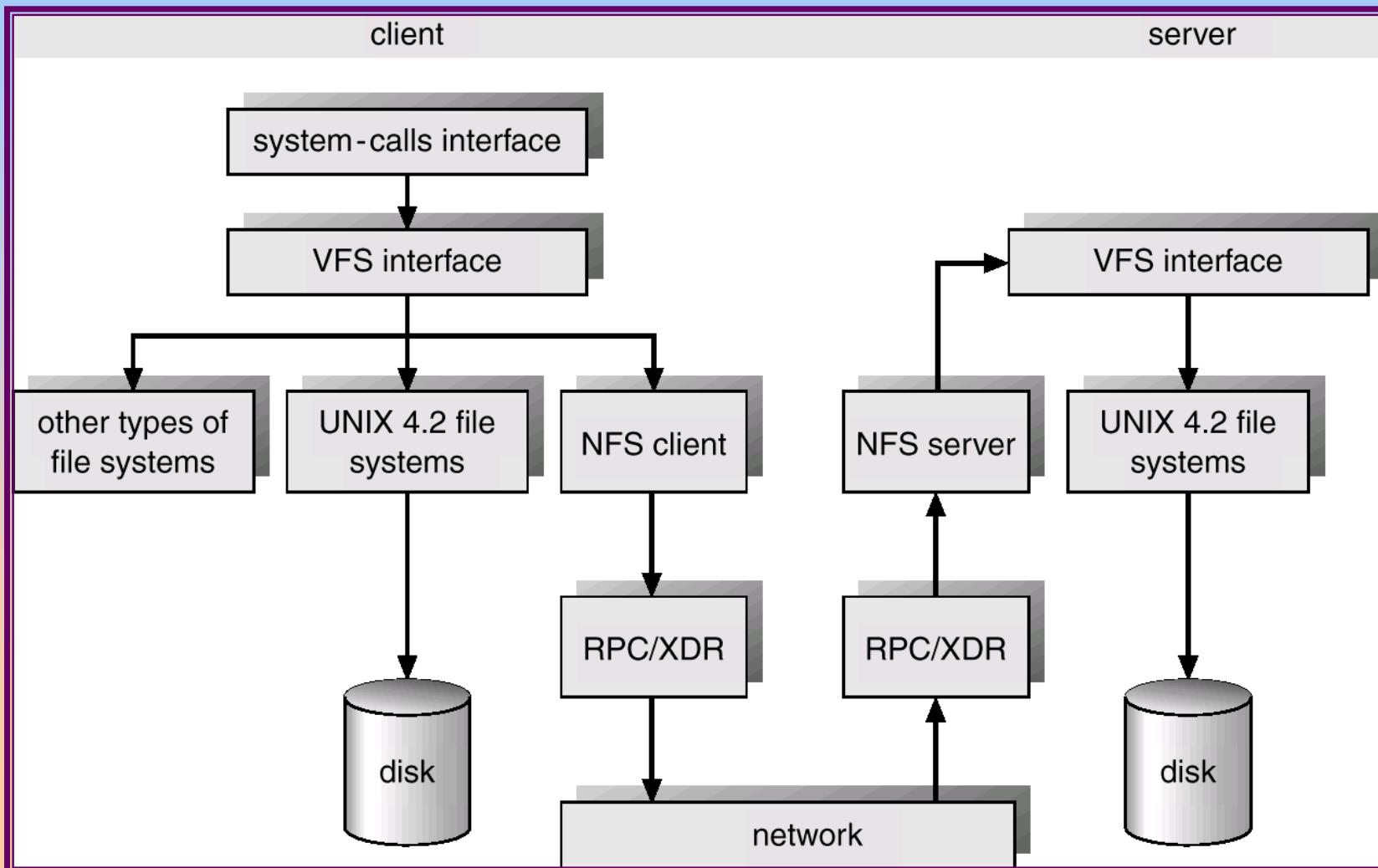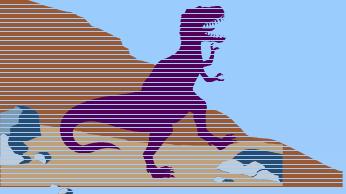
Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and file descriptors).

- *Virtual File System* (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types.
  - Calls the NFS protocol procedures for remote requests.

- NFS service layer – bottom layer of the architecture; implements the NFS protocol.
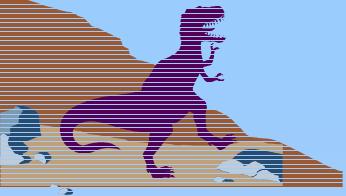
Schematic View of NFS Architecture
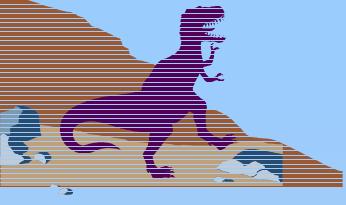
NFS Path-Name Translation

■ Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode.

■ To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names.
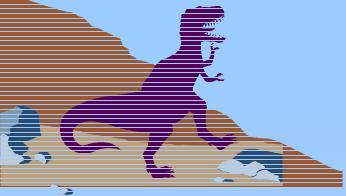
NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files).

- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance.

- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes. Cached file blocks are used only if the corresponding cached attributes are up to date.

- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server.

- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.

Chapter 13: I/O Systems
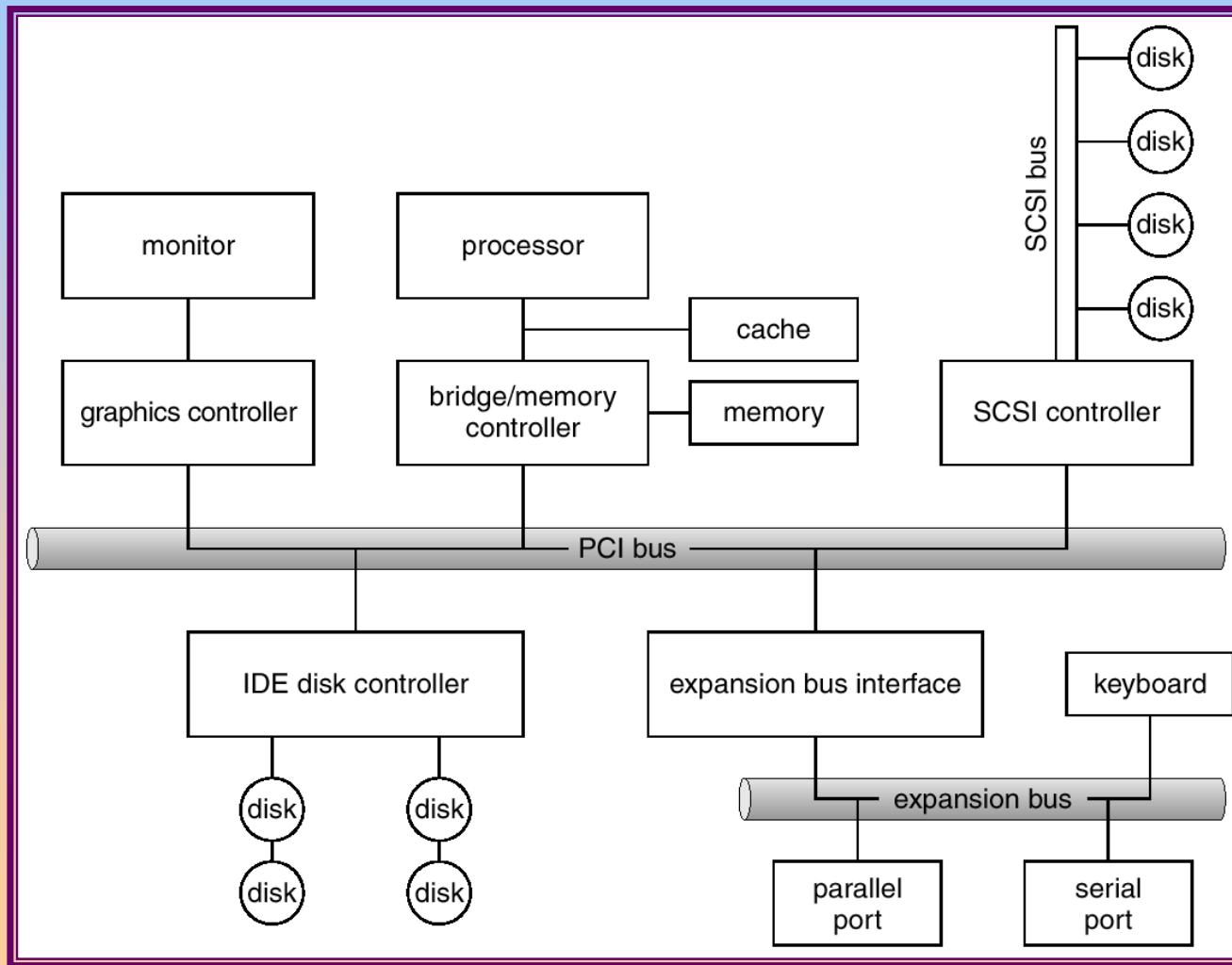
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Streams
- Performance
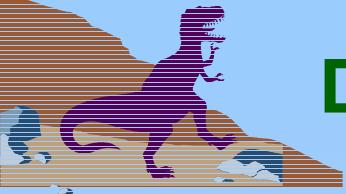
I/O Hardware

- Incredible variety of I/O devices
- Common concepts
    - Port
    - Bus (daisy chain or shared direct access)
    - Controller (host adapter)
- I/O instructions control devices
- Devices have addresses, used by
    - Direct I/O instructions
    - Memory-mapped I/O
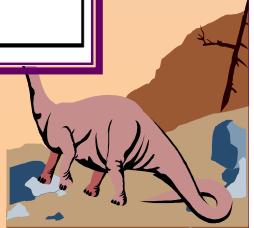
A Typical PC Bus Structure
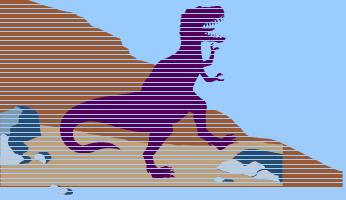
Device I/O Port Locations on PCs (partial)
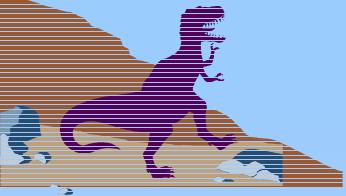
I/O address range (hexadecimal)	device
000-00F	DMA controller
020-021	interrupt controller
040-043	timer
200-20F	game controller
2F8-2FF	serial port (secondary)
320-32F	hard-disk controller
378-37F	parallel port
3D0-3DF	graphics controller
3F0-3F7	diskette-drive controller
3F8-3FF	serial port (primary)

Polling
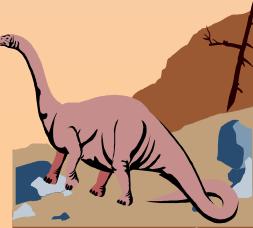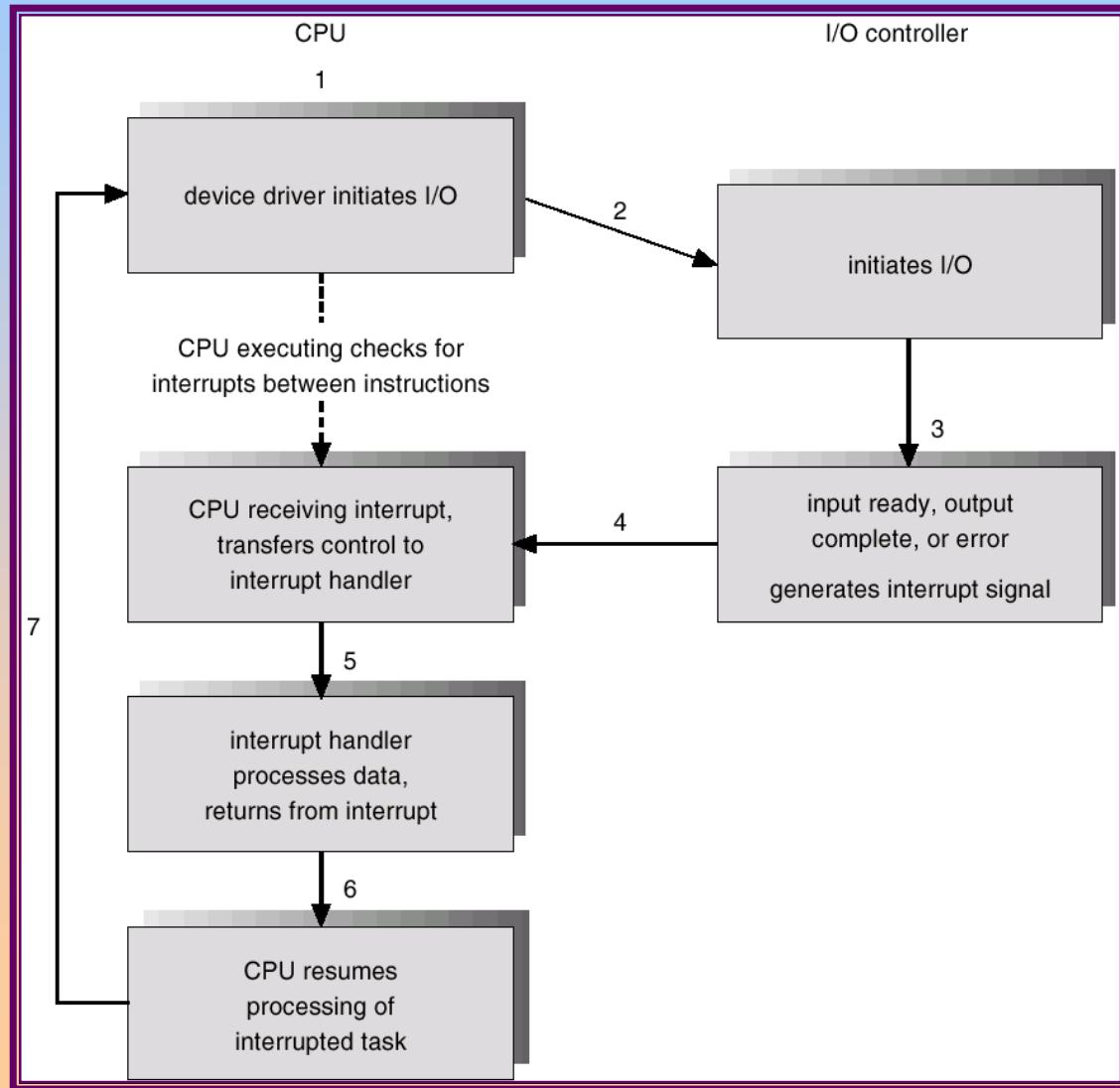
- Determines state of device
  - `command-ready`
  - `busy`
  - `Error`

- Busy-wait cycle to wait for I/O from device
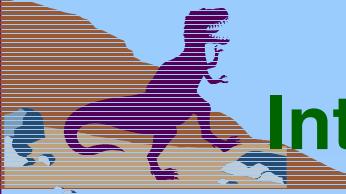
Interrupts

- CPU Interrupt request line triggered by I/O device

- Interrupt handler receives interrupts

- Maskable to ignore or delay some interrupts

- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some unmaskable

- Interrupt mechanism also used for exceptions
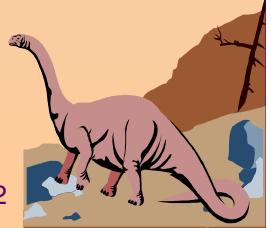
Interrupt-Driven I/O Cycle
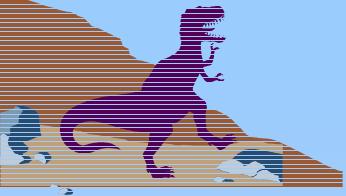
Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19Ð31	(Intel reserved, do not use)
32Ð255	maskable interrupts
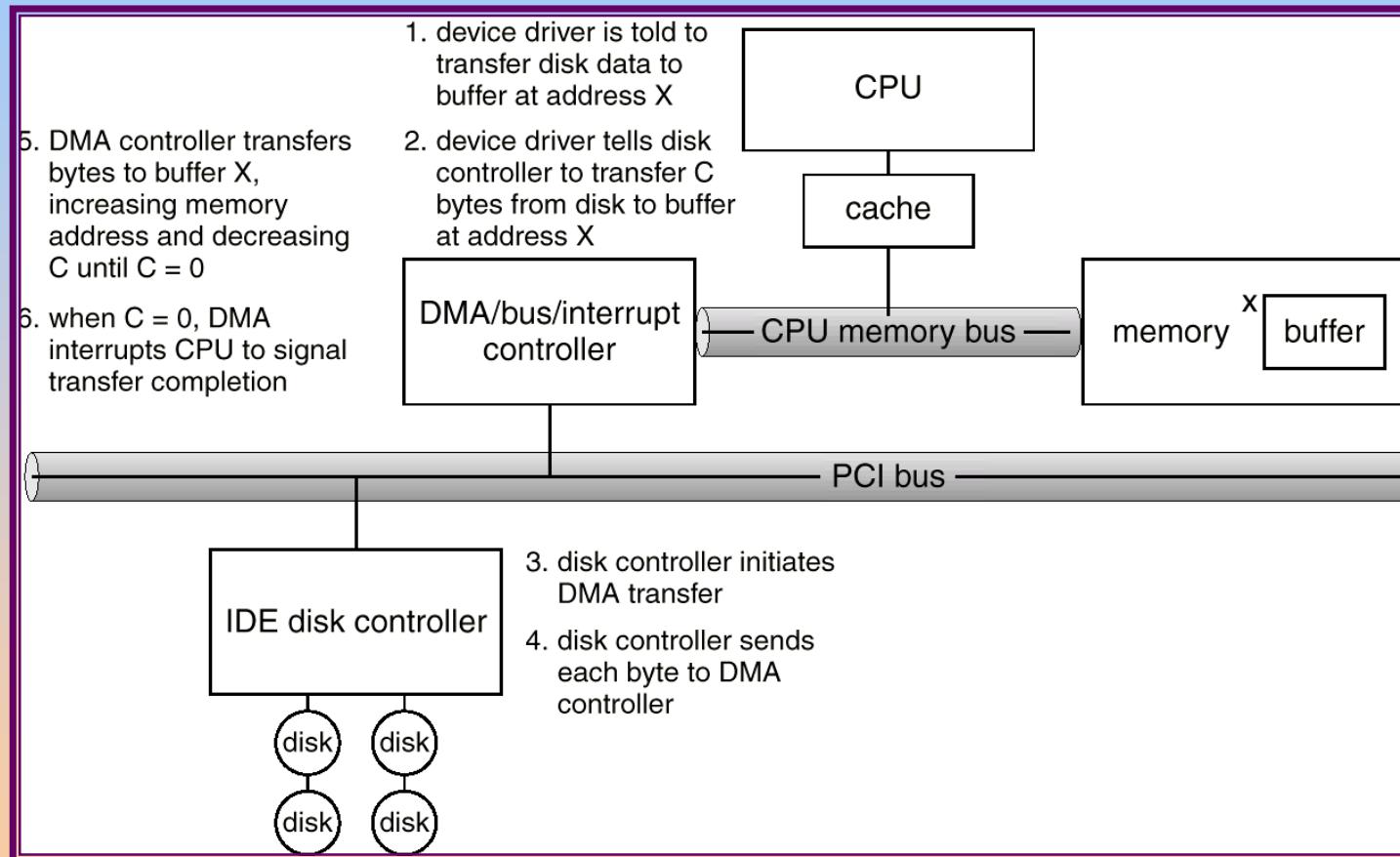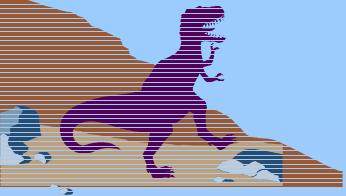
Direct Memory Access

- Used to avoid programmed I/O for large data movement

- Requires DMA controller

- Bypasses CPU to transfer data directly between I/O device and memory

Six Step Process to Perform DMA Transfer

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

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

CPU

cache

DMA/bus/interrupt controller

CPU memory bus

memory

X buffer

PCI bus

IDE disk controller

3. disk controller initiates DMA transfer

4. disk controller sends each byte to DMA controller

disk disk

disk disk

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes

- Device-driver layer hides differences among I/O controllers from kernel

- Devices vary in many dimensions
  - Character-stream or block
  - Sequential or random-access
  - Sharable or dedicated
  - Speed of operation
  - read-write, read only, or write only

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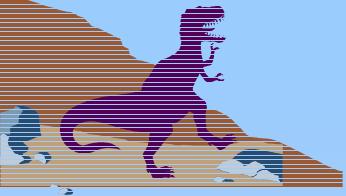
Characteristics of I/O Devices
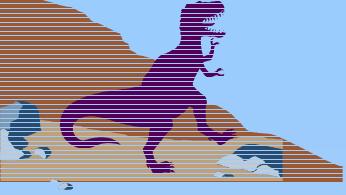
aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction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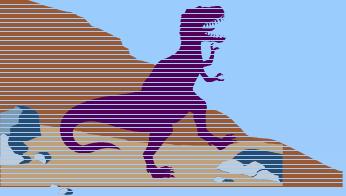
Block and Character Devices

- 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
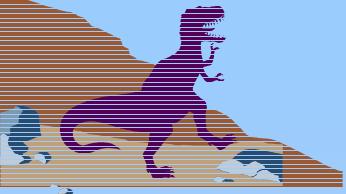
Network Devices

- Varying enough from block and character to have own interface

- Unix and Windows NT/9*i*/2000 include socket interface
  - ✦ Separates network protocol from network operation
  - ✦ Includes `select` functionality

- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)
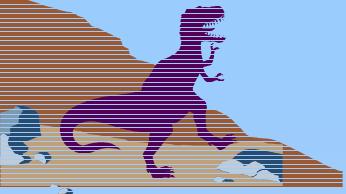
Clocks and Timers

- Provide current time, elapsed time, timer

- 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)
  - ✦ 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
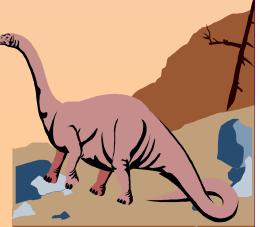
Kernel I/O Subsystem
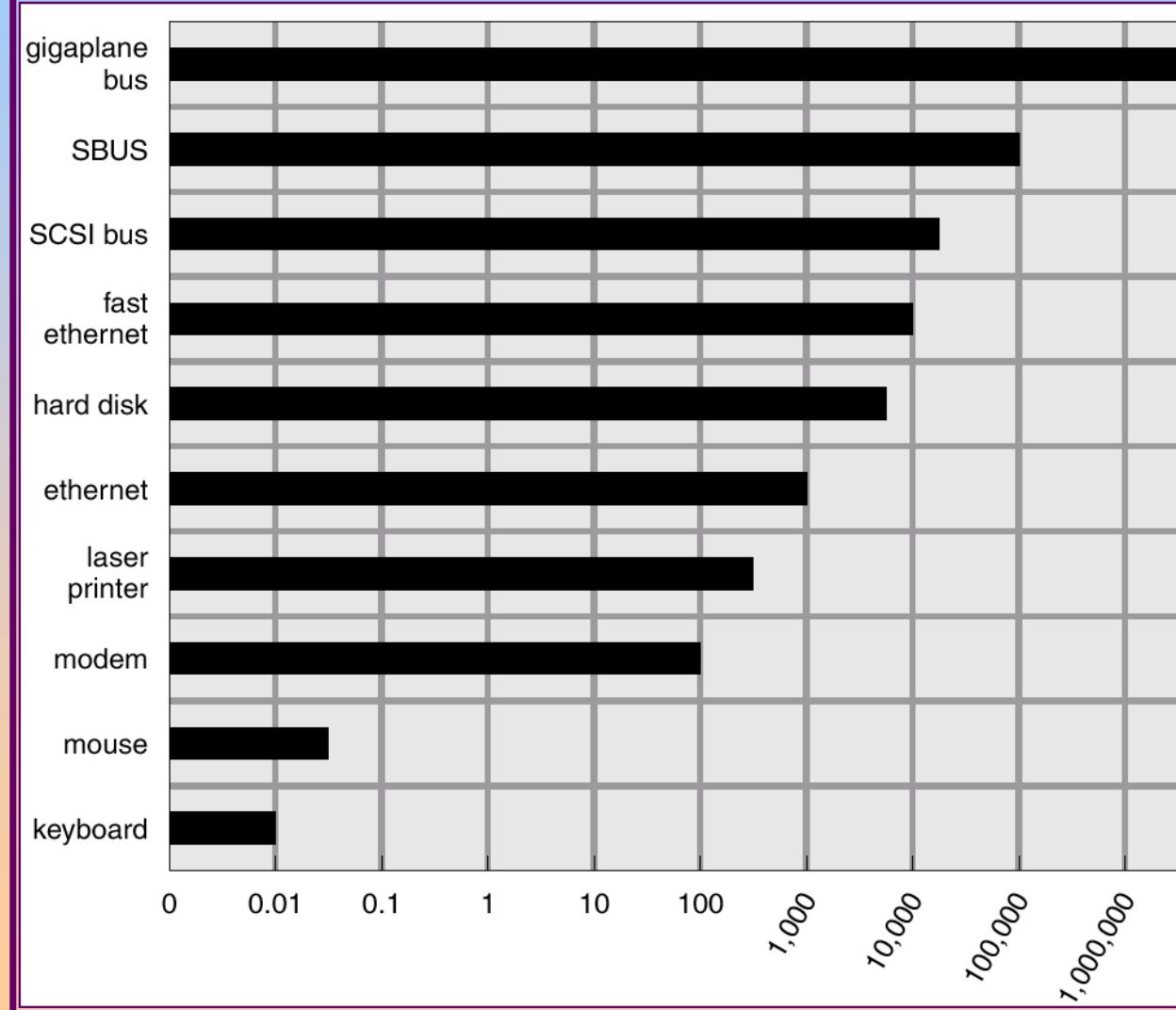
- Scheduling
    - Some I/O request ordering via per-device queue
    - Some OSs try fairness
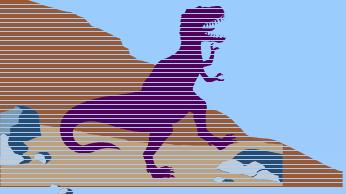
- Buffering - store data in memory while transferring between devices
    - To cope with device speed mismatch
    - To cope with device transfer size mismatch
    - To maintain "copy semantics"
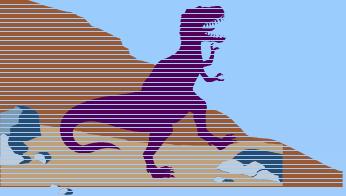
Sun Enterprise 6000 Device-Transfer Rates
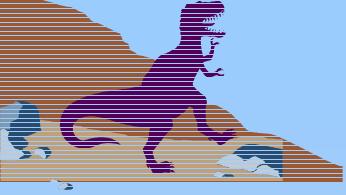
Kernel I/O Subsystem

- Caching - fast memory holding copy of data
  - Always just a copy
  - Key to performance

- Spooling - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing

- Device reservation - provides exclusive access to a device
  - System calls for allocation and deallocation
  - Watch out for deadlock
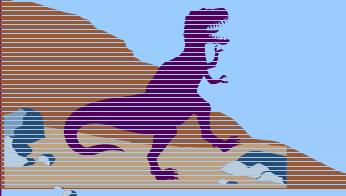
Error Handling

- OS can recover from disk read, device unavailable, transient write failures

- Most return an error number or code when I/O request fails

- System error logs hold problem reports
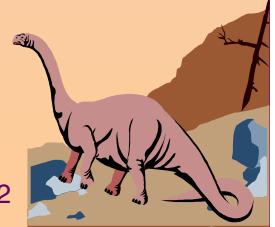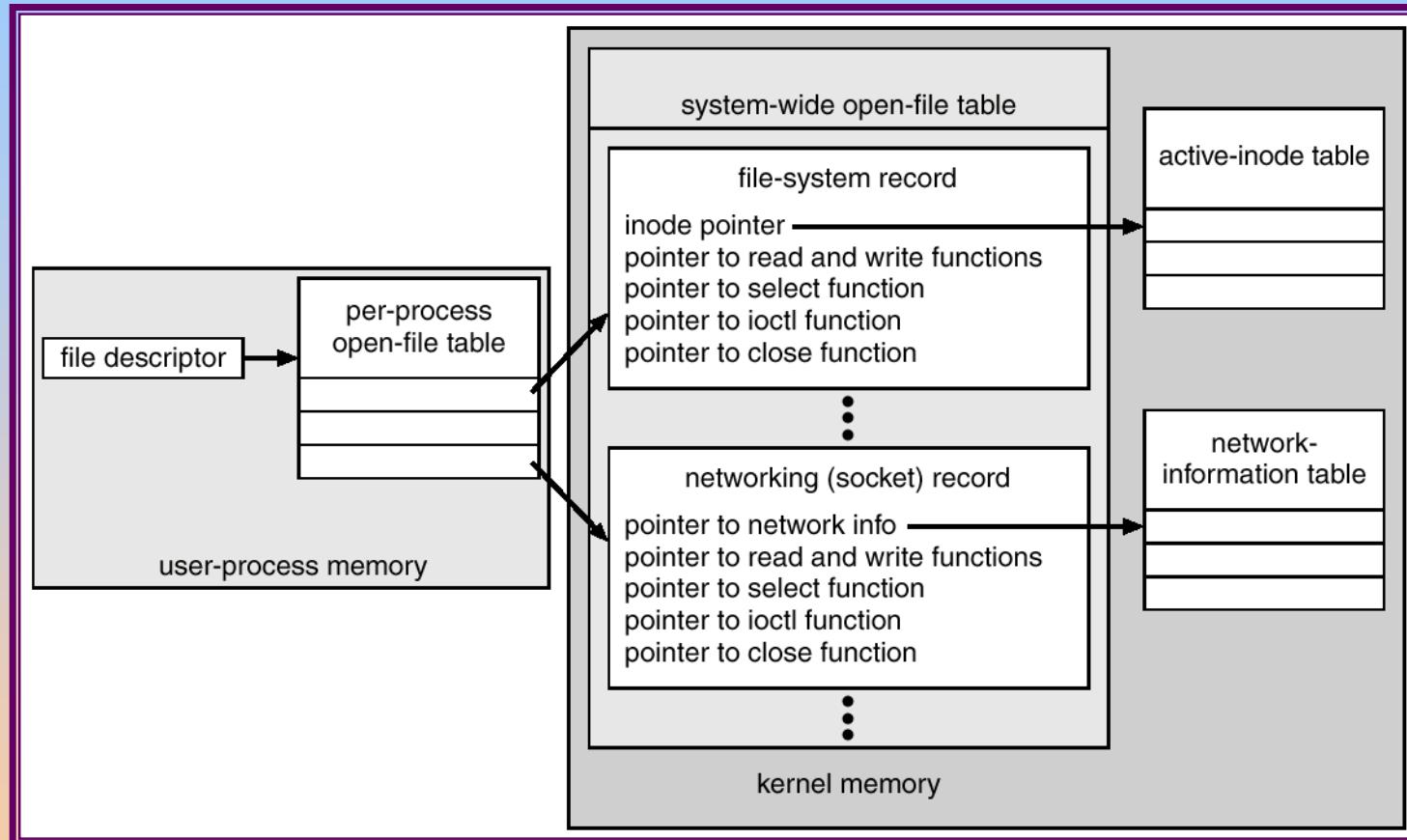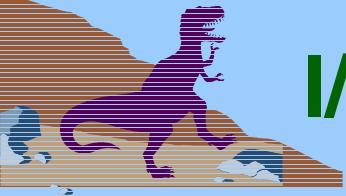
Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state

- Many, many complex data structures to track buffers, memory allocation, "dirty" blocks

- Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure
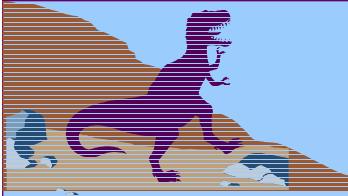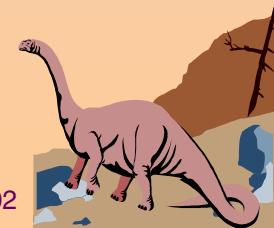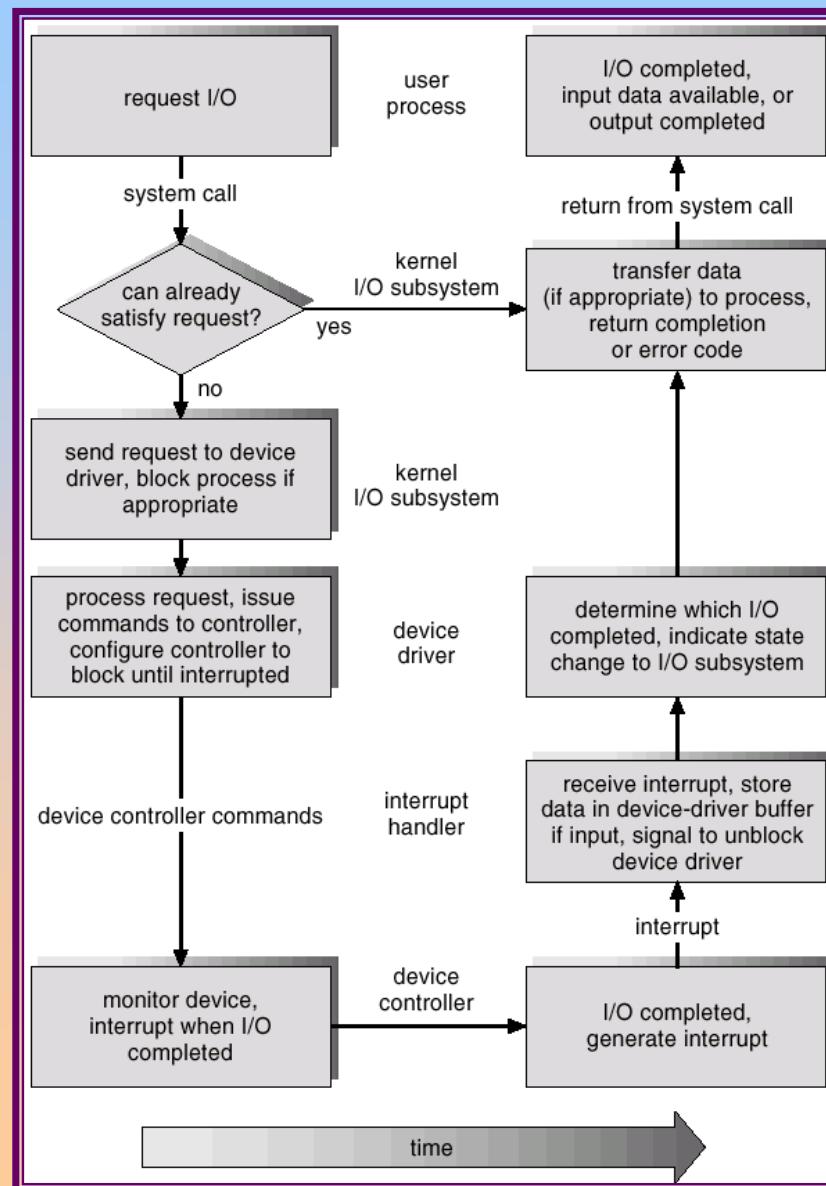
I/O Requests to Hardware Operations
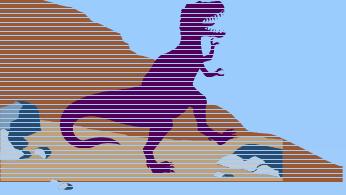
- Consider reading a file from disk for a process:

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

Life Cycle of An I/O Request
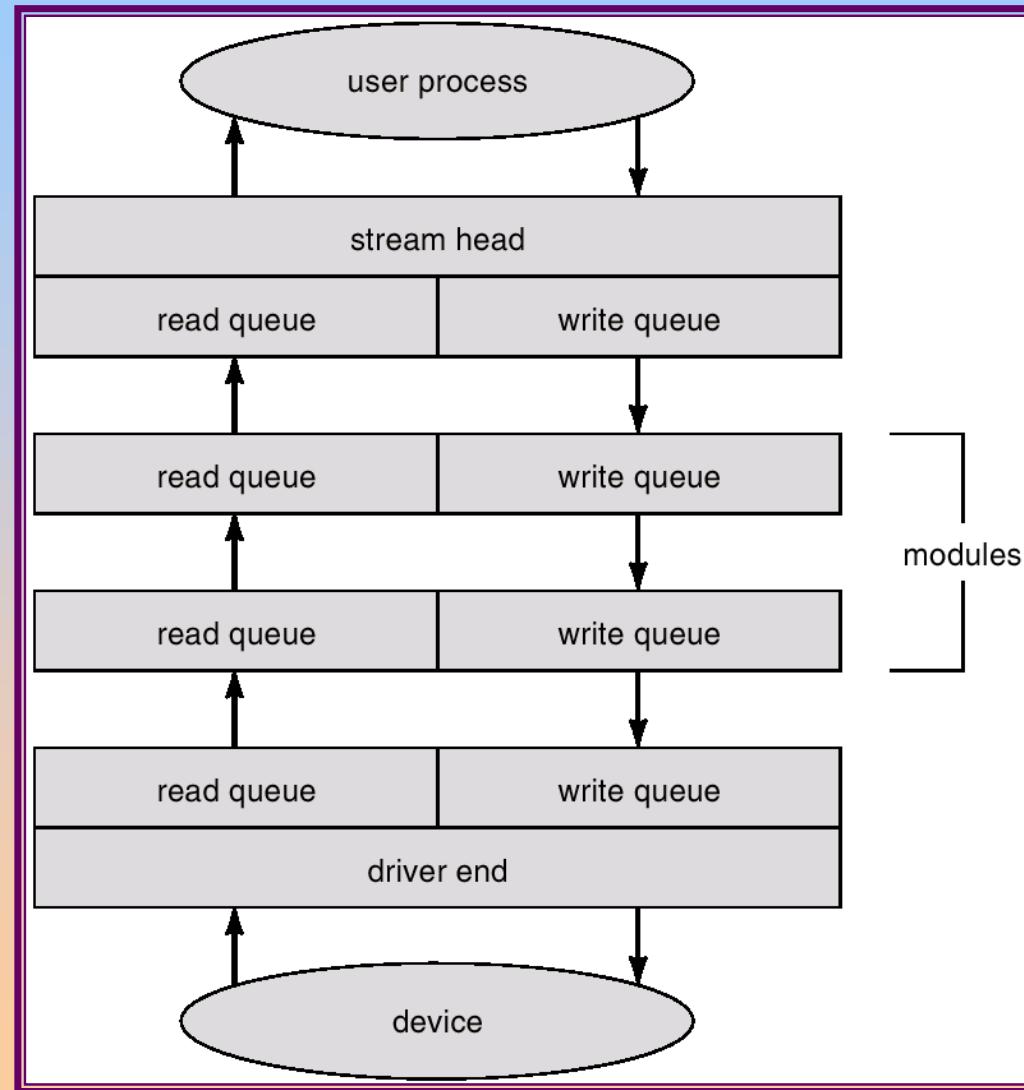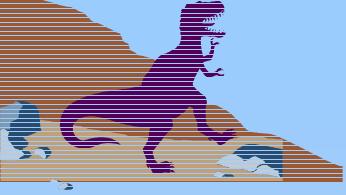
STREAMS

- **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

13.26

The STREAMS Structure
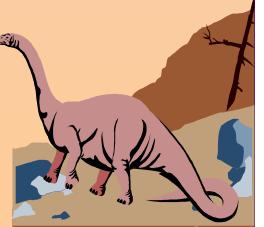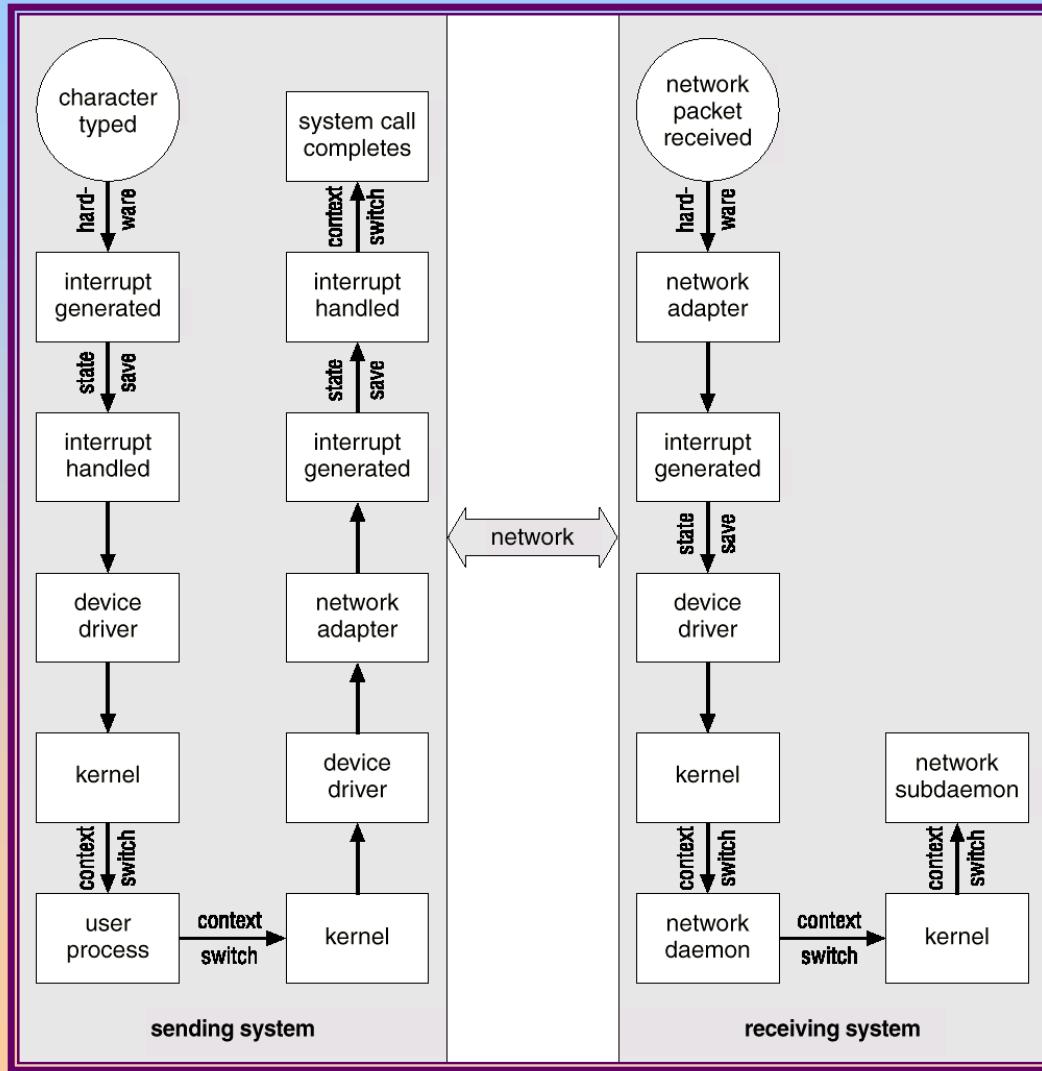
Performance
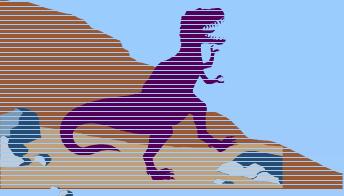
- I/O a major factor in system performance:

  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful
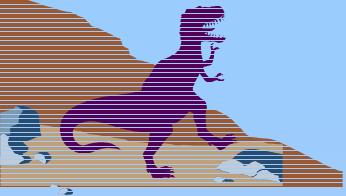
Intercomputer Communications
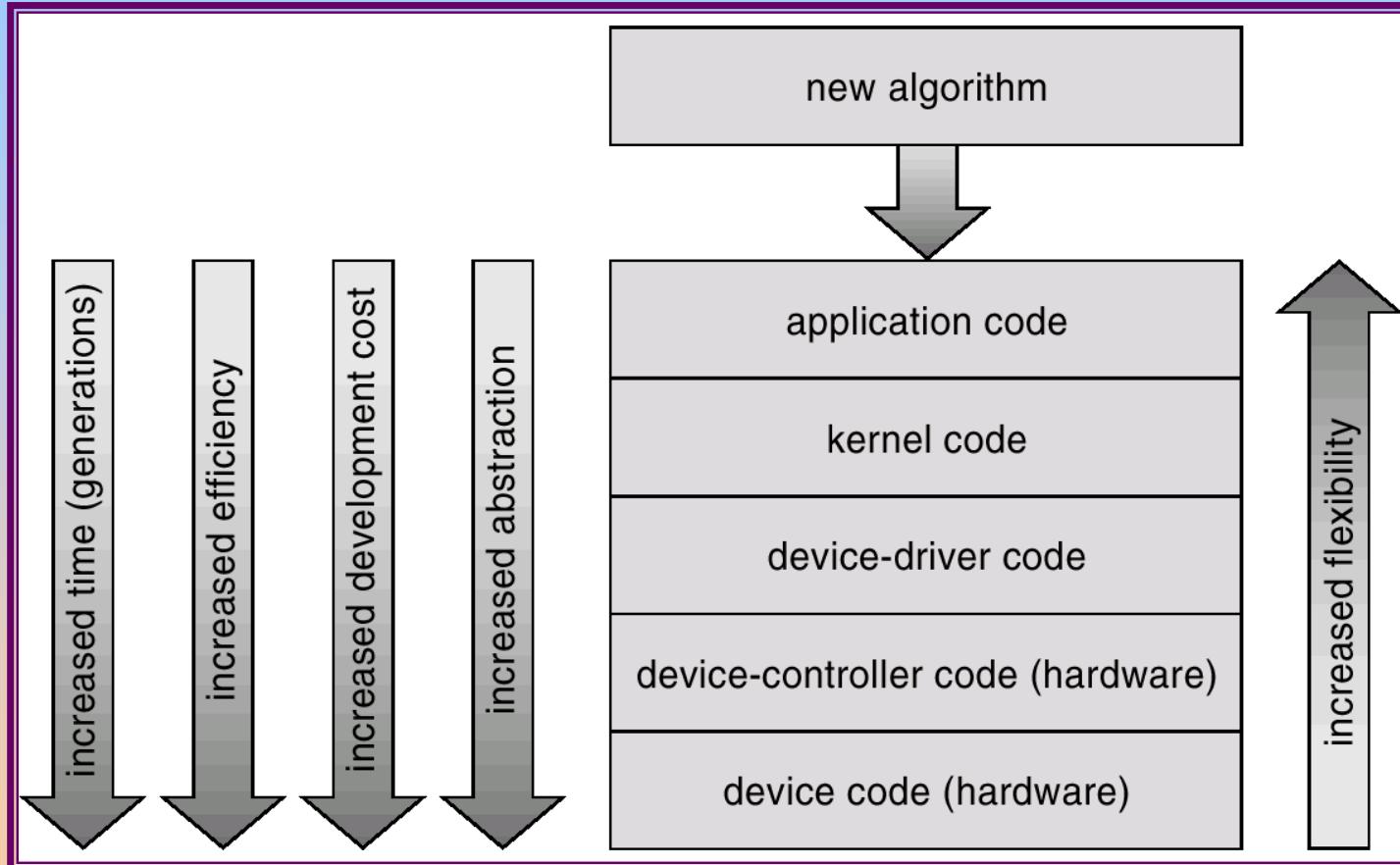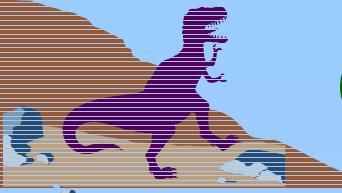
Improving Performance

- Reduce number of context switches

- Reduce data copying

- Reduce interrupts by using large transfers, smart controllers, polling

- Use DMA

- Balance CPU, memory, bus, and I/O performance for highest throughput

Device-Functionality Progression
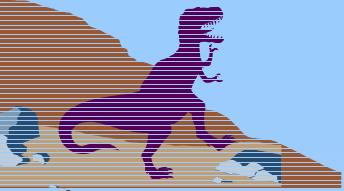
Chapter 14: Mass-Storage Systems
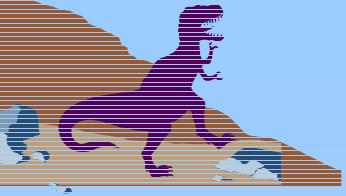
- Disk Structure
- Disk Scheduling
- Disk Management
- Swap-Space Management
- RAID Structure
- Disk Attachment
- Stable-Storage Implementation
- Tertiary Storage Devices
- Operating System Issues
- Performance Issues
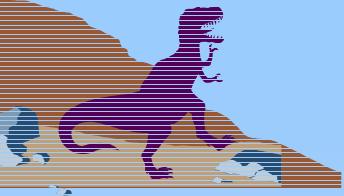
Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.

- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
  - ☞ Sector 0 is the first sector of the first track on the outermost cylinder.
  - ☞ Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

14.2

Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.

- Access time has two major components

  - ☞ *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.

  - ☞ *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.

- Minimize seek time

- Seek time $\approx$ seek distance

- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
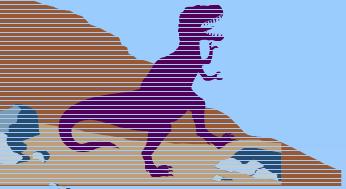
Disk Scheduling (Cont.)

- Several algorithms exist to schedule the servicing of disk I/O requests.

- We illustrate them with a request queue (0-199).
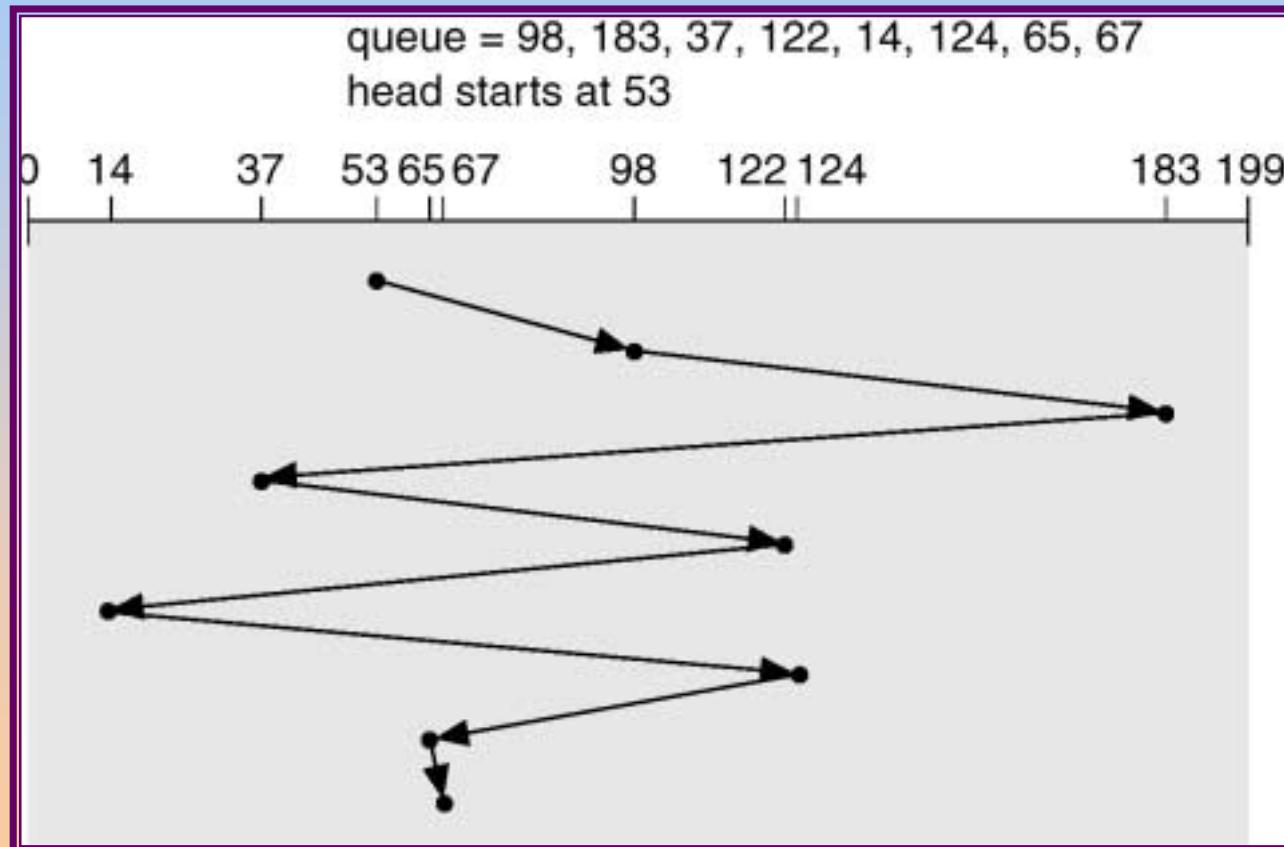
  98, 183, 37, 122, 14, 124, 65, 67
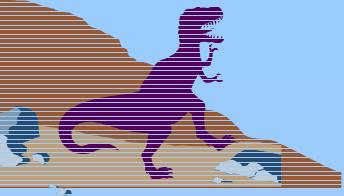
  Head pointer 53

FCFS
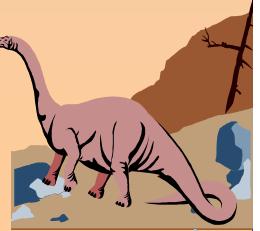
Illustration shows total head movement of 640 cylinders.

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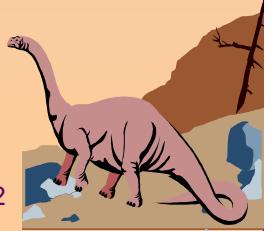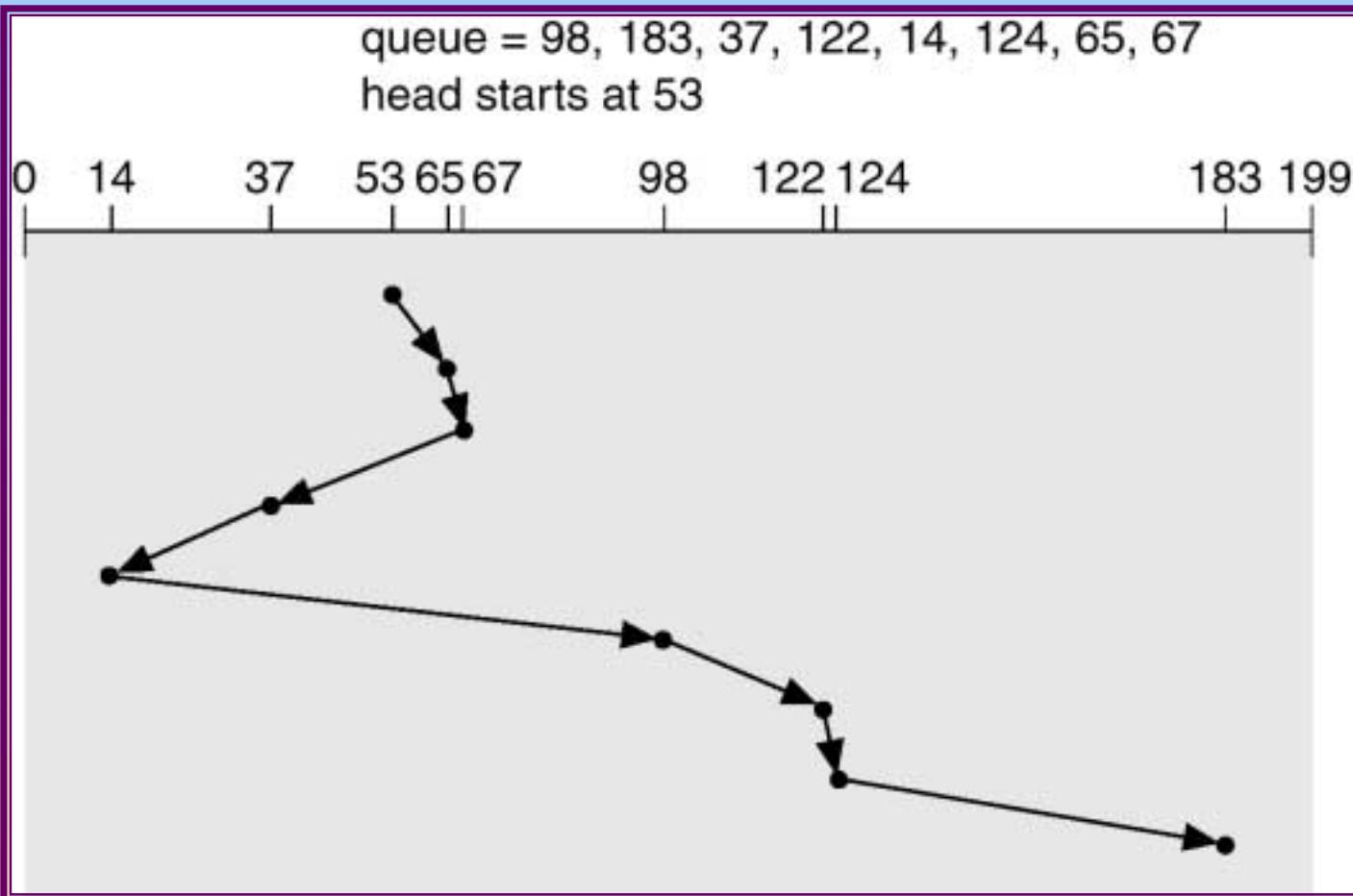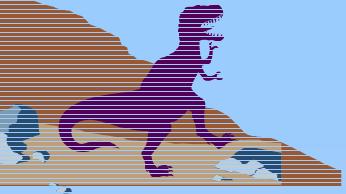
SSTF

- Selects the request with the minimum seek time from the current head position.

- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.

- Illustration shows total head movement of 236 cylinders.

SSTF (Cont.)

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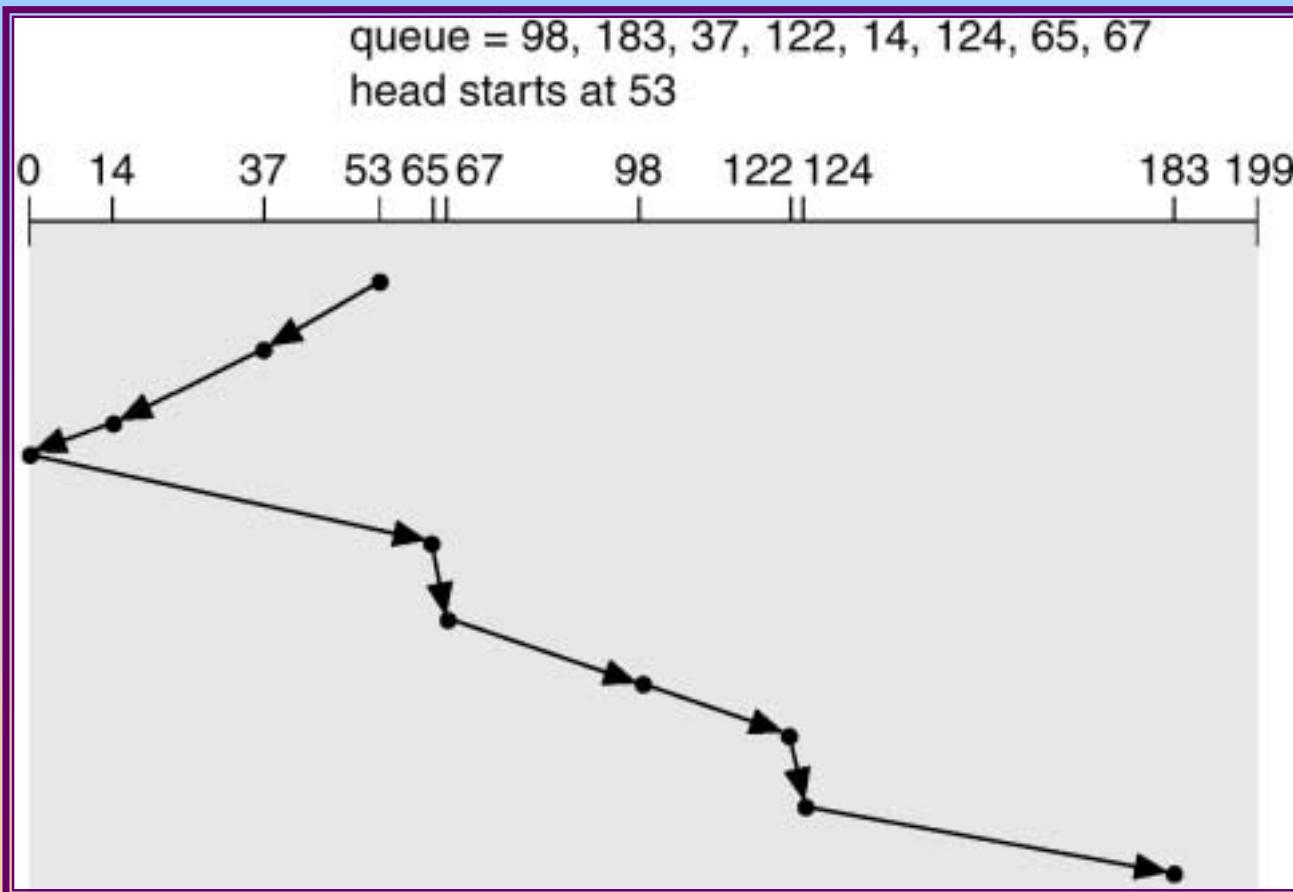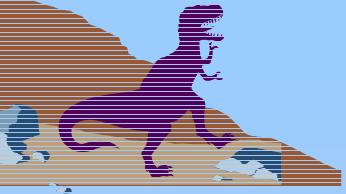
SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

- Sometimes called the *elevator algorithm*.

- Illustration shows total head movement of 208 cylinders.

SCAN (Cont.)

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

0   14   37   53 65 67   98   122 124   183 199
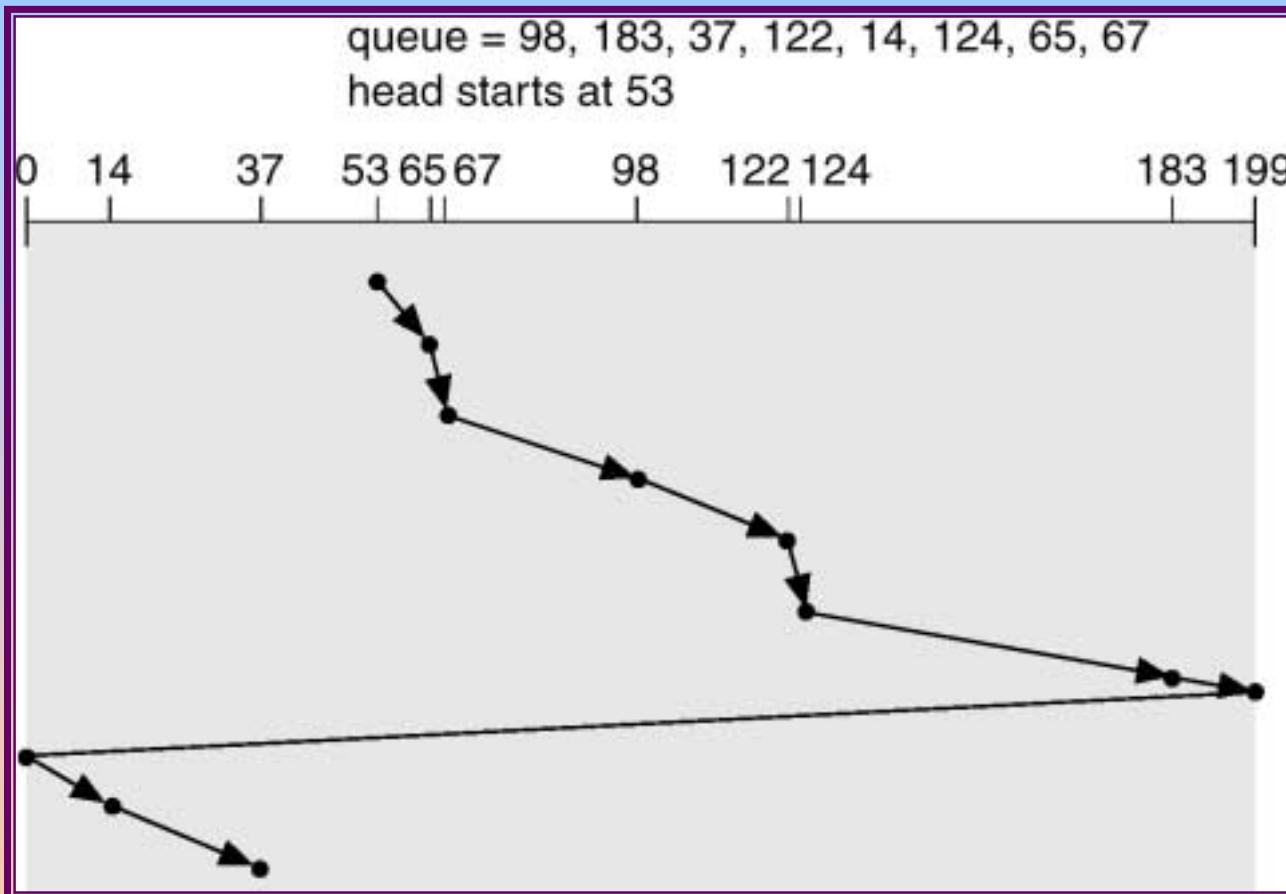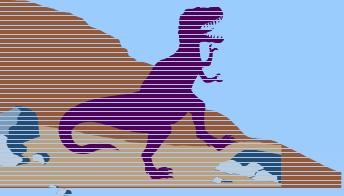
C-SCAN

- 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

0   14   37   53 65 67   98   122 124   183 199
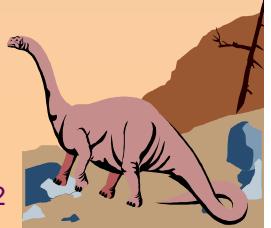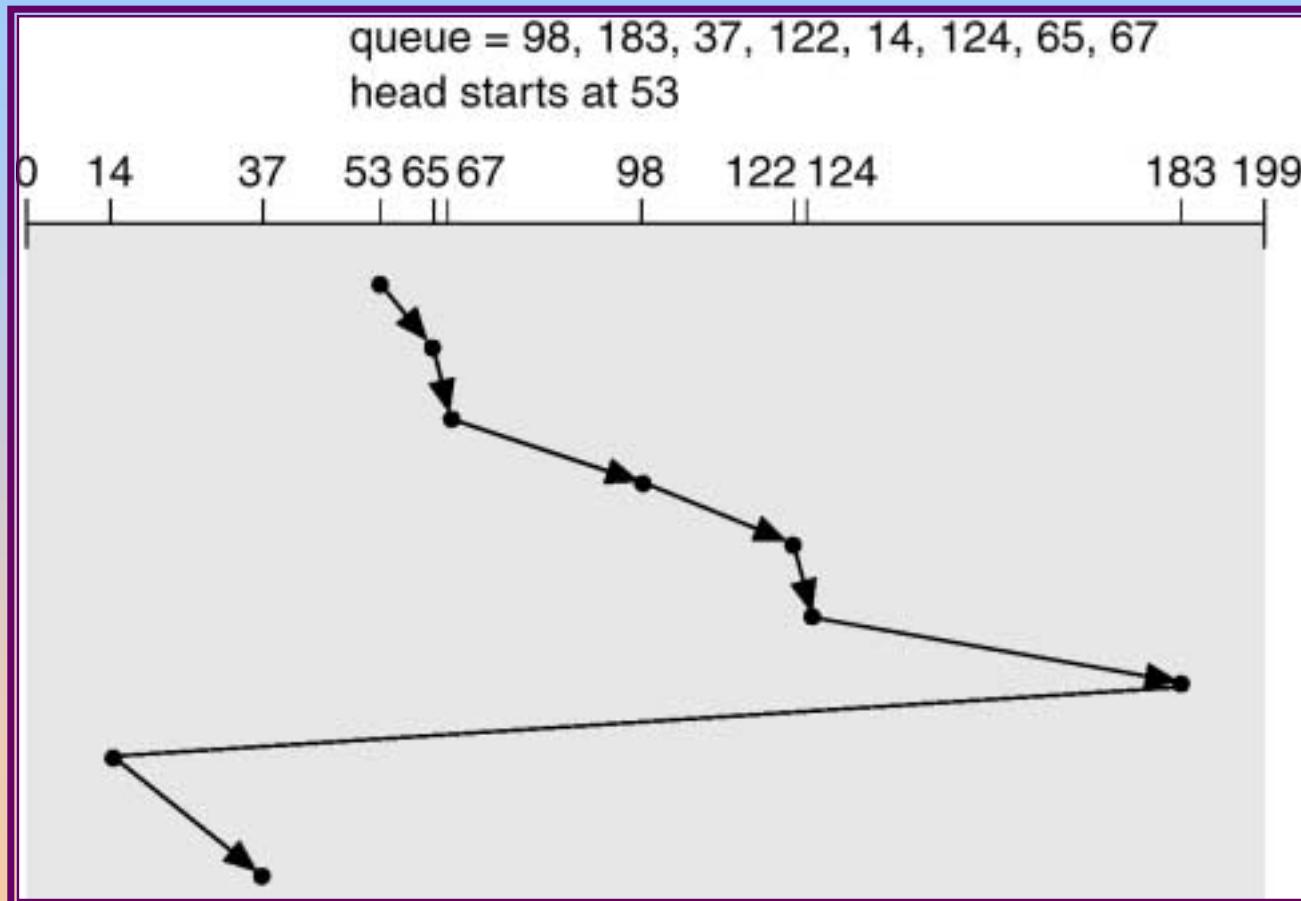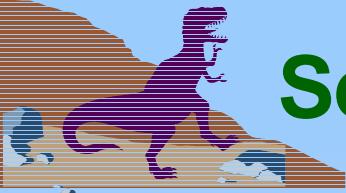
C-LOOK

- Version of 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.

C-LOOK (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
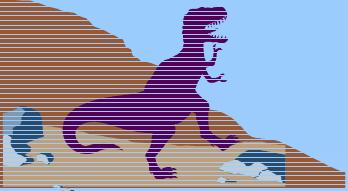
Selecting a Disk-Scheduling Algorithm
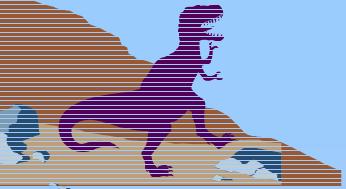
- SSTF is common and has a natural appeal

- 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.
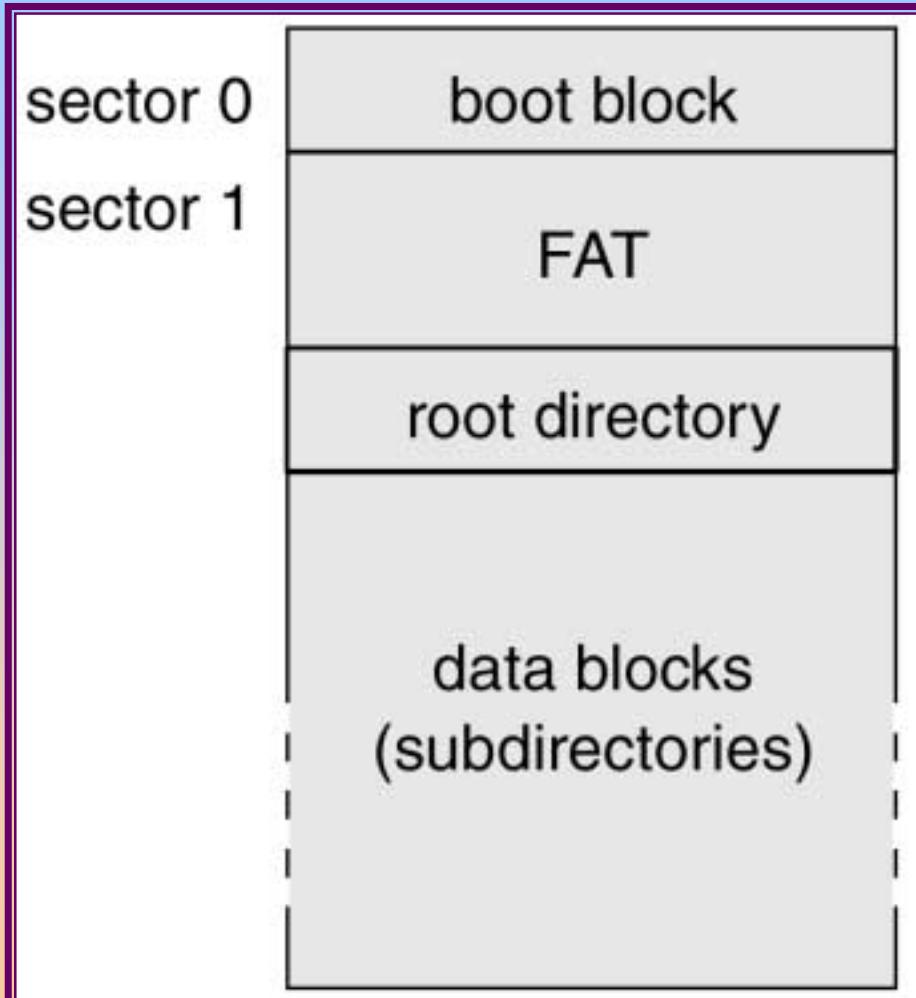
Disk Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.

- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
  - ☞ *Partition* the disk into one or more groups of cylinders.
  - ☞ *Logical formatting* or "making a file system".

- Boot block initializes system.
  - ☞ The bootstrap is stored in ROM.
  - ☞ *Bootstrap loader* program.

- Methods such as *sector sparing* used to handle bad blocks.
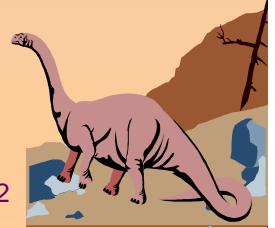
MS-DOS Disk Layout
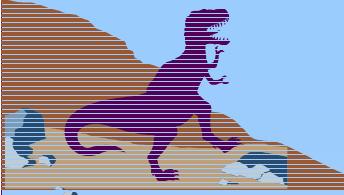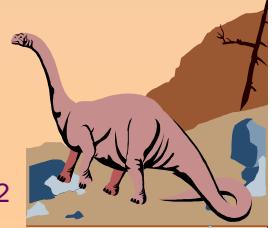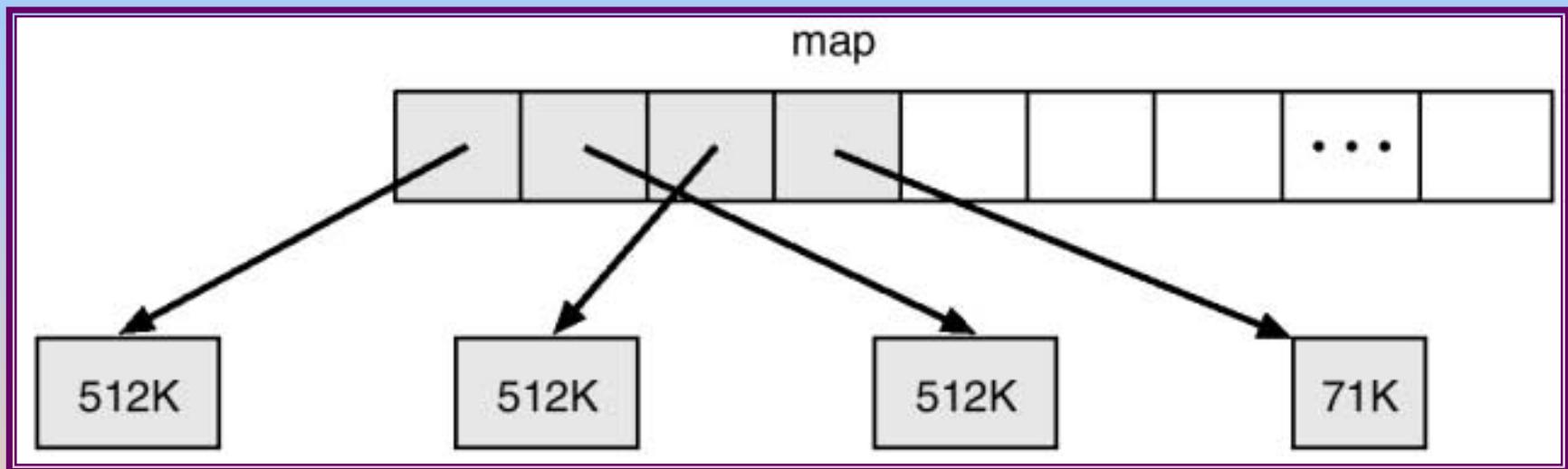
Swap-Space Management
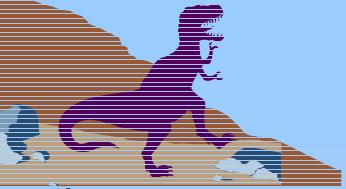
- Swap-space — Virtual memory uses disk space as an extension of main memory.

- Swap-space can be carved out of the normal file system,or, more commonly, it can be in a separate disk partition.

- Swap-space management

  - ☞ 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment.*

  - ☞ Kernel uses *swap maps* to track swap-space use.

  - ☞ Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.
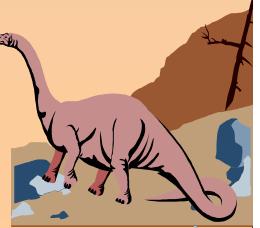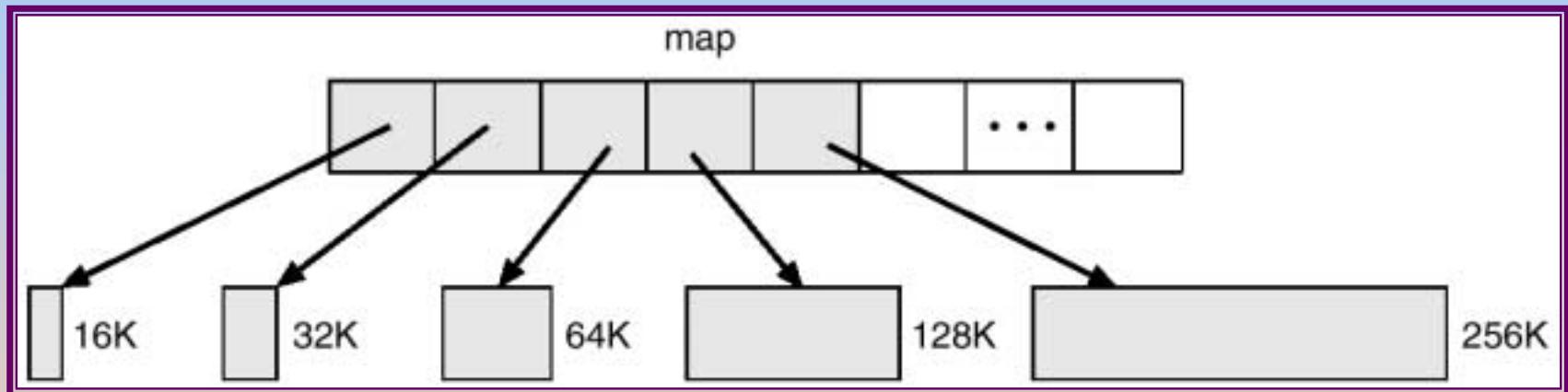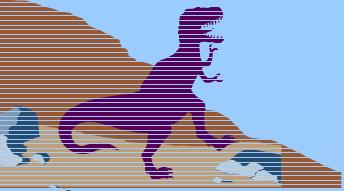
4.3 BSD Text-Segment Swap Map
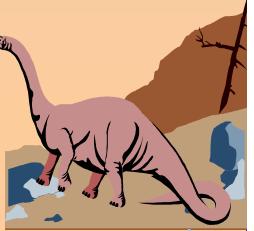
4.3 BSD Data-Segment Swap Map
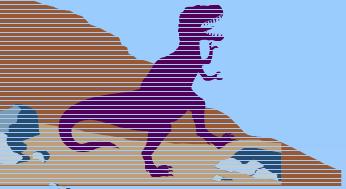
RAID Structure

- **RAID** – multiple disk drives provides **reliability** via **redundancy**.

- RAID is arranged into six different levels.
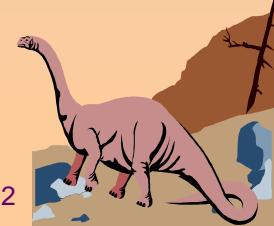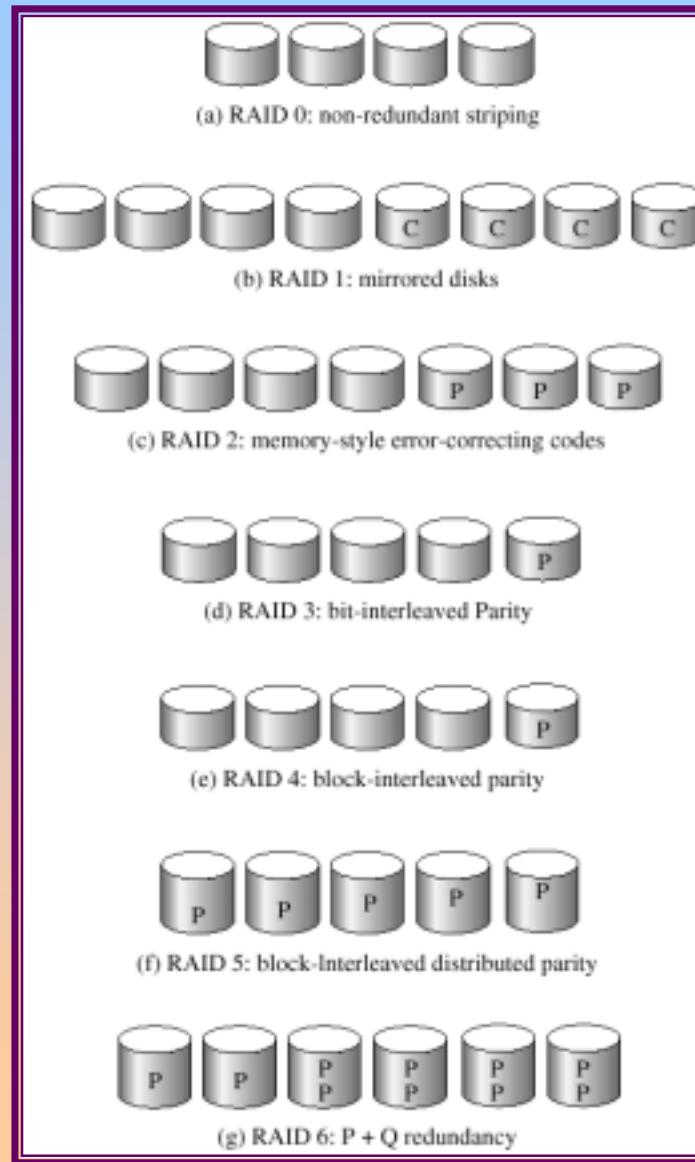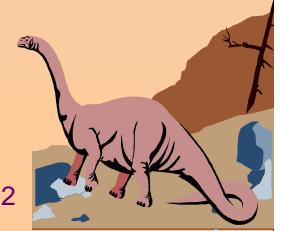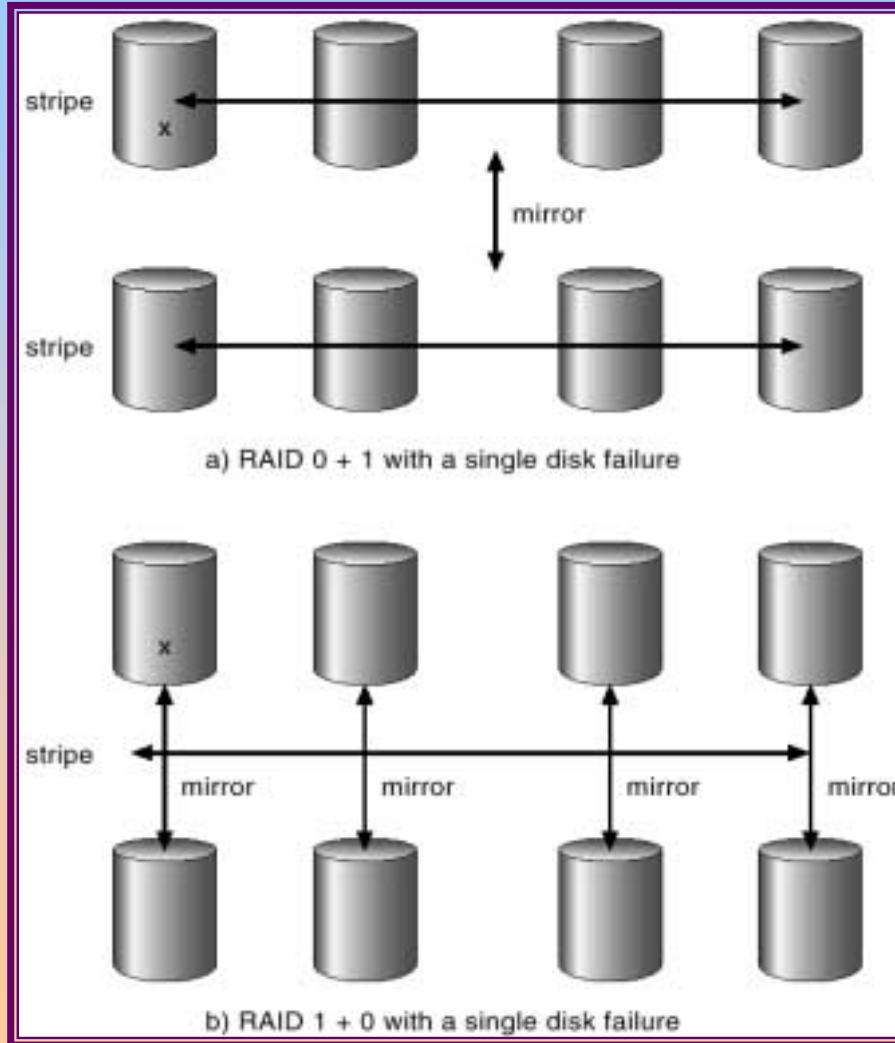
RAID (cont)
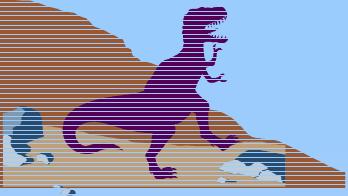
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.

- Disk striping uses a group of disks as one storage unit.

- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
  - *Mirroring* or *shadowing* keeps duplicate of each disk.
  - *Block interleaved parity* uses much less redundancy.

RAID Levels

(a) RAID 0: non-redundant striping

(b) RAID 1: mirrored disks

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

(d) RAID 3: bit-interleaved Parity

(e) RAID 4: block-interleaved parity

(f) RAID 5: block-Interleaved distributed parity

(g) RAID 6: P + Q redundancy

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

a) RAID 0 + 1 with a single disk failure

b) RAID 1 + 0 with a single disk failure
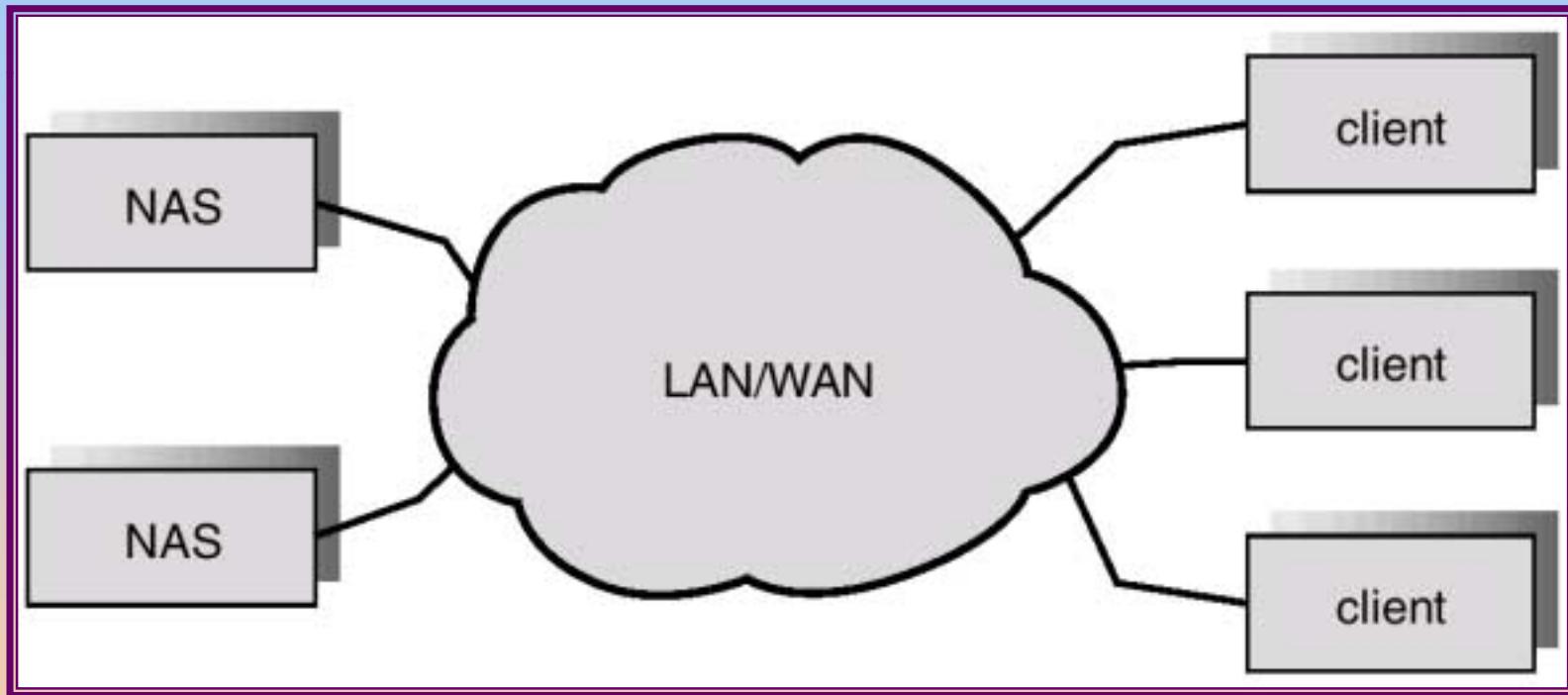
Disk Attachment
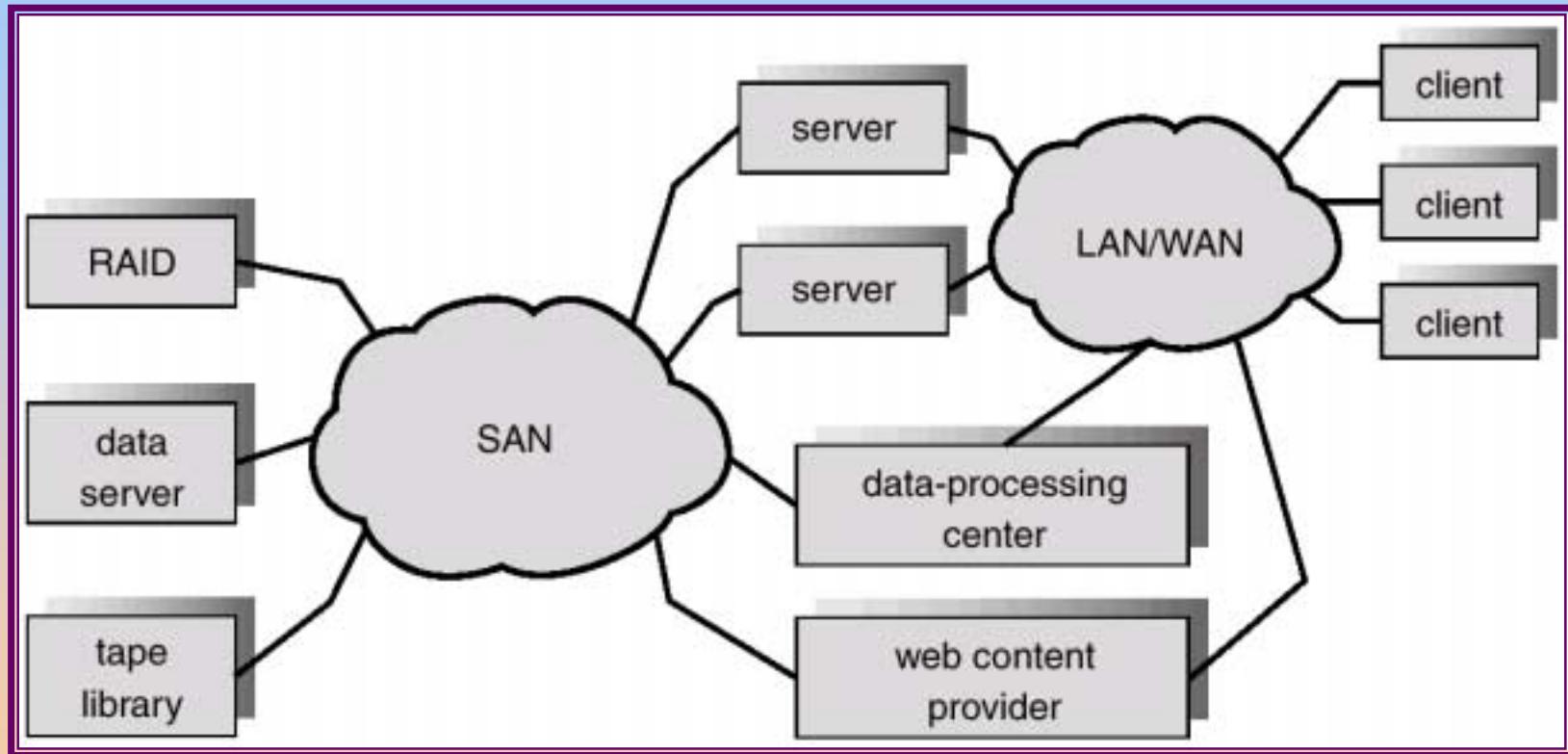
- Disks may be attached one of two ways:

1. **Host attached** via an I/O port

2. **Network attached** via a network connection
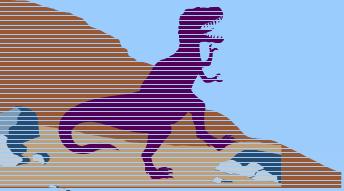
Network-Attached Storage
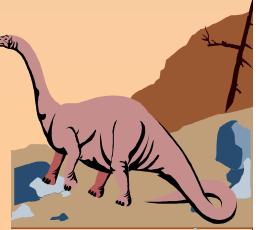
Storage-Area Network
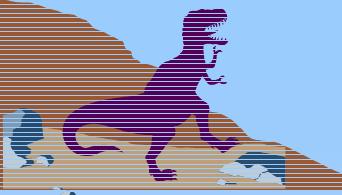
Stable-Storage Implementation
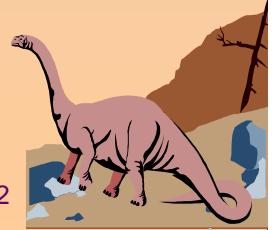
- Write-ahead log scheme requires stable storage.

- To implement stable storage:
  - ☞ Replicate information on more than one nonvolatile storage media with independent failure modes.
  - ☞ Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.
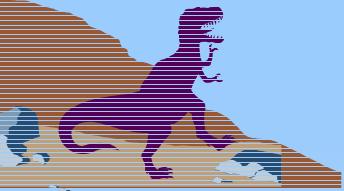
Tertiary Storage Devices
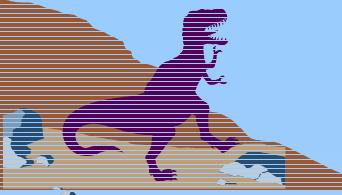
- Low cost is the defining characteristic of tertiary storage.

- Generally, tertiary storage is built using *removable media*

- Common examples of removable media are floppy disks and CD-ROMs; other types are available.

Removable Disks
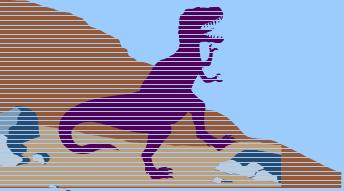
■ Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.

☞ Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.

☞ Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.

Removable Disks (Cont.)
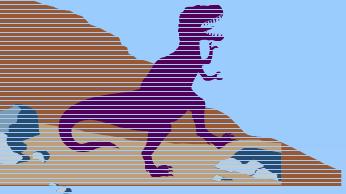
- A magneto-optic disk records data on a rigid platter coated with magnetic material.
  - ☞ Laser heat is used to amplify a large, weak magnetic field to record a bit.
  - ☞ Laser light is also used to read data (Kerr effect).
  - ☞ The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.

- Optical disks do not use magnetism; they employ special materials that are altered by laser light.

WORM Disks

- The data on read-write disks can be modified over and over.

- WORM ("Write Once, Read Many Times") disks can be written only once.

- Thin aluminum film sandwiched between two glass or plastic platters.

- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered.

- Very durable and reliable.

- *Read Only* disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded.

Tapes
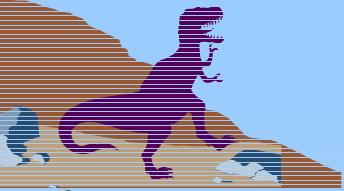
- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.

- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.

- Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
  - ☞ stacker – library that holds a few tapes
  - ☞ silo – library that holds thousands of tapes

- A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.

Operating System Issues
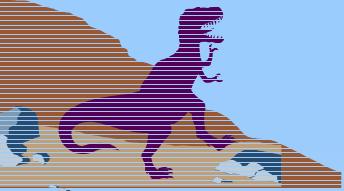
- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications

- For hard disks, the OS provides two abstraction:
  - ☞ Raw device – an array of data blocks.
  - ☞ File system – the OS queues and schedules the interleaved requests from several applications.
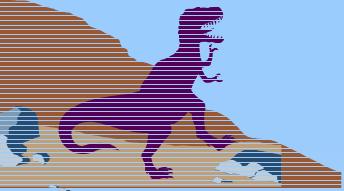
Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk.

- Tapes are presented as a raw storage medium, i.e., and application does not not open a file on the tape, it opens the whole tape drive as a raw device.

- Usually the tape drive is reserved for the exclusive use of that application.

- Since the OS does not provide file system services, the application must decide how to use the array of blocks.

- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it.
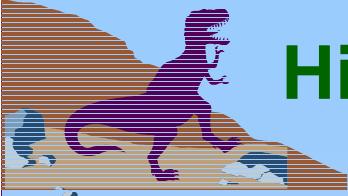
Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.

- **locate** positions the tape to a specific logical block, not an entire track (corresponds to **seek**).

- The **read position** operation returns the logical block number where the tape head is.

- The **space** operation enables relative motion.

- Tape drives are "append-only" devices; updating a block in the middle of the tape also effectively erases everything beyond that block.

- An EOT mark is placed after a block that is written.
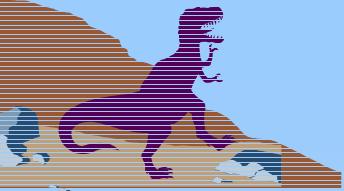
File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.

- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.

- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.
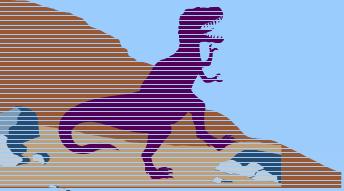
Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.

- Usually incorporate tertiary storage by extending the file system.
  - ☞ Small and frequently used files remain on disk.
  - ☞ Large, old, inactive files are archived to the jukebox.

- HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.
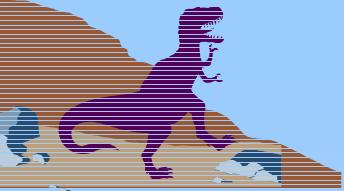
Speed

- Two aspects of speed in tertiary storage are bandwidth and latency.

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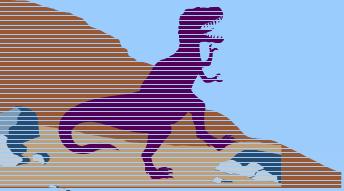
Speed (Cont.)

- **Access latency** – amount of time needed to locate data.

  - ☞ Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds.

  - ☞ Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds.

  - ☞ Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk.

- The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives.

- A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour.

Reliability
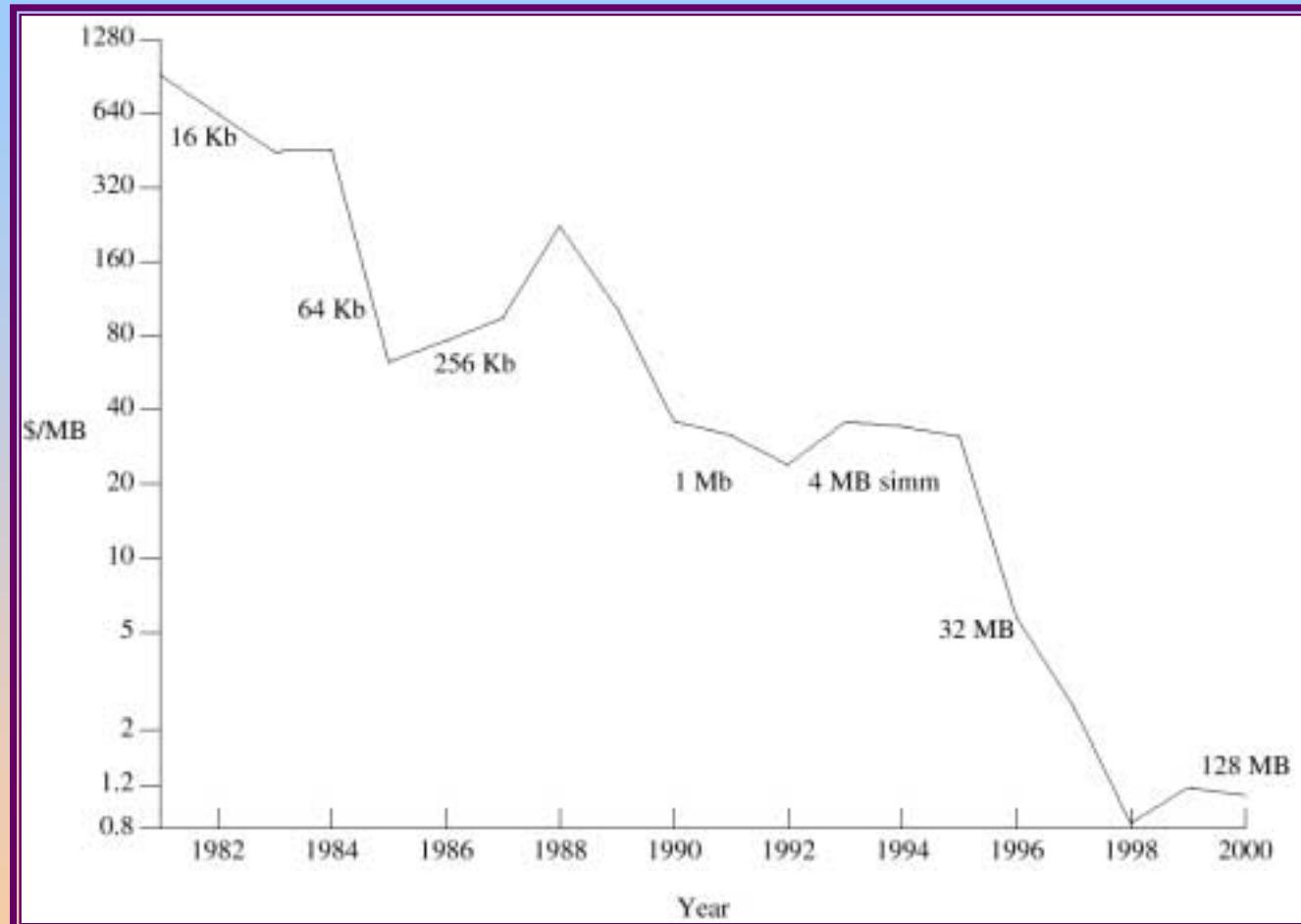
- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.

- An optical cartridge is likely to be more reliable than a magnetic disk or tape.

- A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.
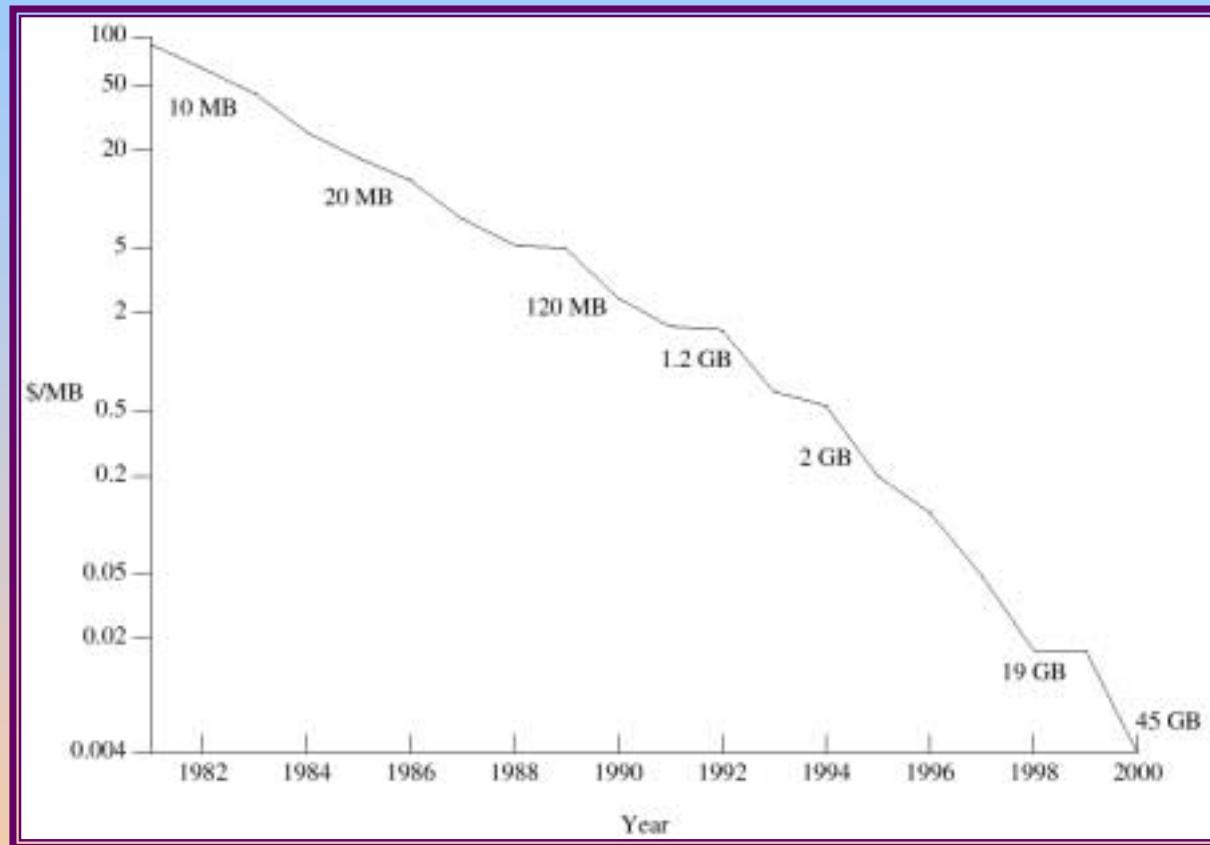
Cost

- Main memory is much more expensive than disk storage

- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive.

- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.

- Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.
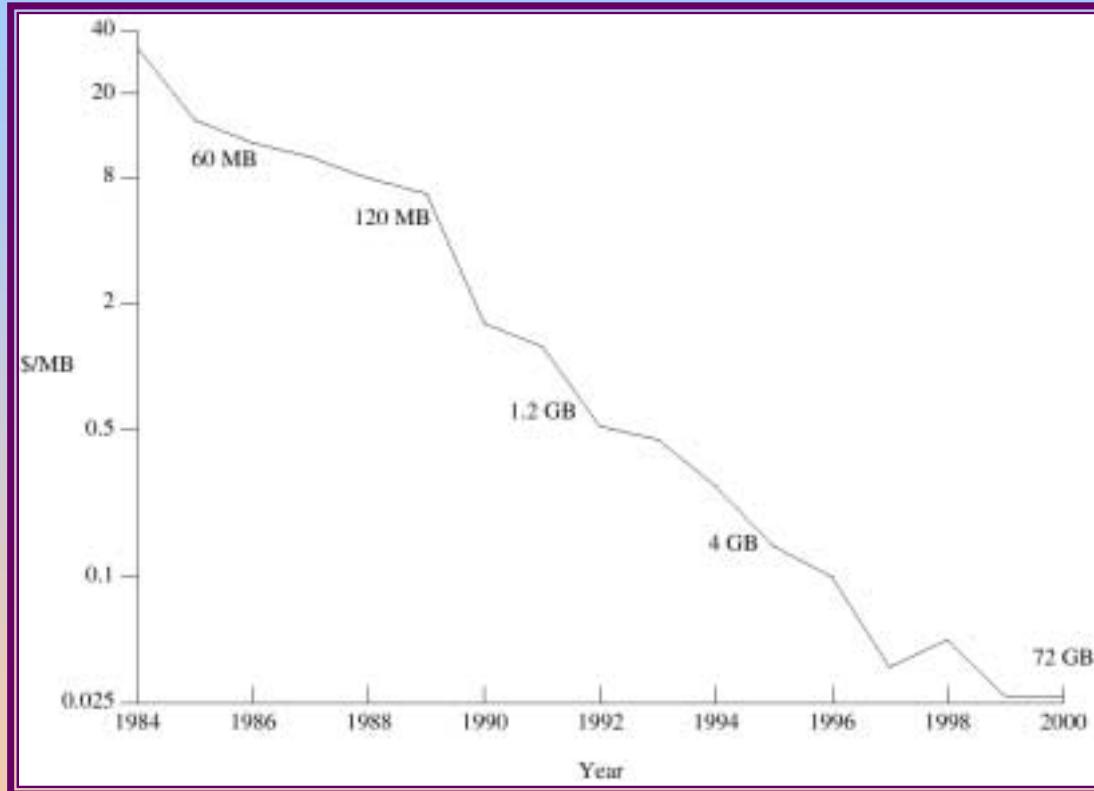
Price per Megabyte of DRAM, From 1981 to 2000
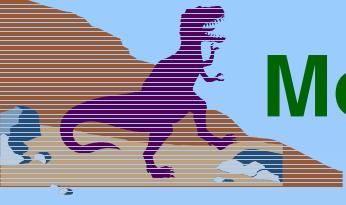
Price per Megabyte of Magnetic Hard Disk, From 1981 to 2000

Price per Megabyte of a Tape Drive, From 1984-2000
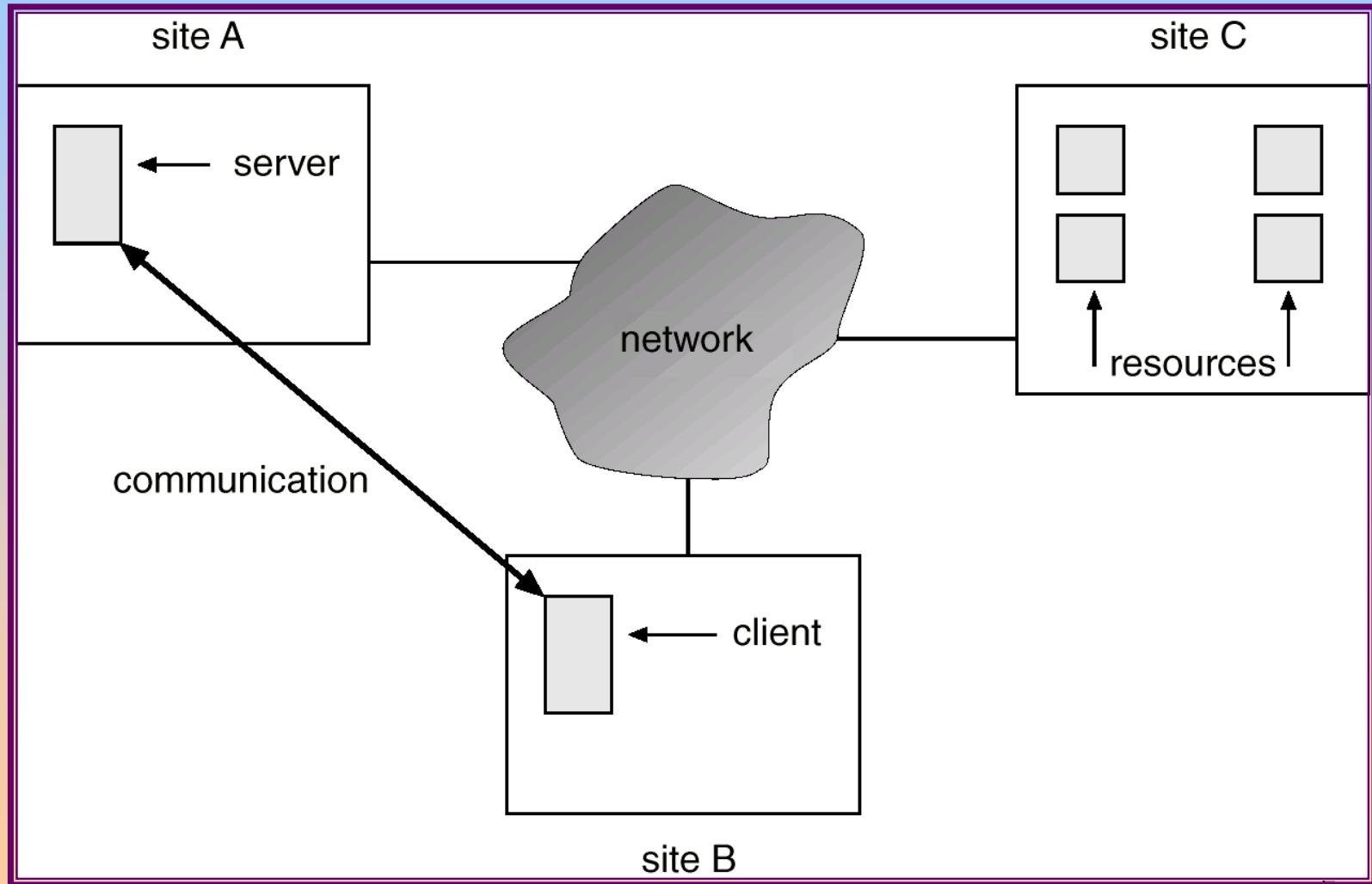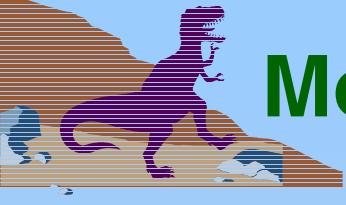
Module 15: Network Structures

- Background
- Topology
- Network Types
- Communication
- Communication Protocol
- Robustness
- Design Strategies
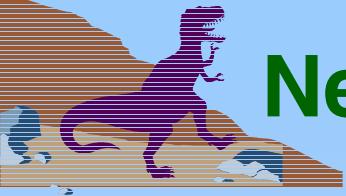
A Distributed System
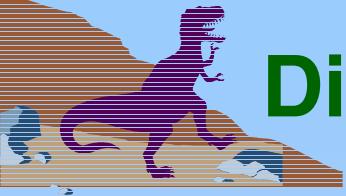
Motivation

- Resource sharing
    - sharing and printing files at remote sites
    - processing information in a distributed database
    - using remote specialized hardware devices
- Computation speedup – *load sharing*
- Reliability – detect and recover from site failure, function transfer, reintegrate failed site
- Communication – message passing

Network-Operating Systems
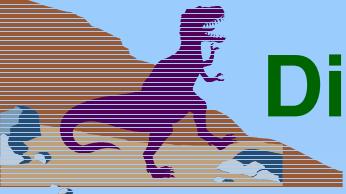
- Users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:
  - Remote logging into the appropriate remote machine.
  - Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism.

Distributed-Operating Systems
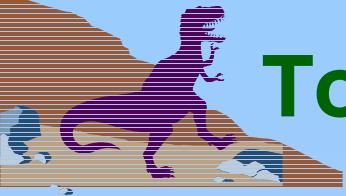
- Users not aware of multiplicity of machines. Access to remote resources similar to access to local resources.

- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task.

- Computation Migration – transfer the computation, rather than the data, across the system.

Distributed-Operating Systems (Cont.)
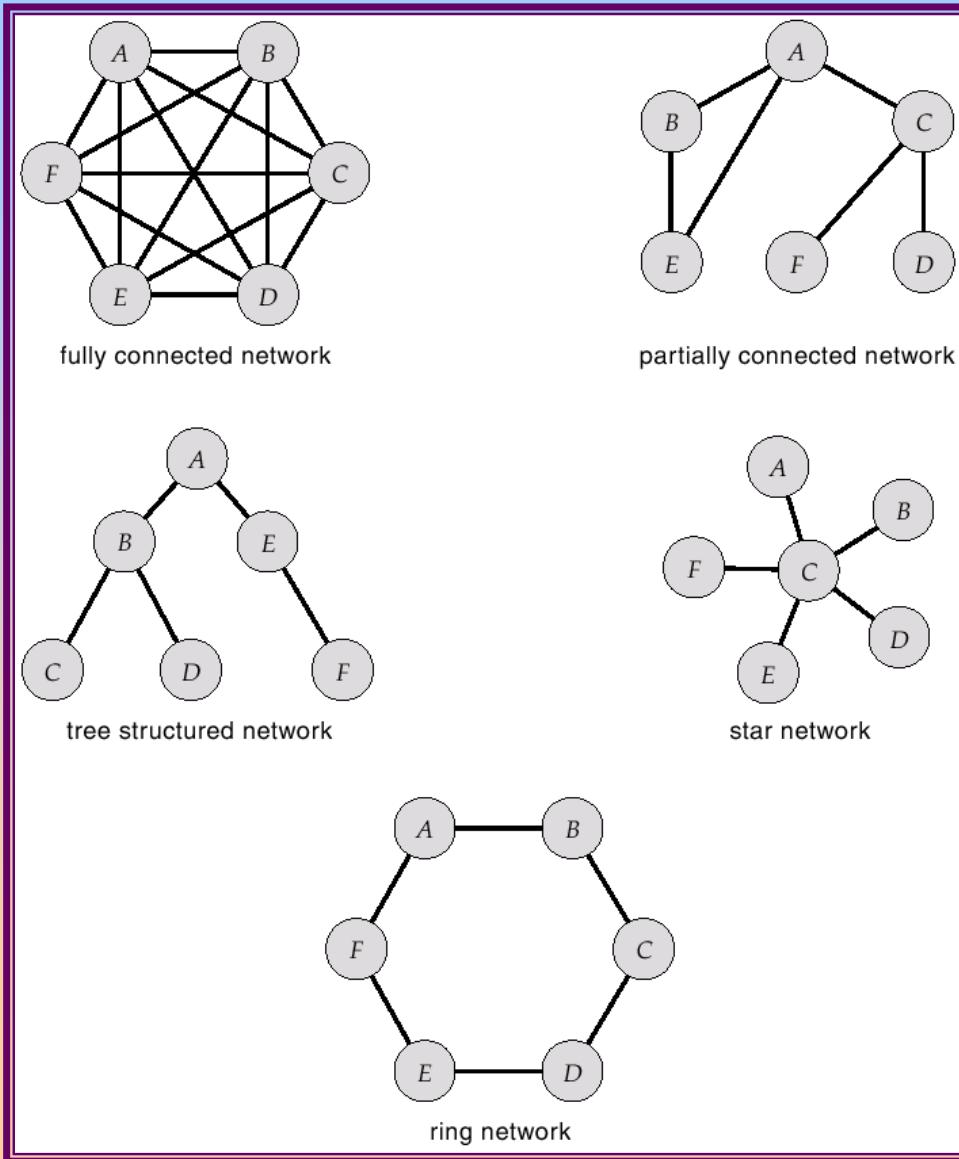
- Process Migration – execute an entire process, or parts of it, at different sites.

  - Load balancing – distribute processes across network to even the workload.

  - Computation speedup – subprocesses can run concurrently on different sites.

  - Hardware preference – process execution may require specialized processor.

  - Software preference – required software may be available at only a particular site.

  - Data access – run process remotely, rather than transfer all data locally.
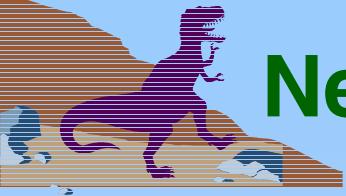
Topology

- Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:
    - **Basic cost**. How expensive is it to link the various sites in the system?
    - **Communication cost**. How long does it take to send a message from site A to site B?
    - **Reliability**. If a link or a site in the system fails, can the remaining sites still communicate with each other?
- The various topologies are depicted as graphs whose nodes correspond to sites. An edge from node A to node B corresponds to a direct connection between the two sites.
- The following six items depict various network topologies.

Network Topology

fully connected network

partially connected network

tree structured network

star network
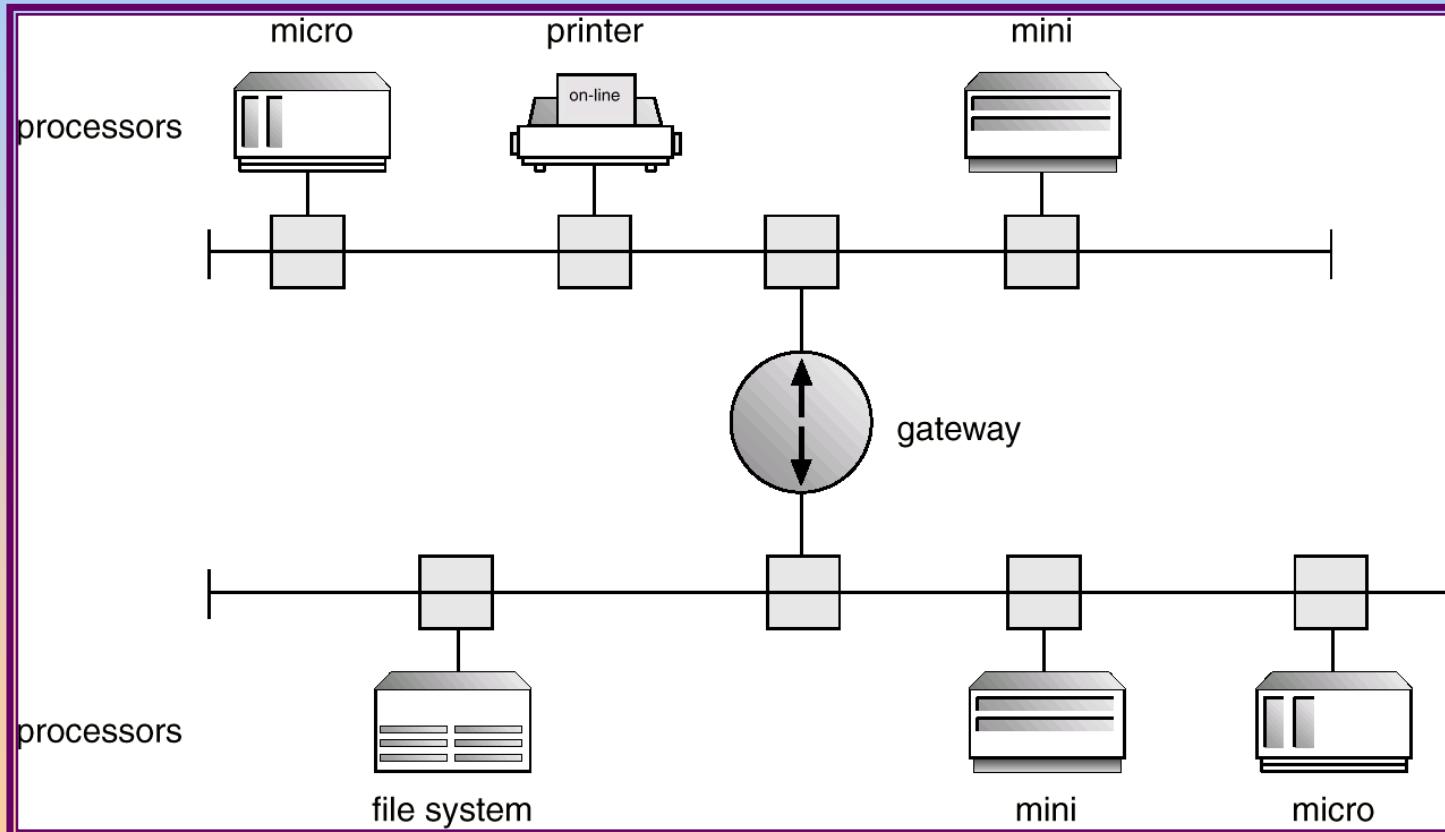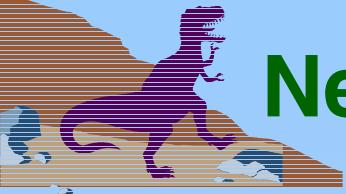
ring network

Network Types

- Local-Area Network (LAN) – designed to cover small geographical area.
    - Multiaccess bus, ring, or star network.
    - Speed ≈ 10 megabits/second, or higher.
    - Broadcast is fast and cheap.
    - Nodes:
        - usually workstations and/or personal computers
        - a few (usually one or two) mainframes.

Network Types (Cont.)

- Depiction of typical LAN:
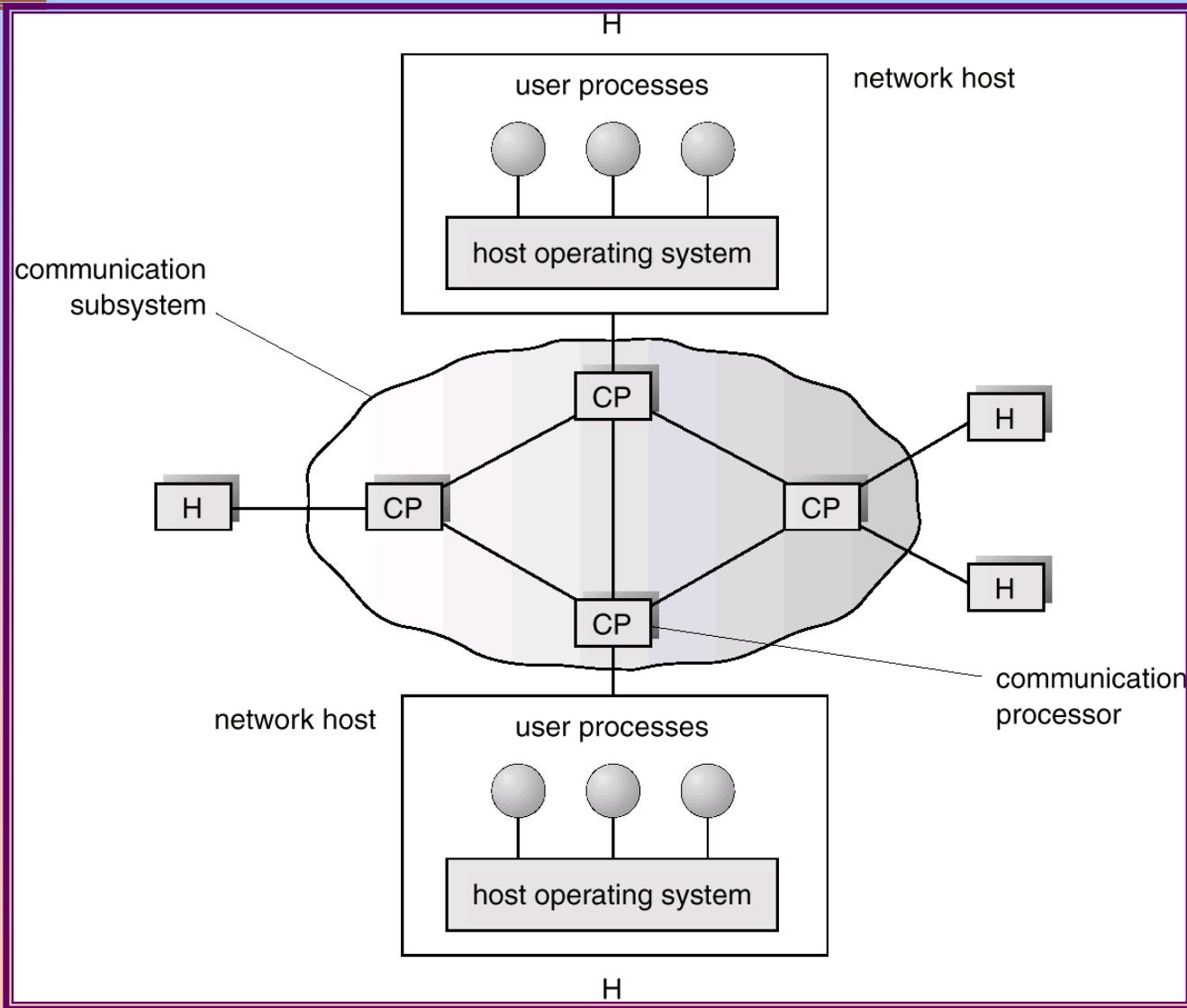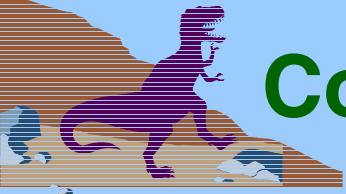
Network Types (Cont.)

- Wide-Area Network (WAN) – links geographically separated sites.
  - ✦ Point-to-point connections over long-haul lines (often leased from a phone company).
  - ✦ Speed $\approx$ 100 kilobits/second.
  - ✦ Broadcast usually requires multiple messages.
  - ✦ Nodes:
    - ✔ usually a high percentage of mainframes

Communication Processors in a Wide-Area Network
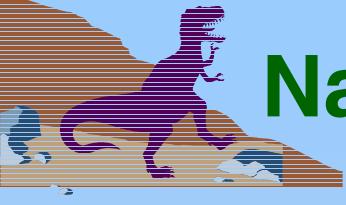
Communication
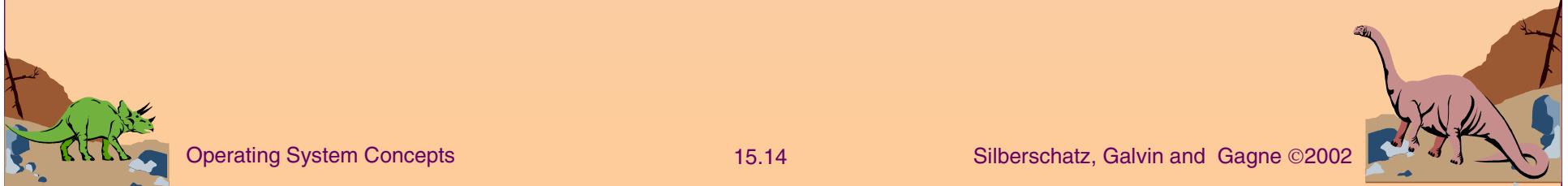
The design of a *communication* network must address four basic issues:
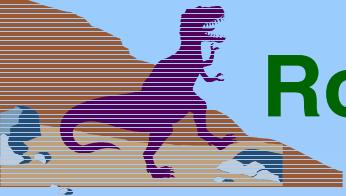
- **Naming and name resolution**: How do two processes locate each other to communicate?

- **Routing strategies**. How are messages sent through the network?

- **Connection strategies**. How do two processes send a sequence of messages?

- **Contention**. The network is a shared resource, so how do we resolve conflicting demands for its use?

Naming and Name Resolution
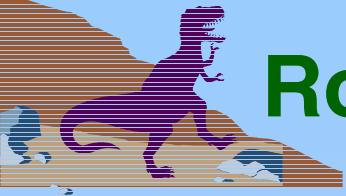
- Name systems in the network
- Address messages with the process-id.
- Identify processes on remote systems by

    <host-name, identifier> pair.

- *Domain name service* (DNS) – specifies the naming structure of the hosts, as well as name to address resolution (Internet).
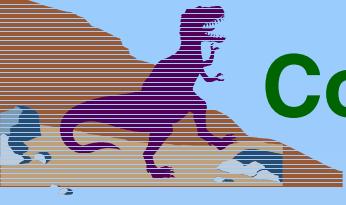
Routing Strategies

- **Fixed routing**. A path from *A* to *B* is specified in advance; path changes only if a hardware failure disables it.
    - ✦ Since the shortest path is usually chosen, communication costs are minimized.
    - ✦ Fixed routing cannot adapt to load changes.
    - ✦ Ensures that messages will be delivered in the order in which they were sent.
- **Virtual circuit**. A path from *A* to *B* is fixed for the duration of one session. Different sessions involving messages from *A* to *B* may have different paths.
    - ✦ Partial remedy to adapting to load changes.
    - ✦ Ensures that messages will be delivered in the order in which they were sent.

15.15
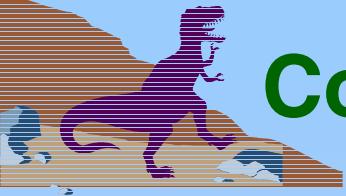
Routing Strategies (Cont.)

- **Dynamic routing**. The path used to send a message form site *A* to site *B* is chosen only when a message is sent.
    - Usually a site sends a message to another site on the link least used at that particular time.
    - Adapts to load changes by avoiding routing messages on heavily used path.
    - Messages may arrive out of order. This problem can be remedied by appending a sequence number to each message.

Connection Strategies
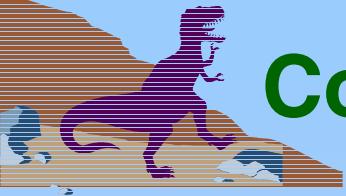
- **Circuit switching**. A permanent physical link is established for the duration of the communication (i.e., telephone system).

- **Message switching**. A temporary link is established for the duration of one message transfer (i.e., post-office mailing system).

- **Packet switching**. Messages of variable length are divided into fixed-length packets which are sent to the destination. Each packet may take a different path through the network. The packets must be reassembled into messages as they arrive.

- Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth. Message and packet switching require less setup time, but incur more overhead per message.

Contention

Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

- CSMA/CD. Carrier sense with multiple access (CSMA); collision detection (CD)
    - A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting.
    - When the system is very busy, many collisions may occur, and thus performance may be degraded.
- SCMA/CD is used successfully in the Ethernet system, the most common network system.

Contention (Cont.)
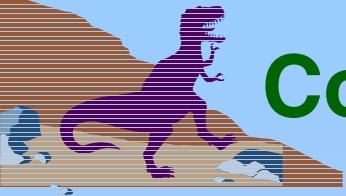
- **Token passing.** A unique message type, known as a token, continuously circulates in the system (usually a ring structure). A site that wants to transmit information must wait until the token arrives. When the site completes its round of message passing, it retransmits the token. A token-passing scheme is used by the IBM and Apollo systems.

- **Message slots.** A number of fixed-length message slots continuously circulate in the system (usually a ring structure). Since a slot can contain only fixed-sized messages, a single logical message may have to be broken down into a number of smaller packets, each of which is sent in a separate slot. This scheme has been adopted in the experimental Cambridge Digital Communication Ring

Communication Protocol
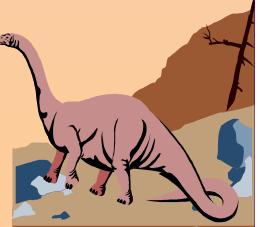
The communication network is partitioned into the following multiple layers;

- Physical layer – handles the mechanical and electrical details of the physical transmission of a bit stream.

- Data-link layer – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.

- Network layer – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels.
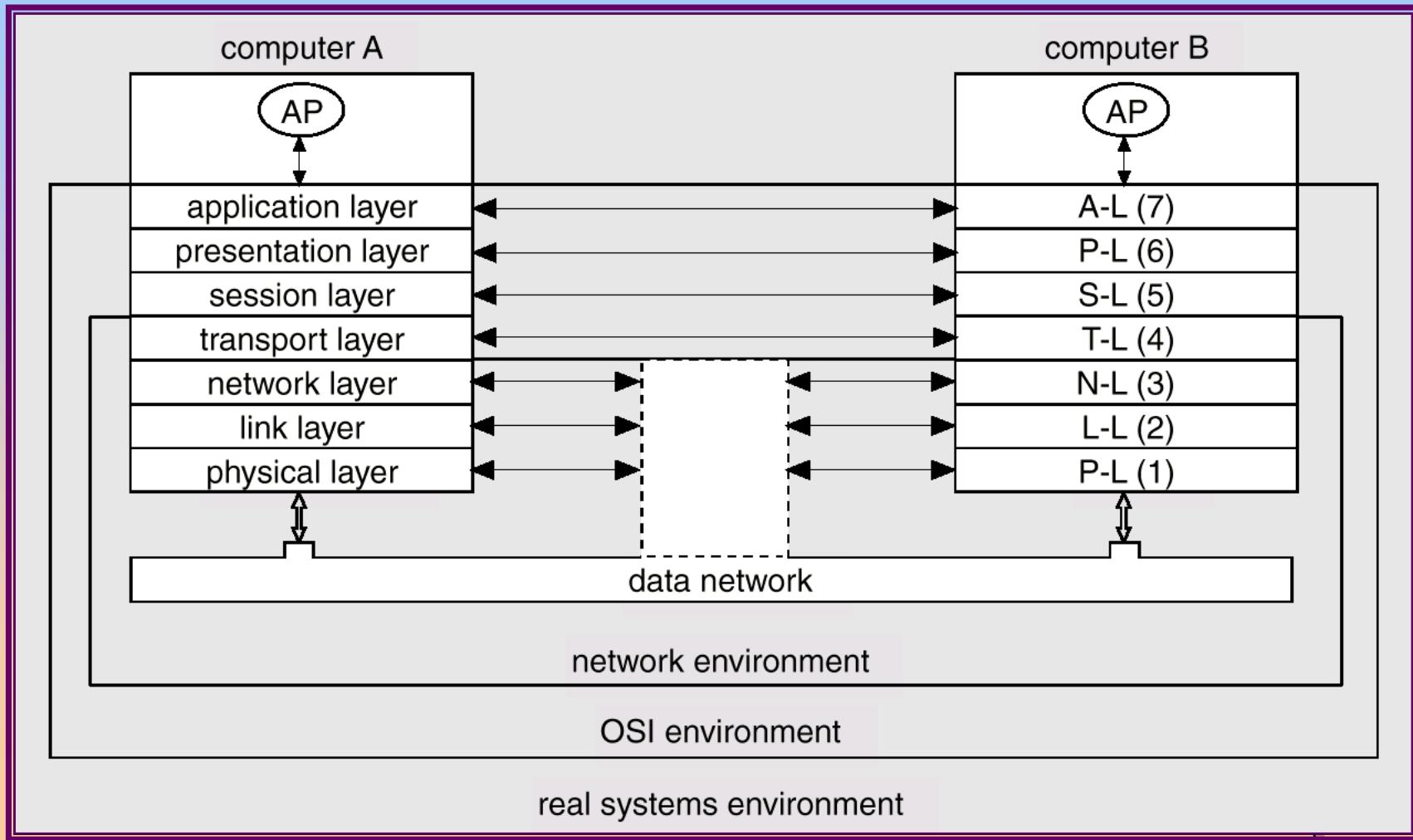
Communication Protocol (Cont.)
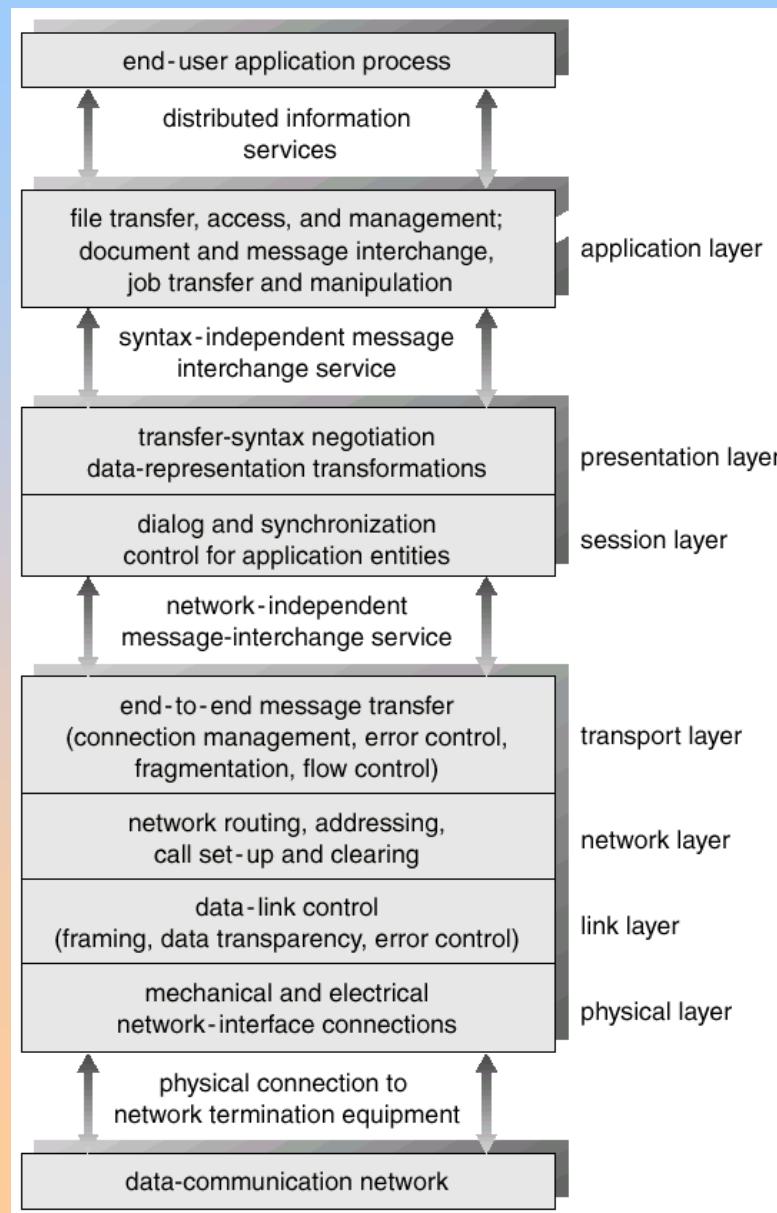
- Transport layer – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.

- Session layer – implements sessions, or process-to-process communications protocols.

- Presentation layer – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing).

- Application layer – interacts directly with the users' deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases.
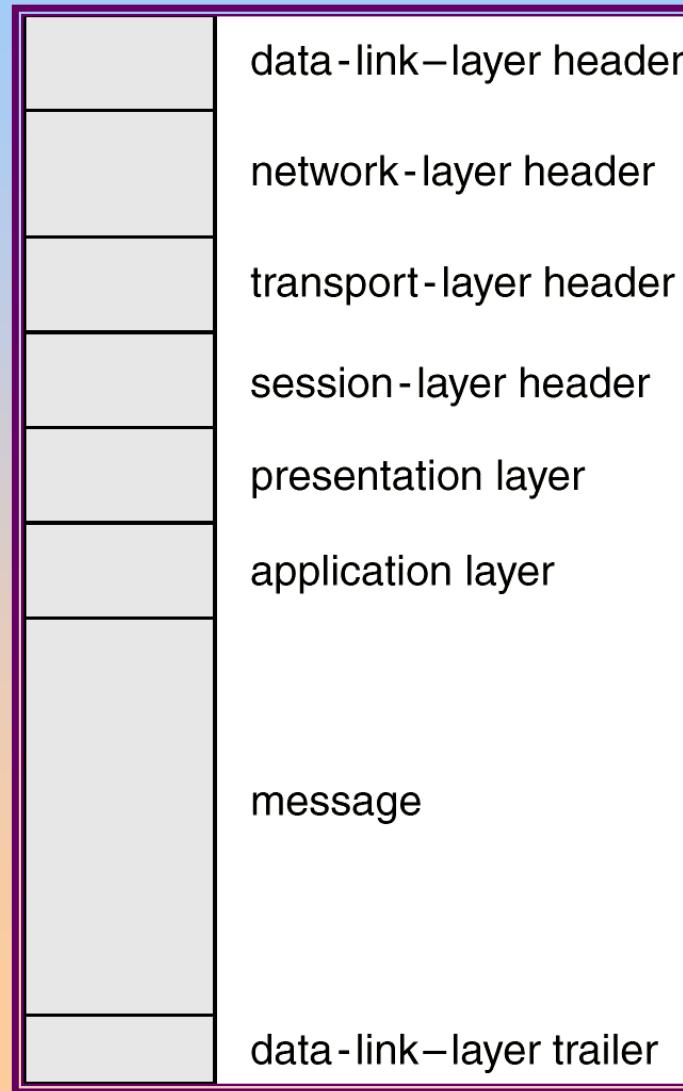
Communication Via ISO Network Model

The ISO Protocol Layer

end-user application process	
↕ distributed information services	
file transfer, access, and management; document and message interchange, job transfer and manipulation	application layer
↕ syntax-independent message interchange service	
transfer-syntax negotiation data-representation transformations	presentation layer
dialog and synchronization control for application entities	session layer
↕ network-independent message-interchange service	
end-to-end message transfer (connection management, error control, fragmentation, flow control)	transport layer
network routing, addressing, call set-up and clearing	network layer
data-link control (framing, data transparency, error control)	link layer
mechanical and electrical network-interface connections	physical layer
↕ physical connection to network termination equipment	
data-communication network	

The ISO Network Message

	data-link−layer header
	network-layer header
	transport-layer header
	session-layer header
	presentation layer
	application layer
	message
	data-link−layer trailer

The TCP/IP Protocol Layers

end-user application process

layers 5-7
- hypertext-transfer protocol, HTTP
- file-transfer protocol, FTP
- remote-terminal protocol, TELNET
- simple mail-transfer protocol, SMTP
- name-server protocol, DNS
- simple network-management protocol, SNMP

layer 4 — TCP | UDP

layers 1-3
- IP
- IEEE802.X/X.25

LAN/WAN

TCP = transmission control protocol
UDP = user datagram protocol
IP = internet protocol
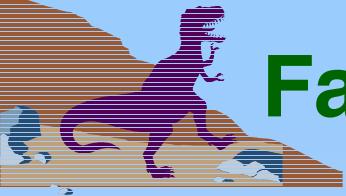
Robustness

- Failure detection

- Reconfiguration

Failure Detection

- Detecting hardware failure is difficult.

- To detect a link failure, a handshaking protocol can be used.

- Assume Site A and Site B have established a link. At fixed intervals, each site will exchange an *I-am-up* message indicating that they are up and running.

- If Site A does not receive a message within the fixed interval, it assumes either (a) the other site is not up or (b) the message was lost.

- Site A can now send an *Are-you-up?* message to Site B.

- If Site A does not receive a reply, it can repeat the message or try an alternate route to Site B.
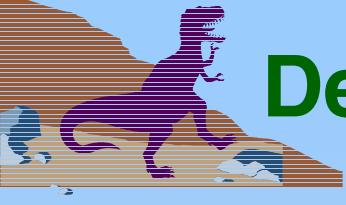
Failure Detection (cont)

- If Site A does not ultimately receive a reply from Site B, it concludes some type of failure has occurred.

- Types of failures:
  - Site B is down

  - The direct link between A and B is down
  - The alternate link from A to B is down

  - The message has been lost

- However, Site A cannot determine exactly **why** the failure has occurred.

Reconfiguration
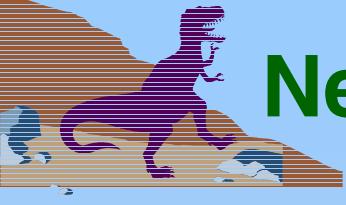
- When Site A determines a failure has occurred, it must reconfigure the system:

  1. If the link from A to B has failed, this must be broadcast to every site in the system.

  2. If a site has failed, every other site must also be notified indicating that the services offered by the failed site are no longer available.

- When the link or the site becomes available again, this information must again be broadcast to all other sites.

Design Issues

- **Transparency** – the distributed system should appear as a conventional, centralized system to the user.

- **Fault tolerance** – the distributed system should continue to function in the face of failure.

- **Scalability** – as demands increase, the system should easily accept the addition of new resources to accommodate the increased demand.

- **Clusters** – a collection of semi-autonomous machines that acts as a single system.
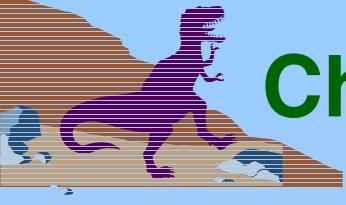
Networking Example

- The transmission of a network packet between hosts on an Ethernet network.

- Every host has a unique IP address and a corresponding Ethernet (MAC) address.

- Communication requires both addresses.

- Domain Name Service (DNS) can be used to acquire IP addresses.

- Address Resolution Protocol (ARP) is used to map MAC addresses to IP addresses.

- If the hosts are on the same network, ARP can be used. If the hosts are on different networks, the sending host will send the packet to a *router* which routes the packet to the destination network.

An Ethernet Packet
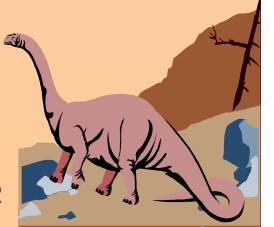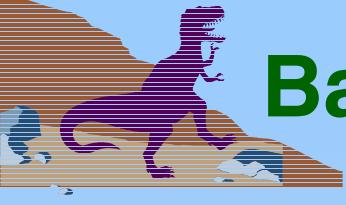
bytes		
7	preamble—start of packet	each byte pattern 10101010
1	start of frame delimiter	pattern 10101011
2 or 6	destination address	ethernet address or broadcast
2 or 6	source address	ethernet address
2	length of data section	length in bytes
0–1500	data	message data
0–46	pad (optional)	message must be > 63 bytes long
4	frame checksum	for error detection

Chapter 16 Distributed-File Systems
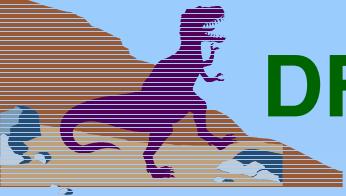
- Background
- Naming and Transparency
- Remote File Access
- Stateful versus Stateless Service
- File Replication
- Example Systems
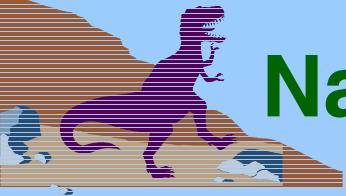
Background

- Distributed file system (DFS) – a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources.

- A DFS manages set of dispersed storage devices

- Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces.

- There is usually a correspondence between constituent storage spaces and sets of files.
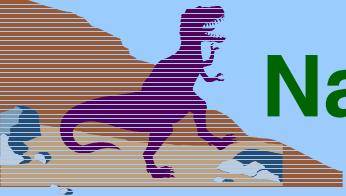
DFS Structure

- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients.

- **Server** – service software running on a single machine.

- **Client** – process that can invoke a service using a set of operations that forms its *client interface.*

- A client interface for a file service is formed by a set of primitive *file operations* (create, delete, read, write).

- Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files.

Naming and Transparency

- *Naming* – mapping between logical and physical objects.

- Multilevel mapping – abstraction of a file that hides the details of how and where on the disk the file is actually stored.

- A *transparent* DFS hides the location where in the network the file is stored.

- For a file being replicated in several sites, the mapping returns a set of the locations of this file's replicas; both the existence of multiple copies and their location are hidden.

Naming Structures
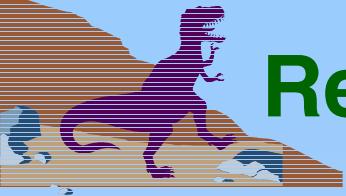
- **Location transparency** – file name does not reveal the file's physical storage location.
    - File name still denotes a specific, although hidden, set of physical disk blocks.
    - Convenient way to share data.
    - Can expose correspondence between component units and machines.

- **Location independence** – file name does not need to be changed when the file's physical storage location changes.
    - Better file abstraction.
    - Promotes sharing the storage space itself.
    - Separates the naming hierarchy form the storage-devices hierarchy.
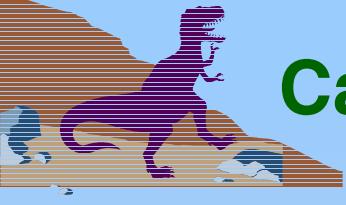
Naming Schemes — Three Main Approaches

- Files named by combination of their host name and local name; guarantees a unique systemwide name.

- Attach remote directories to local directories, giving the appearance of a coherent directory tree; only previously mounted remote directories can be accessed transparently.

- Total integration of the component file systems.
  - A single global name structure spans all the files in the system.
  - If a server is unavailable, some arbitrary set of directories on different machines also becomes unavailable.

Remote File Access

- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally.

  - ✦ If needed data not already cached, a copy of data is brought from the server to the user.

  - ✦ Accesses are performed on the cached copy.

  - ✦ Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches.

  - ✦ *Cache-consistency* problem – keeping the cached copies consistent with the master file.

Cache Location – Disk vs. Main Memory
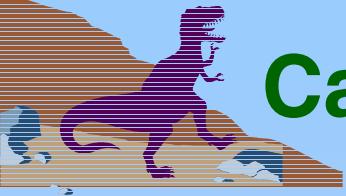
- **Advantages of disk caches**
  - ✦ More reliable.
  - ✦ Cached data kept on disk are still there during recovery and don't need to be fetched again.

- **Advantages of main-memory caches:**
  - ✦ Permit workstations to be diskless.
  - ✦ Data can be accessed more quickly.
  - ✦ Performance speedup in bigger memories.
  - ✦ Server caches (used to speed up disk I/O) are in main memory regardless of where user caches are located; using main-memory caches on the user machine permits a single caching mechanism for servers and users.
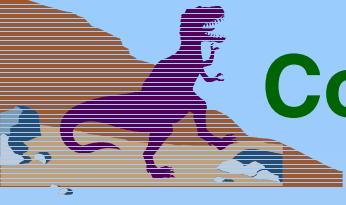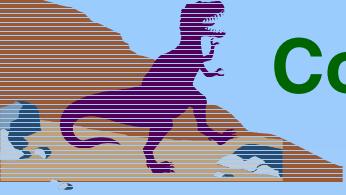
Cache Update Policy

- **Write-through** – write data through to disk as soon as they are placed on any cache. Reliable, but poor performance.

- **Delayed-write** – modifications written to the cache and then written through to the server later. Write accesses complete quickly; some data may be overwritten before they are written back, and so need never be written at all.

  - Poor reliability; unwritten data will be lost whenever a user machine crashes.

  - Variation – scan cache at regular intervals and flush blocks that have been modified since the last scan.

  - Variation – *write-on-close*, writes data back to the server when the file is closed. Best for files that are open for long periods and frequently modified.
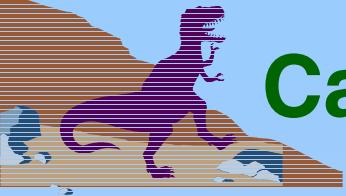
Consistency

- Is locally cached copy of the data consistent with the master copy?

- Client-initiated approach
  - Client initiates a validity check.
  - Server checks whether the local data are consistent with the master copy.

- Server-initiated approach
  - Server records, for each client, the (parts of) files it caches.
  - When server detects a potential inconsistency, it must react.

Comparing Caching and Remote Service

- In caching, many remote accesses handled efficiently by the local cache; most remote accesses will be served as fast as local ones.

- Servers are contracted only occasionally in caching (rather than for each access).
  - Reduces server load and network traffic.
  - Enhances potential for scalability.

- Remote server method handles every remote access across the network; penalty in network traffic, server load, and performance.

- Total network overhead in transmitting big chunks of data (caching) is lower than a series of responses to specific requests (remote-service).

Caching and Remote Service (Cont.)

- Caching is superior in access patterns with infrequent writes. With frequent writes, substantial overhead incurred to overcome cache-consistency problem.

- Benefit from caching when execution carried out on machines with either local disks or large main memories.

- Remote access on diskless, small-memory-capacity machines should be done through remote-service method.

- In caching, the lower intermachine interface is different form the upper user interface.

- In remote-service, the intermachine interface mirrors the local user-file-system interface.

Stateful File Service

- **Mechanism.**
  - Client opens a file.
  - Server fetches information about the file from its disk, stores it in its memory, and gives the client a connection identifier unique to the client and the open file.
  - Identifier is used for subsequent accesses until the session ends.
  - Server must reclaim the main-memory space used by clients who are no longer active.

- **Increased performance.**
  - Fewer disk accesses.
  - Stateful server knows if a file was opened for sequential access and can thus read ahead the next blocks.

Stateless File Server

- Avoids state information by making each request self-contained.

- Each request identifies the file and position in the file.

- No need to establish and terminate a connection by open and close operations.

Distinctions Between Stateful & Stateless Service

■ Failure Recovery.

✦ A stateful server loses all its volatile state in a crash.

✔ Restore state by recovery protocol based on a dialog with clients, or abort operations that were underway when the crash occurred.

✔ Server needs to be aware of client failures in order to reclaim space allocated to record the state of crashed client processes (orphan detection and elimination).

✦ With stateless server, the effects of server failure sand recovery are almost unnoticeable. A newly reincarnated server can respond to a self-contained request without any difficulty.
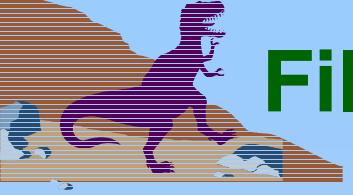
Distinctions (Cont.)

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing
  - additional constraints imposed on DFS design
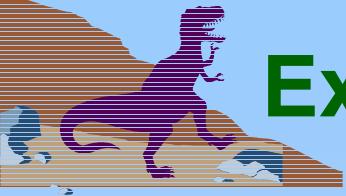
- Some environments require stateful service.
  - A server employing server-initiated cache validation cannot provide stateless service, since it maintains a record of which files are cached by which clients.
  - UNIX use of file descriptors and implicit offsets is inherently stateful; servers must maintain tables to map the file descriptors to inodes, and store the current offset within a file.
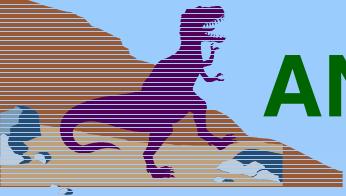
File Replication

- Replicas of the same file reside on failure-independent machines.

- Improves availability and can shorten service time.

- Naming scheme maps a replicated file name to a particular replica.
  - Existence of replicas should be invisible to higher levels.
  - Replicas must be distinguished from one another by different lower-level names.

- Updates – replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas.

- Demand replication – reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica.
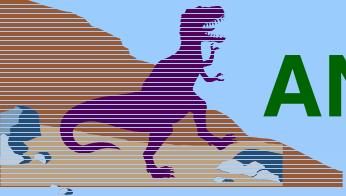
Example System - ANDREW

- A distributed computing environment under development since 1983 at Carnegie-Mellon University.

- Andrew is highly scalable; the system is targeted to span over 5000 workstations.

- Andrew distinguishes between client machines (workstations) and dedicated *server machines*. Servers and clients run the 4.2BSD UNIX OS and are interconnected by an internet of LANs.
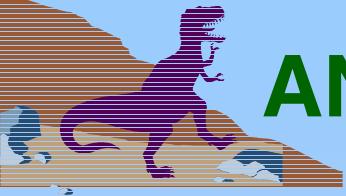
ANDREW (Cont.)

- Clients are presented with a partitioned space of file names: a *local name space* and a *shared name space*.

- Dedicated servers, called Vice, present the shared name space to the clients as an homogeneous, identical, and location transparent file hierarchy.

- The local name space is the root file system of a workstation, from which the shared name space descends.

- Workstations run the Virtue protocol to communicate with Vice, and are required to have local disks where they store their local name space.

- Servers collectively are responsible for the storage and management of the shared name space.
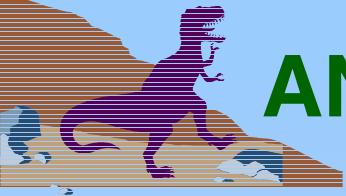
ANDREW (Cont.)

- Clients and servers are structured in clusters interconnected by a backbone LAN.

- A cluster consists of a collection of workstations and a *cluster server* and is connected to the backbone by a *router*.

- A key mechanism selected for remote file operations is *whole file caching*. Opening a file causes it to be cached, in its entirety, on the local disk.
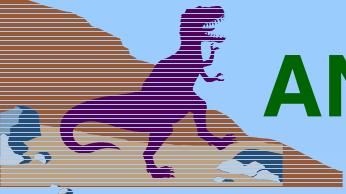
ANDREW Shared Name Space

- Andrew's volumes are small component units associated with the files of a single client.

- A fid identifies a Vice file or directory. A fid is 96 bits long and has three equal-length components:
  - volume number
  - vnode number – index into an array containing the inodes of files in a single volume.
  - uniquifier – allows reuse of vnode numbers, thereby keeping certain data structures, compact.

- Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents.

- Location information is kept on a volume basis, and the information is replicated on each server.
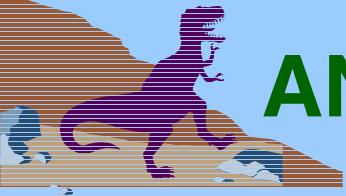
ANDREW File Operations

- Andrew caches entire files form servers. A client workstation interacts with Vice servers only during opening and closing of files.

- Venus – caches files from Vice when they are opened, and stores modified copies of files back when they are closed.

- Reading and writing bytes of a file are done by the kernel without Venus intervention on the cached copy.

- Venus caches contents of directories and symbolic links, for path-name translation.

- Exceptions to the caching policy are modifications to directories that are made directly on the server responsibility for that directory.

ANDREW Implementation

- Client processes are interfaced to a UNIX kernel with the usual set of system calls.

- Venus carries out path-name translation component by component.

- The UNIX file system is used as a low-level storage system for both servers and clients. The client cache is a local directory on the workstation's disk.

- Both Venus and server processes access UNIX files directly by their inodes to avoid the expensive path name-to-inode translation routine.
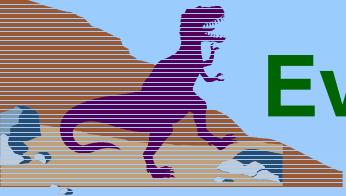
ANDREW Implementation (Cont.)

- Venus manages two separate caches:
    - one for status
    - one for data
- LRU algorithm used to keep each of them bounded in size.
- The status cache is kept in virtual memory to allow rapid servicing of *stat* (file status returning) system calls.
- The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of the disk blocks in memory that are transparent to Venus.

Chapter 17 Distributed Coordination
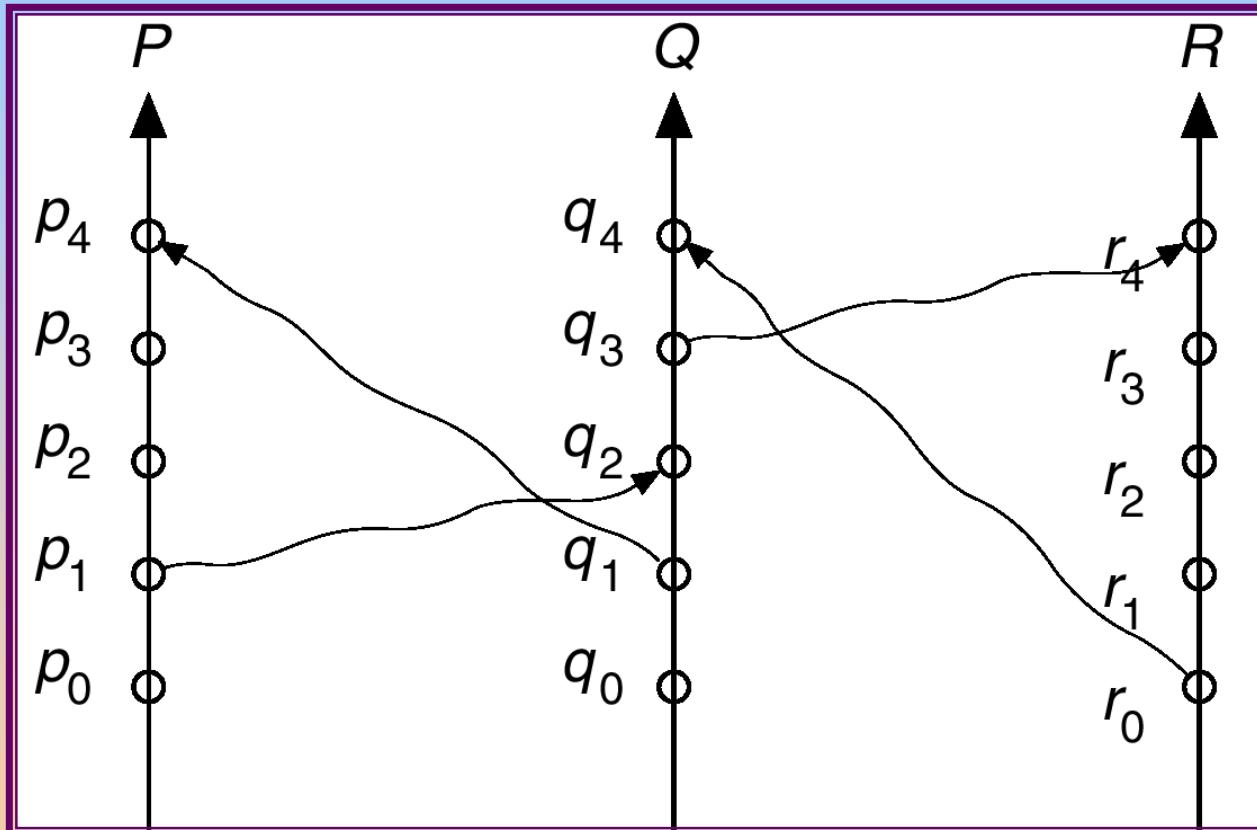
- Event Ordering
- Mutual Exclusion
- Atomicity
- Concurrency Control
- Deadlock Handling
- Election Algorithms
- Reaching Agreement
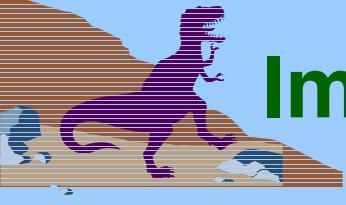
Event Ordering

- *Happened-before* relation (denoted by $\rightarrow$).
    - If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$.
    - If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.
    - If $A \rightarrow B$ and B $\rightarrow C$ then $A \rightarrow C$.
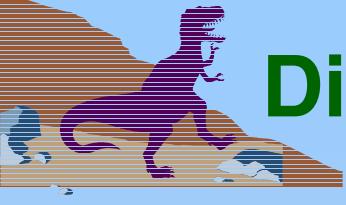
Relative Time for Three Concurrent Processes

Implementation of →

- Associate a timestamp with each system event. Require that for every pair of events A and B, if $A \rightarrow B$, then the timestamp of A is less than the timestamp of B.

- Within *each* process P_i a *logical clock*, LC_i is associated. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process.

- A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock.

- If the timestamps of two events A and B are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering.

Distributed Mutual Exclusion (DME)
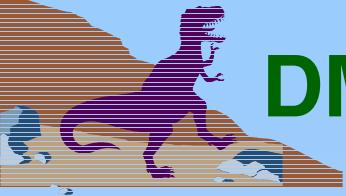
- **Assumptions**
  - ✦ The system consists of n processes; each process P_i resides at a different processor.
  - ✦ Each process has a critical section that requires mutual exclusion.
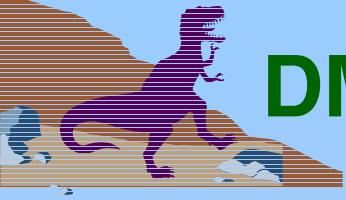
- **Requirement**
  - ✦ If P_i is executing in its critical section, then no other process P_j is executing in its critical section.

- We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections.

DME: Centralized Approach
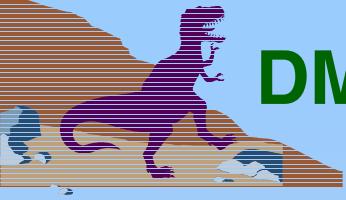
- One of the processes in the system is chosen to coordinate the entry to the critical section.

- A process that wants to enter its critical section sends a *request* message to the coordinator.

- The coordinator decides which process can enter the critical section next, and its sends that process a *reply* message.

- When the process receives a *reply* message from the coordinator, it enters its critical section.

- After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.

- This scheme requires three messages per critical-section entry:
  - ✦ request
  - ✦ reply
  - ✦ release

DME: Fully Distributed Approach
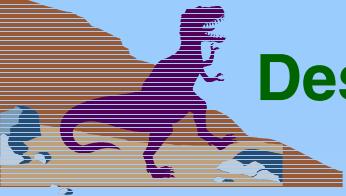
- When process P_i wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request* (P_i, *TS*) to all other processes in the system.

- When process P_j receives a *request* message, it may reply immediately or it may defer sending a reply back.

- When process P_i receives a *reply* message from all other processes in the system, it can enter its critical section.

- After exiting its critical section, the process sends *reply* messages to all its deferred requests.

DME: Fully Distributed Approach (Cont.)

■ The decision whether process P_j replies immediately to a *request*(P_i, *TS*) message or defers its reply is based on three factors:

✦ If P_j is in its critical section, then it defers its reply to P_i.

✦ If P_j does *not* want to enter its critical section, then it sends a *reply* immediately to P_i.

✦ If P_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp *TS*.

✔ If its own request timestamp is greater than *TS*, then it sends a *reply* immediately to P_i (P_i asked first).

✔ Otherwise, the reply is deferred.
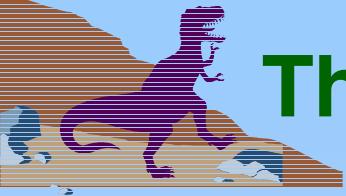
Desirable Behavior of Fully Distributed Approach

- Freedom from Deadlock is ensured.

- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first served order.

- The number of messages per critical-section entry is

$$2 \times (n - 1).$$
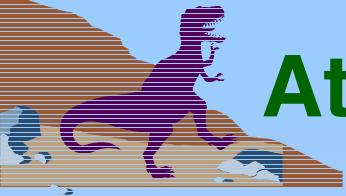
This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

Three Undesirable Consequences
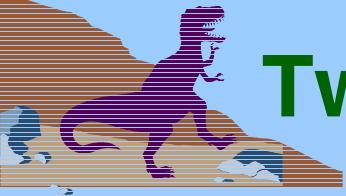
- The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex.

- If one of the processes fails, then the entire scheme collapses. This can be dealt with by continuously monitoring the state of all the processes in the system.

- Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section. This protocol is therefore suited for small, stable sets of cooperating processes.

Atomicity

- Either all the operations associated with a program unit are executed to completion, or none are performed.

- Ensuring atomicity in a distributed system requires a *transaction coordinator*, which is responsible for the following:
  - Starting the execution of the transaction.
  - Breaking the transaction into a number of subtransactions, and distribution these subtransactions to the appropriate sites for execution.
  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

Two-Phase Commit Protocol (2PC)
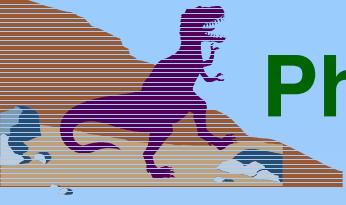
- Assumes fail-stop model.

- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.

- When the protocol is initiated, the transaction may still be executing at some of the local sites.

- The protocol involves all the local sites at which the transaction executed.

- Example: Let T be a transaction initiated at site S_i and let the transaction coordinator at S_i be C_i.

Phase 1: Obtaining a Decision

- C_i adds <prepare T> record to the log.
- C_i sends <prepare T> message to all sites.
- When a site receives a <prepare T> message, the transaction manager determines if it can commit the transaction.
  - If no: add <no T> record to the log and respond to C_i with <abort T>.
  - If yes:
    - add <ready T> record to the log.
    - force *all log records* for T onto stable storage.
    - transaction manager sends <ready T> message to C_i.
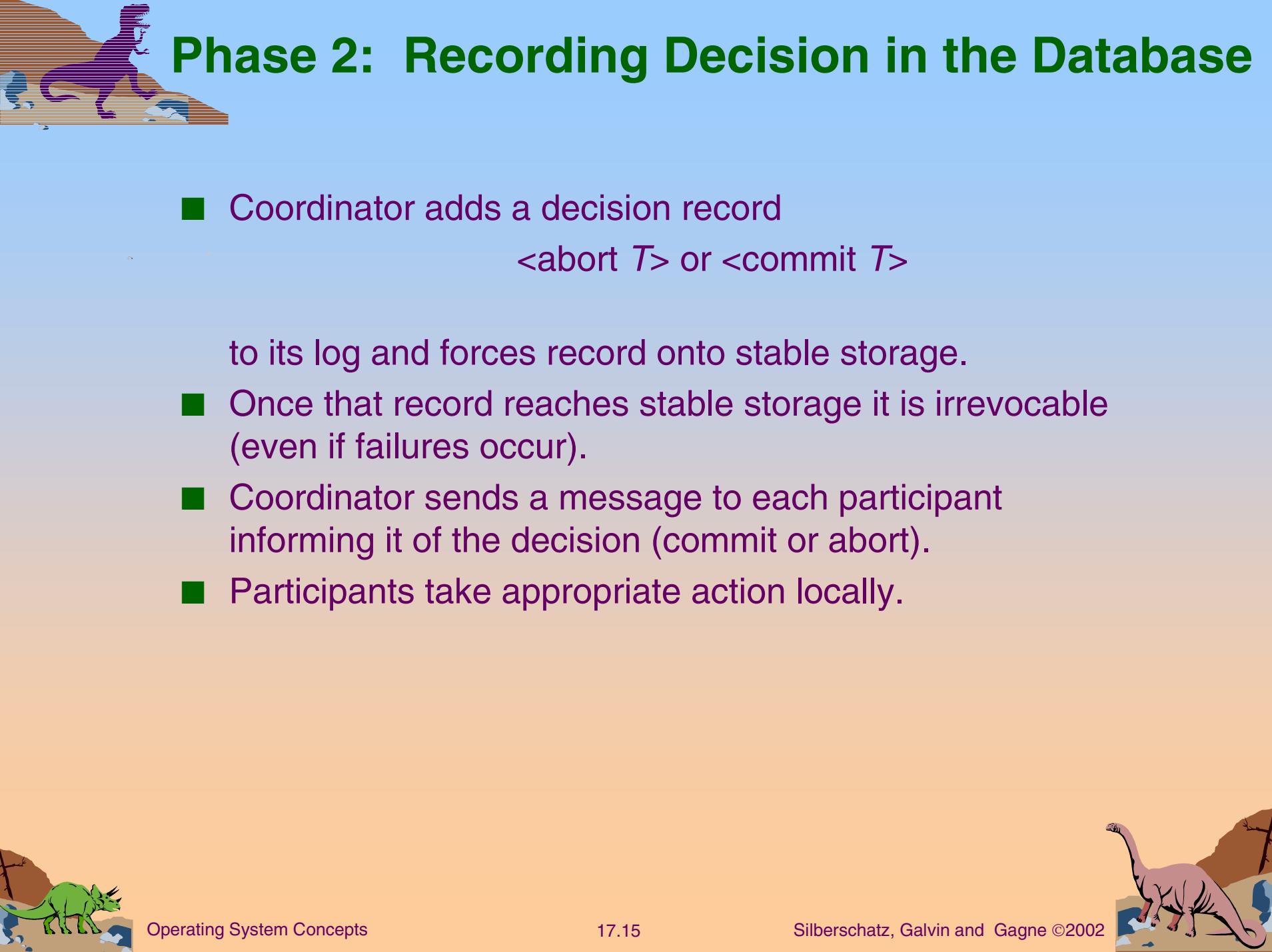
Phase 1 (Cont.)
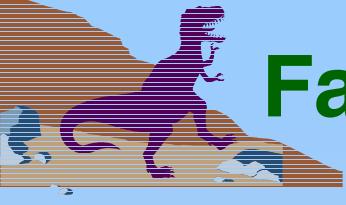
- Coordinator collects responses
    - ✦ All respond "ready",
      decision is *commit*.
    - ✦ At least one response is "abort",
      decision is *abort*.
    - ✦ At least one participant fails to respond within time out
      period,
      decision is *abort*.

Phase 2: Recording Decision in the Database

- Coordinator adds a decision record

  <abort *T*> or <commit *T*>

  to its log and forces record onto stable storage.
- Once that record reaches stable storage it is irrevocable (even if failures occur).
- Coordinator sends a message to each participant informing it of the decision (commit or abort).
- Participants take appropriate action locally.
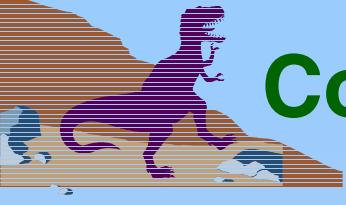
Failure Handling in 2PC – Site Failure

- The log contains a <commit T> record. In this case, the site executes **redo**(T).

- The log contains an <abort T> record. In this case, the site executes **undo**(T).

- The contains a <ready T> record; consult C_i. If C_i is down, site sends **query-status** T message to the other sites.

- The log contains no control records concerning T. In this case, the site executes **undo**(T).
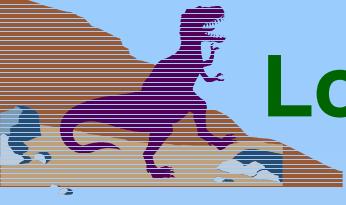
Failure Handling in 2PC – Coordinator C_i Failure

■ If an active site contains a <commit T> record in its log, the T must be committed.

■ If an active site contains an <abort T> record in its log, then T must be aborted.

■ If some active site does *not* contain the record <ready T> in its log then the failed coordinator C_i cannot have decided to
commit T. Rather than wait for C_i to recover, it is preferable to abort T.

■ All active sites have a <ready T> record in their logs, but no additional control records. In this case we must wait for the coordinator to recover.

  ✦ Blocking problem − T is blocked pending the recovery of site S_i.

Concurrency Control

- Modify the centralized concurrency schemes to accommodate the distribution of transactions.

- Transaction manager coordinates execution of transactions (or subtransactions) that access data at local sites.

- Local transaction only executes at that site.

- Global transaction executes at several sites.

Locking Protocols

- Can use the two-phase locking protocol in a distributed environment by changing how the lock manager is implemented.

- Nonreplicated scheme – each site maintains a local lock manager which administers lock and unlock requests for those data items that are stored in that site.

  - Simple implementation involves two message transfers for handling lock requests, and one message transfer for handling unlock requests.

  - Deadlock handling is more complex.
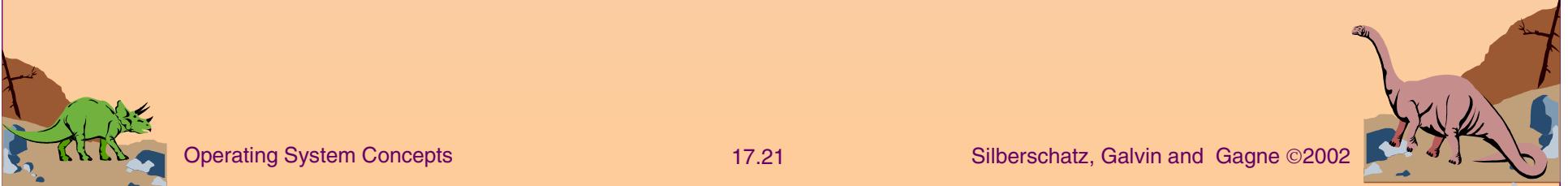
Single-Coordinator Approach
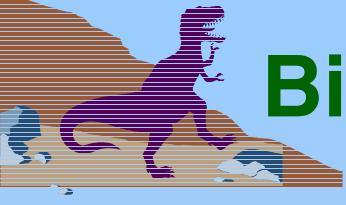
- A single lock manager resides in a single chosen site, all lock and unlock requests are made a that site.

- Simple implementation

- Simple deadlock handling

- Possibility of bottleneck

- Vulnerable to loss of concurrency controller if single site fails

- *Multiple-coordinator approach* distributes lock-manager function over several sites.

Majority Protocol

- Avoids drawbacks of central control by dealing with replicated data in a decentralized manner.

- More complicated to implement

- Deadlock-handling algorithms must be modified; possible for deadlock to occur in locking only one data item.
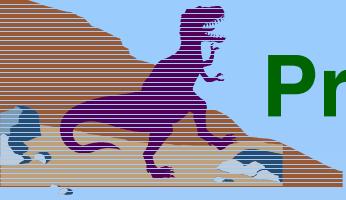
Biased Protocol
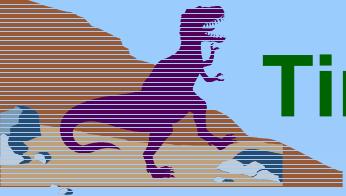
- Similar to majority protocol, but requests for shared locks prioritized over requests for exclusive locks.

- Less overhead on read operations than in majority protocol; but has additional overhead on writes.

- Like majority protocol, deadlock handling is complex.
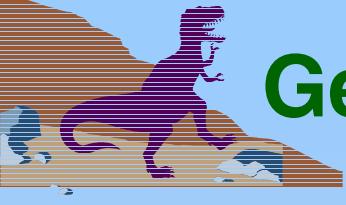
Primary Copy

- One of the sites at which a replica resides is designated as the primary site. Request to lock a data item is made at the primary site of that data item.

- Concurrency control for replicated data handled in a manner similar to that of unreplicated data.

- Simple implementation, but if primary site fails, the data item is unavailable, even though other sites may have a replica.
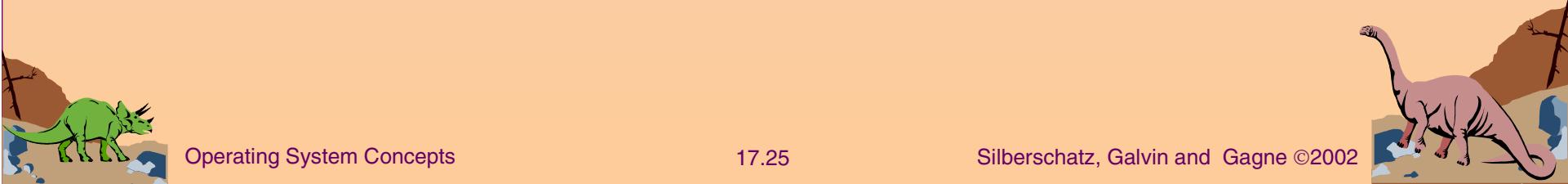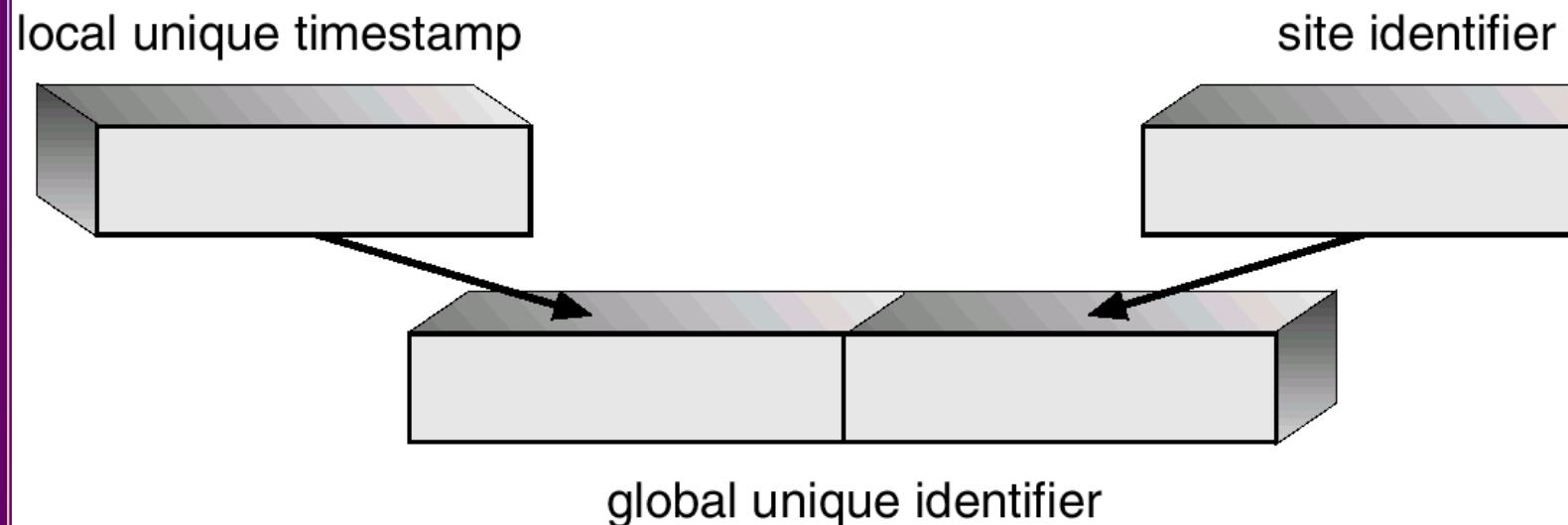
Timestamping

■ Generate unique timestamps in distributed scheme:

  ✦ Each site generates a unique local timestamp.

  ✦ The global unique timestamp is obtained by concatenation of the unique local timestamp with the unique site identifier

  ✦ Use a *logical clock* defined within each site to ensure the fair generation of timestamps.

■ Timestamp-ordering scheme – combine the centralized concurrency control timestamp scheme with the 2PC protocol to obtain a protocol that ensures serializability with no cascading rollbacks.

Generation of Unique Timestamps

local unique timestamp

site identifier

global unique identifier
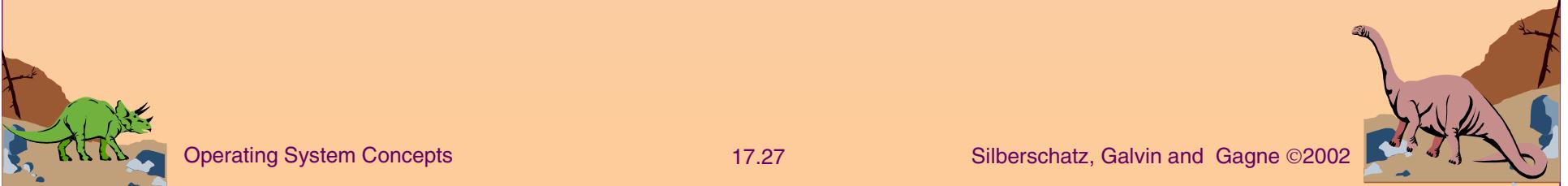
Deadlock Prevention

- Resource-ordering deadlock-prevention – define a *global* ordering among the system resources.

  - ✦ Assign a unique number to all system resources.

  - ✦ A process may request a resource with unique number i only if it is not holding a resource with a unique number grater than i.

  - ✦ Simple to implement; requires little overhead.

- Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm.

  - ✦ Also implemented easily, but may require too much overhead.

Timestamped Deadlock-Prevention Scheme

- Each process P_i is assigned a unique priority number

- Priority numbers are used to decide whether a process P_i should wait for a process P_j; otherwise P_i is rolled back.

- The scheme prevents deadlocks. For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j. Thus a cycle cannot exist.

Wait-Die Scheme
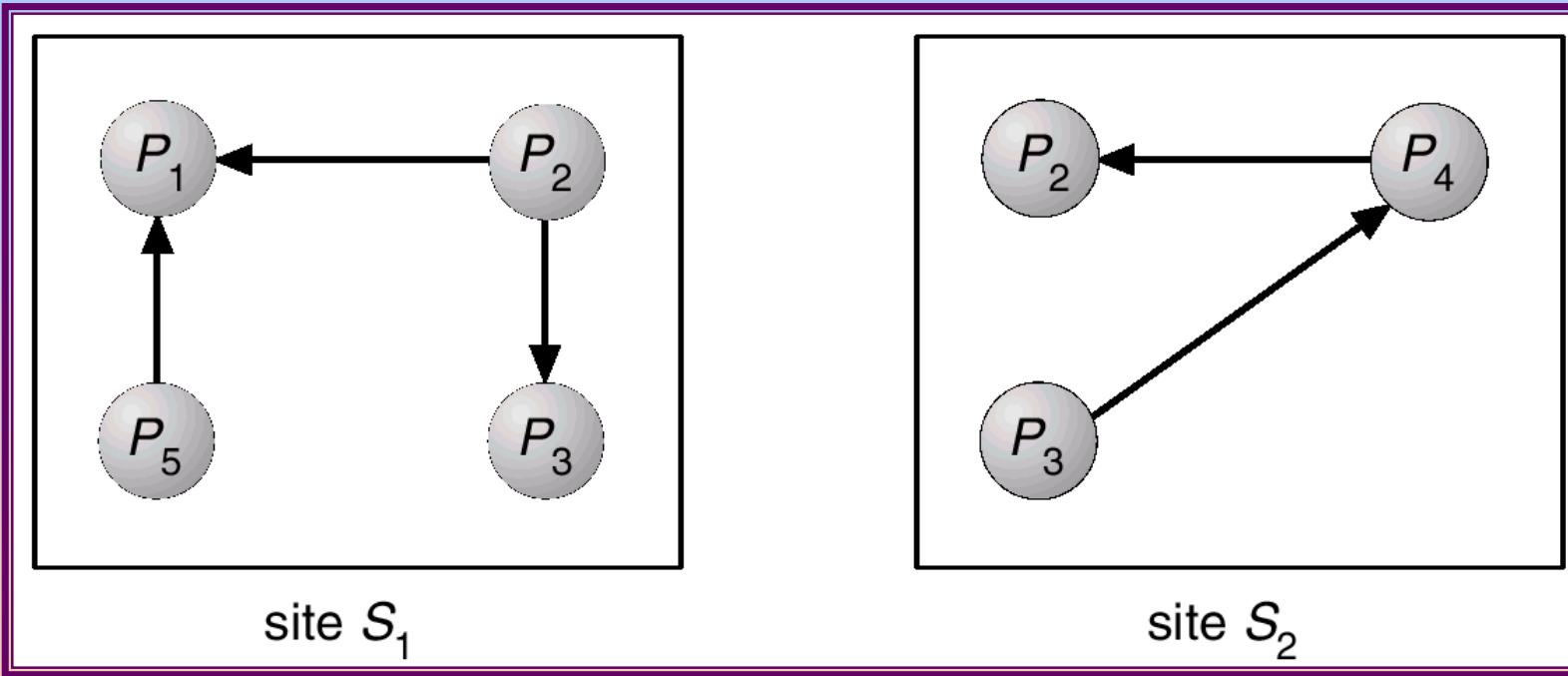
- Based on a nonpreemptive technique.

- If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a smaller timestamp than does P_j (P_i is older than P_j). Otherwise, P_i is rolled back (dies).

- Example: Suppose that processes P_1, P_2, and P_3 have timestamps t, 10, and 15 respectively.
  - if P_1 request a resource held by P_2, then P_1 will wait.
  - If P_3 requests a resource held by P_2, then P_3 will be rolled back.
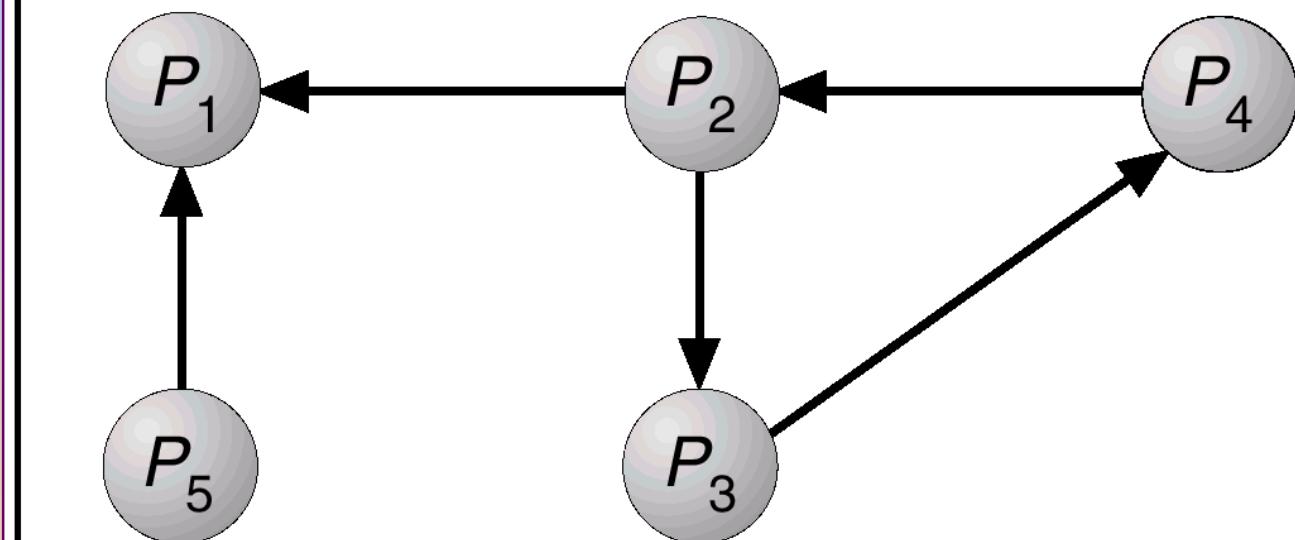
Would-Wait Scheme

- Based on a preemptive technique; counterpart to the wait-die system.

- If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a larger timestamp than does P_j (P_i is younger than P_j). Otherwise P_j is rolled back (P_j is wounded by P_i).

- Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15 respectively.
  - ✦ If P_1 requests a resource held by P_2, then the resource will be preempted from P_2 and P_2 will be rolled back.
  - ✦ If P_3 requests a resource held by P_2, then P_3 will wait.

Two Local Wait-For Graphs

site S_1 site S_2

Global Wait-For Graph
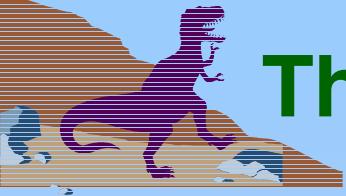
Deadlock Detection – Centralized Approach

- Each site keeps a *local* wait-for graph. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site.

- A global wait-for graph is maintained in a *single* coordination process; this graph is the union of all local wait-for graphs.

- There are three different options (points in time) when the wait-for graph may be constructed:

    1. Whenever a new edge is inserted or removed in one of the local wait-for graphs.

    2. Periodically, when a number of changes have occurred in a wait-for graph.

    3. Whenever the coordinator needs to invoke the cycle-detection algorithm..

- Unnecessary rollbacks may occur as a result of *false cycles*.

Detection Algorithm Based on Option 3
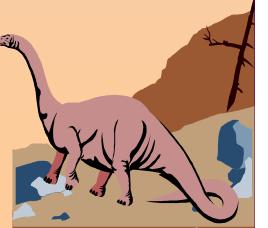
■ Append unique identifiers (timestamps) to requests form different sites.

■ When process P_i, at site A, requests a resource from process P_j, at site B, a request message with timestamp TS is sent.

■ The edge $P_i \rightarrow P_j$ with the label TS is inserted in the local wait-for of A. The edge is inserted in the local wait-for graph of B only if B has received the request message and cannot immediately grant the requested resource.
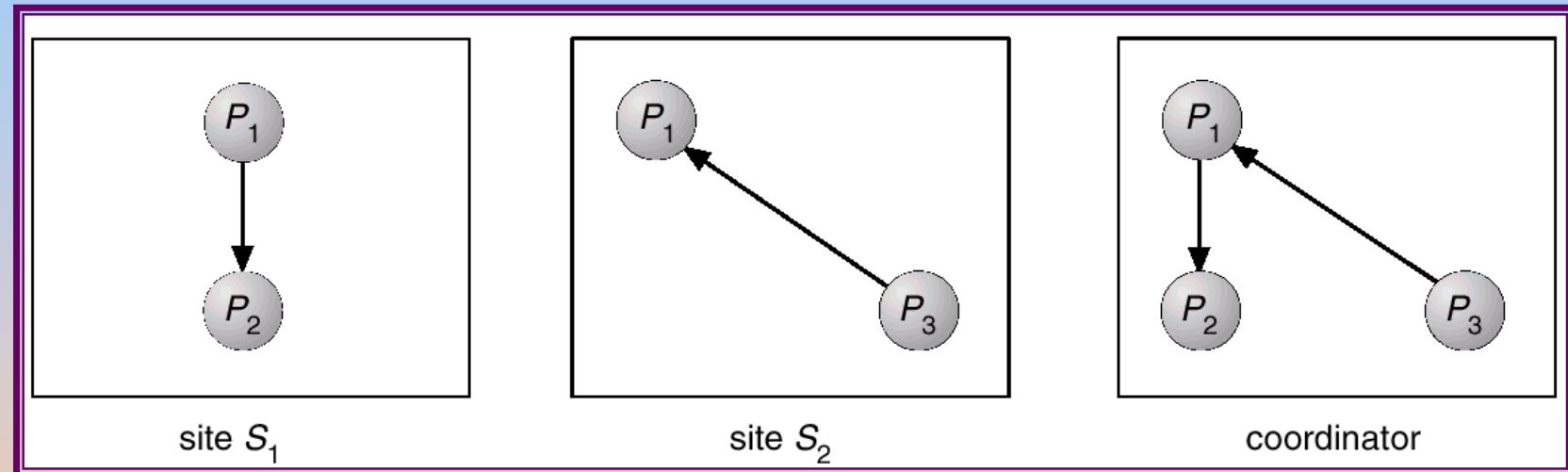
The Algorithm
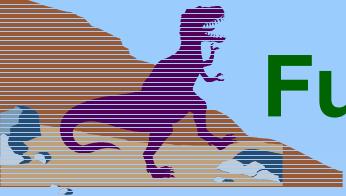
1. The controller sends an initiating message to each site in the system.

2. On receiving this message, a site sends its local wait-for graph to the coordinator.

3. When the controller has received a reply from each site, it constructs a graph as follows:

    (a) The constructed graph contains a vertex for every process in the system.

    (b) The graph has an edge $P_i \rightarrow P_j$ if and only if (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or (2) an edge $P_i \rightarrow P_j$ with some label TS appears in more than one wait-for graph.

    If the constructed graph contains a cycle $\Rightarrow$ deadlock.

Local and Global Wait-For Graphs

site S_1 site S_2 coordinator
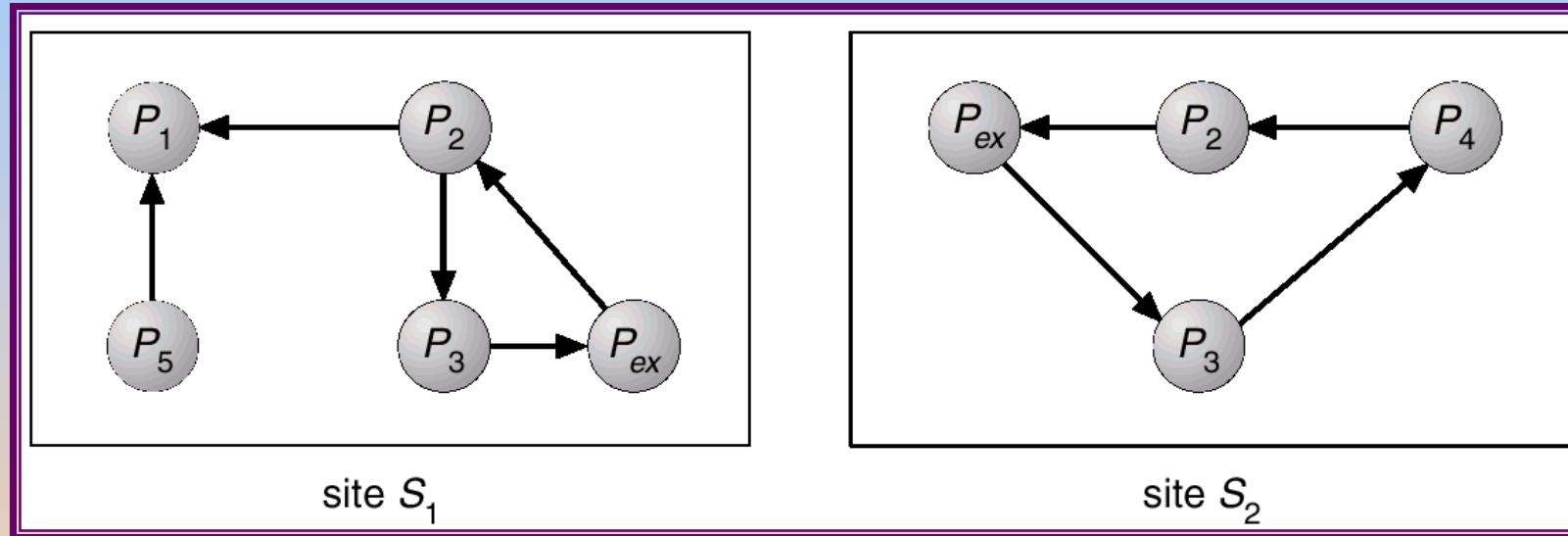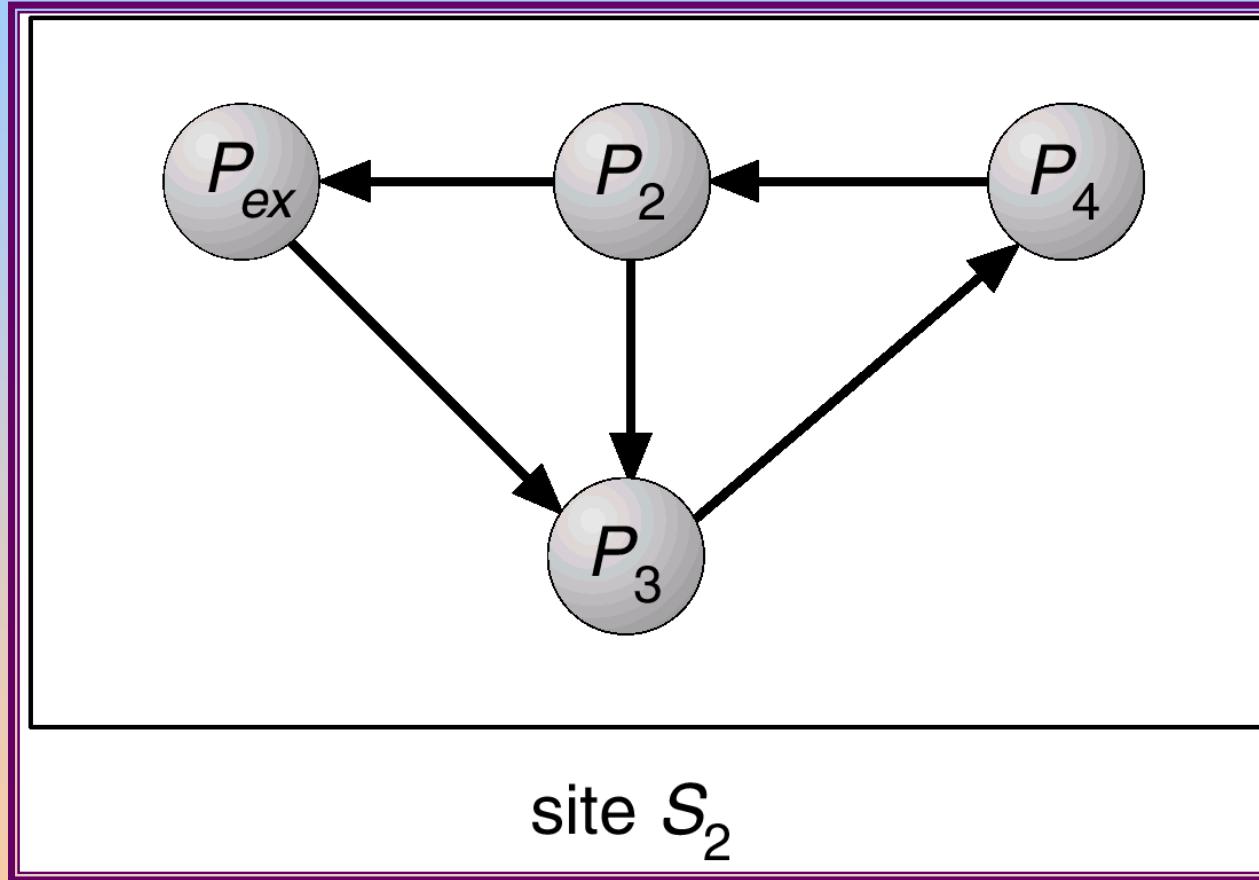
Fully Distributed Approach
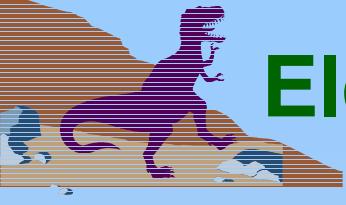
- All controllers share equally the responsibility for detecting deadlock.

- Every site constructs a wait-for graph that represents a part of the total graph.

- We add one additional node P_{ex} to each local wait-for graph.

- If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlock state.

- A cycle involving P_{ex} implies the possibility of a deadlock. To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked.

Augmented Local Wait-For Graphs
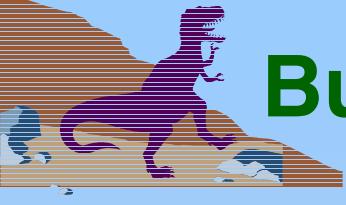
site S_1

site S_2

Augmented Local Wait-For Graph in Site S2

site S_2
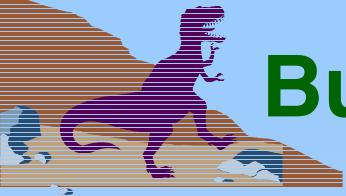
Election Algorithms

- Determine where a new copy of the coordinator should be restarted.

- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process P_i is i.

- Assume a one-to-one correspondence between processes and sites.

- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number.

- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures.
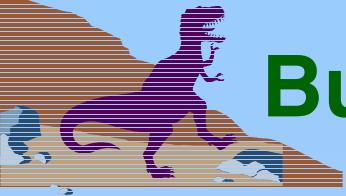
Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system.

- If process P_i sends a request that is not answered by the coordinator within a time interval T, assume that the coordinator has failed; P_i tries to elect itself as the new coordinator.

- P_i sends an election message to every process with a higher priority number, P_i then waits for any of these processes to answer within T.

Bully Algorithm (Cont.)

- If no response within T, assume that all processes with numbers greater than i have failed; P_i elects itself the new coordinator.

- If answer is received, P_i begins time interval T', waiting to receive a message that a process with a higher priority number has been elected.

- If no message is sent within T', assume the process with a higher number has failed; P_i should restart the algorithm
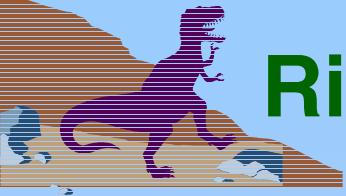
Bully Algorithm (Cont.)

- If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j:

  - P_j is the new coordinator ($j > i$). P_i, in turn, records this information.
  - P_j started an election ($j > i$). P_i, sends a response to P_j and begins its own election algorithm, provided that Pi has not already initiated such an election.

- After a failed process recovers, it immediately begins execution of the same algorithm.

- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number.
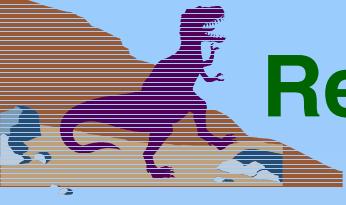
Ring Algorithm

- Applicable to systems organized as a ring (logically or physically).

- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors.

- Each process maintains an *active list*, consisting of all the priority numbers of all active processes in the system when the algorithm ends.

- If process P_i detects a coordinator failure, I creates a new active list that is initially empty. It then sends a message *elect(i)* to its right neighbor, and adds the number *i* to its active list.
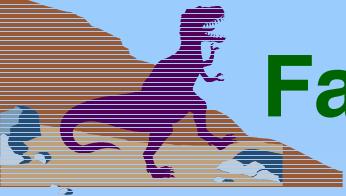
Ring Algorithm (Cont.)

- If P_i receives a message elect(j) from the process on the left, it must respond in one of three ways:

    1. If this is the first *elect* message it has seen or sent, P_i creates a new active list with the numbers i and j. It then sends the message *elect(i)*, followed by the message *elect(j)*.

    - If $i \neq j$, then the active list for P_i now contains the numbers of all the active processes in the system. P_i can now determine the largest number in the active list to identify the new coordinator process.

    - If $i = j$, then P_i receives the message *elect(i)*. The active list for P_i contains all the active processes in the system. P_i can now determine the new coordinator process.
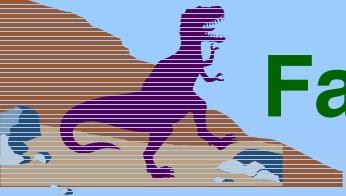
Reaching Agreement

- There are applications where a set of processes wish to agree on a common "value".

- Such agreement may not take place due to:
  - Faulty communication medium
  - Faulty processes
    - ✔ Processes may send garbled or incorrect messages to other processes.
    - ✔ A subset of the processes may collaborate with each other in an attempt to defeat the scheme.
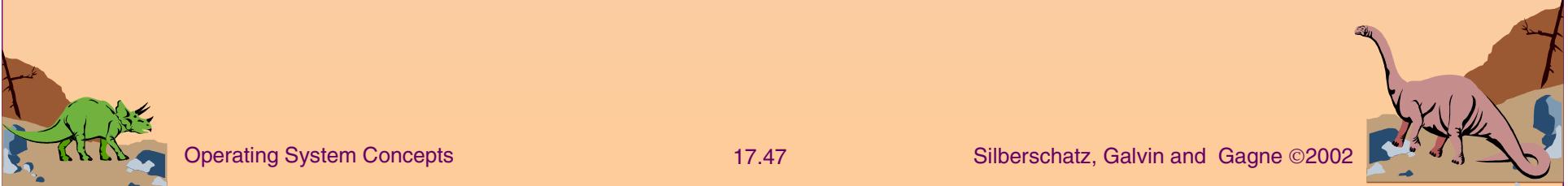
Faulty Communications

- Process P_i at site A, has sent a message to process P_j at site B; to proceed, P_i needs to know if P_j has received the message.

- Detect failures using a time-out scheme.
  - When P_i sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message form P_j.
  - When P_j receives the message, it immediately sends an acknowledgment to P_i.
  - If P_i receives the acknowledgment message within the specified time interval, it concludes that P_j has received its message. If a time-out occurs, P_j needs to retransmit its message and wait for an acknowledgment.
  - Continue until P_i either receives an acknowledgment, or is notified by the system that B is down.

Faulty Communications (Cont.)

- Suppose that P_j also needs to know that P_i has received its acknowledgment message, in order to decide on how to proceed.

  - In the presence of failure, it is not possible to accomplish this task.
  - It is not possible in a distributed environment for processes P_i and P_j to agree completely on their respective states.
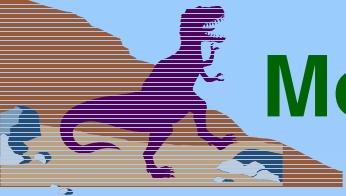
Faulty Processes (Byzantine Generals Problem)

- Communication medium is reliable, but processes can fail in unpredictable ways.

- Consider a system of n processes, of which no more than m are faulty. Suppose that each process P_i has some private value of V_i.

- Devise an algorithm that allows each nonfaulty P_i to construct a vector $X_i = (A_{i,1}, A_{i,2}, \ldots, A_{i,n})$ such that::
  - If P_j is a nonfaulty process, then $A_{ij} = V_j$.
  - If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.

- Solutions share the following properties.
  - A correct algorithm can be devised only if $n \geq 3 \times m + 1$.
  - The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays.

Faulty Processes (Cont.)
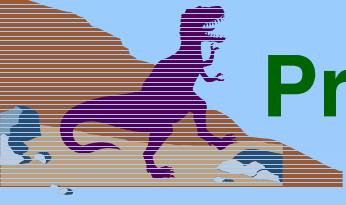
- An algorithm for the case where $m = 1$ and $n = 4$ requires two rounds of information exchange:

  - Each process sends its private value to the other 3 processes.
  - Each process sends the information it has obtained in the first round to all other processes.

- If a faulty process refuses to send messages, a nonfaulty process can choose an arbitrary value and pretend that that value was sent by that process.

- After the two rounds are completed, a nonfaulty process P_i can construct its vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ as follows:

  - $A_{i,j} = V_i.$
  - For $j \neq i$, if at least two of the three values reported for process P_j agree, then the majority value is used to set the value of $A_{ij}.$ Otherwise, a default value (*nil*) is used.

Module 18: Protection

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

Protection
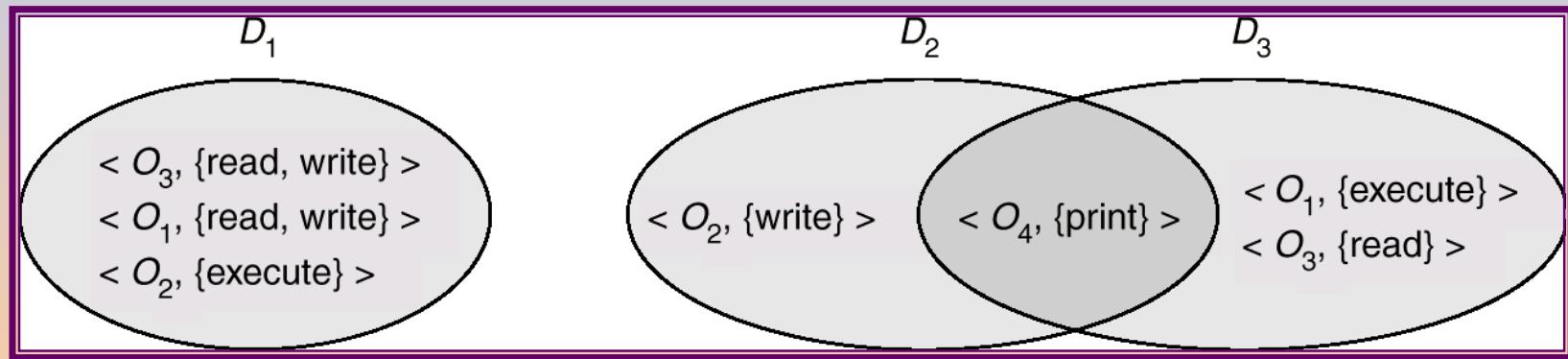
- Operating system consists of a collection of objects, hardware or software

- Each object has a unique name and can be accessed through a well-defined set of operations.

- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.

Domain Structure

- Access-right = *<object-name, rights-set>*
  where *rights-set* is a subset of all valid operations that
  can be performed on the object.

- Domain = set of access-rights

Domain Implementation (UNIX)

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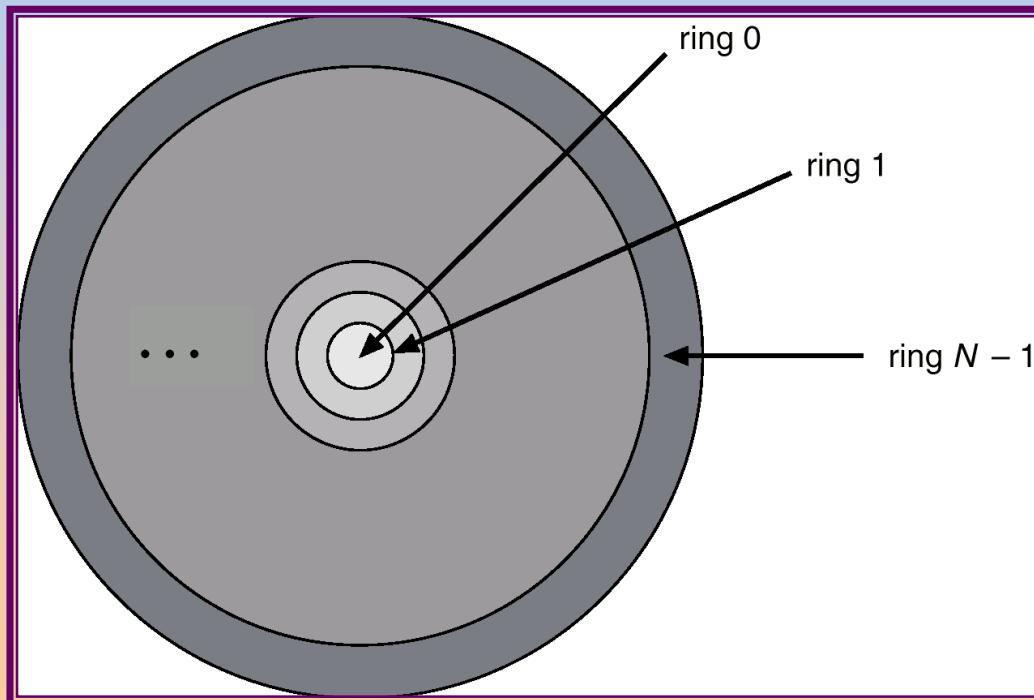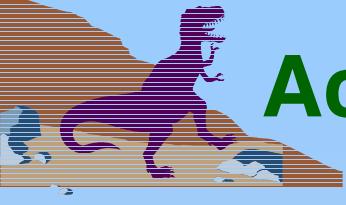
- UNIX
  - Domain = user-id
  - Domain switch accomplished via file system.
    - Each file has associated with it a domain bit (setuid bit).
    - When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset.

Domain Implementation (Multics)

- Let D_i and D_j be any two domain rings.
- If $j < I \Rightarrow D_i \subseteq D_j$

ring 0

ring 1

ring $N - 1$

Multics Rings
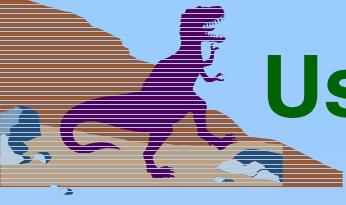
Access Matrix

- View protection as a matrix (*access matrix*)

- Rows represent domains

- Columns represent objects

- *Access(i, j)* is the set of operations that a process executing in Domain$_i$ can invoke on Object$_j$

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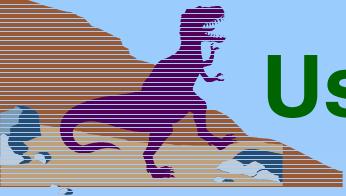
object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure A
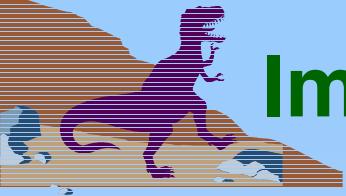
Use of Access Matrix

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

- Can be expanded to dynamic protection.
  - Operations to add, delete access rights.
  - Special access rights:
    - *owner of O_i*
    - *copy op from O_i to O_j*
    - *control – D_i can modify D_j access rights*
    - *transfer – switch from domain D_i to D_j*

Use of Access Matrix (Cont.)

■ Access matrix design separates mechanism from policy.

✦ Mechanism

✔ Operating system provides access-matrix + rules.

✔ If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced.

✦ Policy

✔ User dictates policy.

✔ Who can access what object and in what mode.

Implementation of Access Matrix

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

  > Domain 1 = Read, Write
  > Domain 2 = Read
  > Domain 3 = Read

  $\vdots$

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

  > Object 1 – Read
  >
  > Object 4 – Read, Write, Execute
  >
  > Object 5 – Read, Write, Delete, Copy

Access Matrix of Figure A With Domains as Objects

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

Figure B

Access Matrix with *Copy* Rights

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

(a)

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

(b)

Access Matrix With *Owner* Rights

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

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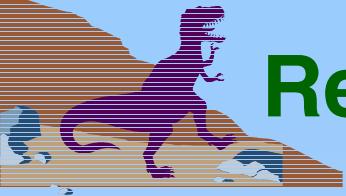
object domain	F_1	F_2	F_3
D_1	owner execute		
D_2		owner read* write*	read* owner write*
D_3		write	write

(b)
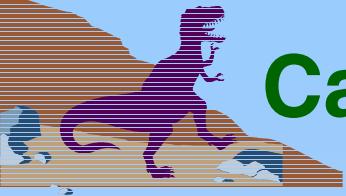
Modified Access Matrix of Figure B

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

Revocation of Access Rights

- *Access List* – Delete access rights from access list.
  - ✦ Simple
  - ✦ Immediate

- *Capability List* – Scheme required to locate capability in the system before capability can be revoked.
  - ✦ Reacquisition
  - ✦ Back-pointers
  - ✦ Indirection
  - ✦ Keys
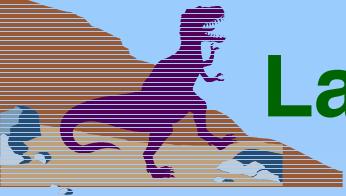
Capability-Based Systems

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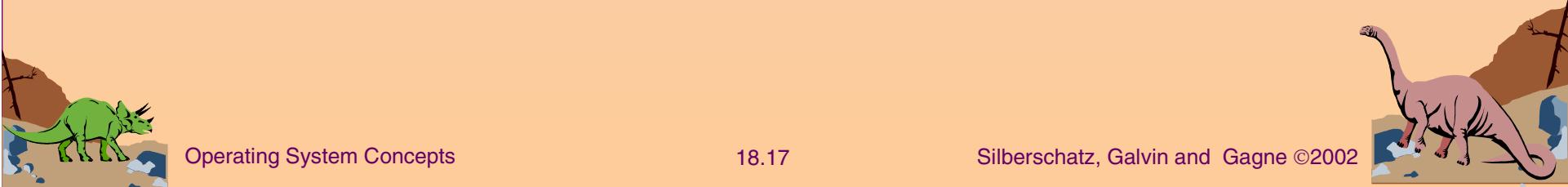
- Cambridge CAP System
  - Data capability - provides standard read, write, execute of individual storage segments associated with object.
  - Software capability -interpretation left to the subsystem, through its protected procedures.
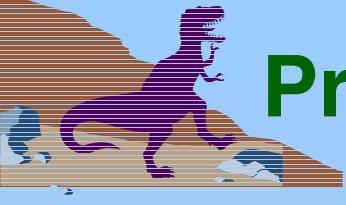
Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.

- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.

- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

Protection in Java 2

- Protection is handled by the Java Virtual Machine (JVM)

- A class is assigned a protection domain when it is loaded by the JVM.

- The protection domain indicates what operations the class can (and cannot) perform.

- If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library.
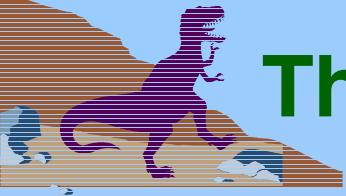
Stack Inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	
	   . . .
	      get(url);
	      open(addr);
	   . . .
		get(URL u):
		
		   . . .
		     doPrivileged {
		       open('proxy.lucent.com:80');
		     }
		    <request u from proxy>
		
		    . . .
			open(Addr a):
			
			   . . .
			      checkPermission(a, connect);
			      connect (a);
			   . . .
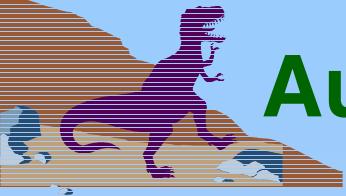
Module 19: Security

- The Security Problem
- Authentication
- Program Threats
- System Threats
- Securing Systems
- Intrusion Detection
- Encryption
- Windows NT

The Security Problem
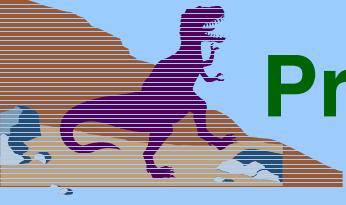
- Security must consider external environment of the system, and protect it from:
    - unauthorized access.
    - malicious modification or destruction
    - accidental introduction of inconsistency.

- Easier to protect against accidental than malicious misuse.

Authentication

- User identity most often established through *passwords*, can be considered a special case of either keys or capabilities.

- Passwords must be kept secret.
  - Frequent change of passwords.
  - Use of "non-guessable" passwords.
  - Log all invalid access attempts.

- Passwords may also either be encrypted or allowed to be used only once.

Program Threats

- Trojan Horse
  - ✦ Code segment that misuses its environment.
  - ✦ Exploits mechanisms for allowing programs written by users to be executed by other users.

- Trap Door
  - ✦ Specific user identifier or password that circumvents normal security procedures.
  - ✦ Could be included in a compiler.

- Stack and Buffer Overflow
  - ✦ Exploits a bug in a program (overflow either the stack or memory buffers.)
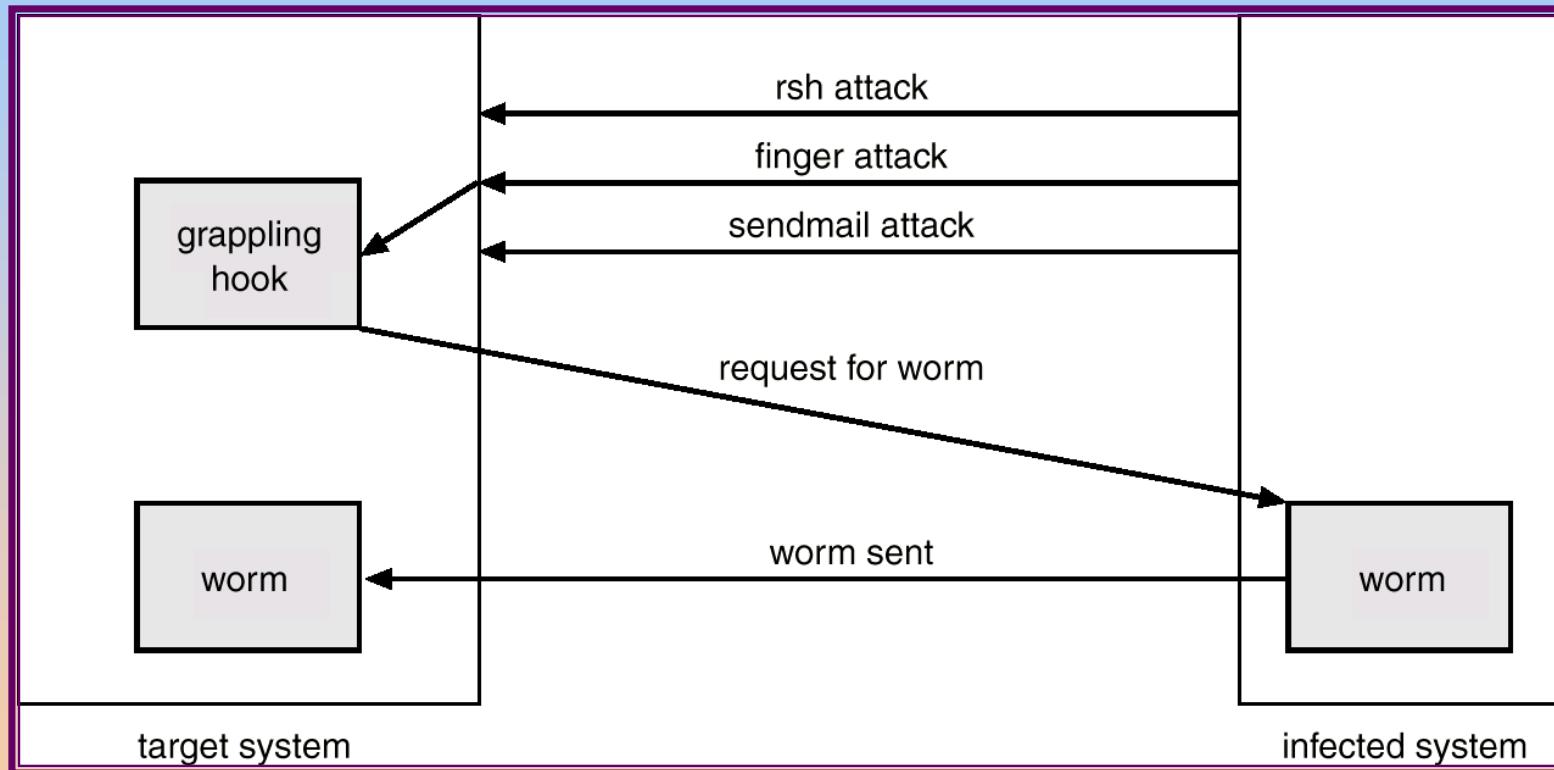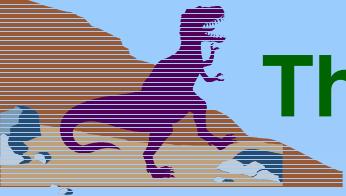
System Threats

- Worms – use spawn mechanism; standalone program
- Internet worm
  - ✦ Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs.
  - ✦ Grappling hook program uploaded main worm program.
- Viruses – fragment of code embedded in a legitimate program.
  - ✦ Mainly effect microcomputer systems.
  - ✦ Downloading viral programs from public bulletin boards or exchanging floppy disks containing an infection.
  - ✦ *Safe computing*.
- Denial of Service
  - ✦ Overload the targeted computer preventing it from doing any sueful work.

The Morris Internet Worm
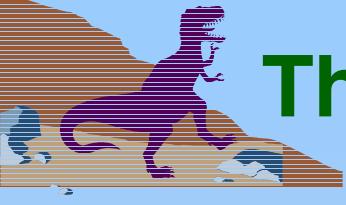
Threat Monitoring

- Check for suspicious patterns of activity – i.e., several incorrect password attempts may signal password guessing.

- Audit log – records the time, user, and type of all accesses to an object; useful for recovery from a violation and developing better security measures.

- Scan the system periodically for security holes; done when the computer is relatively unused.
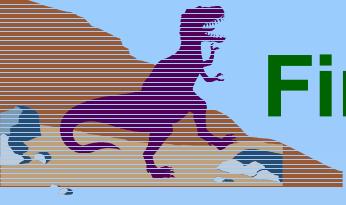
Threat Monitoring (Cont.)
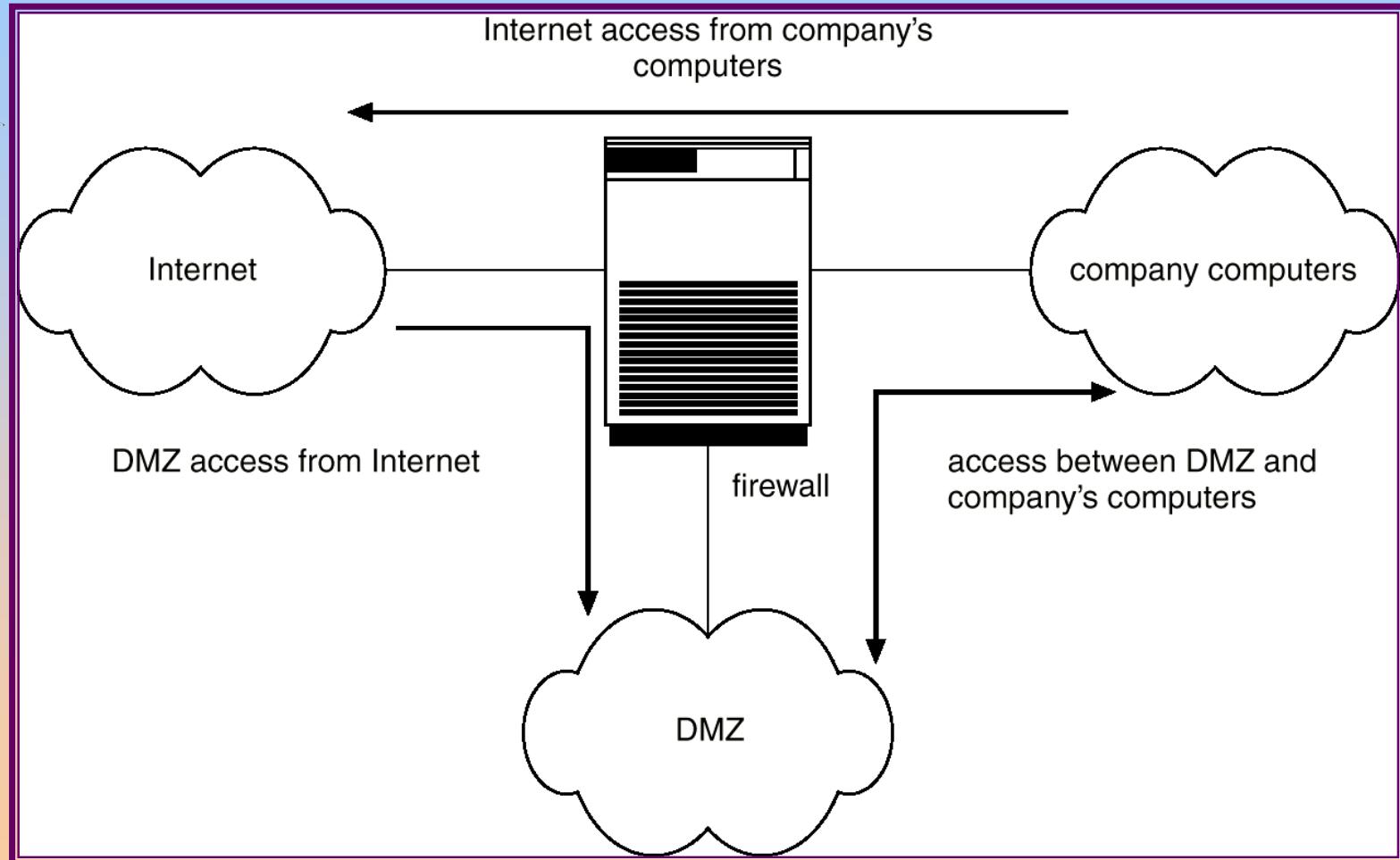
- Check for:
    - ✦ Short or easy-to-guess passwords
    - ✦ Unauthorized set-uid programs
    - ✦ Unauthorized programs in system directories
    - ✦ Unexpected long-running processes
    - ✦ Improper directory protections
    - ✦ Improper protections on system data files
    - ✦ Dangerous entries in the program search path (Trojan horse)
    - ✦ Changes to system programs: monitor checksum values
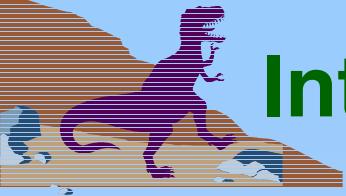
FireWall

- A firewall is placed between trusted and untrusted hosts.

- The firewall limits network access between these two security domains.

Network Security Through Domain Separation Via Firewall

Internet access from company's computers

Internet

DMZ access from Internet

company computers

firewall

access between DMZ and company's computers

DMZ
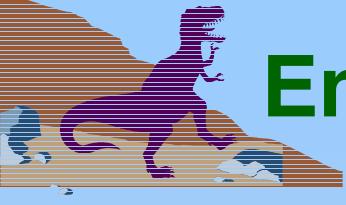
Intrusion Detection

- Detect attempts to intrude into computer systems.

- Detection methods:
  - Auditing and logging.
  - Tripwire (UNIX software that checks if certain files and directories have been altered – I.e. password files)

- System call monitoring
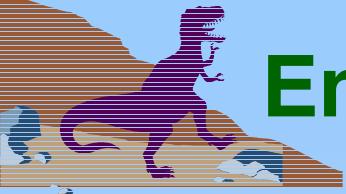
Data Structure Derived From System-Call Sequence

system call	distance = 1	distance = 2	distance = 3
open	read getrlimit	mmap	mmap close
read	mmap	mmap	open
mmap	mmap open close	open getrlimit	getrlimit mmap
getrlimit	mmap	close	
close			
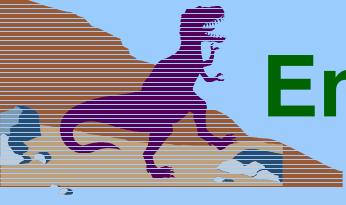
Encryption

- Encrypt clear text into cipher text.

- Properties of good encryption technique:

  - Relatively simple for authorized users to incrypt and decrypt data.

  - Encryption scheme depends not on the secrecy of the algorithm but on a parameter of the algorithm called the encryption key.

  - Extremely difficult for an intruder to determine the encryption key.

- *Data Encryption Standard* substitutes characters and rearranges their order on the basis of an encryption key provided to authorized users via a secure mechanism. Scheme only as secure as the mechanism.
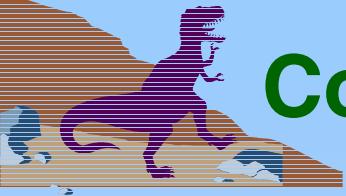
Encryption (Cont.)

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

Encryption Example - SSL

- SSL – Secure Socket Layer

- Cryptographic protocol that limits two computers to only exchange messages with each other.

- Used between web servers and browsers for secure communication (credit card numbers)

- The server is verified with a **certificate**.

- Communication between each computers uses symmetric key cryptography.

Computer Security Classifications

- U.S. Department of Defense outlines four divisions of computer security: **A**, **B**, **C**, and **D**.

- **D** – Minimal security.

- **C** – Provides discretionary protection through auditing. Divided into **C1** and **C2**. **C1** identifies cooperating users with the same level of protection. **C2** allows user-level access control.

- **B** – All the properties of **C**, however each object may have unique sensitivity labels. Divided into **B1**, **B2**, and **B3**.

- **A** – Uses formal design and verification techniques to ensure security.

Windows NT Example

- Configurable security allows policies ranging from D to C2.

- Security is based on user accounts where each user has a security ID.

- Uses a subject model to ensure access security. A subject tracks and manages permissions for each program that a user runs.

- Each object in Windows NT has a security attribute defined by a security descriptor. For example, a file has a security descriptor that indicates the access permissions for all users.
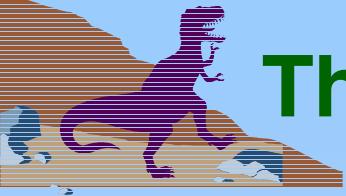
Module 20: The Linux System

- History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling
- Memory Management
- File Systems
- Input and Output
- Interprocess Communication
- Network Structure
- Security
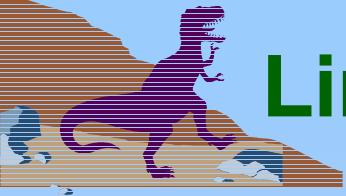
History

- Linux is a modem, free operating system based on UNIX standards.

- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility.

- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet.

- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms.

- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code.
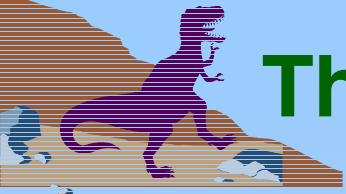
The Linux Kernel

- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system.

- Linux 1.0 (March 1994) included these new features:
  - Support for UNIX's standard TCP/IP networking protocols
  - BSD-compatible socket interface for networking programming
  - Device-driver support for running IP over an Ethernet
  - Enhanced file system
  - Support for a range of SCSI controllers for high-performance disk access
  - Extra hardware support

- Version 1.2 (March 1995) was the final PC-only Linux kernel.
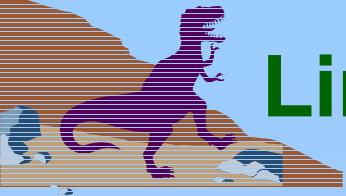
Linux 2.0

- Released in June 1996, 2.0 added two major new capabilities:
    - Support for multiple architectures, including a fully 64-bit native Alpha port.
    - Support for multiprocessor architectures
- Other new features included:
    - Improved memory-management code
    - Improved TCP/IP performance
    - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand.
    - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems.
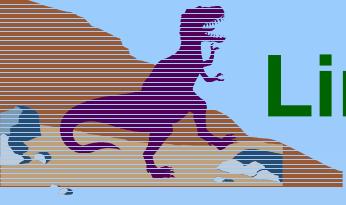
The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.

- The min system libraries were started by the GNU project, with improvements provided by the Linux community.

- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return.

- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories.

Linux Distributions
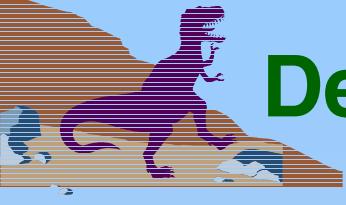
- Standard, precompiled sets of packages, or *distributions*, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools.

- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management.

- Early distributions included SLS and Slackware. *Red Hat* and *Debian* are popular distributions from commercial and noncommercial sources, respectively.

- The RPM Package file format permits compatibility among the various Linux distributions.

Linux Licensing

- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation.

- Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product.
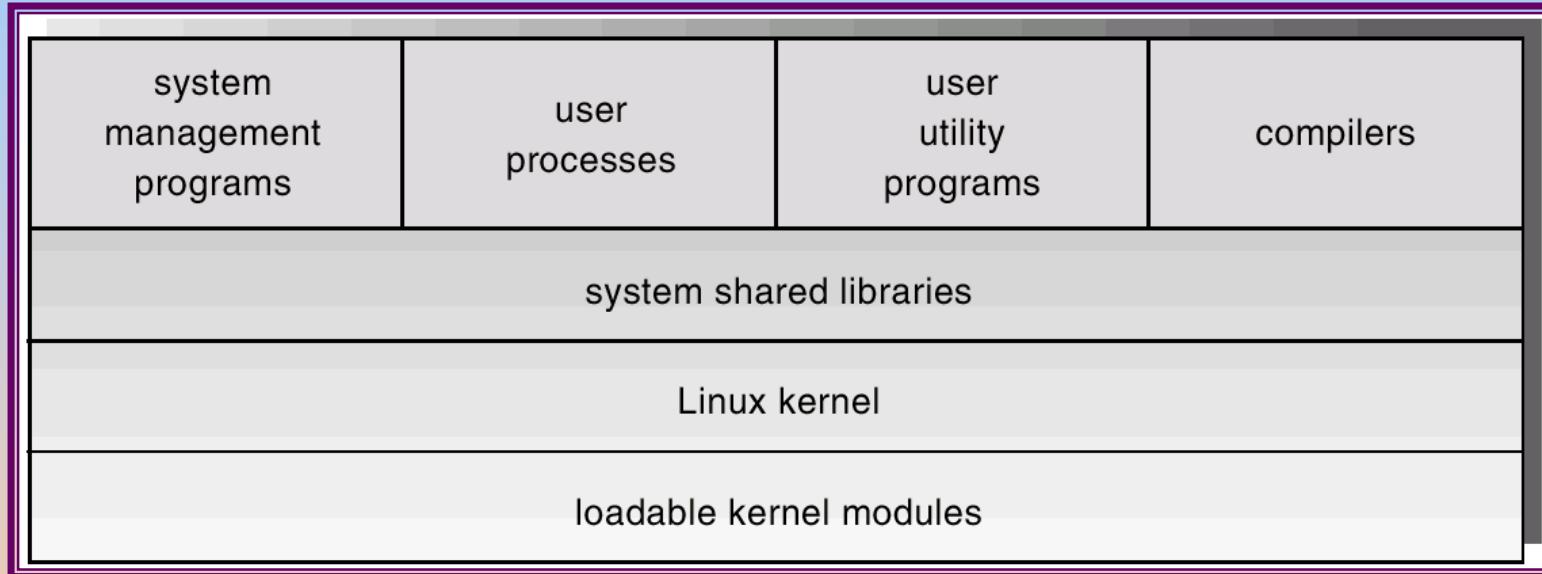
Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools..

- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.

- Main design goals are speed, efficiency, and standardization.

- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.

- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior.

Components of a Linux System

system management programs	user processes	user utility programs	compilers
system shared libraries			
Linux kernel			
loadable kernel modules			
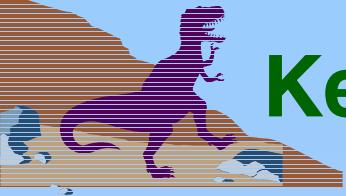
Components of a Linux System (Cont.)

- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components.

- The **kernel** is responsible for maintaining the important abstractions of the operating system.
    - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer.
    - All kernel code and data structures are kept in the same single address space.
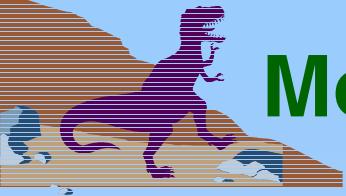
Components of a Linux System (Cont.)

- The **system libraries** define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code.

- The **system utilities** perform individual specialized management tasks.

Kernel Modules
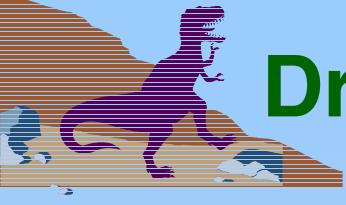
- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.

- A kernel module may typically implement a device driver, a file system, or a networking protocol.

- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.

- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.

- Three components to Linux module support:
  - module management
  - driver registration
  - conflict resolution

Module Management
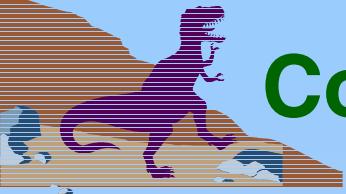
- Supports loading modules into memory and letting them talk to the rest of the kernel.

- Module loading is split into two separate sections:
  - Managing sections of module code in kernel memory
  - Handling symbols that modules are allowed to reference

- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.
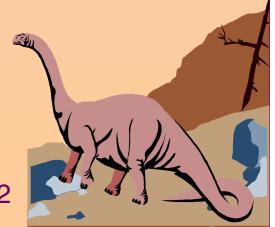
Driver Registration
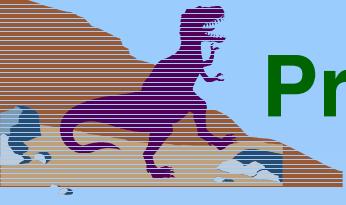
- Allows modules to tell the rest of the kernel that a new driver has become available.

- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.

- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

Conflict Resolution
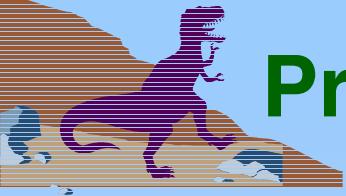
■ A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver

■ The conflict resolution module aims to:

   ✦ Prevent modules from clashing over access to hardware resources

   ✦ Prevent *autoprobes* from interfering with existing device drivers

   ✦ Resolve conflicts with multiple drivers trying to access the same hardware

Process Management
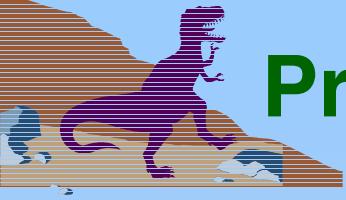
- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - ✦ The **fork** system call creates a new process.
  - ✦ A new program is run after a call to **execve**.
- Under UNIX, a process encompasses all the information that the operating system must maintain *t* track the context of a single execution of a single program.
- Under Linux, process properties fall into three groups: the process's identity, environment, and context.

Process Identity
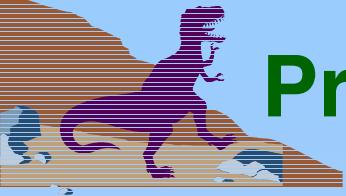
- **Process ID** (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process.

- **Credentials**. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files.

- **Personality**. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls.
Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX.
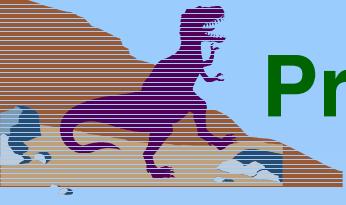
Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
  - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.

- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software.

- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.
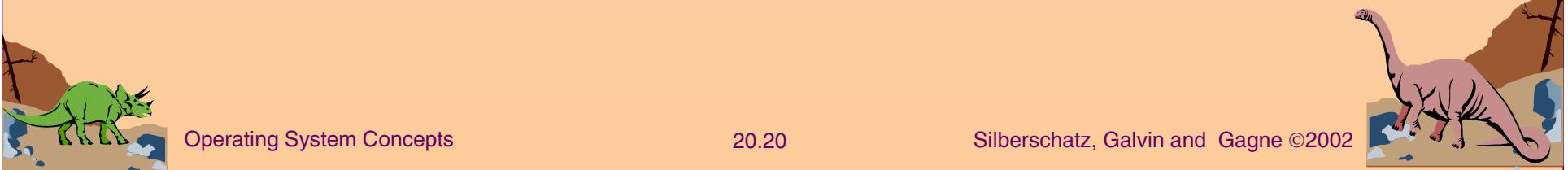
Process Context
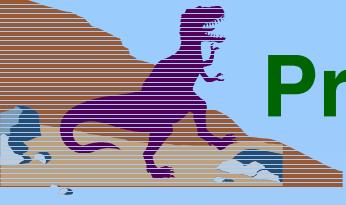
- The (constantly changing) state of a running program at any point in time.

- The **scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.

- The kernel maintains **accounting** information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.

- The **file table** is an array of pointers to kernel file structures. When making file I/O system calls, processes refer to files by their index into this table.

Process Context (Cont.)

- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.

- The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive.

- The **virtual-memory context** of a process describes the full contents of the its private address space.

Processes and Threads
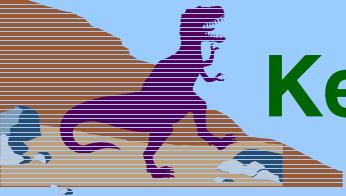
- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.

- A distinction is only made when a new thread is created by the **clone** system call.

  - **fork** creates a new process with its own entirely new process context

  - **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent

- Using **clone** gives an application fine-grained control over exactly what is shared between two threads.

Scheduling
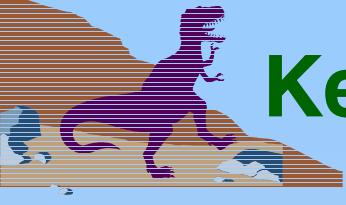
- The job of allocating CPU time to different tasks within an operating system.

- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks.

- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

Kernel Synchronization
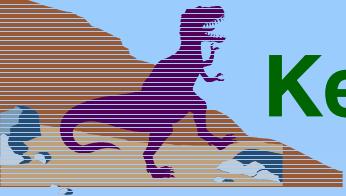
- A request for kernel-mode execution can occur in two ways:
    - ✦ A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs.
    - ✦ A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section.

Kernel Synchronization (Cont.)

- Linux uses two techniques to protect critical sections:

  1. Normal kernel code is nonpreemptible
     - when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode.

  2. The second technique applies to critical sections that occur in an interrupt service routines.

     - By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.
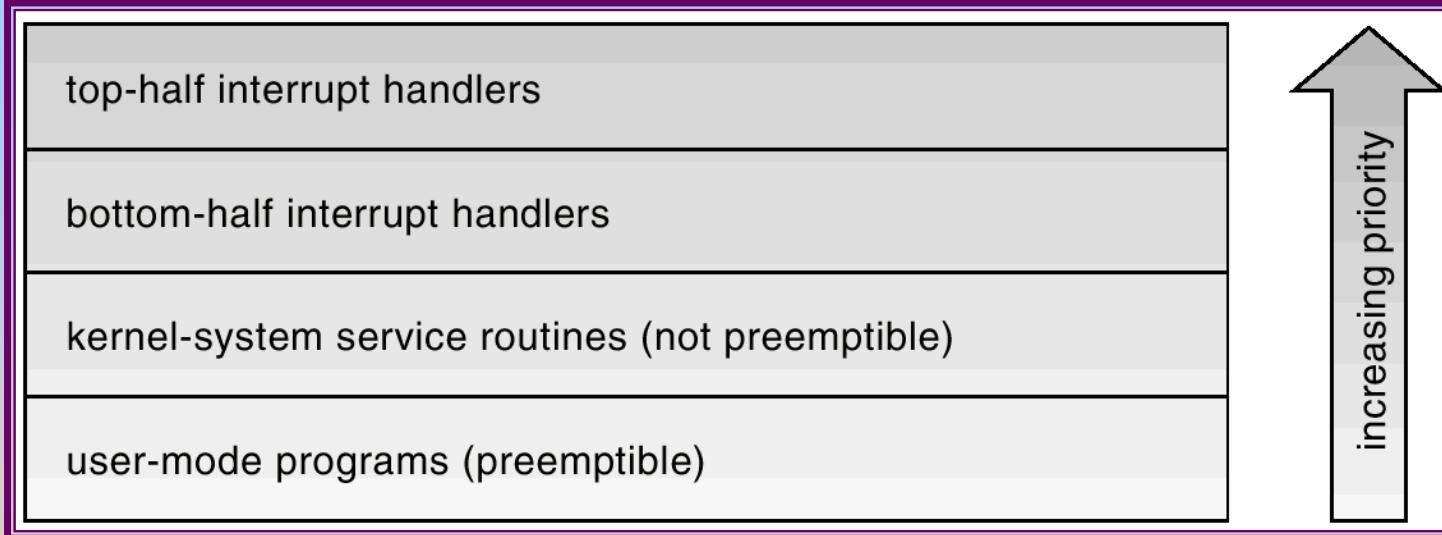
Kernel Synchronization (Cont.)

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration.

- Interrupt service routines are separated into a *top half* and a *bottom half.*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled.
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves.
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code.

Interrupt Protection Levels

top-half interrupt handlers	
bottom-half interrupt handlers	increasing priority ↑
kernel-system service routines (not preemptible)	
user-mode programs (preemptible)	

- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level.

- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.
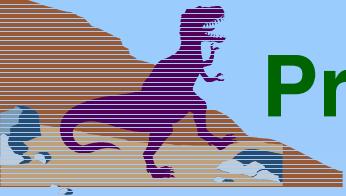
Process Scheduling

- Linux uses two process-scheduling algorithms:
  - ✦ A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - ✦ A real-time algorithm for tasks where absolute priorities are more important than fairness
- A process's scheduling class defines which algorithm to apply.
- For time-sharing processes, Linux uses a prioritized, credit based algorithm.
  - ✦ The crediting rule

$$\text{credits} := \frac{\text{credits}}{2} + \text{priority}$$

  factors in both the process's history and its priority.
  - ✦ This crediting system automatically prioritizes interactive or I/O-bound processes.

Process Scheduling (Cont.)
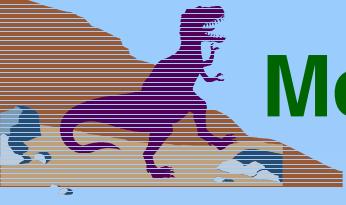
- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class.
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the process waiting the longest
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robing processes of equal priority automatically time-share between themselves.

Symmetric Multiprocessing
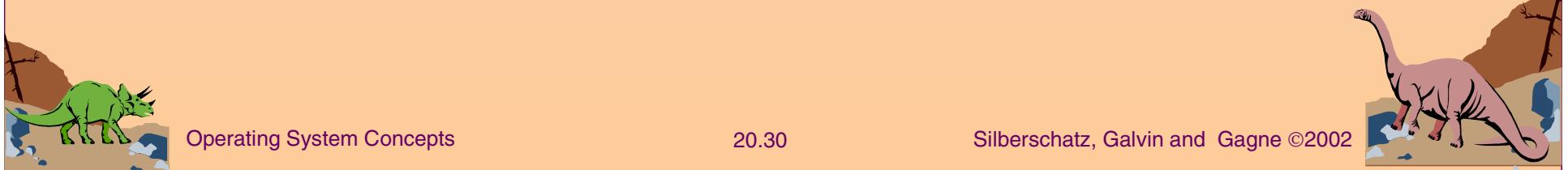
- Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors.

- To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.
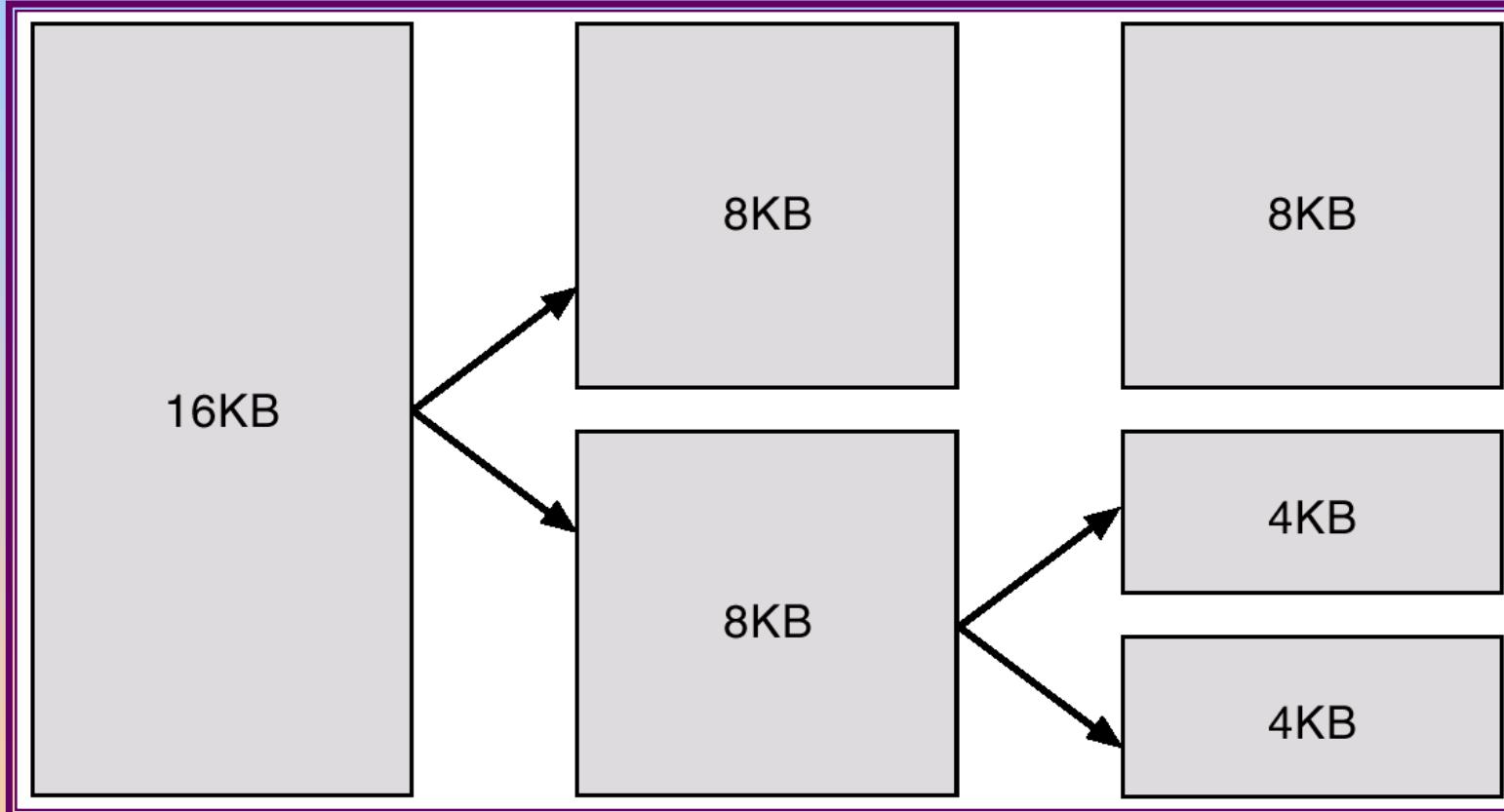
Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory.

- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes.
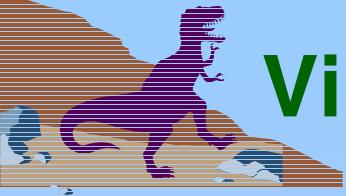
Splitting of Memory in a Buddy Heap
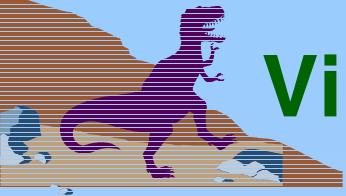
Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request.

- The allocator uses a *buddy-heap* algorithm to keep track of available physical pages.

  - Each allocatable memory region is paired with an adjacent partner.

  - Whenever two allocated partner regions are both freed up they are combined to form a larger region.

  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request.

- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator).
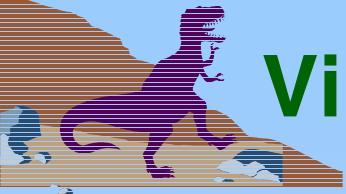
Virtual Memory

■ The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required.

■ The VM manager maintains two separate views of a process's address space:

✦ A logical view describing instructions concerning the layout of the address space.
The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space.

✦ A physical view of each address space which is stored in the hardware page tables for the process.
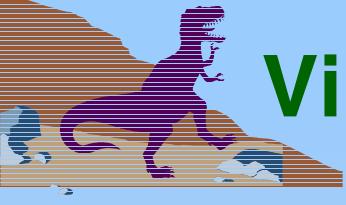
Virtual Memory (Cont.)

- Virtual memory regions are characterized by:
  - ✦ The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
  - ✦ The region's reaction to writes (page sharing or copy-on-write).

- The kernel creates a new virtual address space
  1. When a process runs a new program with the **exec** system call
  2. Upon creation of a new process by the **fork** system call
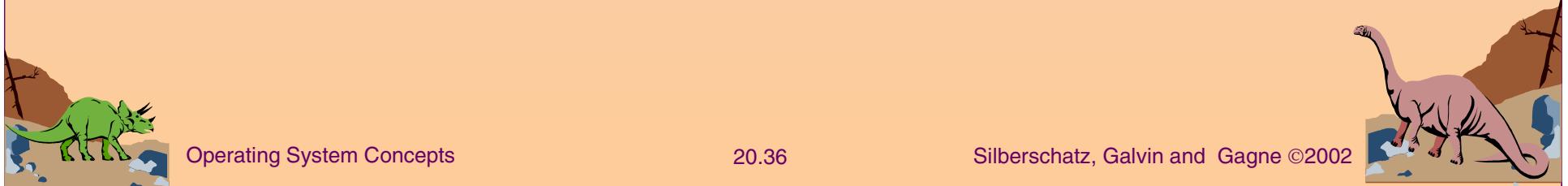
Virtual Memory (Cont.)
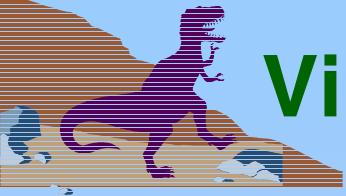
- On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions.

- Creating a new process with **fork** involves creating a complete copy of the existing process's virtual address space.

  - ✦ The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child.

  - ✦ The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented.

  - ✦ After the fork, the parent and child share the same physical pages of memory in their address spaces.

Virtual Memory (Cont.)

- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else.

- The VM paging system can be divided into two sections:
  - The pageout-policy algorithm decides which pages to write out to disk, and when.
  - The paging mechanism actually carries out the transfer, and pages data back into physical memory as needed.

Virtual Memory (Cont.)

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use.

- This kernel virtual-memory area contains two regions:
  - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code.
  - The reminder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory.
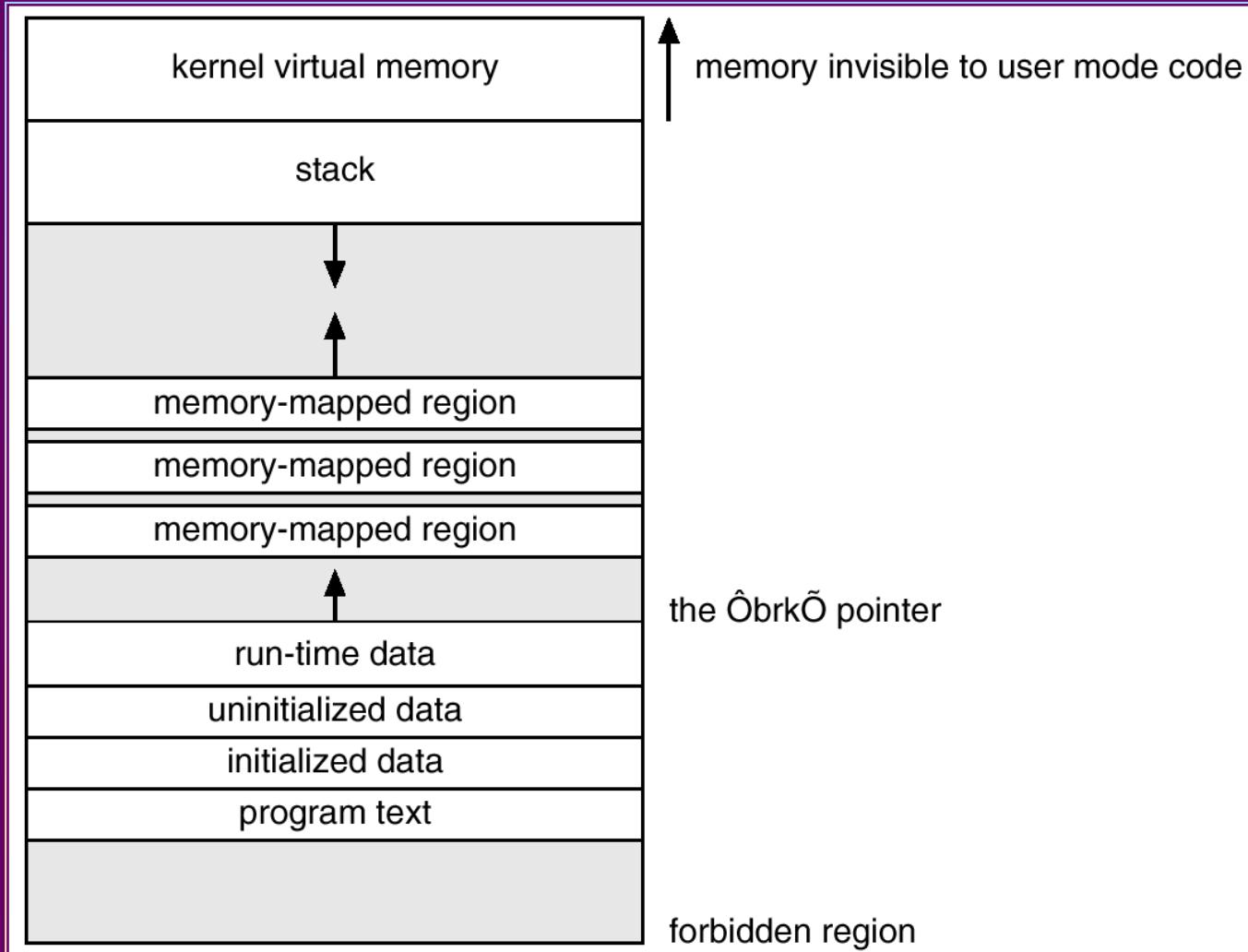
Executing and Loading User Programs

- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made.

- The registration of multiple loader routines allows Linux to support both the ELF and **a.out** binary formats.

- Initially, binary-file pages are mapped into virtual memory; only when a program tries to access a given page will a page fault result in that page being loaded into physical memory.

- An ELF-format binary file consists of a header followed by several page-aligned sections; the ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.
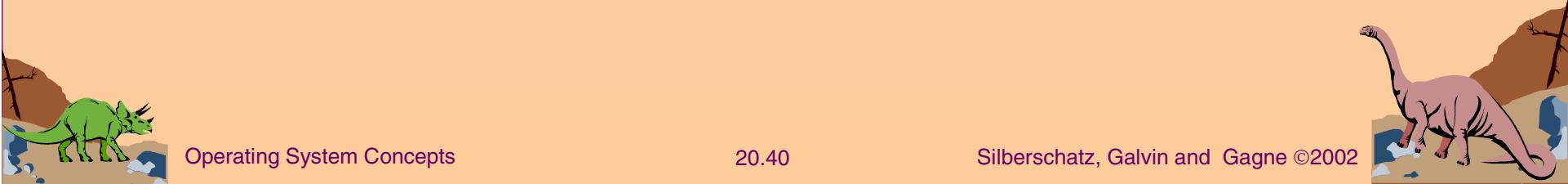
Memory Layout for ELF Programs

kernel virtual memory	memory invisible to user mode code
stack	
↓	
↑	
memory-mapped region	
memory-mapped region	
memory-mapped region	
↑	the ÔbrkÕ pointer
run-time data	
uninitialized data	
initialized data	
program text	
	forbidden region
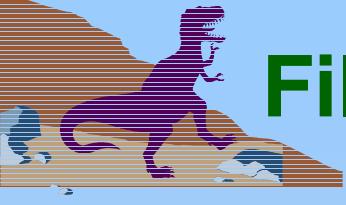
Static and Dynamic Linking
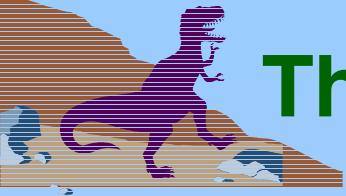
- A program whose necessary library functions are embedded directly in the program's executable binary file is *statically* linked to its libraries.

- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions.

- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once.

File Systems
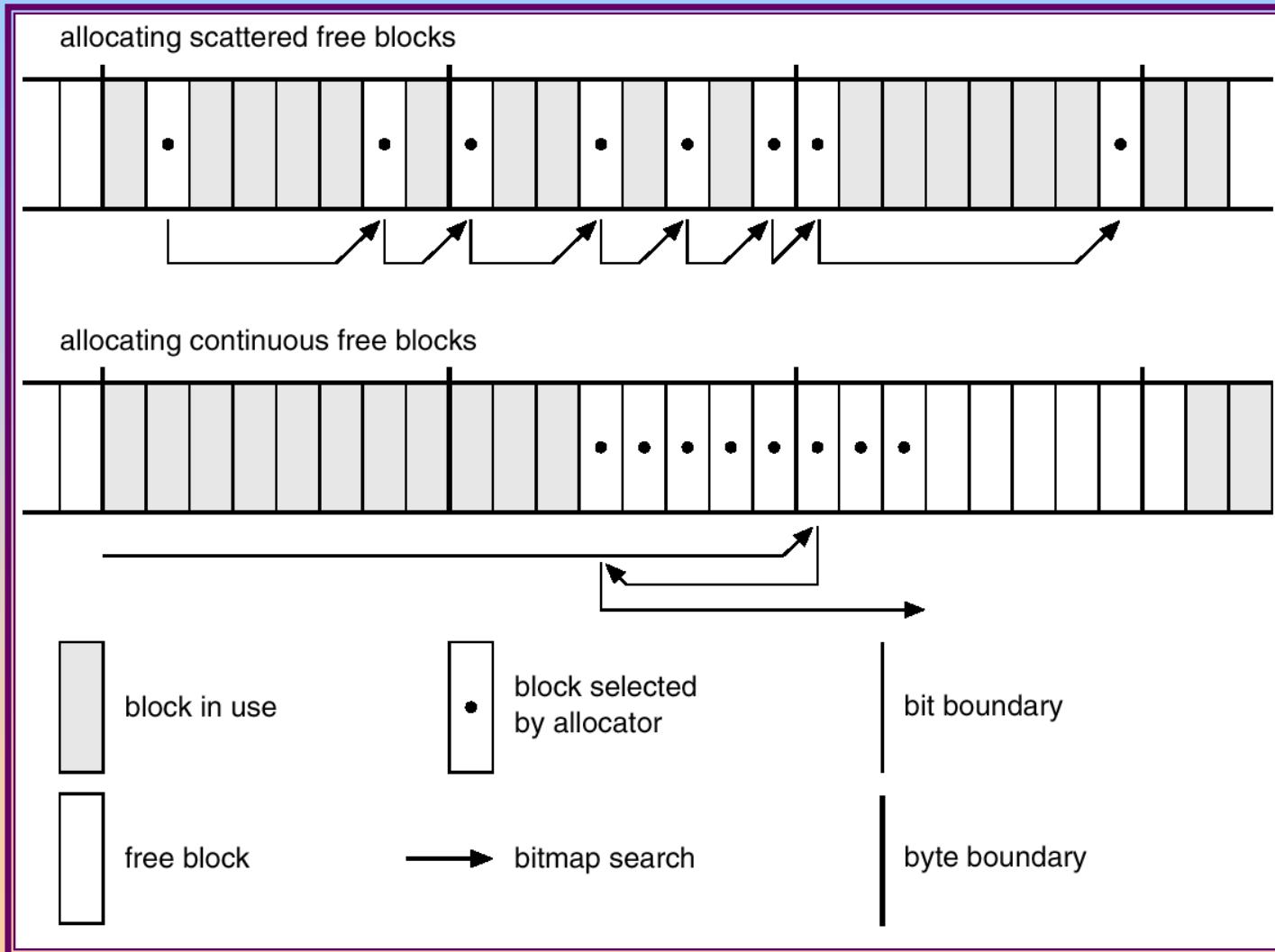
- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics.

- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*.

- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - The *inode-object* and the *file-object* structures represent individual files
    - the *file system object* represents an entire file system
  - A layer of software to manipulate those objects.
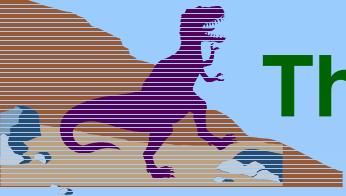
The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file.

- The main differences between ext2fs and ffs concern their disk allocation policies.

    - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file.

    - Ext2fs does not use fragments; it performs its allocations in smaller units. The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.

    - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.
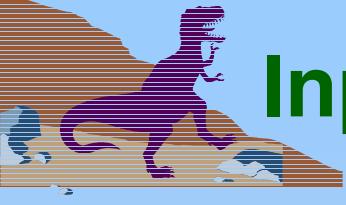
Ext2fs Block-Allocation Policies

allocating scattered free blocks

allocating continuous free blocks

| | block in use | | block selected by allocator | | bit boundary |
| | free block | | bitmap search | | byte boundary |
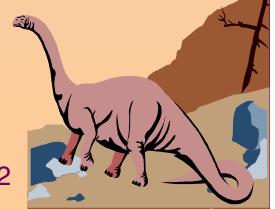
The Linux Proc File System

- The **proc** file system does not store data, rather, its contents are computed on demand according to user file I/O requests.

- **proc** must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains.

    - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode.

    - When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer.
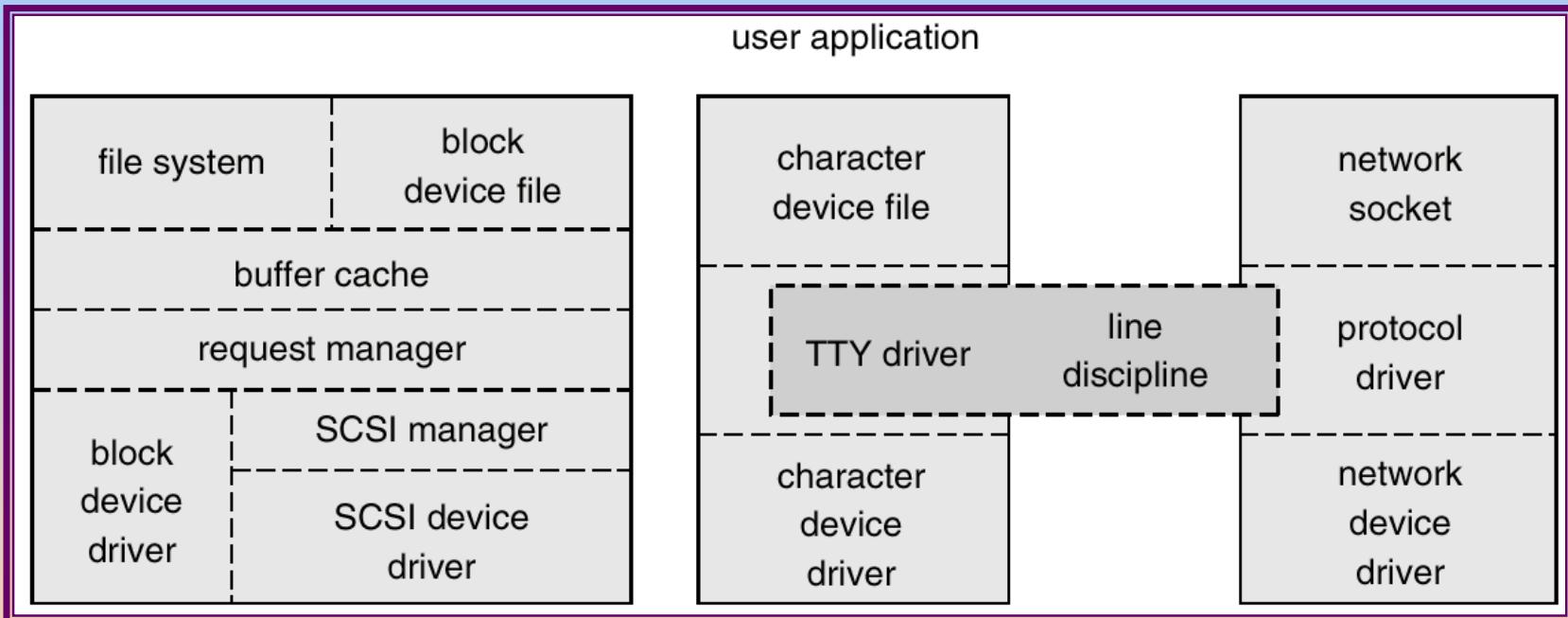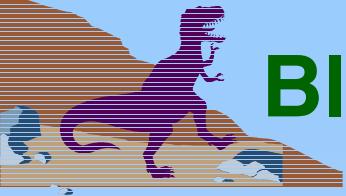
Input and Output

- The Linux device-oriented file system accesses disk storage through two caches:
  - ✦ Data is cached in the page cache, which is unified with the virtual memory system
  - ✦ Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.

- Linux splits all devices into three classes:
  - ✦ *block devices* allow random access to completely independent, fixed size blocks of data
  - ✦ *character devices* include most other devices; they don't need to support the functionality of regular files.
  - ✦ *network devices* are interfaced via the kernel's networking subsystem
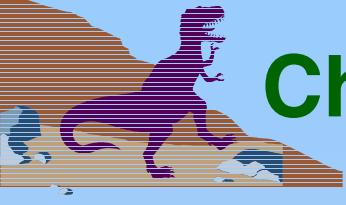
Device-Driver Block Structure

user application

file system	block device file
buffer cache	
request manager	
block device driver	SCSI manager
	SCSI device driver

character device file
TTY driver   line discipline
character device driver

network socket
protocol driver
network device driver
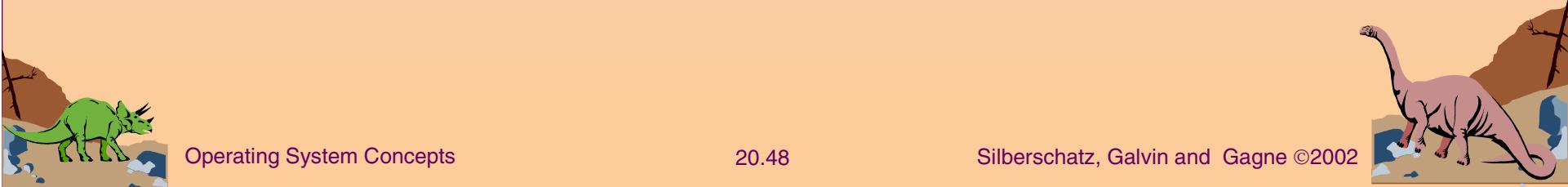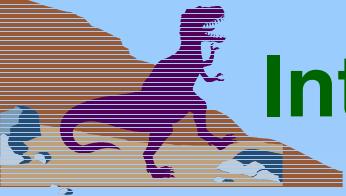
Block Devices

- Provide the main interface to all disk devices in a system.

- The *block buffer* cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O

- The *request manager* manages the reading and writing of buffer contents to and from a block device driver.
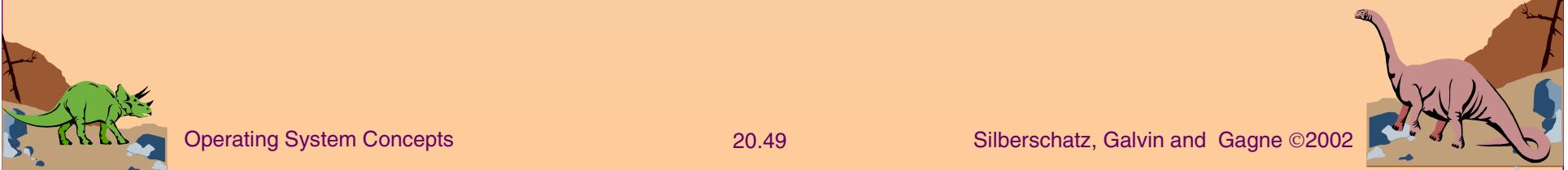
Character Devices

- A device driver which does not offer random access to fixed blocks of data.

- A character device driver must register a set of functions which implement the driver's various file I/O operations.

- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device.

- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface.
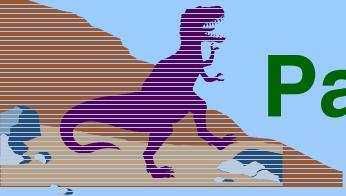
Interprocess Communication
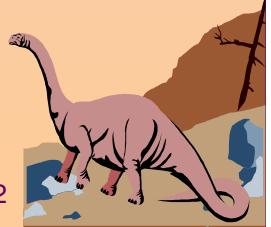
- Like UNIX, Linux informs processes that an event has occurred via signals.

- There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process.

- The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and **wait.queue** structures.

Passing Data Between Processes
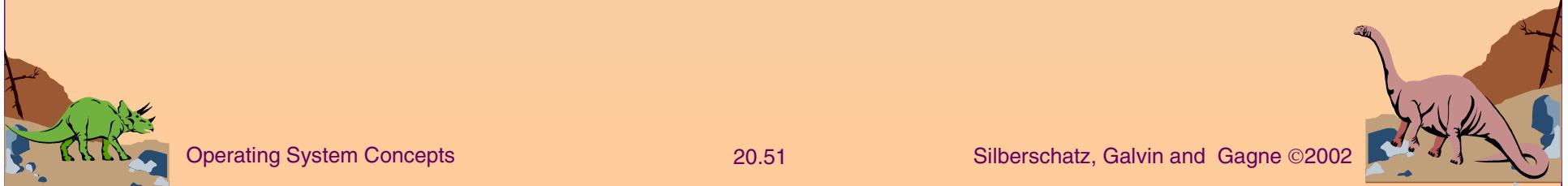
- The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read a the other.

- Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space.

- To obtain synchronization, however, shared memory must be used in conjunction with another Interprocess-communication mechanism.
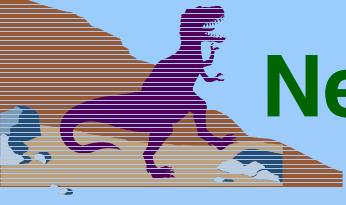
Shared Memory Object
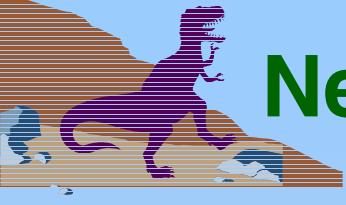
- The shared-memory object acts as a backing store for shared-memory regions in the same way as a file can act as backing store for a memory-mapped memory region.

- Shared-memory mappings direct page faults to map in pages from a persistent shared-memory object.

- Shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

Network Structure

- Networking is a key area of functionality for Linux.
    - It supports the standard Internet protocols for UNIX to UNIX communications.
    - It also implements protocols native to nonUNIX operating systems, in particular, protocols used on PC networks, such as Appletalk and IPX.

- Internally, networking in the Linux kernel is implemented by three layers of software:
    - The socket interface
    - Protocol drivers
    - Network device drivers

Network Structure (Cont.)

■ The most important set of protocols in the Linux networking system is the internet protocol suite.

✦ It implements routing between different hosts anywhere on the network.

✦ On top of the routing protocol are built the UDP, TCP and ICMP protocols.

Security
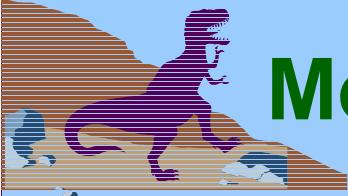
- The *pluggable authentication modules (PAM)* system is available under Linux.

- PAM is based on a shared library that can be used by any system component that needs to authenticate users.

- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**).

- Access control is performed by assigning objects a *protections mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access.

Security (Cont.)

- Linux augments the standard UNIX **setuid** mechanism in two ways:
    - It implements the POSIX specification's saved *user-id* mechanism, which allows a process to repeatedly drop and reacquire its effective uid.
    - It has added a process characteristic that grants just a subset of the rights of the effective uid.

- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process without granting it any other privileges.
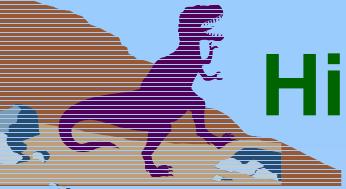
Module 21: Windows 2000

- History
- Design Principles
- System Components
- Environmental Subsystems
- File system
- Networking
- Programmer Interface

Windows 2000

- 32-bit preemptive multitasking operating system for Intel microprocessors.
- Key goals for the system:
  - ☞ portability
  - ☞ security
  - ☞ POSIX compliance
  - ☞ multiprocessor support
  - ☞ extensibility
  - ☞ international support
  - ☞ compatibility with MS-DOS and MS-Windows applications.
- Uses a micro-kernel architecture.
- Available in four versions, Professional, Server, Advanced Server, National Server.
- In 1996, more NT server licenses were sold than UNIX licenses

History

- In 1988, Microsoft decided to develop a "new technology" (NT) portable operating system that supported both the OS/2 and POSIX APIs.

- Originally, NT was supposed to use the OS/2 API as its native environment but during development NT was changed to use the Win32 API, reflecting the popularity of Windows 3.0.

Design Principles

- Extensibility — layered architecture.
  - ☞ Executive, which runs in protected mode, provides the basic system services.
  - ☞ On top of the executive, several server subsystems operate in user mode.
  - ☞ Modular structure allows additional environmental subsystems to be added without affecting the executive.
- Portability — 2000 can be moved from on hardware architecture to another with relatively few changes.
  - ☞ Written in C and C++.
  - ☞ Processor-dependent code is isolated in a dynamic link library (DLL) called the "hardware abstraction layer" (HAL).

Design Principles (Cont.)
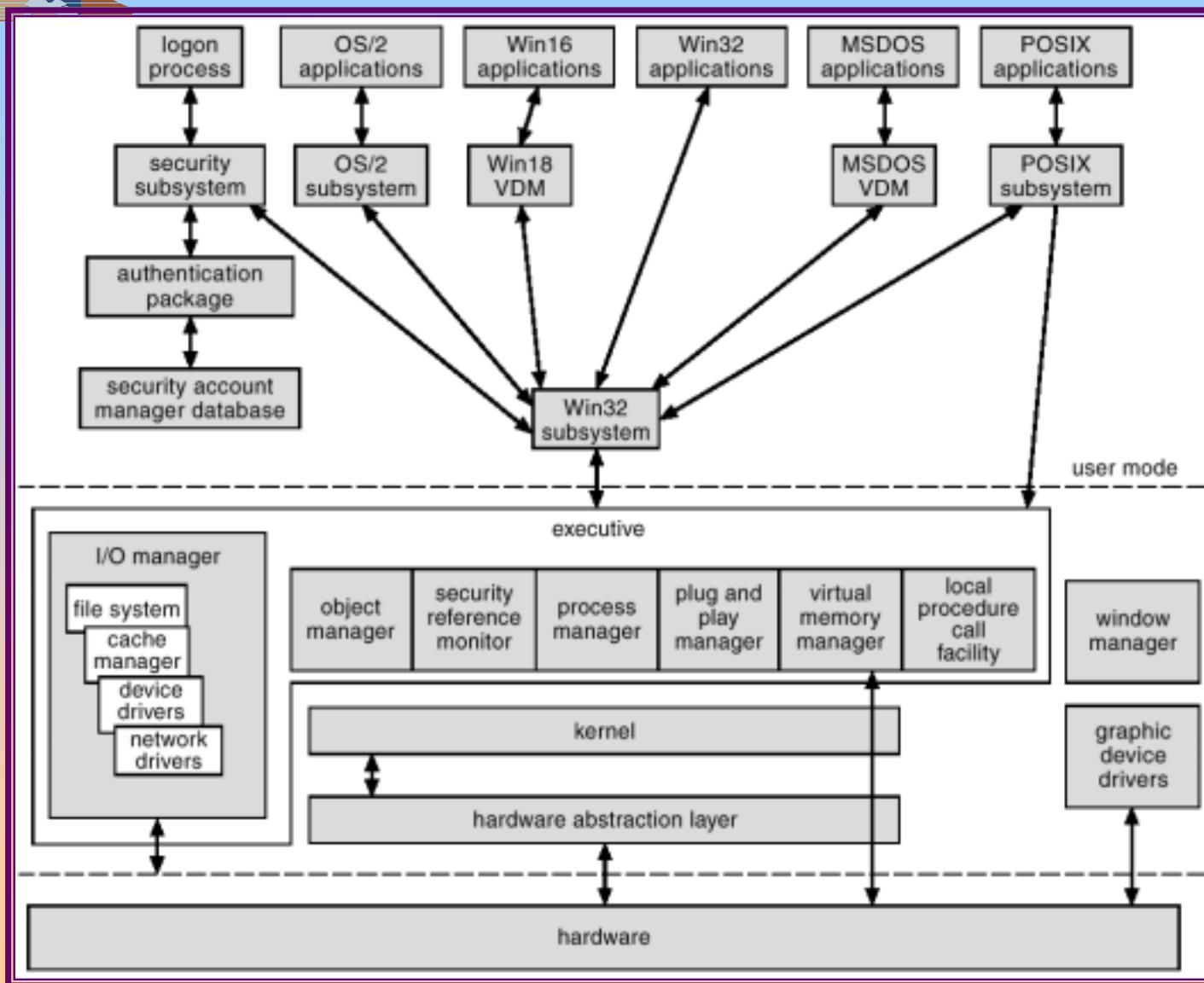
- Reliability — 2000 uses hardware protection for virtual memory, and software protection mechanisms for operating system resources.

- Compatibility — applications that follow the IEEE 1003.1 (POSIX) standard can be complied to run on 2000 without changing the source code.

- Performance — 2000 subsystems can communicate with one another via high-performance message passing.
  - ☞ Preemption of low priority threads enables the system to respond quickly to external events.
  - ☞ Designed for symmetrical multiprocessing

- International support — supports different locales via the national language support (NLS) API.

2000 Architecture

- Layered system of modules.

- Protected mode — HAL, kernel, executive.

- User mode — collection of subsystems
  - Environmental subsystems emulate different operating systems.
  - Protection subsystems provide security functions.
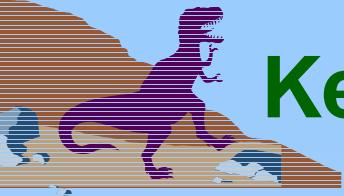
Depiction of 2000 Architecture
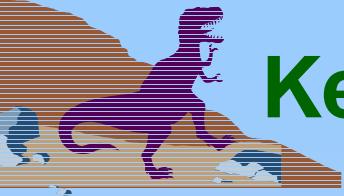
System Components — Kernel

- Foundation for the executive and the subsystems.
- Never paged out of memory; execution is never preempted.
- Four main responsibilities:
  - ☞ thread scheduling
  - ☞ interrupt and exception handling
  - ☞ low-level processor synchronization
  - ☞ recovery after a power failure
- Kernel is object-oriented, uses two sets of objects.
  - ☞ *dispatcher objects* control dispatching and synchronization (events, mutants, mutexes, semaphores, threads and timers).
  - ☞ *control objects* (asynchronous procedure calls, interrupts, power notify, power status, process and profile objects.)
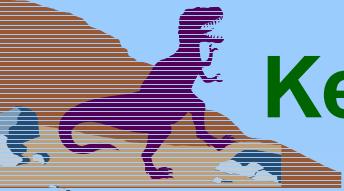
Kernel — Process and Threads

- The process has a virtual memory address space, information (such as a base priority), and an affinity for one or more processors.

- Threads are the unit of execution scheduled by the kernel's dispatcher.

- Each thread has its own state, including a priority, processor affinity, and accounting information.

- A thread can be one of six states: *ready, standby, running, waiting, transition*, and *terminated*.

Kernel — Scheduling

- The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes..
  - The real-time class contains threads with priorities ranging from 16 to 31.
  - The variable class contains threads having priorities from 0 to 15.
- Characteristics of 2000's priority strategy.
  - Trends to give very good response times to interactive threads that are using the mouse and windows.
  - Enables I/O-bound threads to keep the I/O devices busy.
  - Complete-bound threads soak up the spare CPU cycles in the background.

Kernel — Scheduling (Cont.)

- Scheduling can occur when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity.

- Real-time threads are given preferential access to the CPU; but 2000 does not guarantee that a real-time thread will start to execute within any particular time limit. (This is known as *soft realtime*.)

Windows 2000 Interrupt Request Levels

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3-26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive
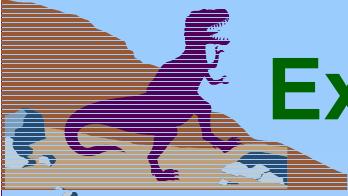
Kernel — Trap Handling

- The kernel provides trap handling when exceptions and interrupts are generated by hardware of software.

- Exceptions that cannot be handled by the trap handler are handled by the kernel's *exception dispatcher*.

- The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (such as in a device driver) or an internal kernel routine.

- The kernel uses spin locks that reside in global memory to achieve multiprocessor mutual exclusion.

Executive — Object Manager

- 2000 uses objects for all its services and entities; the object manger supervises the use of all the objects.
  - ☞ Generates an object *handle*
  - ☞ Checks security.
  - ☞ Keeps track of which processes are using each object.
- Objects are manipulated by a standard set of methods, namely `create, open, close, delete, query name, parse` and `security.`

Executive — Naming Objects
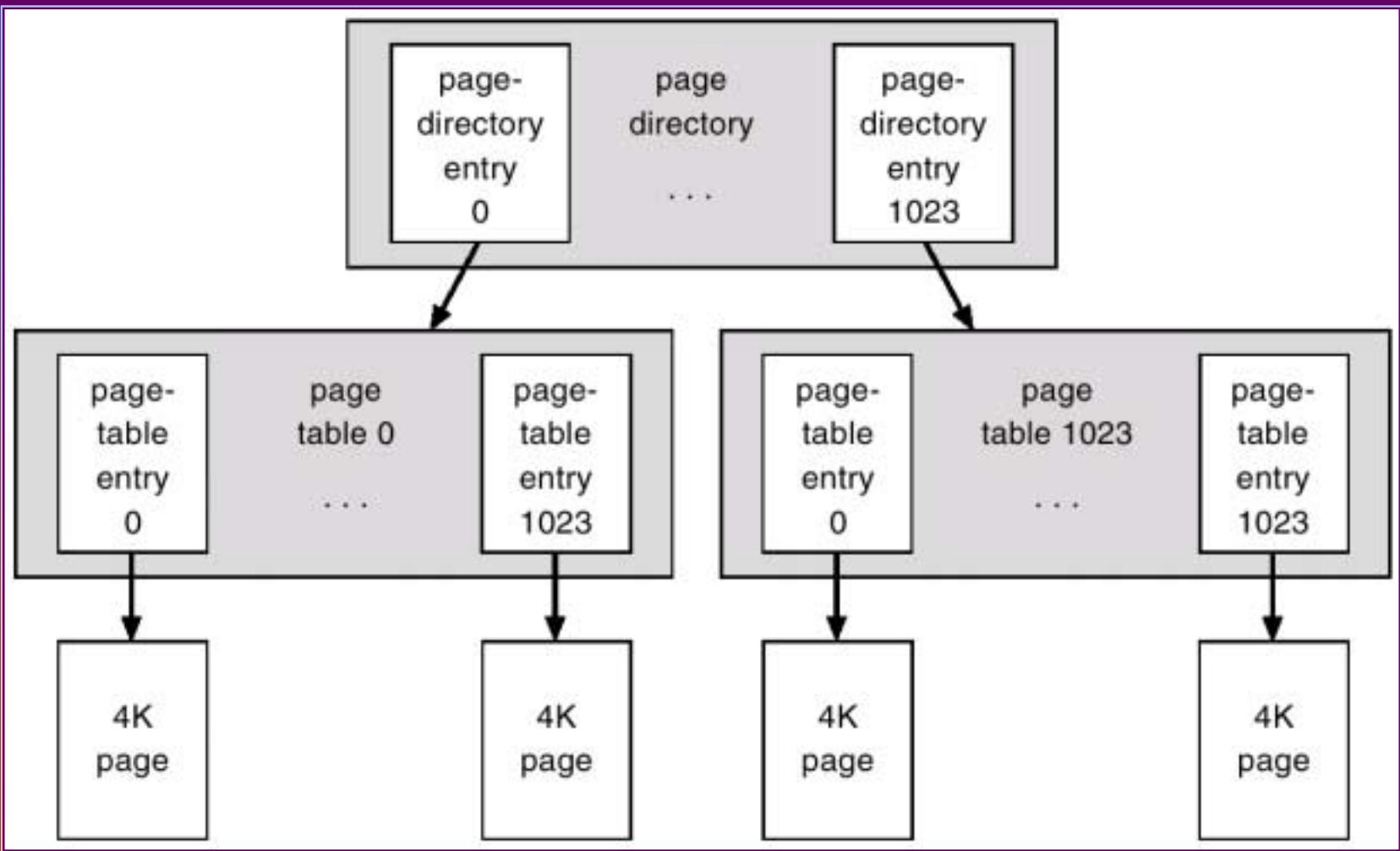
- The 2000 executive allows any object to be given a name, which may be either permanent or temporary.

- Object names are structured like file path names in MS-DOS and UNIX.

- 2000 implements a *symbolic link object*, which is similar to *symbolic links* in UNIX that allow multiple nicknames or aliases to refer to the same file.

- A process gets an object handle by creating an object by opening an existing one, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process.

- Each object is protected by an access control list.

Executive — Virtual Memory Manager

- The design of the VM manager assumes that the underlying hardware supports virtual to physical mapping a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual addressing aliasing.

- The VM manager in 2000 uses a page-based management scheme with a page size of 4 KB.

- The 2000 VM manager uses a two step process to allocate memory.

  - ☞ The first step reserves a portion of the process's address space.

  - ☞ The second step commits the allocation by assigning space in the 2000 paging file.

Virtual-Memory Layout
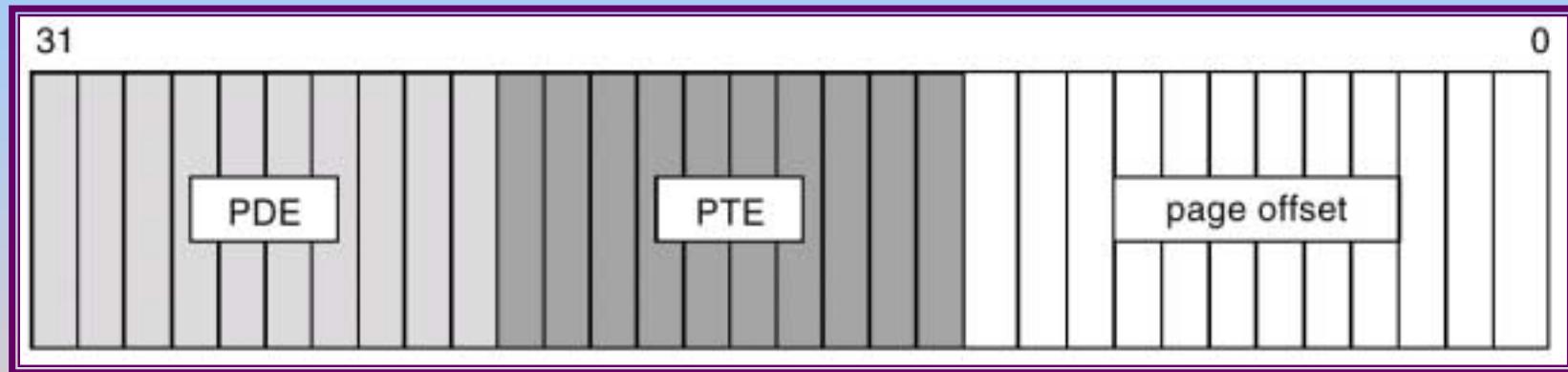
Virtual Memory Manager (Cont.)
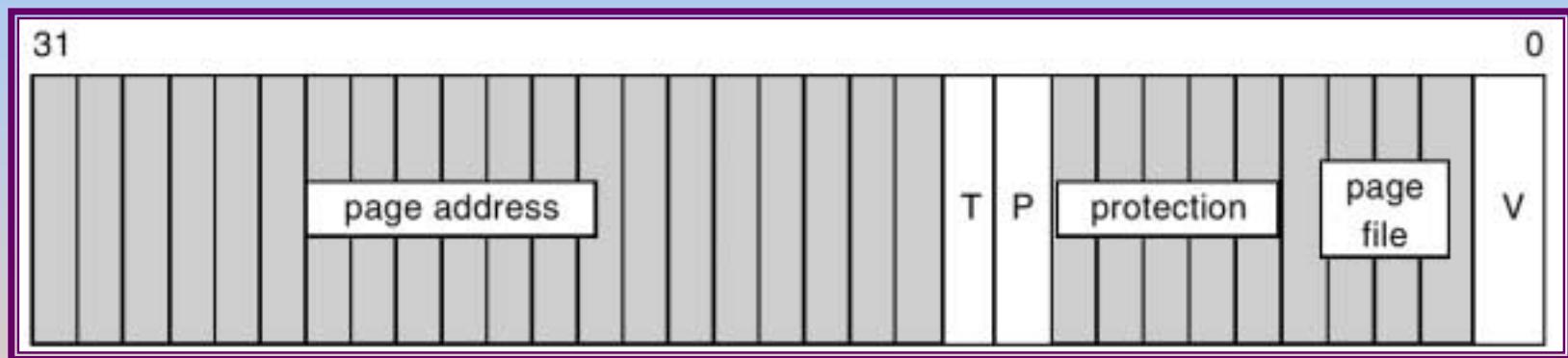
- The virtual address translation in 2000 uses several data structures.

  - ☞ Each process has a *page directory* that contains 1024 *page directory entries* of size 4 bytes.

  - ☞ Each page directory entry points to a *page table* which contains 1024 *page table entries* (PTEs) of size 4 bytes.

  - ☞ Each PTE points to a 4 KB *page frame* in physical memory.

- A 10-bit integer can represent all the values form 0 to 1023, therefore, can select any entry in the page directory, or in a page table.

- This property is used when translating a virtual address pointer to a bye address in physical memory.

- A page can be in one of six states: valid, zeroed, free standby, modified and bad.

Virtual-to-Physical Address Translation
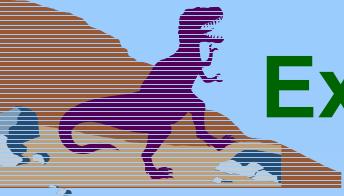
- 10 bits for page directory entry, 20 bits for page table entry, and 12 bits for byte offset in page.

Page File Page-Table Entry


```
31                                                                    0
┌──────────────────────────────┬─┬─┬──────────┬──────┬─┐
│         page address         │T│P│protection│ page │V│
│                              │ │ │          │ file │ │
└──────────────────────────────┴─┴─┴──────────┴──────┴─┘
```

■ 5 bits for page protection, 20 bits for page frame address, 4 bits to select a paging file, and 3 bits that describe the page state. V = 0

Executive — Process Manager

- Provides services for creating, deleting, and using threads and processes.

- Issues such as parent/child relationships or process hierarchies are left to the particular environmental subsystem that owns the process.

Executive — Local Procedure Call Facility
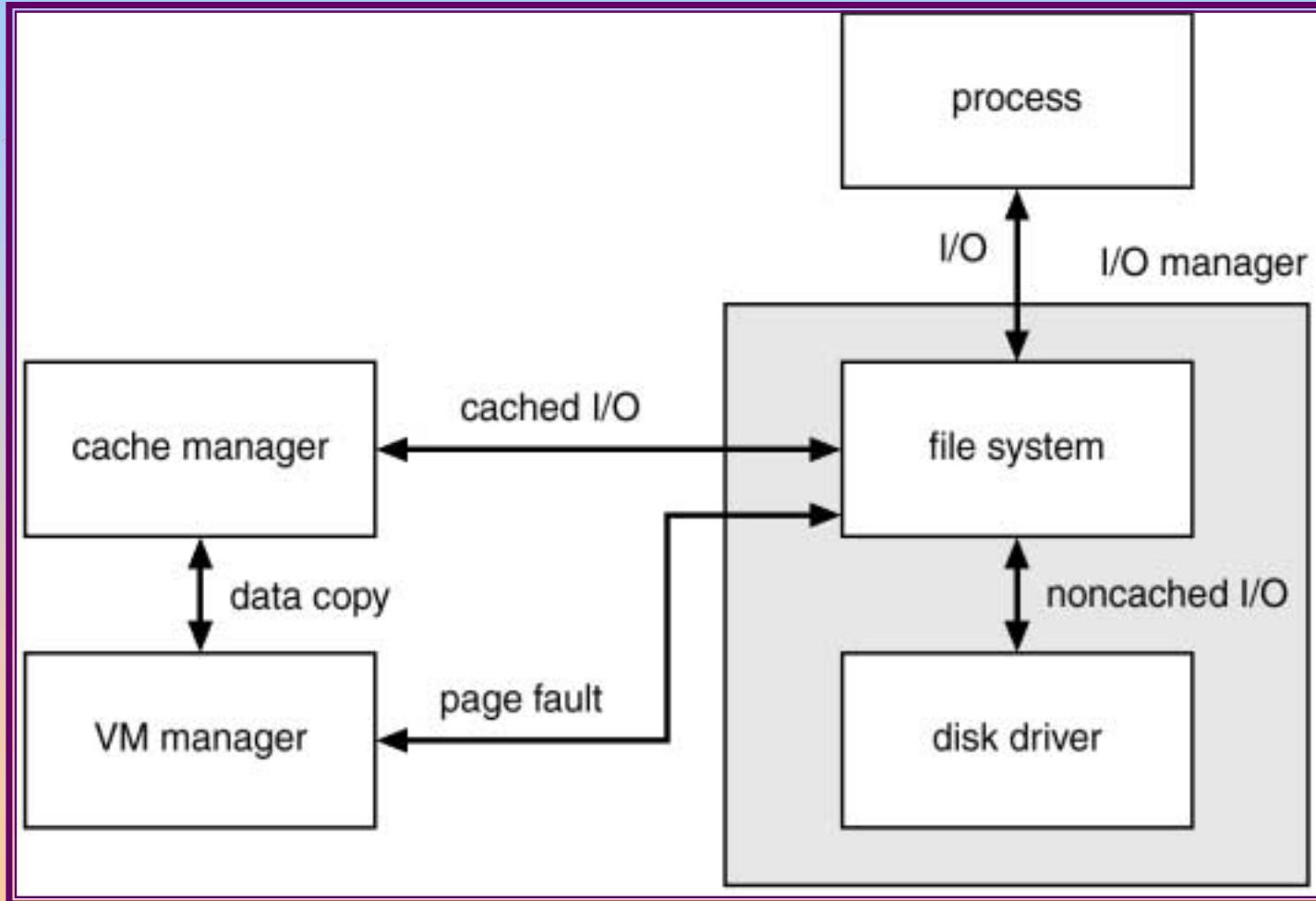
- The LPC passes requests and results between client and server processes within a single machine.

- In particular, it is used to request services from the various 2000 subsystems.

- When a LPC channel is created, one of three types of message passing techniques must be specified.

  - ☞ First type is suitable for small messages, up to 256 bytes; port's message queue is used as intermediate storage, and the messages are copied from one process to the other.

  - ☞ Second type avoids copying large messages by pointing to a shared memory section object created for the channel.

  - ☞ Third method, called *quick* LPC was used by graphical display portions of the Win32 subsystem.

Executive — I/O Manager

- The I/O manager is responsible for
  - ☞ file systems
  - ☞ cache management
  - ☞ device drivers
  - ☞ network drivers

- Keeps track of which installable file systems are loaded, and manages buffers for I/O requests.

- Works with VM Manager to provide memory-mapped file I/O.

- Controls the 2000 cache manager, which handles caching for the entire I/O system.

- Supports both synchronous and asynchronous operations, provides time outs for drivers, and has mechanisms for one driver to call another.
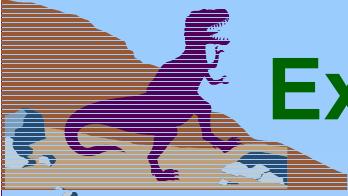
File I/O

Executive — Security Reference Monitor

- The object-oriented nature of 2000 enables the use of a uniform mechanism to perform runtime access validation and audit checks for every entity in the system.

- Whenever a process opens a handle to an object, the security reference monitor checks the process's security token and the object's access control list to see whether the process has the necessary rights.
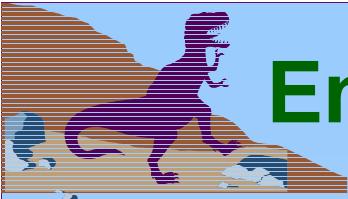
Executive – Plug-and-Play Manager

- Plug-and-Play (PnP) manager is used to recognize and adapt to changes in the hardware configuration.

- When new devices are added (for example, PCI or USB), the PnP manager loads the appropriate driver.

- The manager also keeps track of the resources used by each device.
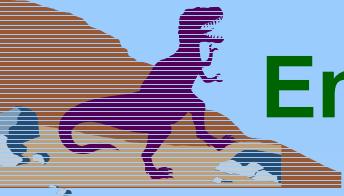
Environmental Subsystems

- User-mode processes layered over the native 2000 executive services to enable 2000 to run programs developed for other operating system.

- 2000 uses the Win32 subsystem as the main operating environment; Win32 is used to start all processes. It also provides all the keyboard, mouse and graphical display capabilities.

- MS-DOS environment is provided by a Win32 application called the *virtual dos machine* (VDM), a user-mode process that is paged and dispatched like any other 2000 thread.

Environmental Subsystems (Cont.)

- 16-Bit Windows Environment:
  - ☞ Provided by a VDM that incorporates *Windows on Windows.*
  - ☞ Provides the Windows 3.1 kernel routines and sub routines for window manager and GDI functions.
- The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model.
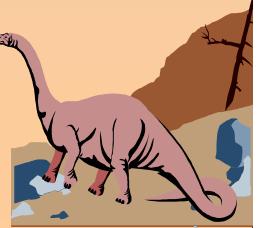
Environmental Subsystems (Cont.)
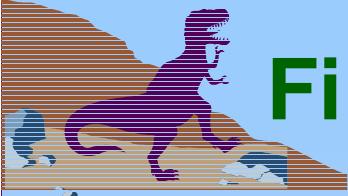
- OS/2 subsystems runs OS/2 applications.

- Logon and Security Subsystems authenticates users logging to to Windows 2000 systems. Users are required to have account names and passwords.

  - The authentication package authenticates users whenever they attempt to access an object in the system. Windows 2000 uses Kerberos as the default authentication package.

File System
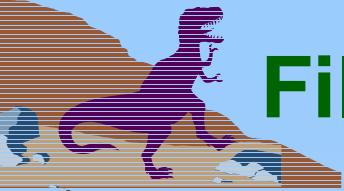
- The fundamental structure of the 2000 file system (NTFS) is a *volume*.

  - ☞ Created by the 2000 disk administrator utility.
  - ☞ Based on a logical disk partition.
  - ☞ May occupy a portions of a disk, an entire disk, or span across several disks.

- All *metadata*, such as information about the volume, is stored in a regular file.

- NTFS uses *clusters* as the underlying unit of disk allocation.

  - ☞ A cluster is a number of disk sectors that is a power of two.
  - ☞ Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced.

File System — Internal Layout

- NTFS uses *logical cluster numbers* (LCNs) as disk addresses.

- A file in NTFS is not a simple byte stream, as in MS-DOS or UNIX, rather, it is a structured object consisting of *attributes*.

- Every file in NTFS is described by one or more records in an array stored in a special file called the Master File Table (MFT).

- Each file on an NTFS volume has a unique ID called a *file reference.*
  - ☞ 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number.
  - ☞ Can be used to perform internal consistency checks.

- The NTFS name space is organized by a hierarchy of directories; the *index root* contains the top level of the B+ tree.

File System — Recovery

- All file system data structure updates are performed inside transactions that are logged.
  - ☞ Before a data structure is altered, the transaction writes a log record that contains redo and undo information.
  - ☞ After the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
  - ☞ After a crash, the file system data structures can be restored to a consistent state by processing the log records.

File System — Recovery (Cont.)

- This scheme does not guarantee that all the user file data can be recovered after a crash, just that the file system data structures (the metadata files) are undamaged and reflect some consistent state prior to the crash.

- The log is stored in the third metadata file at the beginning of the volume.

- The logging functionality is provided by the 2000 *log file service*.

File System — Security
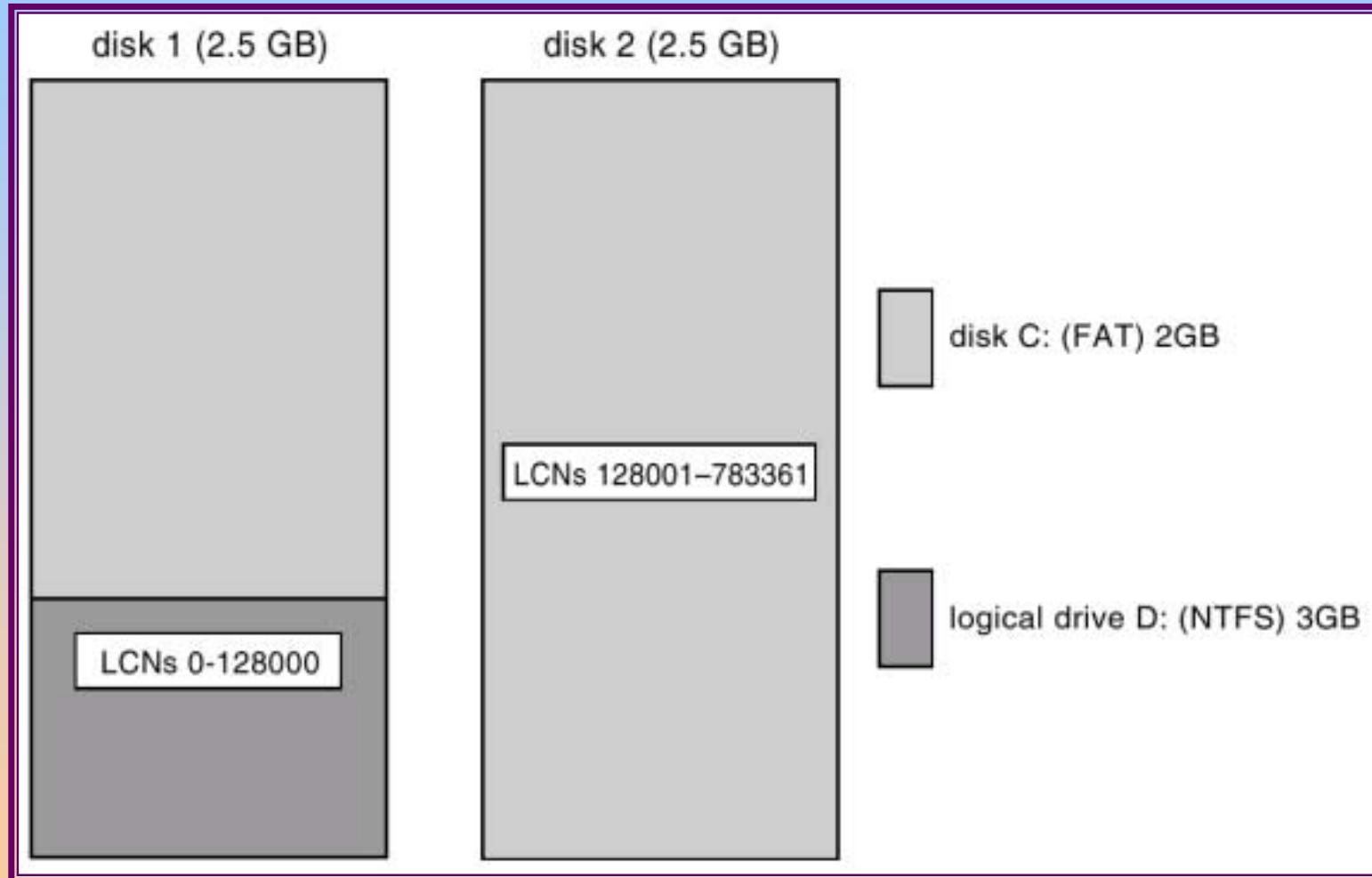
- Security of an NTFS volume is derived from the 2000 object model.

- Each file object has a security descriptor attribute stored in this MFT record.

- This attribute contains the access token of the owner of the file, and an access control list that states the access privileges that are granted to each user that has access to the file.
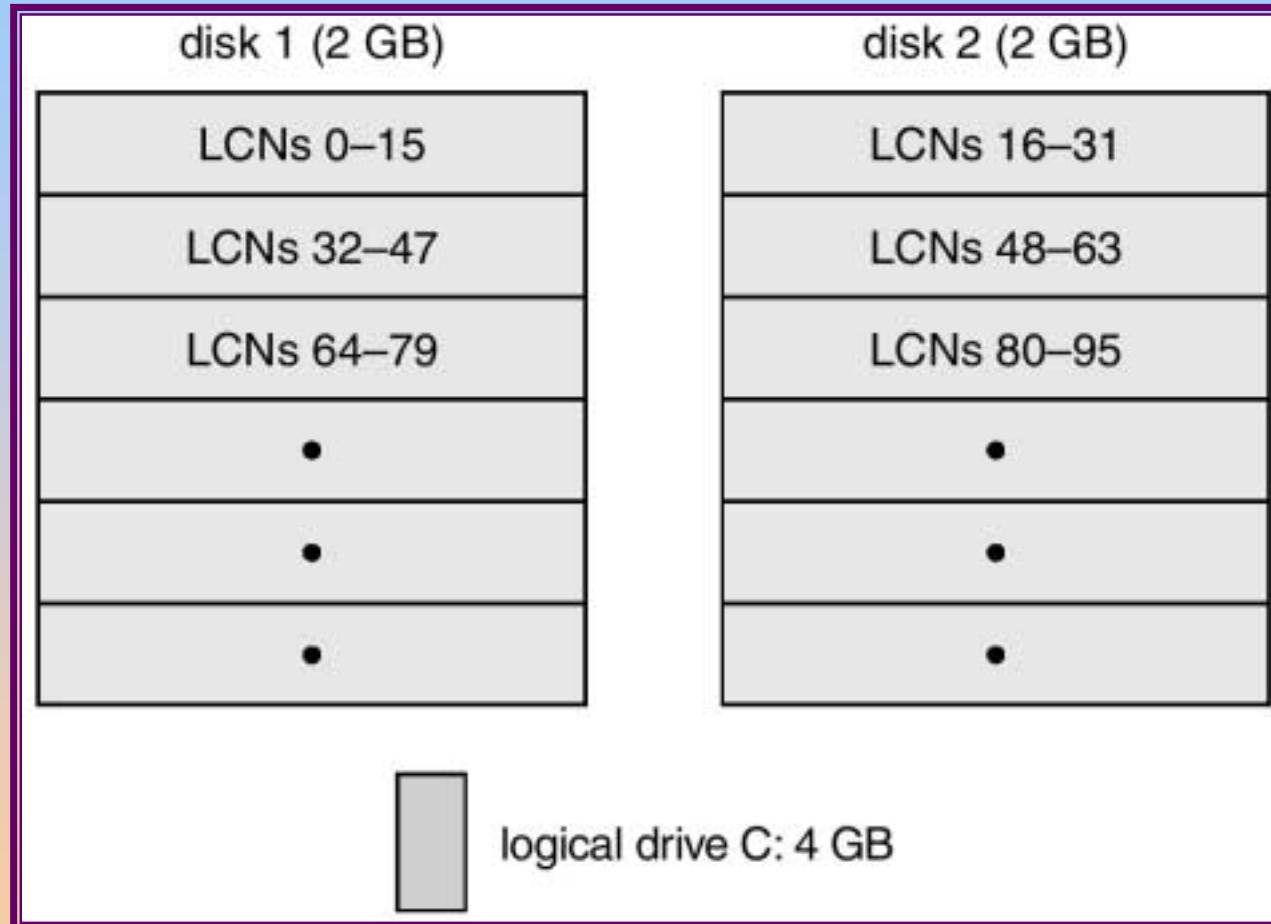
Volume Management and Fault Tolerance

- `FtDisk`, the fault tolerant disk driver for 2000, provides several ways to combine multiple SCSI disk drives into one logical volume.

- Logically concatenate multiple disks to form a large logical volume, a *volume set*.

- Interleave multiple physical partitions in round-robin fashion to form a *stripe set* (also called RAID level 0, or "disk striping").

  ☞ Variation: *stripe set with parity,* or RAID level 5.

- Disk mirroring, or RAID level 1, is a robust scheme that uses a *mirror set* — two equally sized partitions on tow disks with identical data contents.

- To deal with disk sectors that go bad, `FtDisk`, uses a hardware technique called *sector sparing* and NTFS uses a software technique called *cluster remapping*.

Volume Set On Two Drives

disk 1 (2.5 GB)　　disk 2 (2.5 GB)

disk C: (FAT) 2GB

LCNs 128001–783361

logical drive D: (NTFS) 3GB

LCNs 0-128000
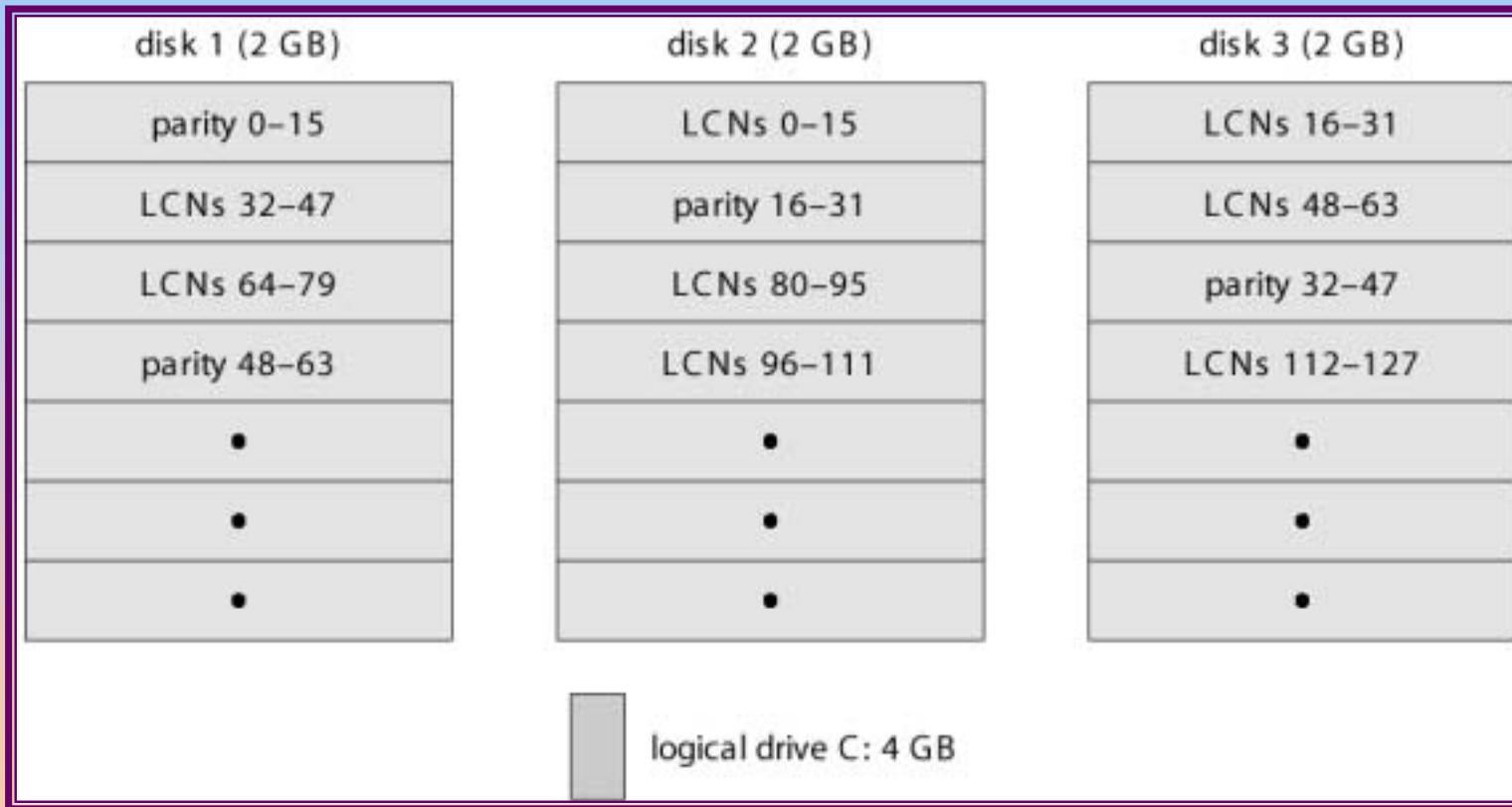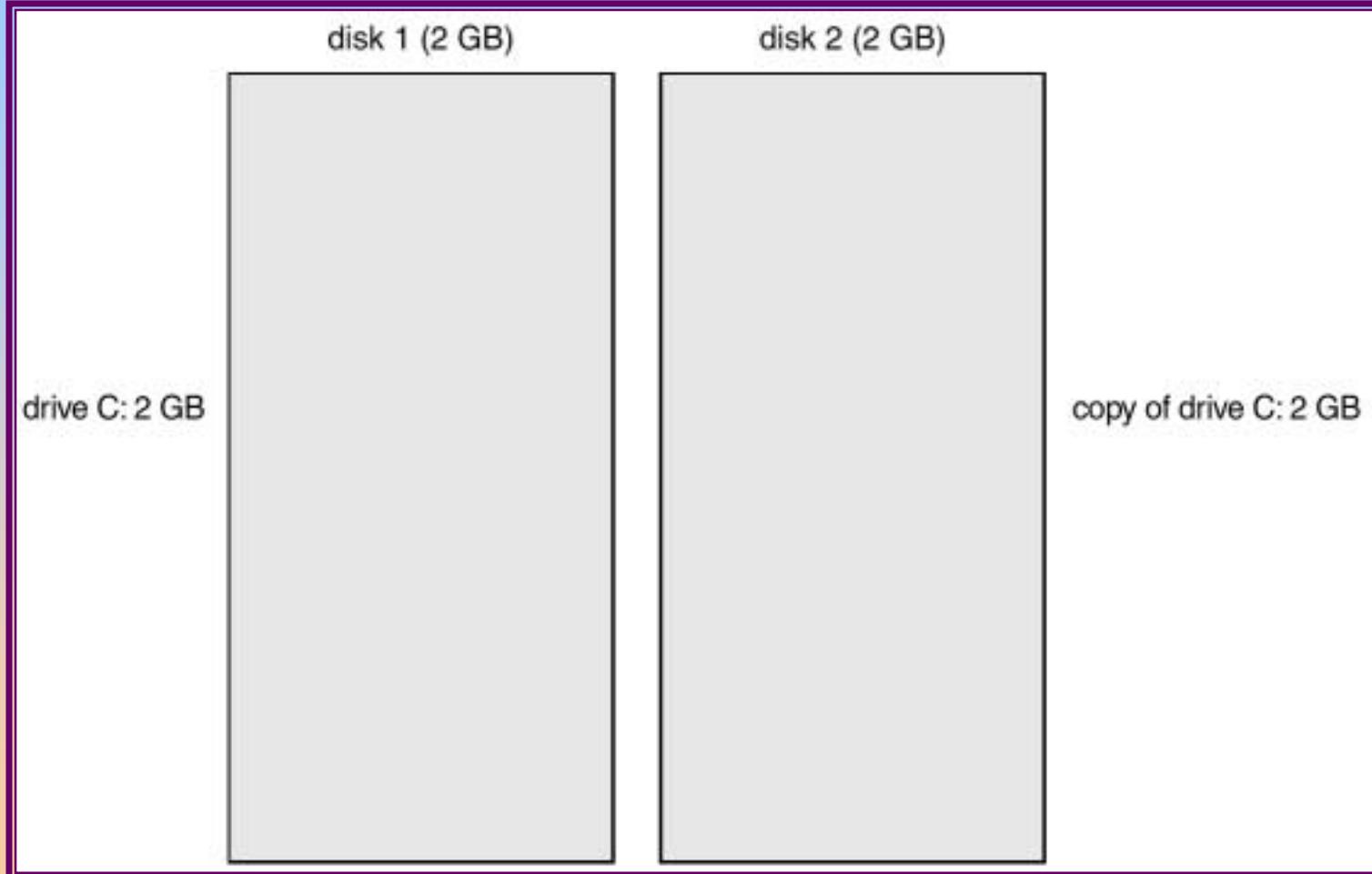
Stripe Set on Two Drives
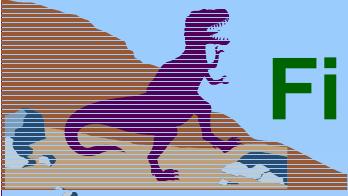
Stripe Set With Parity on Three Drives

Mirror Set on Two Drives

disk 1 (2 GB)   disk 2 (2 GB)

drive C: 2 GB   copy of drive C: 2 GB
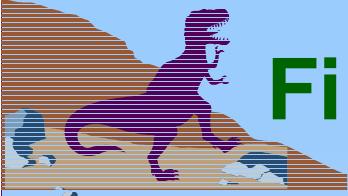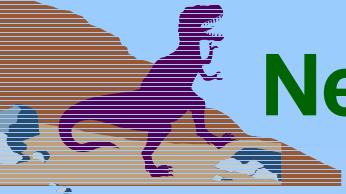
File System — Compression

- To compress a file, NTFS divides the file's data into *compression units*, which are blocks of 16 contiguous clusters.

- For sparse files, NTFS uses another technique to save space.
  - ☞ Clusters that contain all zeros are not actually allocated or stored on disk.
  - ☞ Instead, gaps are left in the sequence of virtual cluster numbers stored in the MFT entry for the file.
  - ☞ When reading a file, if a gap in the virtual cluster numbers is found, NTFS just zero-fills that portion of the caller's buffer.
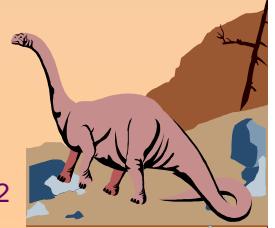
File System — Reparse Points
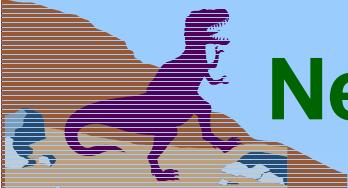
- A reparse point returns an error code when accessed. The reparse data tells the I/O manager what to do next.

- Reparse points can be used to provide the functionality of UNIX *mounts*

- Reparse points can also be used to access files that have been moved to offline storage.

Networking
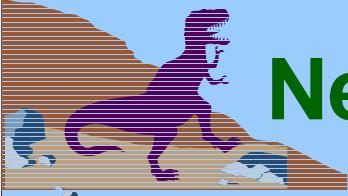
- 2000 supports both peer-to-peer and client/server networking; it also has facilities for network management.

- To describe networking in 2000, we refer to two of the internal networking interfaces:

  - ☞ NDIS (Network Device Interface Specification) — Separates network adapters from the transport protocols so that either can be changed without affecting the other.

  - ☞ TDI (Transport Driver Interface) — Enables any session layer component to use any available transport mechanism.

- 2000 implements transport protocols as drivers that can be loaded and unloaded from the system dynamically.
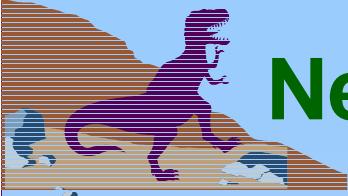
Networking — Protocols

■ The server message block (SMB) protocol is used to send I/O requests over the network. It has four message types:

  - Session control

  - File

  - Printer

  - Message

■ The network basic Input/Output system (NetBIOS) is a hardware abstraction interface for networks. Used to:

  ☞ Establish logical names on the network.

  ☞ Establish logical connections of sessions between two logical names on the network.

  ☞ Support reliable data transfer for a session via NetBIOS requests or *SMBs*

Networking — Protocols (Cont.)

- NetBEUI (NetBIOS Extended User Interface): default protocol for Windows 95 peer networking and Windows for Workgroups; used when 2000 wants to share resources with these networks.

- 2000 uses the TCP/IP Internet protocol to connect to a wide variety of operating systems and hardware platforms.

- PPTP (Point-to-Point Tunneling Protocol) is used to communicate between Remote Access Server modules running on 2000 machines that are connected over the Internet.

- The 2000 NWLink protocol connects the NetBIOS to Novell NetWare networks.

Networking — Protocols (Cont.)

- The Data Link Control protocol (DLC) is used to access IBM mainframes and HP printers that are directly connected to the network.

- 2000 systems can communicate with Macintosh computers via the Apple Talk protocol if an 2000 Server on the network is running the Windows 2000 Services for Macintosh package.

Networking — Dist. Processing Mechanisms

- 2000 supports distributed applications via named NetBIOS,named pipes and mailslots, Windows Sockets, Remote Procedure Calls (RPC), and Network Dynamic Data Exchange (NetDDE).

- NetBIOS applications can communicate over the network using NetBEUI, NWLink, or TCP/IP.

- Named pipes are connection-oriented messaging mechanism that are named via the *uniform naming convention* (UNC).

- Mailslots are a connectionless messaging mechanism that are used for broadcast applications, such as for finding components on the network,

- Winsock, the windows sockets API, is a session-layer interface that provides a standardized interface to many transport protocols that may have different addressing schemes.
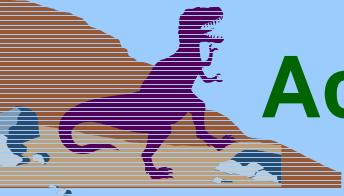
Distributed Processing Mechanisms (Cont.)

- The 2000 RPC mechanism follows the widely-used Distributed Computing Environment standard for RPC messages, so programs written to use 2000 RPCs are very portable.

  - ☞ RPC messages are sent using NetBIOS, or Winsock on TCP/IP networks, or named pipes on LAN Manager networks.

  - ☞ 2000 provides the Microsoft *Interface Definition Language* to describe the remote procedure names, arguments, and results.
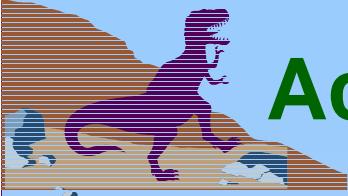
Networking — Redirectors and Servers

- In 2000, an application can use the 2000 I/O API to access files from a remote computer as if they were local, provided that the remote computer is running an MS-NET server.

- A *redirector* is the client-side object that forwards I/O requests to remote files, where they are satisfied by a server.

- For performance and security, the redirectors and servers run in kernel mode.

Access to a Remote File

- The application calls the I/O manager to request that a file be opened (we assume that the file name is in the standard UNC format).

- The I/O manager builds an I/O request packet.

- The I/O manager recognizes that the access is for a remote file, and calls a driver called a Multiple Universal Naming Convention Provider (MUP).

- The MUP sends the I/O request packet asynchronously to all registered redirectors.

- A redirector that can satisfy the request responds to the MUP.

  - ☞ To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember with redirector can handle this file.

Access to a Remote File (Cont.)

- The redirector sends the network request to the remote system.

- The remote system network drivers receive the request and pass it to the server driver.

- The server driver hands the request to the proper local file system driver.

- The proper device driver is called to access the data.

- The results are returned to the server driver, which sends the data back to the requesting redirector.
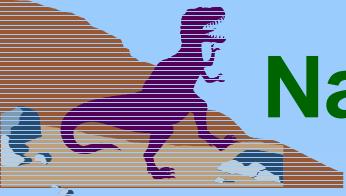
Networking — Domains

- NT uses the concept of a domain to manage global access rights within groups.

- A domain is a group of machines running NT server that share a common security policy and user database.

- 2000 provides three models of setting up trust relationships.
  - *One way, A trusts B*
  - *Two way, transitive, A trusts B, B trusts C so A, B, C trust each other*
  - *Crosslink – allows authentication to bypass hierarchy to cut down on authentication traffic.*

Name Resolution in TCP/IP Networks

- On an IP network, name resolution is the process of converting a computer name to an IP address.

    e.g., `www.bell-labs.com` resolves to `135.104.1.14`

- 2000 provides several methods of name resolution:
  - Windows Internet Name Service (WINS)
  - broadcast name resolution
  - domain name system (DNS)
  - a host file
  - an LMHOSTS file

Name Resolution (Cont.)

- WINS consists of two or more WINS servers that maintain a dynamic database of name to IP address bindings, and client software to query the servers.

- WINS uses the Dynamic Host Configuration Protocol (DHCP), which automatically updates address configurations in the WINS database, without user or administrator intervention.
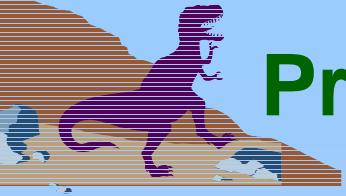
Programmer Interface — Access to Kernel Obj.

- A process gains access to a kernel object named xxx by calling the CreateXXX function to open a *handle* to xxx; the handle is unique to that process.

- A handle can be closed by calling the CloseHandle function; the system may delete the object if the count of processes using the object drops to 0.

- 2000 provides three ways to share objects between processes.

  - ☞ A child process inherits a handle to the object.

  - ☞ One process gives the object a name when it is created and the second process opens that name.

  - • DuplicateHandle function:

    - Given a handle to process and the handle's value a second process can get a handle to the same object, and thus share it.

Programmer Interface — Process Management

- Process is started via the `CreateProcess` routine which loads any dynamic link libraries that are used by the process, and creates a *primary thread*.

- Additional threads can be created by the `CreateThread` function.

- Every dynamic link library or executable file that is loaded into the address space of a process is identified by an *instance handle.*
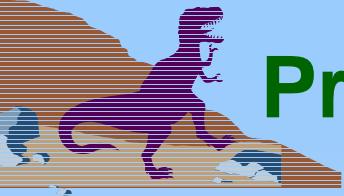
Process Management (Cont.)

- Scheduling in Win32 utilizes four priority classes:
  - IDLE_PRIORITY_CLASS (priority level 4)
  - NORMAL_PRIORITY_CLASS (level8 — typical for most processes
  - HIGH_PRIORITY_CLASS (level 13)
  - REALTIME_PRIORITY_CLASS (level 24)
- To provide performance levels needed for interactive programs, 2000 has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS.
  - ☞ 2000 distinguishes between the *foreground process* that is currently selected on the screen, and the *background processes* that are not currently selected.
  - ☞ When a process moves into the foreground, 2000 increases the scheduling quantum by some factor, typically 3.

Process Management (Cont.)

- The kernel dynamically adjusts the priority of a thread depending on whether it is I/O-bound or CPU-bound.

- To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes.

    - ☞ In addition, threads can synchronize by using the `WaitForSingleObject` or `WaitForMultipleObjects` functions.

    - ☞ Another method of synchronization in the Win32 API is the critical section.

Process Management (Cont.)

- A fiber is user-mode code that gets scheduled according to a user-defined scheduling algorithm.
  - ☞ Only one fiber at a time is permitted to execute, even on multiprocessor hardware.
  - ☞ 2000 includes fibers to facilitate the porting of legacy UNIX applications that are written for a fiber execution model.

Programmer Interface — Interprocess Comm.

- Win32 applications can have interprocess communication by sharing kernel objects.

- An alternate means of interprocess communications is message passing, which is particularly popular for Windows GUI applications.

  - ☞ One thread sends a message to another thread or to a window.

  - ☞ A thread can also send data with the message.

- Every Win32 thread has its own input queue from which the thread receives messages.

- This is more reliable than the shared input queue of 16-bit windows, because with separate queues, one stuck application cannot block input to the other applications.
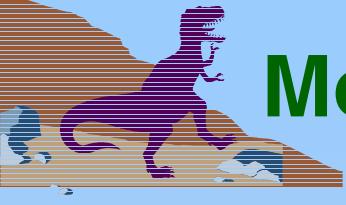
Programmer Interface — Memory Management

- Virtual memory:
  - `VirtualAlloc` reserves or commits virtual memory.
  - `VirtualFree` decommits or releases the memory.
  - ☞ These functions enable the application to determine the virtual address at which the memory is allocated.
- An application can use memory by memory mapping a file into its address space.
  - ☞ Multistage process.
  - ☞ Two processes share memory by mapping the same file into their virtual memory.
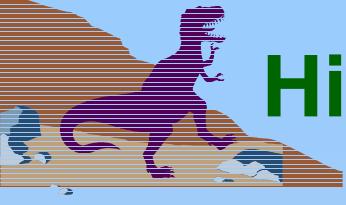
Memory Management (Cont.)

- A heap in the Win32 environment is a region of reserved address space.
  - ☞ A Win 32 process is created with a 1 MB *default heap.*
  - ☞ Access is synchronized to protect the heap's space allocation data structures from damage by concurrent updates by multiple threads.

- Because functions that rely on global or static data typically fail to work properly in a multithreaded environment, the thread-local storage mechanism allocates global storage on a per-thread basis.
  - ☞ The mechanism provides both dynamic and static methods of creating thread-local storage.

Module A: The FreeBSD System
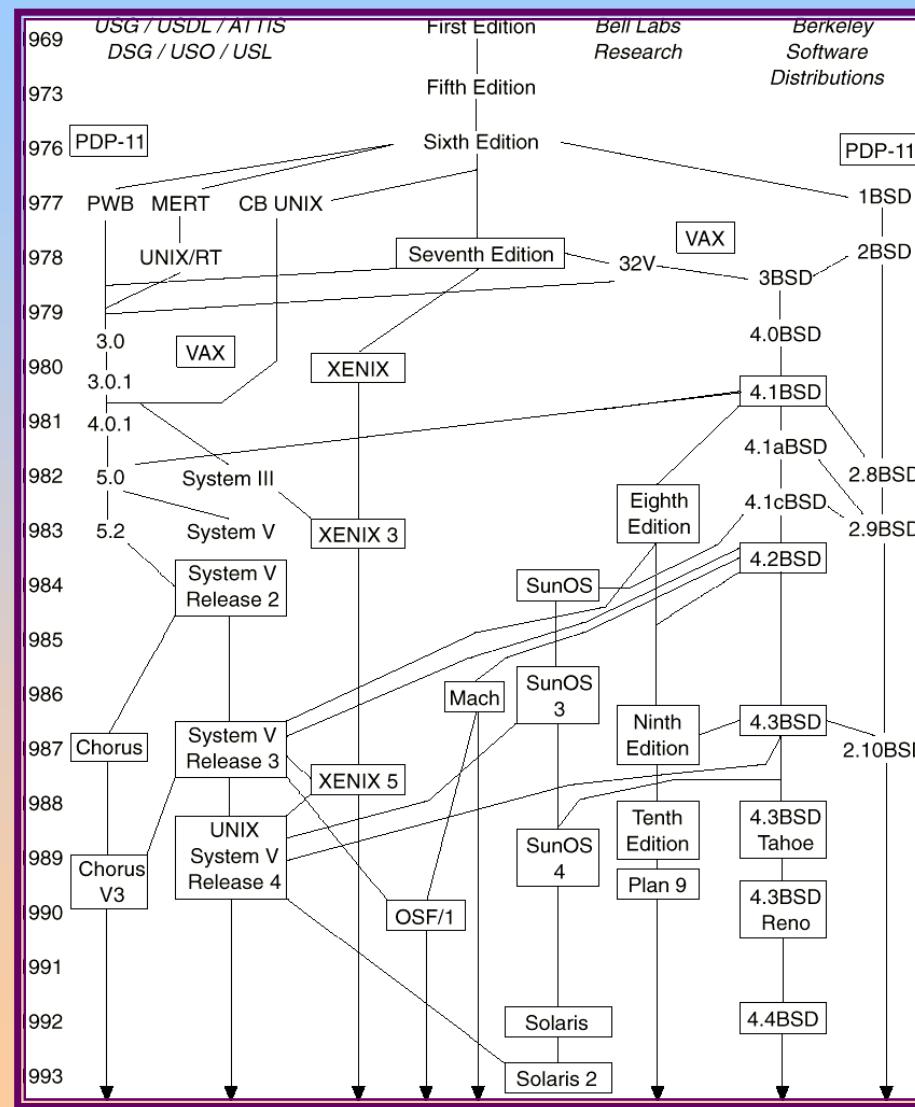
- History
- Design Principles
- Programmer Interface
- User Interface
- Process Management
- Memory Management
- File System
- I/O System
- Interprocess Communication
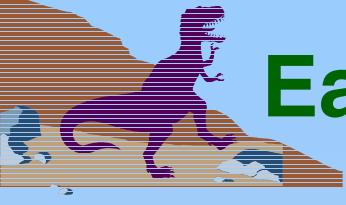
History

- First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS.

- The third version was written in C, which was developed at Bell Labs specifically to support UNIX.

- The most influential of the non-Bell Labs and non-AT&T UNIX development groups — University of California at Berkeley (Berkeley Software Distributions).

  - 4BSD UNIX resulted from DARPA funding to develop a standard UNIX system for government use.

  - Developed for the VAX, 4.3BSD is one of the most influential versions, and has been ported to many other platforms.

- Several standardization projects seek to consolidate the variant flavors of UNIX leading to one programming interface to UNIX.
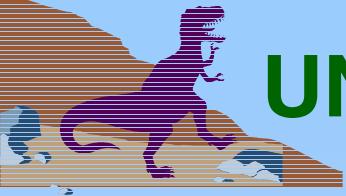
Early Advantages of UNIX

- Written in a high-level language.

- Distributed in source form.

- Provided powerful operating-system primitives on an inexpensive platform.

- Small size, modular, clean design.
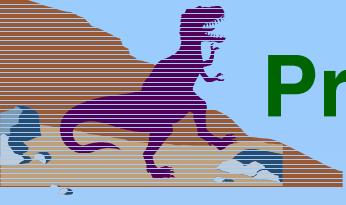
UNIX Design Principles

- Designed to be a time-sharing system.

- Has a simple standard user interface (shell) that can be replaced.

- File system with multilevel tree-structured directories.

- Files are supported by the kernel as unstructured sequences of bytes.

- Supports multiple processes; a process can easily create new processes.

- High priority given to making system interactive, and providing facilities for program development.
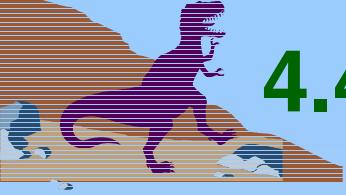
Programmer Interface
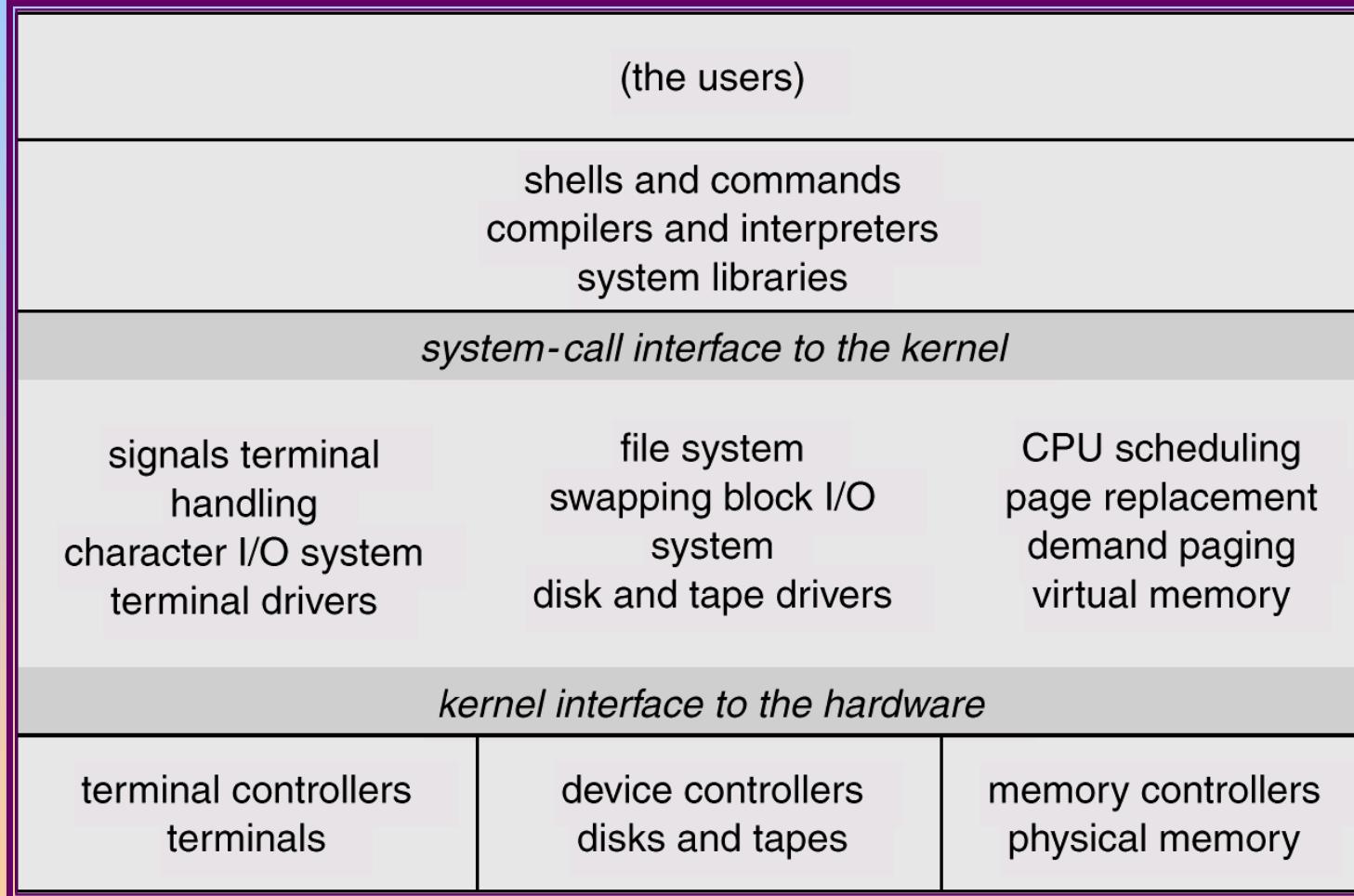
Like most computer systems, UNIX consists of two separable parts:

- Kernel: everything below the system-call interface and above the physical hardware.
    - Provides file system, CPU scheduling, memory management, and other OS functions through system calls.
- Systems programs: use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

4.4BSD Layer Structure

(the users)		
shells and commands compilers and interpreters system libraries		
system-call interface to the kernel		
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
kernel interface to the hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

System Calls

- System calls define the programmer interface to UNIX
- The set of systems programs commonly available defines the user interface.
- The programmer and user interface define the context that the kernel must support.
- Roughly three categories of system calls in UNIX.
  - File manipulation (same system calls also support device manipulation)
  - Process control
  - Information manipulation.
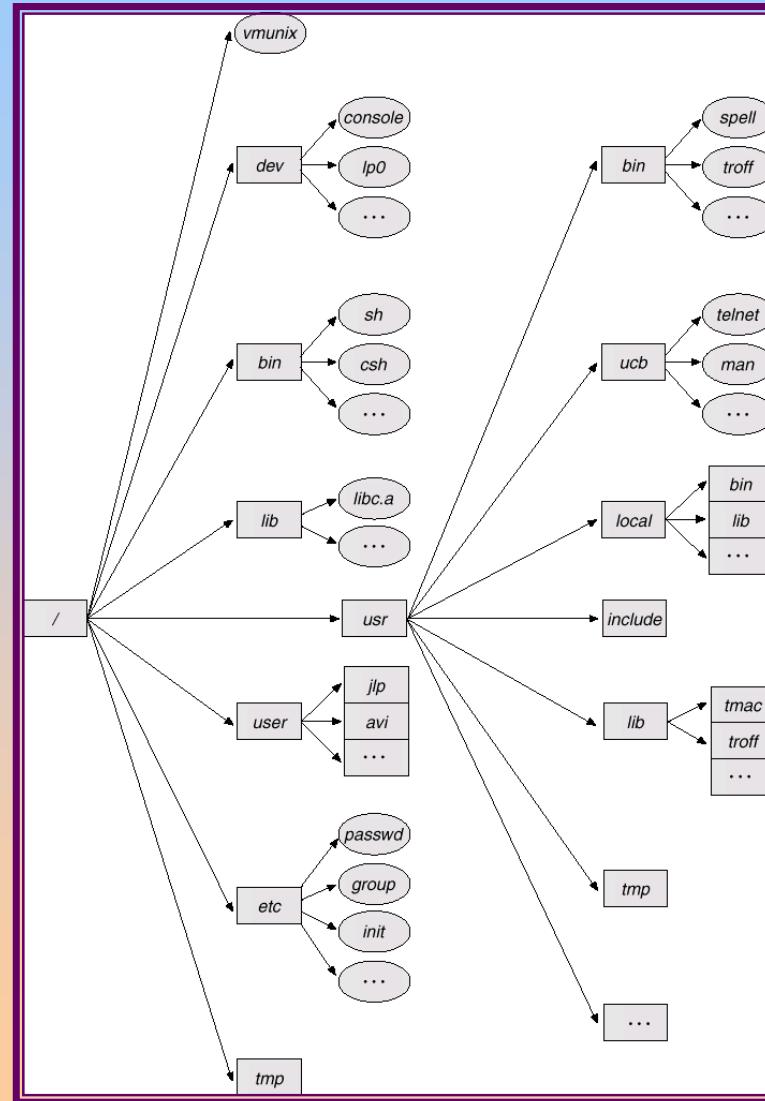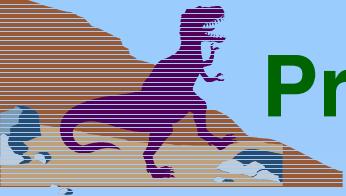
File Manipulation

- A *file* is a sequence of bytes; the kernel does not impose a structure on files.

- Files are organized in tree-structured *directories*.

- Directories are files that contain information on how to find other files.

- *Path name*: identifies a file by specifying a path through the directory structure to the file.
  - Absolute path names start at root of file system
  - Relative path names start at the current directory

- System calls for basic file manipulation: **create, open, read, write, close, unlink, trunc.**

Typical UNIX Directory Structure
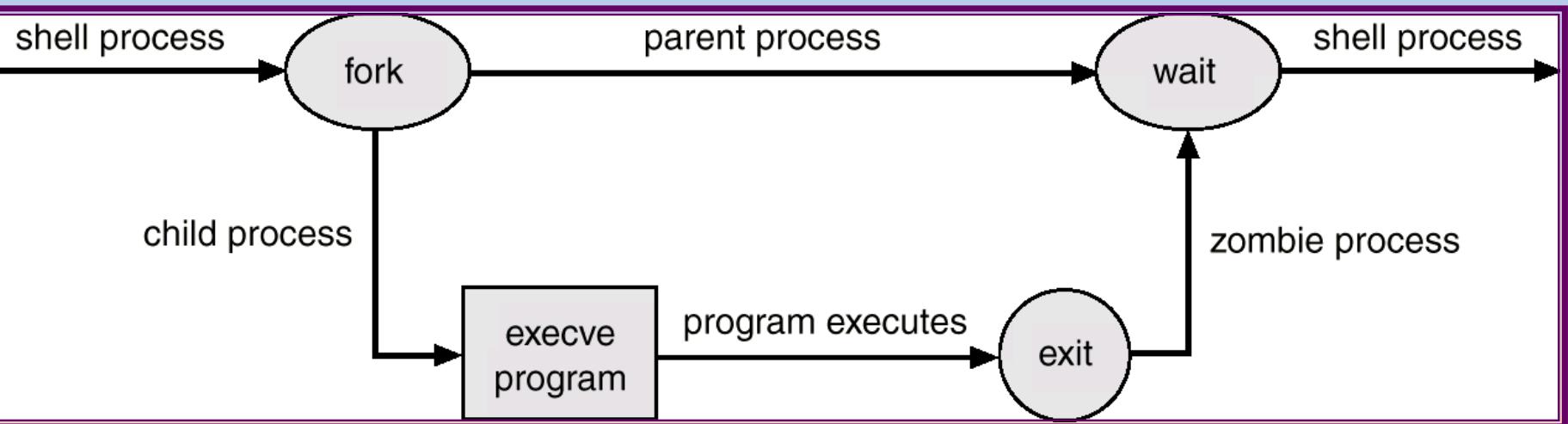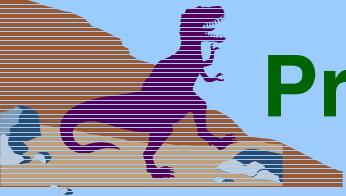
Process Control

- A process is a program in execution.
- Processes are identified by their process identifier, an integer.
- Process control system calls
  - **fork** creates a new process
  - **execve** is used after a fork to replace on of the two processes's virtual memory space with a new program
  - exit terminates a process
  - A parent may **wait** for a child process to terminate; **wait** provides the process id of a terminated child so that the parent can tell which child terminated.
  - **wait3** allows the parent to collect performance statistics about the child
- A *zombie* process results when the parent of a *defunct* child process exits before the terminated child.
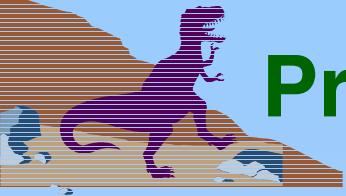
Illustration of Process Control Calls

Process Control (Cont.)

- Processes communicate via pipes; queues of bytes between two processes that are accessed by a file descriptor.

- All user processes are descendants of one original process, *init*.

- *init* forks a *getty* process: initializes terminal line parameters and passes the user's *login name* to *login*.
  - *login* sets the numeric *user identifier* of the process to that of the user
  - executes a *shell* which forks subprocesses for user commands.
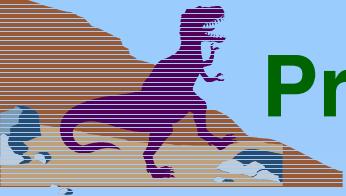
Process Control (Cont.)

- **setuid** bit sets the effective user identifier of the process to the user identifier of the owner of the file, and leaves the *real user identifier* as it was.

- **setuid** scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users.

Signals

- Facility for handling exceptional conditions similar to software interrupts.

- The *interrupt* signal, SIGINT, is used to stop a command before that command completes (usually produced by ^C).

- Signal use has expanded beyond dealing with exceptional events.

  - Start and stop subprocesses on demand

  - SIGWINCH informs a process that the window in which output is being displayed has changed size.

  - Deliver urgent data from network connections.
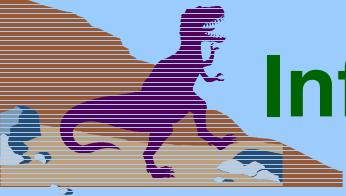
Process Groups

- Set of related processes that cooperate to accomplish a common task.

- Only one process group may use a terminal device for I/O at any time.

  - The foreground job has the attention of the user on the terminal.

  - Background jobs – nonattached jobs that perform their function without user interaction.

- Access to the terminal is controlled by process group signals.

Process Groups (Cont.)
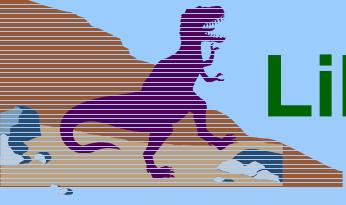
- Each job inherits a controlling terminal from its parent.
  - If the process group of the controlling terminal matches the group of a process, that process is in the foreground.
  - SIGTTIN or SIGTTOU freezes a background process that attempts to perform I/O; if the user foregrounds that process, SIGCONT indicates that the process can now perform I/O.
  - SIGSTOP freezes a foreground process.

Information Manipulation

- System calls to set and return an interval timer: **getitmer/setitmer.**

- Calls to set and return the current time: **gettimeofday/settimeofday.**

- Processes can ask for
  - their process identifier: **getpid**
  - their group identifier: **getgid**
  - the name of the machine on which they are executing: **gethostname**
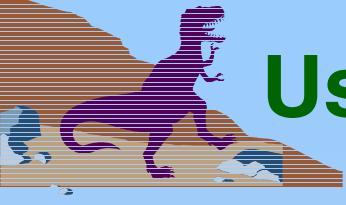
Library Routines

- The system-call interface to UNIX is supported and augmented by a large collection of library routines

- Header files provide the definition of complex data structures used in system calls.

- Additional library support is provided for mathematical functions, network access, data conversion, etc.

User Interface

- Programmers and users mainly deal with already existing systems programs: the needed system calls are embedded within the program and do not need to be obvious to the user.

- The most common systems programs are file or directory oriented.

  - Directory: *mkdir, rmdir, cd, pwd*
  - File: *ls, cp, mv, rm*

- Other programs relate to editors (e.g., *emacs*, *vi*) text formatters (e.g., troff, TEX), and other activities.

Shells and Commands

- *Shell* – the user process which executes programs (also called command interpreter).

- Called a shell, because it surrounds the kernel.

- The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line.

- A typical command is an executable binary object file.

- The shell travels through the *search path* to find the command file, which is then loaded and executed.

- The directories /bin and /usr/bin are almost always in the search path.

Shells and Commands (Cont.)

■ Typical search path on a BSD system:

   (./home/prof/avi/bin /usr/local/bin /usr/ucb/bin/usr/bin)

■ The shell usually suspends its own execution until the command completes.
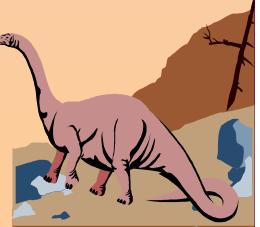
Standard I/O

- Most processes expect three file descriptors to be open when they start:
    - *standard input* – program can read what the user types
    - *standard output* – program can send output to user's screen
    - *standard error* – error output
- Most programs can also accept a file (rather than a terminal) for standard input and standard output.
- The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process — I/O *redirection*.

Standard I/O Redirection

Command	Meaning of command
% ls > filea	direct output of *ls* to file *filea*
% pr < filea > fileb	input from *filea* and output to *fileb*
% lpr < fileb	input from *fileb*
%% make program > & errs	save both standard output and standard error in a file
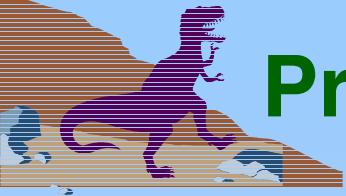
Pipelines, Filters, and Shell Scripts

- Can coalesce individual commands via a vertical bar that tells the shell to pass the previous command's output as input to the following command
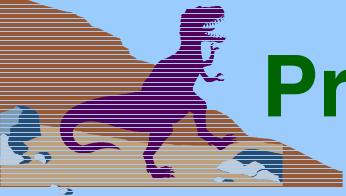
$$\% \ ls \mid pr \mid lpr$$

- Filter – a command such as pr that passes its standard input to its standard output, performing some processing on it.

- Writing a new shell with a different syntax and semantics would change the user view, but not change the kernel or programmer interface.

- X Window System is a widely accepted iconic interface for UNIX.

Process Management

- Representation of processes is a major design problem for operating system.
- UNIX is distinct from other systems in that multiple processes can be created and manipulated with ease.
- These processes are represented in UNIX by various control blocks.
  - Control blocks associated with a process are stored in the kernel.
  - Information in these control blocks is used by the kernel for process control and CPU scheduling.
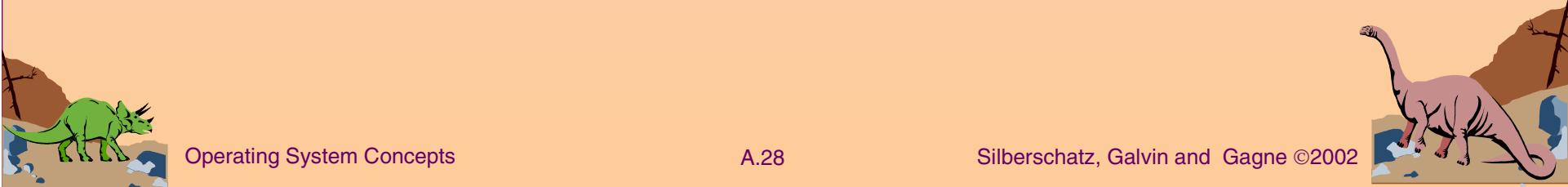
Process Control Blocks

- The most basic data structure associated with processes is the *process structure*.
  - ◆ unique process identifier
  - ◆ scheduling information (e.g., priority)
  - ◆ pointers to other control blocks
- The *virtual address space* of a user process is divided into text (program code), data, and stack segments.
- Every process with sharable text has a pointer form its process structure to a *text structure*.
  - ◆ always resident in main memory.
  - ◆ records how many processes are using the text segment
  - ◆ records were the page table for the text segment can be found on disk when it is swapped.
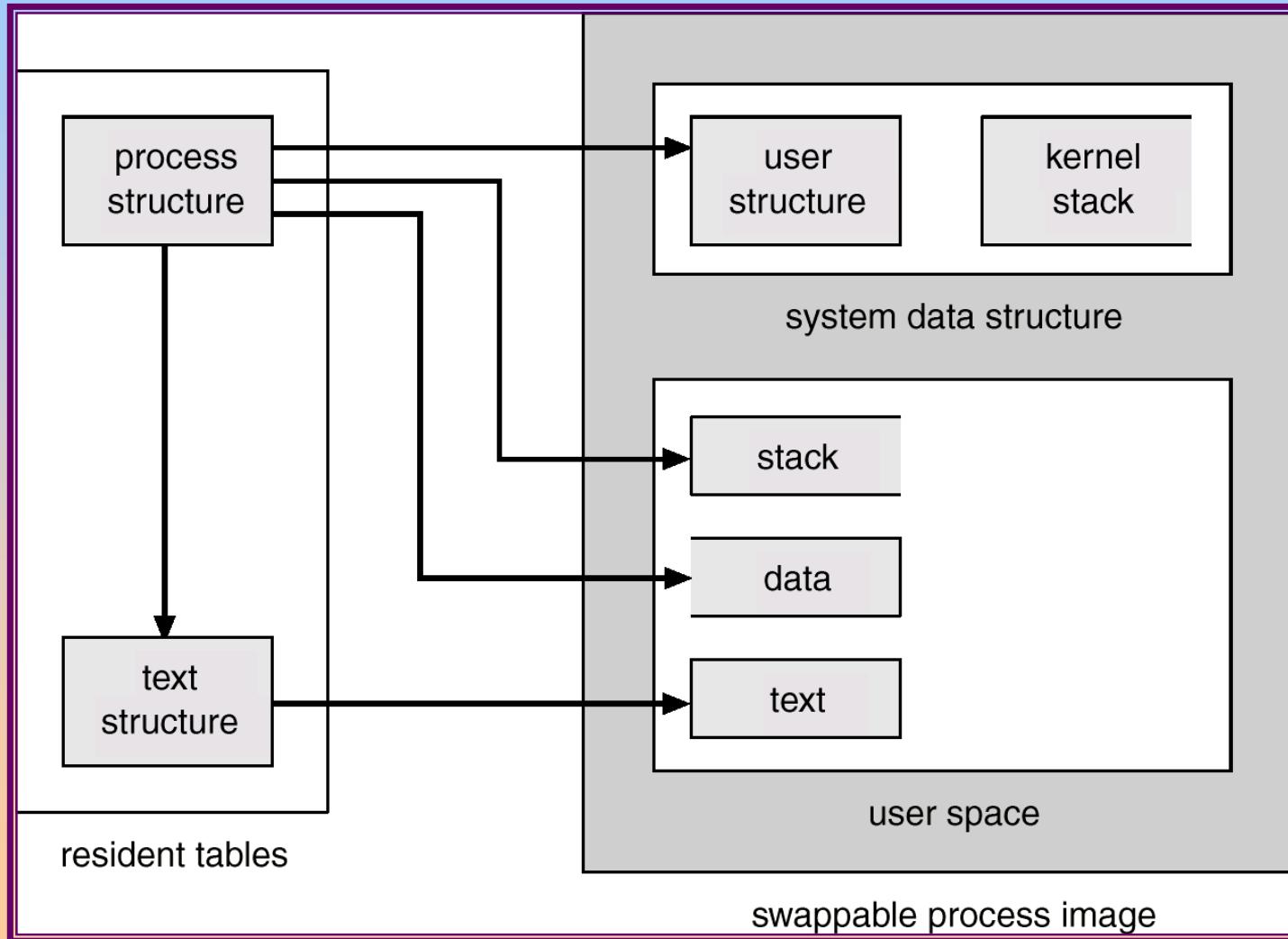
System Data Segment

- Most ordinary work is done in *user mode*; system calls are performed in *system mode*.

- The system and user phases of a process never execute simultaneously.

- a *kernel stack* (rather than the user stack) is used for a process executing in system mode.

- The kernel stack and the user structure together compose the *system data* segment for the process.

Finding parts of a process using process structure

process structure → user structure, kernel stack (system data structure)

process structure → stack, data (user space)

process structure → text structure

text structure → text (user space)

resident tables

swappable process image
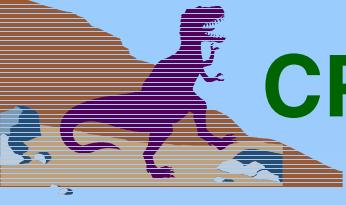
Allocating a New Process Structure

■ fork allocates a new process structure for the child process, and copies the user structure.

  ✦ new page table is constructed

  ✦ new main memory is allocated for the data and stack segments of the child process

  ✦ copying the user structure preserves open file descriptors, user and group identifiers, signal handling, etc.
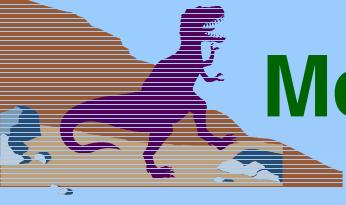
Allocating a New Process Structure (Cont.)

- **vfork** does *not* copy the data and stack to t he new process; the new process simply shares the page table of the old one.
  - ✦ new user structure and a new process structure are still created
  - ✦ commonly used by a shell to execute a command and to wait for its completion
- A parent process uses **vfork** to produce a child process; the child uses **execve** to change its virtual address space, so there is no need for a copy of the parent.
- Using **vfork** with a large parent process saves CPU time, but can be dangerous since any memory change occurs in both processes until **execve** occurs.
- **execve** creates no new process or user structure; rather the text and data of the process are replaced.
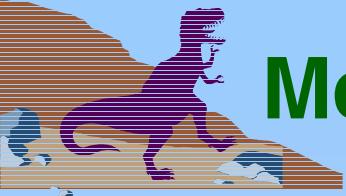
CPU Scheduling

- Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority.

- Negative feedback in CPU scheduling makes it difficult for a single process to take all the CPU time.

- Process aging is employed to prevent starvation.

- When a process chooses to relinquish the CPU, it goes to *sleep* on an *event*.

- When that event occurs, the system process that knows about it calls *wakeup* with the address corresponding to the event, and *all* processes that had done a *sleep* on the same address are put in the ready queue to be run.

Memory Management
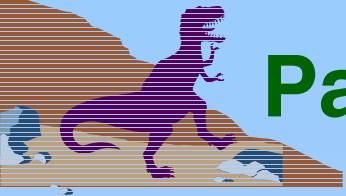
■ The initial memory management schemes were constrained in size by the relatively small memory resources of the PDP machines on which UNIX was developed.

■ Pre 3BSD system use swapping exclusively to handle memory contention among processes: If there is too much contention, processes are swapped out until enough memory is available.

■ Allocation of both main memory and swap space is done first-fit.

Memory Management (Cont.)
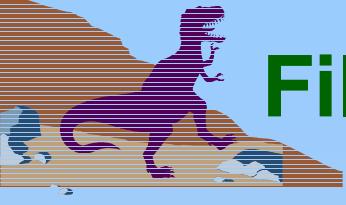
- Sharable text segments do not need to be swapped; results in less swap traffic and reduces the amount of main memory required for multiple processes using the same text segment.

- The *scheduler process* (or *swapper*) decides which processes to swap in or out, considering such factors as time idle, time in or out of main memory, size, etc.

- In f.3BSD, swap space is allocated in pieces that are multiples of power of 2 and minimum size, up to a maximum size determined by the size or the swap-space partition on the disk.
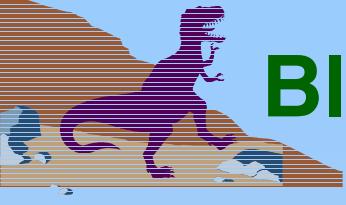
Paging

- Berkeley UNIX systems depend primarily on paging for memory-contention management, and depend only secondarily on swapping.

- *Demand paging* – When a process needs a page and the page is not there, a page fault tot he kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

- A *pagedaemon* process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

- If the scheduler decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved.
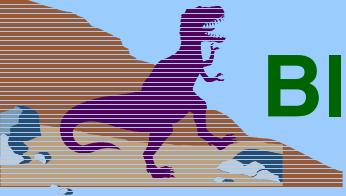
File System

- The UNIX file system supports two main objects: files and directories.

- Directories are just files with a special format, so the representation of a file is the basic UNIX concept.
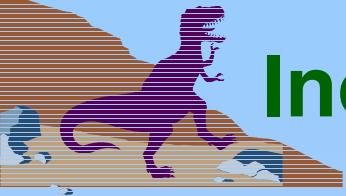
Blocks and Fragments

■ Most of the file system is taken up by *data blocks*.

■ 4.2BSD uses *two* block sized for files which have no indirect blocks:

  ✦ All the blocks of a file are of a large *block* size (such as 8K), except the last.

  ✦ The last block is an appropriate multiple of a smaller *fragment size* (i.e., 1024) to fill out the file.

  ✦ Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

A.37   Silberschatz, Galvin and Gagne ©2002

Blocks and Fragments (Cont.)

- The *block* and *fragment* sizes are set during file-system creation according to the intended use of the file system:
    - If many small files are expected, the fragment size should be small.
    - If repeated transfers of large files are expected, the basic block size should be large.

- The maximum block-to-fragment ratio is 8 : 1; the minimum block size is 4K (typical choices are 4096 : 512 and 8192 : 1024).

Inodes
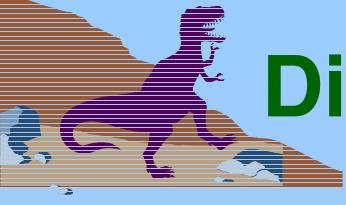
■ A file is represented by an *inode* — a record that stores information about a specific file on the disk.

■ The inode also contains 15 pointer to the disk blocks containing the file's data contents.

✦ First 12 point to *direct blocks*.

✦ Next three point to *indirect blocks*

✔ First indirect block pointer is the address of a *single indirect block* — an index block containing the addresses of blocks that do contain data.

✔ Second is a *double-indirect-block pointe*r, the address of a block that contains the addresses of blocks that contain pointer to the actual data blocks.

✔ A *triple indirect* pointer is not needed; files with as many as 232 bytes will use only double indirection.

Directories

- The inode type field distinguishes between plain files and directories.

- Directory entries are of variable length; each entry contains first the length of the entry, then the file name and the inode number.

- The user refers to a file by a path name,whereas the file system uses the inode as its definition of a file.
  - The kernel has to map the supplied user path name to an inode
  - Directories are used for this mapping.

Directories (Cont.)
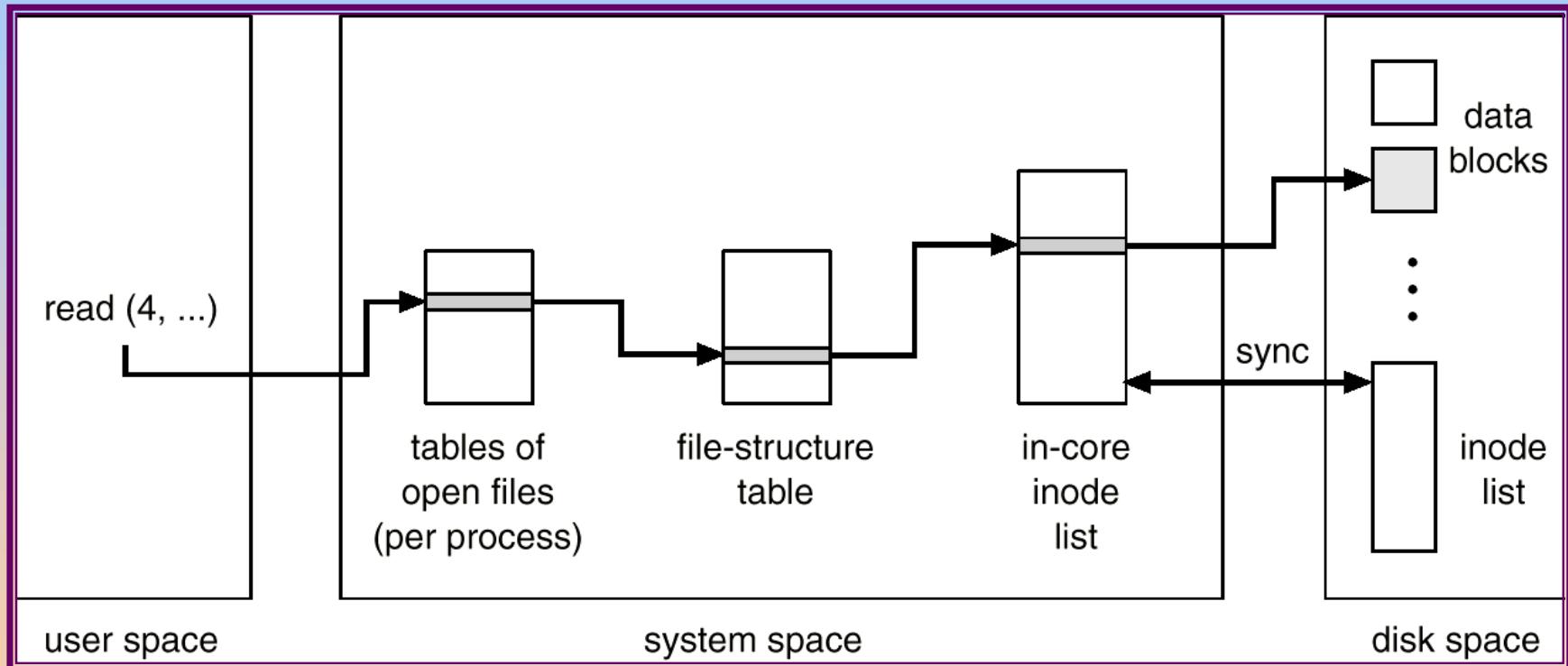
- First determine the starting directory:
  - If the first character is "/", the starting directory is the root directory.
  - For any other starting character, the starting directory is the current directory.
- The search process continues until the end of the path name is reached and the desired inode is returned.
- Once the inode is found, a file structure is allocated to point to the inode.
- 4.3BSD improved file system performance by adding a directory name cache to hold recent directory-to-inode translations.
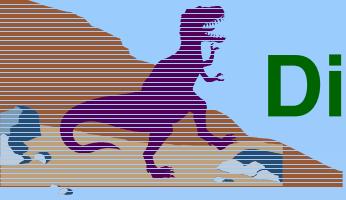
Mapping of a File Descriptor to an Inode

- System calls that refer to open files indicate the file is passing a file descriptor as an argument.

- The file descriptor is used by the kernel to index a table of open files for the current process.

- Each entry of the table contains a pointer to a file structure.

- This file structure in turn points to the inode.

- Since the open file table has a fixed length which is only setable at boot time, there is a fixed limit on the number of concurrently open files in a system.

File-System Control Blocks
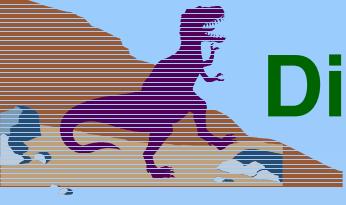
Disk Structures

- The one file system that a user ordinarily sees may actually consist of several physical file systems, each on a different device.

- Partitioning a physical device into multiple file systems has several benefits.
  - Different file systems can support different uses.
  - Reliability is improved
  - Can improve efficiency by varying file-system parameters.
  - Prevents one program form using all available space for a large file.
  - Speeds up searches on backup tapes and restoring partitions from tape.
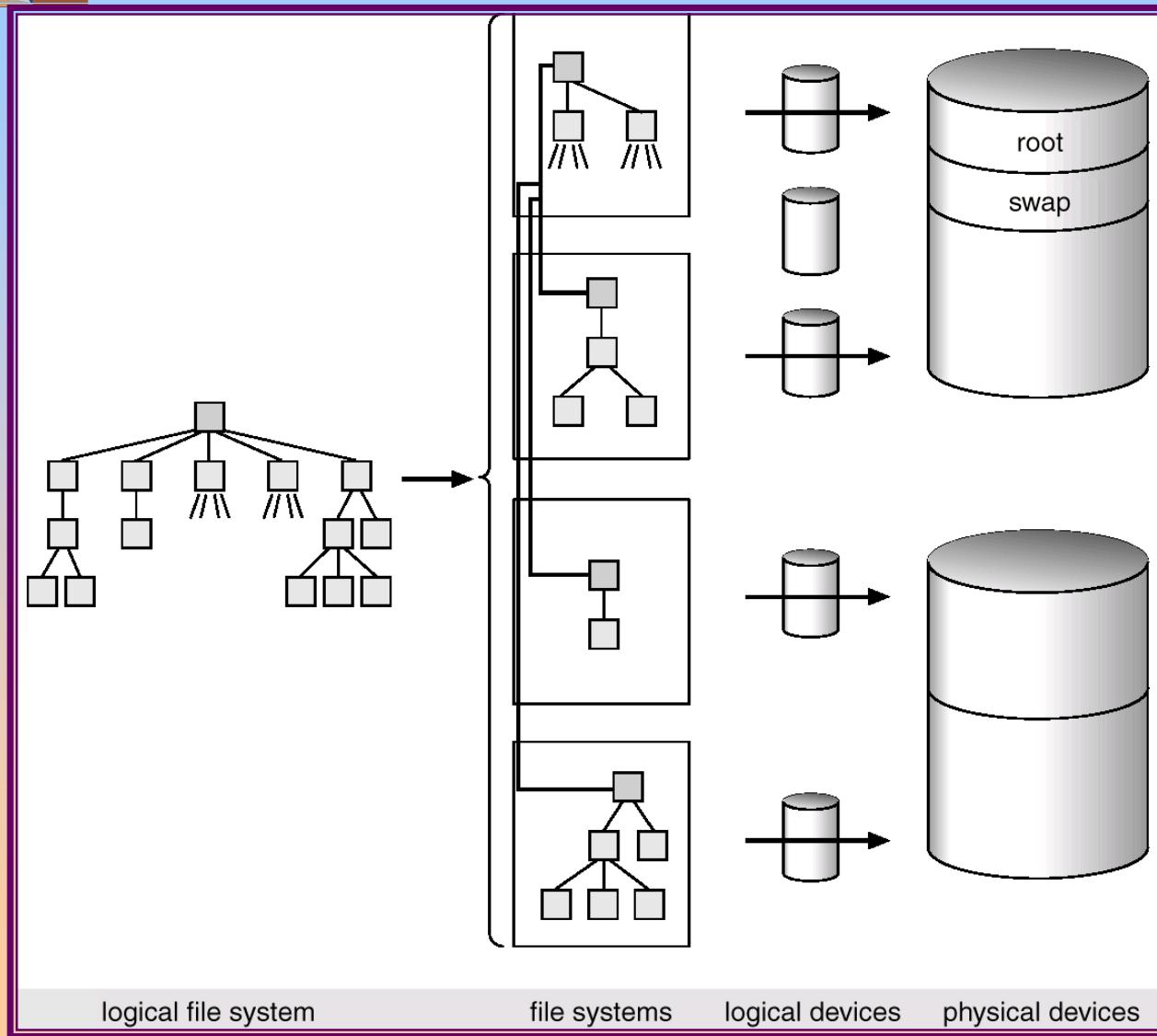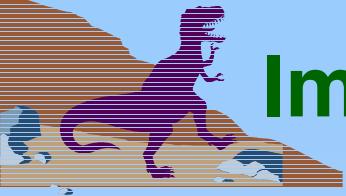
Disk Structures (Cont.)

- The *root file* system is always available on a drive.

- Other file systems may be *mounted* — i.e., integrated into the directory hierarchy of the root file system.

- The following figure illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices.

Mapping File System to Physical Devices

logical file system   file systems   logical devices   physical devices

Implementations

- The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user.

- For Version 7, the size of inodes doubled, the maximum file and file system sized increased, and the details of free-list handling and superblock information changed.

- In 4.0BSD, the size of blocks used in the file system was increased form 512 bytes to 1024 bytes — increased internal fragmentation, but doubled throughput.

- 4.2BSD added the Berkeley Fast File System, which increased speed, and included new features.

  - New directory system calls
  - **truncate** calls
  - Fast File System found in most implementations of UNIX.

Layout and Allocation Policy
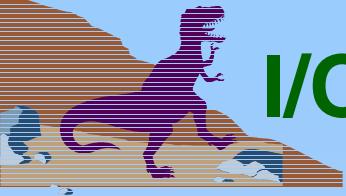
- The kernel uses a *<logical device number, inode number>* pair to identify a file.
    - The logical device number defines the file system involved.
    - The inodes in the file system are numbered in sequence.
- 4.3BSD introduced the *cylinder group* — allows localization of the blocks in a file.
    - Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement.
    - Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks.

4.3BSD Cylinder Group

data blocks
superblock
cylinder block
inodes
data blocks
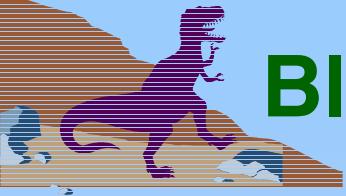
I/O System

- The I/O system hides the peculiarities of I/O devices from the bulk of the kernel.

- Consists of a buffer caching system, general device driver code, and drivers for specific hardware devices.

- Only the device driver knows the peculiarities of a specific device.
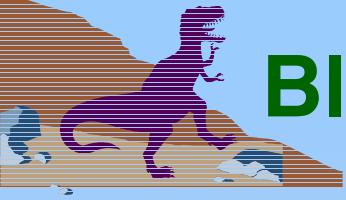
4.3 BSD Kernel I/O Structure

system-call interface to the kernel					
socket	plain file	cooked block interface	raw block interface	raw tty interface	cooked TTY
protocols	file system				line discipline
network interface	block-device driver			character-device driver	
the hardware					
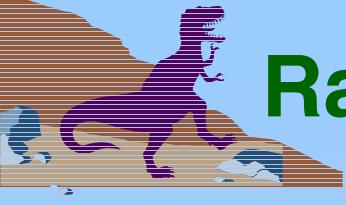
Block Buffer Cache

- Consist of buffer headers, each of which can point to a piece of physical memory, as well as to a device number and a block number on the device.

- The buffer headers for blocks not currently in use are kept in several linked lists:
  - Buffers recently used, linked in LRU order (LRU list).
  - Buffers not recently used, or without valid contents (AGE list).
  - EMPTY buffers with no associated physical memory.

- When a block is wanted from a device, the cache is searched.

- If the block is found it is used, and no I/O transfer is necessary.

- If it is not found, a buffer is chosen from the AGE list, or the LRU list if AGE is empty.
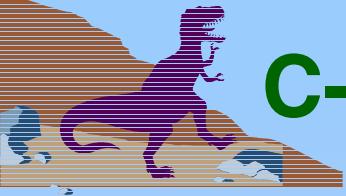
Block Buffer Cache (Cont.)

- Buffer cache size effects system performance; if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low.

- Data written to a disk file are buffered in the cache, and the disk driver sorts its output queue according to disk address — these actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation.
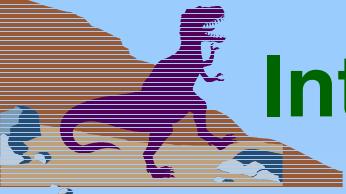
Raw Device Interfaces

■ Almost every block device has a character interface, or *raw device interface* — unlike the block interface, it bypasses the block buffer cache.

■ Each disk driver maintains a queue of pending transfers.

■ Each record in the queue specifies:

- ✦ whether it is a read or a write
- ✦ a main memory address for the transfer
- ✦ a device address for the transfer
- ✦ a transfer size

■ It is simple to map the information from a block buffer to what is required for this queue.
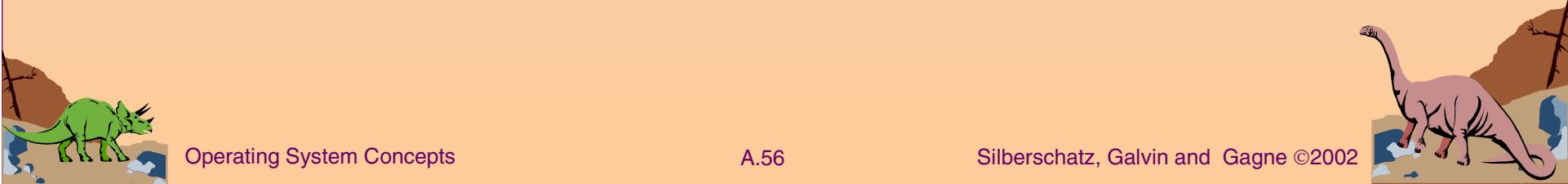
C-Lists

- Terminal drivers use a character buffering system which involves keeping small blocks of characters in linked lists.

- A **write** system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeueing of characters and further transfers.

- Input is similarly interrupt driven.

- It is also possible to have the device driver bypass the canonical queue and return characters directly form the raw queue — *raw mode* (used by full-screen editors and other programs that need to react to every keystroke).

Interprocess Communication

- The *pipe* is the IPC mechanism most characteristic of UNIX.
    - Permits a reliable unidirectional byte stream between two processes.
    - A benefit of pipes small size is that pipe data are seldom written to disk; they usually are kept in memory by the normal block buffer cache.

- In 4.3BSD, pipes are implemented as a special case of the *socket* mechanism which provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

- The socket mechanism can be used by unrelated processes.

Sockets

- A socket is an endpont of communication.

- An in-use socket it usually bound with an address; the nature of the address depends on the *communication domain* of the socket.

- A characteristic property of a domain is that processes communication in the same domain use the same *address format.*

- A single socket can communicate in only one domain — the three domains currently implemented in 4.3BSD are:
  - ✦ the UNIX domain (AF_UNIX)
  - ✦ the Internet domain (AF_INET)
  - ✦ the XEROX Network Service (NS) domain (AF_NS)

Socket Types

- *Stream sockets* provide reliable, duplex, sequenced data streams. Supported in Internet domain by the TCP protocol. In UNIX domain, pipes are implemented as a pair of communicating stream sockets.

- **Sequenced packet sockets** provide similar data streams, except that record boundaries are provided. Used in XEROX AF_NS protocol.

- **Datagram sockets** transfer messages of variable size in either direction. Supported in Internet domain by UDP protocol

- **Reliably delivered message sockets** transfer messages that are guaranteed to arrive. Currently unsupported.

- **Raw sockets** allow direct access by processes to the protocols that support the other socket types; e.g., in the Internet domain, it is possible to reach TCP, IP beneath that, or a deeper Ethernet protocol. Useful for developing new protocols.
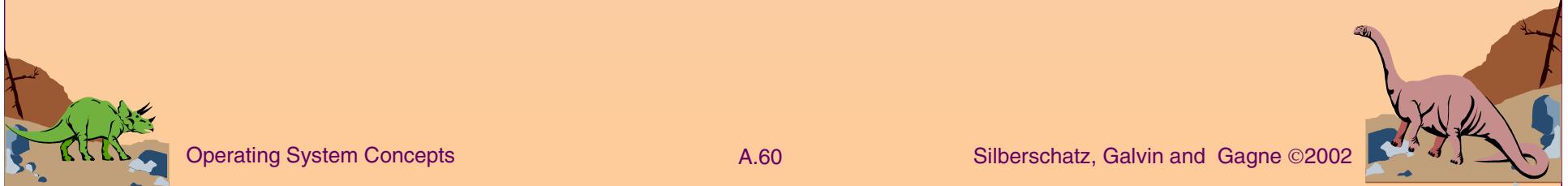
Socket System Calls
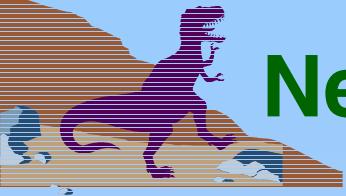
- The **socket** call creates a socket; takes as arguments specifications of the communication domain, socket type, and protocol to be used and returns a small integer called a *socket descriptor.*

- A name is bound to a socket by the **bind** system call.

- The **connect** system call is used to initiate a connection.

- A server process uses **socket** to create a socket and **bind** to bind the well-known address of its service to that socket.

  - Uses **listen** to tell the kernel that it is ready to accept connections from clients.

  - Uses **accept** to accept individual connections.

  - Uses **fork** to produce a new process after the **accept** to service the client while the original server process continues to listen for more connections.

Socket System Calls (Cont.)

- The simplest way to terminate a connection and to destroy the associated socket is to use the **close** system call on its socket descriptor.

- The **select** system call can be used to multiplex data transfers on several file descriptors and /or socket descriptors

Network Support

- Networking support is one of the most important features in 4.3BSD.

- The socket concept provides the programming mechanism to access other processes, even across a network.

- Sockets provide an interface to several sets of protocols.

- Almost all current UNIX systems support UUCP.

- 4.3BSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces.

- The 4.3BSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPANET Reference Model (ARM).

Network Reference models and Layering

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation			
session		sockets	sock_stream
transport	hostÐhos	protocol	TCP
network data link			IP
hardware	network interface	network interfaces	Ethernet driver
	network hardware	network hardware	interlan controller

Appendix A

THE
FREEBSD
SYSTEM

Although operating-system concepts can be considered in purely theoretical terms, it is often useful to see how they are implemented in practice. This chapter presents an in-depth examination of the FreeBSD operating system, a version of UNIX, as an example of the various concepts presented in this book. By examining a complete, real system, we can see how the various concepts discussed in this book relate both to one another and to practice. We consider first a brief history of UNIX, and present the system's user and programmer interfaces. Then, we discuss the internal data structures and algorithms used by the UNIX kernel to support the user–programmer interface.

A.1 ■ History

The first version of UNIX was developed in 1969 by Ken Thompson of the Research Group at Bell Laboratories to use an otherwise idle PDP-7. He was soon joined by Dennis Ritchie. Thompson, Ritchie, and other members of the Research Group produced the early versions of UNIX.

Ritchie had previously worked on the MULTICS project, and MULTICS had a strong influence on the newer operating system. Even the name *UNIX* is merely a pun on *MULTICS*. The basic organization of the file system, the idea of the command interpreter (the shell) as a user process, the use of a separate process for each command, the original line-editing characters (# to erase the last character and @ to erase the entire line), and numerous other features came

directly from MULTICS. Ideas from various other operating systems, such as from MIT's CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. Their work on the first version allowed them to move it to a PDP-11/20, for a second version. A third version resulted from their rewriting most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available outside Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX Programmer's Manual* that was current when the distribution was made; the code and the manuals were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32, and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group continued to develop their own version of UNIX, however, to support their own internal computing. Next came Version 8, which included a facility called the *stream I/O system* that allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. Next came Versions 9 and 10 (the latter version, released in 1989, is available only within Bell Laboratories).

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7, and 32V, and also of several UNIX systems developed by groups other than Research. Features of UNIX/RT, a real-time UNIX system, as well as numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T has left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging

and shared memory. USDL was in turn replaced by AT&T Information Systems (ATTIS), which distributed System V Release 3 (V.3) in 1987. V.3 adapts the V8 implementation of the stream I/O system and makes it available as *STREAMS*. It also includes RFS, an NFS-like remote file system.

The small size, modularity, and clean design of early UNIX systems led to UNIX-based work at numerous other computer-science organizations, such as at Rand, BBN, the University of Illinois, Harvard, Purdue, and even DEC. The most influential of the non-Bell Laboratories and non–AT&T UNIX development groups, however, has been the University of California at Berkeley.

The first Berkeley VAX UNIX work was the addition in 1978 of virtual memory, demand paging, and page replacement to 32V; this work was done by Bill Joy and Ozalp Babaoglu to produce 3BSD UNIX. This version was the first implementation of any of these facilities on any UNIX system. The large virtual-memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz LISP. The memory-management work convinced the Defense Advanced Research Projects Agency (DARPA) to fund Berkeley for the development of a standard UNIX system for government use; 4BSD UNIX was the result.

The 4BSD work for DARPA was guided by a steering committee that included many notable people from the UNIX and networking communities. One of the goals of this project was to provide support for the DARPA Internet networking protocols (TCP/IP). This support was provided in a general manner. It is possible in 4.2BSD to communicate uniformly among diverse network facilities, including local-area networks (such as Ethernets and token rings) and wide-area networks (such as NSFNET). This implementation was the most important reason for the current popularity of these protocols. It was used as the basis for the implementations of many vendors of UNIX computer systems, and even other operating systems. It permitted the Internet to grow from 60 connected networks in 1984 to more than 8000 networks and an estimated 10 million users in 1993.

In addition, Berkeley adapted many features from contemporary operating systems to improve the design and implementation of UNIX. Many of the terminal line-editing functions of the TENEX (TOPS-20) operating system were provided by a new terminal driver. A new user interface (the C Shell), a new text editor (ex/vi), compilers for Pascal and LISP, and many new systems programs were written at Berkeley. For 4.2BSD, certain efficiency improvements were inspired by the VMS operating system.

UNIX software from Berkeley is released in *Berkeley Software Distributions*. It is convenient to refer to the Berkeley VAX UNIX systems following 3BSD as 4BSD, although there were actually several specific releases, most notably 4.1BSD and 4.2BSD. The generic numbers BSD and 4BSD are used for the PDP-11 and VAX distributions of Berkeley UNIX. 4.2BSD, first distributed in 1983, was the culmination of the original Berkeley DARPA UNIX project. 2.9BSD is the equivalent version for PDP-11 systems.

In 1986, 4.3BSD was released. It was so similar to 4.2BSD that its manuals described 4.2BSD more comprehensively than the 4.2BSD manuals did. It did include numerous internal changes, however, including bug fixes and performance improvements. Some new facilities also were added, including support for the Xerox Network System protocols.

4.3BSD Tahoe was the next version, released in 1988. It included various new developments, such as improved networking congestion control and TCP/IP performance. Also, disk configurations were separated from the device drivers, and are now read off the disks themselves. Expanded time-zone support is also included. 4.3BSD Tahoe was actually developed on and for the CCI Tahoe system (Computer Console, Inc., Power 6 computer), rather than for the usual VAX base. The corresponding PDP-11 release is 2.10.1BSD, which is distributed by the USENIX Association, which also publishes the 4.3BSD manuals. The 4.32BSD Reno release saw the inclusion of an implementation of ISO/OSI networking.

The last Berkeley release, 4.4BSD, was finalized in June of 1993. It includes new X.25 networking support, and POSIX standard compliance. It also has a radically new file system organization, with a new virtual file system interface and support for *stackable* file systems, allowing file systems to be layered on top of each other for easy inclusion of new features. An implementation of NFS is also included in the release (Chapter 16), as is a new log-based file system (see Chapter 14). The 4.4BSD virtual memory system is derived from Mach (described in Section 22.9). Several other changes, such as enhanced security and improved kernel structure, are also included. With the release of version 4.4, Berkeley has halted its research efforts.

4BSD was the operating system of choice for VAXes from its initial release (1979) until the release of Ultrix, DEC's BSD implementation. 4BSD is still the best choice for many research and networking installations. Many organizations would buy a 32V license and order 4BSD from Berkeley without even bothering to get a 32V tape.

The current set of UNIX operating systems is not limited to those by Bell Laboratories (which is currently owned by Lucent Technology), and Berkeley, however. Sun Microsystems helped popularize the BSD flavor of UNIX by shipping it on their workstations. As UNIX has grown in popularity, it has been moved to many different computers and computer systems. A wide variety of UNIX, and UNIX-like, operating systems have been created. DEC supports its UNIX (called Ultrix) on its workstations and is replacing Ultrix with another UNIX-derived operating system, OSF/1; Microsoft rewrote UNIX for the Intel 8088 family and called it XENIX, and its new Windows NT operating system is heavily influenced by UNIX; IBM has UNIX (AIX) on its PCs, workstations, and mainframes. In fact, UNIX is available on almost all general-purpose computers; it runs on personal computers, workstations, minicomputers, mainframes, and supercomputers, from Apple Macintosh IIs to Cray IIs. Because of its wide availability, it is used in environments ranging from academic to military to

manufacturing process control. Most of these systems are based on Version 7, System III, 4.2BSD, or System V.
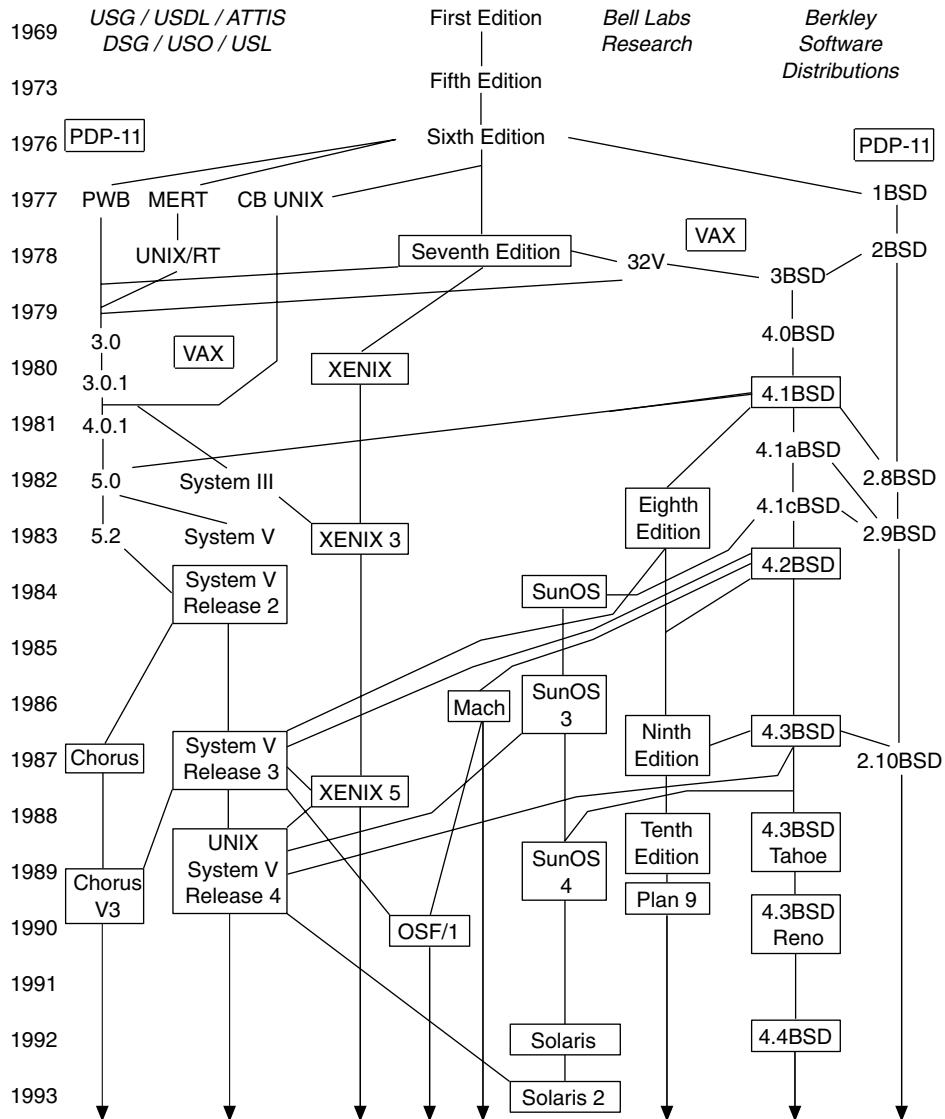
The wide popularity of UNIX with computer vendors has made UNIX the most portable of operating systems, and has made it possible for users to expect a UNIX environment independent of any specific computer manufacturer. But the large number of implementations of the system has led to remarkable variation in the programming and user interfaces distributed by the vendors. For true vendor independence, application-program developers need consistent interfaces. Such interfaces would allow all "UNIX" applications to run on all UNIX systems, which is certainly not the current situation. This issue has become important as UNIX has become the preferred program-development platform for applications ranging from databases to graphics and networking, and has led to a strong market demand for UNIX standards.

There are several standardization projects underway, starting with the */usr/group 1984 Standard* sponsored by the UniForum industry user's group. Since then, many official standards bodies have continued the effort, including IEEE and ISO (the POSIX standard). The X/Open Group international consortium completed XPG3, a Common Application Environment, which subsumes the IEEE interface standard. Unfortunately, XPG3 is based on a draft of the ANSI C standard, rather than the final specification, and therefore needs to be redone. The XPG4 is due out in 1993. In 1989, the ANSI standards body standardized the C programming language, producing an ANSI C specification that vendors were quick to adopt. As these projects continue, the variant flavors of UNIX will converge and there will be one programming interface to UNIX, allowing UNIX to become even more popular. There are in fact two separate sets of powerful UNIX vendors working on this problem: the AT&T-guided UNIX International (UI) and the Open Software Foundation (OSF) have both agreed to follow the POSIX standard. Recently, many of the vendors involved in those two groups have agreed on further standardization (the COSE agreement) on the Motif window environment, and ONC+ (which includes Sun RPC and NFS) and DCE network facilities (which includes AFS and an RPC package).

AT&T replaced its ATTIS group in 1989 with the UNIX Software Organization (USO), which shipped the first merged UNIX, System V Release 4. This system combines features from System V, 4.3BSD, and Sun's SunOS, including long file names, the Berkeley file system, virtual memory management, symbolic links, multiple access groups, job control, and reliable signals; it also conforms to the published POSIX standard, POSIX.1. After USO produced SVR4, it became an independent AT&T subsidiary named Unix System Laboratories (USL); in 1993, it was purchased by Novell, Inc.

Figure A.1 summarizes the relationships among the various versions of UNIX.

The UNIX system has grown from a personal project of two Bell Laboratories employees to an operating system being defined by multinational standardization bodies. Yet this system is still of interest to academia. We believe that

Figure A.1   History of UNIX versions.

UNIX has become and will remain an important part of operating-system theory and practice. UNIX is an excellent vehicle for academic study. For example, the Tunis operating system, the Xinu operating system, and the Minix operating system are based on the concepts of UNIX, but were developed explicitly for classroom study. There is a plethora of ongoing UNIX-related research systems, including Mach, Chorus, Comandos, and Roisin. The original developers,

Ritchie and Thompson, were honored in 1983 by the Association for Computing Machinery Turing award for their work on UNIX.

The specific UNIX version used in this chapter is the Intel version of FreeBSD. This system is used because it implements many interesting operating-system concepts, such as demand paging with clustering, and networking. The FreeBSD project began in early 1993 to produce a snapshot of 386BSD to solve problems that were unable to be resolved using the existing patch mechanism. 386BSD was derived from 4.3BSD-Lite (Net/2) and was originally released in June 1992 by William Jolitz. FreeBSD (coined by David Greenman) 1.0 was released in December 1993. FreeBSD 1.1 was released in May 1994 and both versions were based on 4.3BSD-Lite. Legal issues between UCB and Novell required that 4.3BSD-Lite code could no longer be used, so the final 4.3BSD-Lite Release was made in July 1994 (FreeBSD 1.1.5.1).

FreeBSD was reinvented based on the 4.4BSD-Lite code base, which was incomplete, releasing FreeBSD 2.0 in November 1994. Later releases include releases 2.0.5 in June 1995, 2.1.5 in August 1996, 2.1.7.1 in February 1997, 2.2.1 in April 1997, 2.2.8 in November 1998, 3.0 in October 1998, 3.1 in February 1999, 3.2 in May 1999, 3.3 in September 1999, 3.4 in December 1999, 3.5 in June 2000, 4.0 in March 2000, 4.1 in July 2000 and 4.2 in November 2000.

The goal of the FreeBSD project is to provide software that may be used for any purpose with no strings attached. The idea is that the code will get the widest possible use and provide the most benefit. Fundamentally, it is the same as described in McKusick et al. [1984] with the addition of a merged virtual memory and filesystem buffer cache, kernel queues, and soft filesystem updates. At the present, it runs primarily on Intel platforms, although Alpha platforms are supported. Work is underway to port to other processor platforms as well.

A.2   ■   Design Principles

UNIX was designed to be a time-sharing system. The standard user interface (the shell) is simple and can be replaced by another, if desired. The file system is a multilevel tree, which allows users to create their own subdirectories. Each user data file is simply a sequence of bytes.

Disk files and I/O devices are treated as similarly as possible. Thus, device dependencies and peculiarities are kept in the kernel as much as possible; even in the kernel, most of them are confined to the device drivers.

UNIX supports multiple processes. A process can easily create new processes. CPU scheduling is a simple priority algorithm. FreeBSD uses demand paging as a mechanism to support memory-management and CPU-scheduling decisions. Swapping is used if a system is suffering from excess paging.

Because UNIX was originated first by one programmer, Ken Thompson, and then by another, Dennis Ritchie, as a system for their own convenience, it was

small enough to understand. Most of the algorithms were selected for *simplicity,* not for speed or sophistication. The intent was to have the kernel and libraries provide a small set of facilities that was sufficiently powerful to allow a person to build a more complex system if one were needed. UNIX's clean design has resulted in many imitations and modifications.

Although the designers of UNIX had a significant amount of knowledge about other operating systems, UNIX had no elaborate design spelled out before its implementation. This flexibility appears to have been one of the key factors in the development of the system. Some design principles were involved, however, even though they were not made explicit at the outset.

The UNIX system was designed by programmers for programmers. Thus, it has always been interactive, and facilities for program development have always been a high priority. Such facilities include the program *make* (which can be used to check to see which of a collection of source files for a program need to be compiled, and then to do the compiling) and the *Source Code Control System (SCCS)* (which is used to keep successive versions of files available without having to store the entire contents of each step). The primary version control system used by freebsd is the *Concurrent Versions System* (CVS) due to the large number of developers operating on and using the code.

The operating system is written mostly in C, which was developed to support UNIX, since neither Thompson nor Ritchie enjoyed programming in assembly language. The avoidance of assembly language was also necessary because of the uncertainty about the machine or machines on which UNIX would be run. It has greatly simplified the problems of moving UNIX from one hardware system to another.

From the beginning, UNIX development systems have had all the UNIX sources available on-line, and the developers have used the systems under development as their primary systems. This pattern of development has greatly facilitated the discovery of deficiencies and their fixes, as well as of new possibilities and their implementations. It has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages: if something is broken, it can be fixed at a local site; there is no need to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Where other systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash called *panic*. Instead of attempting to cure such conditions, UNIX tries to prevent them. Where other systems would use brute force or macro-expansion, UNIX mostly has had to develop more subtle, or at least simpler, approaches.

These early strengths of UNIX produced much of its popularity, which in turn produced new demands that challenged those strengths. UNIX was used for tasks such as networking, graphics, and real-time operation, which did not always fit into its original text-oriented model. Thus, changes were made to

(the users)		
shells and commands compilers and interpreters system libraries		
system-call interface to the kernel		
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
kernel interface to the hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

Figure A.2   4.4BSD layer structure.

certain internal facilities and new programming interfaces were added. These new facilities, and others—particularly window interfaces—required large amounts of code to support them, radically increasing the size of the system. For instance, networking and windowing both doubled the size of the system. This pattern in turn pointed out the continued strength of UNIX—whenever a new development occurred in the industry, UNIX could usually absorb it, but still remain UNIX.

A.3 ■ Programmer Interface

As do most operating systems, UNIX consists of two separable parts: the kernel and the systems programs. We can view the UNIX operating system as being layered, as shown in Figure A.2. Everything below the system-call interface and above the physical hardware is the *kernel*. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the *programmer interface* to UNIX; the set of systems programs commonly available defines the *user interface*. The programmer and user interface define the context that the kernel must support.

Most systems programs are written in C, and the *UNIX Programmer's Manual* presents all system calls as C functions. A system program written in C for FreeBSD on the Pentium can generally be moved to an Alpha freebsd system and simply recompiled, even though the two systems are quite different. The details

of system calls are known only to the compiler. This feature is a major reason for the portability of UNIX programs.

System calls for UNIX can be roughly grouped into three categories: file manipulation, process control, and information manipulation. In Chapter 3, we listed a fourth category, device manipulation, but since devices in UNIX are treated as (special) files, the same system calls support both files and devices (although there is an extra system call for setting device parameters).

A.3.1 File Manipulation

A *file* in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose a structure on files. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (which is the linefeed character in ASCII), but the kernel knows nothing of this convention.

Files are organized in tree-structured *directories*. Directories are themselves files that contain information on how to find other files. A *path name* to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically, it consists of individual file-name elements separated by the slash character. For example, in */usr/local/font,* the first slash indicates the root of the directory tree, called the *root* directory. The next element, *usr,* is a subdirectory of the root, *local* is a subdirectory of *usr,* and *font* is a file or directory in the directory *local*. Whether *font* is an ordinary file or a directory cannot be determined from the path-name syntax.

The UNIX file system has both *absolute path names* and *relative path names*. Absolute path names start at the root of the file system and are distinguished by a slash at the beginning of the path name; */usr/local/font* is an absolute path name. Relative path names start at the *current directory*, which is an attribute of the process accessing the path name. Thus, *local/font* indicates a file or directory named *font* in the directory *local* in the current directory, which might or might not be */usr*.

A file may be known by more than one name in one or more directories. Such multiple names are known as *links,* and all links are treated equally by the operating system. FreeBSD also supports *symbolic links,* which are files containing the path name of another file. The two kinds of links are also known as *hard links* and *soft links*. Soft (symbolic) links, unlike hard links, may point to directories and may cross file-system boundaries.

The file name "." in a directory is a hard link to the directory itself. The file name ".." is a hard link to the parent directory. Thus, if the current directory is */user/jlp/programs,* then *../bin/wdf* refers to */user/jlp/bin/wdf*.

Hardware devices have names in the file system. These *device special files* or *special files* are known to the kernel as device interfaces, but are nonetheless accessed by the user by much the same system calls as are other files.
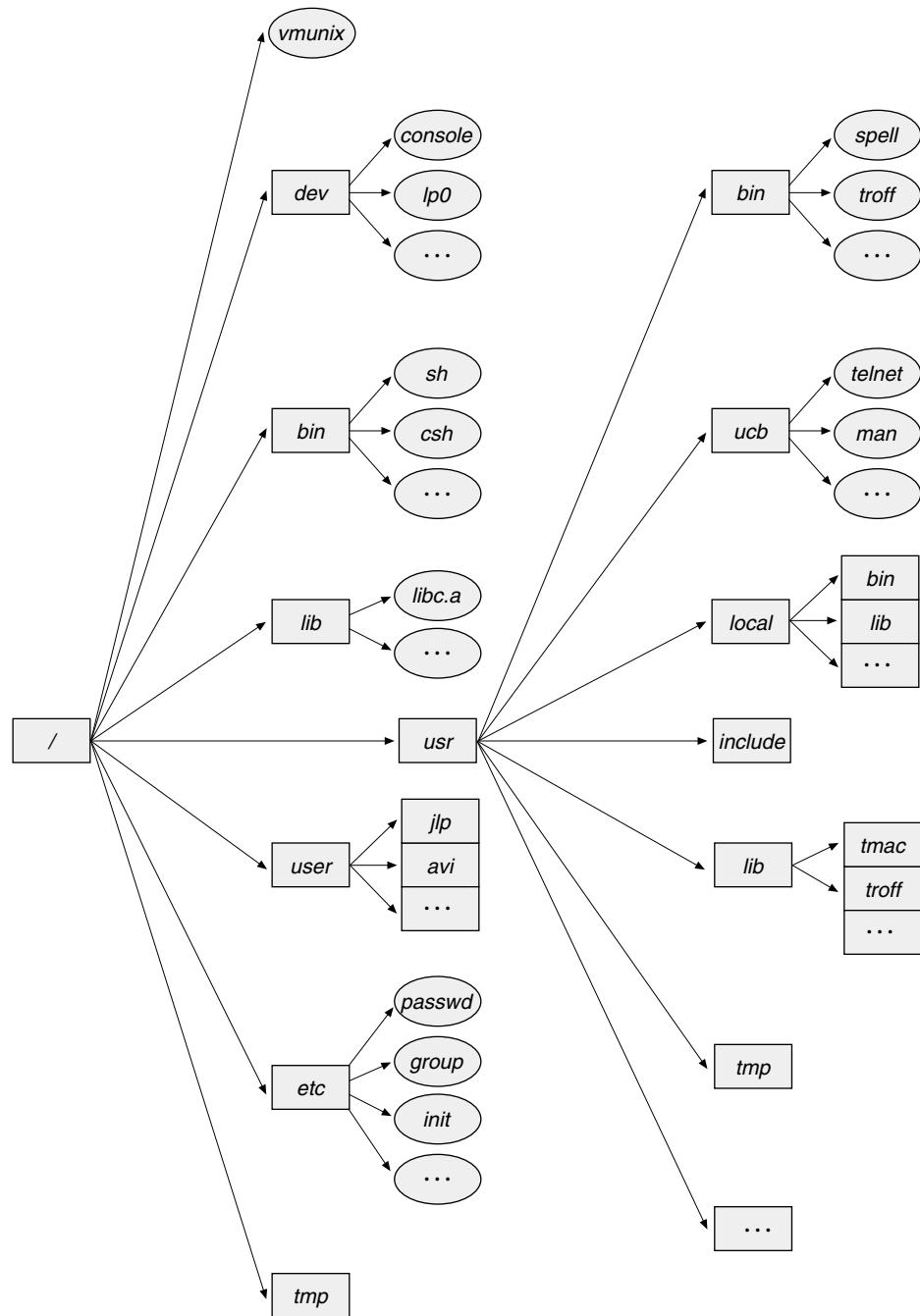
Figure A.3   Typical UNIX directory structure.

Figure A.3 shows a typical UNIX file system. The root (/) normally contains a small number of directories as well as */kernel,* the binary boot image of the operating system; */dev* contains the device special files, such as */dev/console, /dev/lp0, /dev/mt0,* and so on; */bin* contains the binaries of the essential UNIX systems programs. Other binaries may be in */usr/bin* (for applications systems programs, such as text formatters), */usr/compat* for programs from other operating systems such as Linux or */usr/local/bin* (for systems programs written at the local site). Library files—such as the C, Pascal, and FORTRAN subroutine libraries—are kept in */lib* (or */usr/lib* or */usr/local/lib*).

The files of users themselves are stored in a separate directory for each user, typically in */usr*. Thus, the user directory for *carol* would normally be in */usr/carol*. For a large system, these directories may be further grouped to ease administration, creating a file structure with */usr/prof/avi* and */usr/staff/carol*. Administrative files and programs, such as the password file, are kept in */etc*. Temporary files can be put in */tmp,* which is normally erased during system boot, or in */usr/tmp*.

Each of these directories may have considerably more structure. For example, the font-description tables for the troff formatter for the Merganthaler 202 typesetter are kept in */usr/lib/troff/dev202*. All the conventions concerning the location of specific files and directories have been defined by programmers and their programs; the operating-system kernel needs only */etc/init,* which is used to initialize terminal processes, to be operable.

System calls for basic file manipulation are **creat, open, read, write, close, unlink,** and **trunc**. The **creat** system call, given a path name, creates an (empty) file (or truncates an existing one). An existing file is opened by the **open** system call, which takes a path name and a mode (such as read, write, or read–write) and returns a small integer, called a *file descriptor*. A file descriptor may then be passed to a **read** or **write** system call (along with a buffer address and the number of bytes to transfer) to perform data transfers to or from the file. A file is closed when its file descriptor is passed to the **close** system call. The **trunc** call reduces the length of a file to 0.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each **read** or **write** updates the current offset into the file, which is associated with the file-table entry and is used to determine the position in the file for the next **read** or **write**. The **lseek** system call allows the position to be reset explicitly. It also allows the creation of sparse files (files with "holes" in them). The **dup** and **dup2** system calls can be used to produce a new file descriptor that is a copy of an existing one. The **fcntl** system call can also do that, and in addition can examine or set various parameters of an open file. For example, it can make each succeeding **write** to an open file append to the end of that file. There is an additional system call, **ioctl,** for manipulating device parameters. It can set the baud rate of a serial port, or rewind a tape, for instance.

Information about the file (such as its size, protection modes, owner, and so on) can be obtained by the **stat** system call. Several system calls allow some of this information to be changed: **rename** (change file name), **chmod** (change the protection mode), and **chown** (change the owner and group). Many of these system calls have variants that apply to file descriptors instead of file names. The **link** system call makes a hard link for an existing file, creating a new name for an existing file. A link is removed by the **unlink** system call; if it is the last link, the file is deleted. The **symlink** system call makes a symbolic link.

Directories are made by the **mkdir** system call and are deleted by **rmdir**. The current directory is changed by **cd**.

Although it is possible to use the standard file calls on directories, it is inadvisable to do so, since directories have an internal structure that must be preserved. Instead, another set of system calls is provided to open a directory, to step through each file entry within the directory, to close the directory, and to perform other functions; these are **opendir, readdir, closedir,** and others.

A.3.2 Process Control

A *process* is a program in execution. Processes are identified by their *process identifier,* which is an integer. A new process is created by the **fork** system call. The new process consists of a copy of the address space of the original process (the same program and the same variables with the same values). Both processes (the parent and the child) continue execution at the instruction after the **fork** with one difference: The return code for the **fork** is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the **execve** system call is used after a fork by one of the two processes to replace that process' virtual memory space with a new program. The **execve** system call loads a binary file into memory (destroying the memory image of the program containing the **execve** system call) and starts its execution.

A process may terminate by using the **exit** system call, and its parent process may wait for that event by using the **wait** system call. If the child process crashes, the system simulates the **exit** call. The **wait** system call provides the process id of a terminated child so that the parent can tell which of possibly many children terminated. A second system call, **wait3,** is similar to **wait** but also allows the parent to collect performance statistics about the child. Between the time the child exits, and the time the parent completes one of the **wait** system calls, the child is *defunct*. A defunct process can do nothing, but exists merely so that the parent can collect its status information. If the parent process of a defunct process exits before a child, the defunct process is inherited by the *init* process (which in turn **wait**s on it) and becomes a *zombie* process. A typical use of these facilities is shown in Figure A.4.
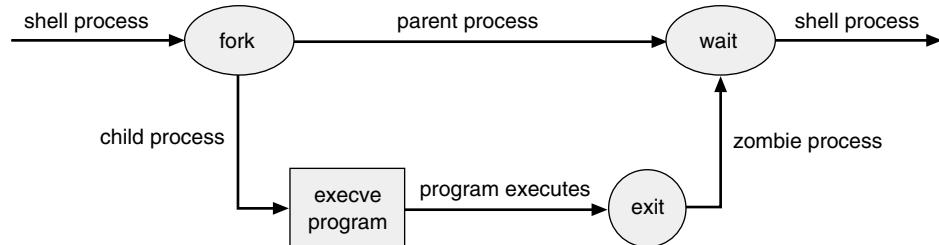
The simplest form of communication between processes is by *pipes,* which may be created before the **fork,** and whose endpoints are then set up between

the **fork** and the **execve**. A pipe is essentially a queue of bytes between two processes. The pipe is accessed by a file descriptor, like an ordinary file. One process writes into the pipe, and the other reads from the pipe. The size of the original pipe system was fixed by the system. With FreeBSD , pipes are implemented on top of the socket system, which has variable-sized buffers. Reading from an empty pipe or writing into a full pipe causes the process to be blocked until the state of the pipe changes. Special arrangements are needed for a pipe to be placed between a parent and child (so only one is reading and one is writing).

All user processes are descendants of one original process, called *init*. Each terminal port available for interactive use has a *getty* process forked for it by *init*. The *getty* process initializes terminal line parameters and waits for a user's *login name,* which it passes through an **execve** as an argument to a *login* process. The *login* process collects the user's password, encrypts the password, and compares the result to an encrypted string taken from the file */etc/passwd*. If the comparison is successful, the user is allowed to log in. The *login* process executes a *shell,* or command interpreter, after setting the numeric *user identifier* of the process to that of the user logging in. (The shell and the user identifier are found in */etc/passwd* by the user's login name.) It is with this shell that the user ordinarily communicates for the rest of the login session; the shell itself forks subprocesses for the commands the user tells it to execute.

The user identifier is used by the kernel to determine the user's permissions for certain system calls, especially those involving file accesses. There is also a *group identifier,* which is used to provide similar privileges to a collection of users. In FreeBSD a process may be in several groups simultaneously. The *login* process puts the shell in all the groups permitted to the user by the files */etc/passwd* and */etc/group*.

There are actually two user identifiers used by the kernel: the *effective user identifier* is the identifier used to determine file access permissions. If the file of a program being loaded by an **execve** has the **setuid** bit set in its inode, the effective user identifier of the process is set to the user identifier of the owner of the file, whereas the *real user identifier* is left as it was. This scheme allows certain processes to have more than ordinary privileges while still being executable by

Figure A.4   A shell forks a subprocess to execute a program.

ordinary users. The **setuid** idea was patented by Dennis Ritchie (U.S. Patent 4,135,240) and is one of the distinctive features of UNIX. There is a similar **setgid** bit for groups. A process may determine its real and effective user identifier with the **getuid** and **geteuid** calls, respectively. The **getgid** and **getegid** calls determine the process' real and effective group identifier, respectively. The rest of a process' groups may be found with the **getgroups** system call.

A.3.3 Signals

Signals are a facility for handling exceptional conditions similar to software interrupts. There are 20 different signals, each corresponding to a distinct condition. A signal may be generated by a keyboard interrupt, by an error in a process (such as a bad memory reference), or by a number of asynchronous events (such as timers or job-control signals from the shell). Almost any signal may also be generated by the **kill** system call.

The *interrupt* signal, SIGINT, is used to stop a command before that command completes. It is usually produced by the ˆC character (ASCII 3). As of 4.2BSD, the important keyboard characters are defined by a table for each terminal and can be redefined easily. The *quit* signal, SIGQUIT, is usually produced by the ˆbs character (ASCII 28). The *quit* signal both stops the currently executing program and dumps its current memory image to a file named *core* in the current directory. The core file can be used by debuggers. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of the legal virtual-memory space of a process.

Arrangements can be made either for most signals to be ignored (to have no effect), or for a routine in the user process (a signal handler) to be called. A signal handler may safely do one of two things before returning from catching a signal: call the **exit** system call, or modify a global variable. There is one signal (the *kill* signal, number 9, SIGKILL) that cannot be ignored or caught by a signal handler. SIGKILL is used, for example, to kill a runaway process that is ignoring other signals such as SIGINT or SIGQUIT.

Signals can be lost: If another signal of the same kind is sent before a previous signal has been accepted by the process to which it is directed, the first signal will be overwritten and only the last signal will be seen by the process. In other words, a call to the signal handler tells a process that there has been at least one occurance of the signal. Also, there is no relative priority among UNIX signals. If two different signals are sent to the same process at the same time, it is indeterminate which one the process will receive first.

Signals were originally intended to deal with exceptional events. As is true of the use of most other features in UNIX, however, signal use has steadily expanded. 4.1BSD introduced job control, which uses signals to start and stop subprocesses on demand. This facility allows one shell to control multiple processes: starting, stopping, and backgrounding them as the user wishes. The SIGWINCH signal, invented by Sun Microsystems, for informing a process that

the window in which output is being displayed has changed size. Signals are also used to deliver urgent data from network connections.

Users also wanted more reliable signals, and a bug fix in an inherent race condition in the old signals implementation. Thus, 4.2BSD also brought with it a race-free, reliable, separately implemented signal capability. It allows individual signals to be blocked during critical sections, and has a new system call to let a process sleep until interrupted. It is similar to hardware-interrupt functionality. This capability is now part of the POSIX standard.

A.3.4 Process Groups

Groups of related processes frequently cooperate to accomplish a common task. For instance, processes may create, and communicate over, pipes. Such a set of processes is termed a *process group*, or a *job*. Signals may be sent to all processes in a group. A process usually inherits its process group from its parent, but the **setpgrp** system call allows a process to change its group.

Process groups are used by the C shell to control the operation of multiple jobs. Only one process group may use a terminal device for I/O at any time. This *foreground* job has the attention of the user on that terminal while all other nonattached jobs (*background* jobs) perform their function without user interaction. Access to the terminal is controlled by process group signals. Each job has a *controlling terminal* (again, inherited from its parent). If the process group of the controlling terminal matches the group of a process, that process is in the foreground, and is allowed to perform I/O. If a nonmatching (background) process attempts the same, a SIGTTIN or SIGTTOU signal is sent to its process group. This signal usually results in the process group freezing until it is foregrounded by the user, at which point it receives a SIGCONT signal, indicating that the process can perform the I/O. Similarly, a SIGSTOP may be sent to the foreground process group to freeze it.

A.3.5 Information Manipulation

System calls exist to set and return both an interval timer (**getitimer**/**setitimer**) and the current time (**gettimeofday**/**settimeofday**) in microseconds. In addition, processes can ask for their process identifier (**getpid**), their group identifier (**getgid**), the name of the machine on which they are executing (**gethostname**), and many other values.

A.3.6 Library Routines

The system-call interface to UNIX is supported and augmented by a large collection of library routines and header files. The header files provide the definition of complex data structures used in system calls. In addition, a large library of functions provides additional program support.

For example, the UNIX I/O system calls provide for the reading and writing of blocks of bytes. Some applications may want to read and write only 1 byte at a time. Although it would be possible to read and write 1 byte at a time, that would require a system call for each byte—a very high overhead. Instead, a set of standard library routines (the standard I/O package accessed through the header file *<stdio.h>*) provides another interface, which reads and writes several thousand bytes at a time using local buffers, and transfers between these buffers (in user memory) when I/O is desired. Formatted I/O is also supported by the standard I/O package.

Additional library support is provided for mathematical functions, network access, data conversion, and so on. The FreeBSD kernel supports over 300 system calls; the C program library has over 300 library functions. Although the library functions eventually result in system calls where necessary (for example, the *getchar* library routine will result in a **read** system call if the file buffer is empty), it is generally unnecessary for the programmer to distinguish between the basic set of kernel system calls and the additional functions provided by library functions.

A.4   ■   User Interface

Both the programmer and the user of a UNIX system deal mainly with the set of systems programs that have been written and are available for execution. These programs make the necessary system calls to support their function, but the system calls themselves are contained within the program and do not need to be obvious to the user.

The common systems programs can be grouped into several categories; most of them are file or directory oriented. For example, the systems programs to manipulate directories are *mkdir* to create a new directory, *rmdir* to remove a directory, *cd* to change the current directory to another, and *pwd* to print the absolute path name of the current (working) directory.

The *ls* program lists the names of the files in the current directory. Any of 28 options can ask that properties of the files be displayed also. For example, the *-l* option asks for a long listing, showing the file name, owner, protection, date and time of creation, and size. The *cp* program creates a new file that is a copy of an existing file. The *mv* program moves a file from one place to another in the directory tree. In most cases, this move simply requires a renaming of the file; if necessary, however, the file is copied to the new location and the old copy is deleted. A file is deleted by the *rm* program (which makes an **unlink** system call).

To display a file on the terminal, a user can run *cat*. The *cat* program takes a list of files and concatenates them, copying the result to the standard output, commonly the terminal. On a high-speed cathode-ray tube (CRT) display, of course, the file may speed by too fast to be read. The *more* program displays the file one screen at a time, pausing until the user types a character to continue to

the next screen. The *head* program displays just the first few lines of a file; *tail* shows the last few lines.

These are the basic systems programs widely used in UNIX. In addition, there are a number of editors (*ed, sed, emacs, vi,* and so on), compilers (C, Pascal, FORTRAN, and so on), and text formatters (troff, TEX, scribe, and so on). There are also programs for sorting (*sort*) and comparing files (*cmp, diff*), looking for patterns (*grep, awk*), sending mail to other users (*mail*), and many other activities.

A.4.1 Shells and Commands

Both user-written and systems programs are normally executed by a command interpreter. The command interpreter in UNIX is a user process like any other. It is called a *shell,* as it surrounds the kernel of the operating system. Users can write their own shell, and there are, in fact, several shells in general use. The *Bourne shell,* written by Steve Bourne, is probably the most widely used—or, at least, it is the most widely available. The *C shell,* mostly the work of Bill Joy, a founder of Sun Microsystems, is the most popular on BSD systems. The Korn shell, by Dave Korn, has become popular because it combines the features of the Bourne shell and the C shell.

The common shells share much of their command-language syntax. UNIX is normally an interactive system. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line. For instance, in the line

<p style="text-align: center;">*% ls -l*</p>

the percent sign is the usual C shell prompt, and the *ls -l* (typed by the user) is the (long) list-directory command. Commands can take arguments, which the user types after the command name on the same line, separated by white space (spaces or tabs).

Although there are a few commands built into the shells (such as *cd*), a typical command is an executable binary object file. A list of several directories, the *search path,* is kept by the shell. For each command, each of the directories in the search path is searched, in order, for a file of the same name. If a file is found, it is loaded and executed. The search path can be set by the user. The directories */bin* and */usr/bin* are almost always in the search path, and a typical search path on a FreeBSD system might be

<p style="text-align: center;">*(. /usr/avi/bin /usr/local/bin /bin /usr/bin)*</p>

The *ls* command's object file is */bin/ls,* and the shell itself is */bin/sh* (the Bourne shell) or */bin/csh* (the C shell).

Execution of a command is done by a **fork** system call followed by an **execve** of the object file. The shell usually then does a **wait** to suspend its own execution until the command completes (Figure A.4). There is a simple syntax (an ampersand [&] at the end of the command line) to indicate that the shell should *not* wait for the completion of the command. A command left running in this manner while the shell continues to interpret further commands is said to be a *background* command, or to be running in the background. Processes for which the shell *does* wait are said to run in the *foreground*.

The C shell in FreeBSD systems provides a facility called *job control* (partially implemented in the kernel), as mentioned previously. Job control allows processes to be moved between the foreground and the background. The processes can be stopped and restarted on various conditions, such as a background job wanting input from the user's terminal. This scheme allows most of the control of processes provided by windowing or layering interfaces, but requires no special hardware. Job control is also useful in window systems, such as the X Window System developed at MIT. Each window is treated as a terminal, allowing multiple processes to be in the foreground (one per window) at any one time. Of course, background processes may exist on any of the windows. The Korn shell also supports job control, and it is likely that job control (and process groups) will be standard in future versions of UNIX.

A.4.2 Standard I/O

Processes can open files as they like, but most processes expect three file descriptors (numbers 0, 1, and 2) to be open when they start. These file descriptors are inherited across the **fork** (and possibly the **execve**) that created the process. They are known as *standard input* (0), *standard output* (1), and *standard error* (2). All three are frequently open to the user's terminal. Thus, the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard-error file descriptor is also open for writing and is used for error output; standard output is used for ordinary output. Most programs can also accept a file (rather than a terminal) for standard input and standard output. The program does not care where its input is coming from and where its output is going. This is one of the elegant design features of UNIX.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process. Changing a standard file is called *I/O redirection*. The syntax for I/O redirection is shown in Figure A.5. In this example, the *ls* command produces a listing of the names of files in the current directory, the *pr* command formats that list into pages suitable for a printer, and the *lpr* command spools the formatted output to a printer, such as */dev/lp0*. The subsequent command forces all output and all error messages to be redirected to a file. Without the ampersand, error messages appear on the terminal.

command	meaning of command
% *ls* > *filea*	direct output of *ls* to file *filea*
% *pr* < *filea* > *fileb*	input from *filea* and output to *fileb*
% *lpr* < *fileb*	input from *fileb*
% % make program > & errs	save both standard output and standard error in a file

Figure A.5   Standard I/O redirection.

A.4.3 Pipelines, Filters, and Shell Scripts

The first three commands of Figure A.5 could have been coalesced into the one command

% ls | pr | lpr

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. A pipe is used to carry the data from one process to the other. One process writes into one end of the pipe, and another process reads from the other end. In the example, the write end of one pipe would be set up by the shell to be the standard output of *ls,* and the read end of the pipe would be the standard input of *pr;* there would be another pipe between *pr* and *lpr*.

A command such as *pr* that passes its standard input to its standard output, performing some processing on it, is called a *filter*. Many UNIX commands can be used as filters. Complicated functions can be pieced together as pipelines of common commands. Also, common functions, such as output formatting, do not need to be built into numerous commands, because the output of almost any program can be piped through *pr* (or some other appropriate filter).

Both of the common UNIX shells are also programming languages, with shell variables and the usual higher-level programming-language control constructs (loops, conditionals). The execution of a command is analogous to a subroutine call. A file of shell commands, a *shell script,* can be executed like any other command, with the appropriate shell being invoked automatically to read it. *Shell programming* thus can be used to combine ordinary programs conveniently for sophisticated applications without the necessity of any programming in conventional languages.

This external user view is commonly thought of as the definition of UNIX, yet it is the most easily changed definition. Writing a new shell with a quite different syntax and semantics would greatly change the user view while not

changing the kernel or even the programmer interface. Several menu-driven and iconic interfaces for UNIX now exist, and the X Window System is rapidly becoming a standard. The heart of UNIX is, of course, the kernel. This kernel is much more difficult to change than is the user interface, because all programs depend on the system calls that it provides to remain consistent. Of course, new system calls can be added to increase functionality, but programs must then be modified to use the new calls.

A.5 ■ Process Management

A major design problem for operating systems is the representation of processes. One substantial difference between UNIX and many other systems is the ease with which multiple processes can be created and manipulated. These processes are represented in UNIX by various control blocks. There are no system control blocks accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The information in these control blocks is used by the kernel for process control and CPU scheduling.

A.5.1 Process Control Blocks

The most basic data structure associated with processes is the *process structure*. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (such as the priority of the process), and pointers to other control blocks. There is an array of process structures whose length is defined at system linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue), and there are pointers from each process structure to the process' parent, to its youngest living child, and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The *virtual address space* of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but may grow separately, and usually in opposite directions: most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack, and is usually read-only. The debugger puts a text segment in read-write mode to be able to allow insertion of breakpoints.

Every process with sharable text (almost all, under FreeBSD) has a pointer from its process structure to a *text structure*. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures, and where the page table for the text segment can be

found on disk when it is swapped. The text structure itself is always resident in main memory: an array of such structures is allocated at system link time. The text, data, and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The *page tables* record information on the mapping from the process' virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process' page table.

Information about the process that is needed only when the process is resident (that is, not swapped out) is kept in the *user structure* (or *u structure*), rather than in the process structure. The *u* structure is mapped read-only into user virtual address space, so user processes can read its contents. It is writable by the kernel. The *u* structure contains a copy of the Process Control Block or PCB is kept here for saving the process' general registers, stack pointer, program counter, and page-table base registers when the process is not running. There is space to keep system-call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary user, the current directory and the table of open files are maintained in the user structure.

Every process has both a user and a system mode. Most ordinary work is done in *user mode,* but, when a system call is made, it is performed in *system mode*. The system and user phases of a process never execute simultaneously. When a process is executing in system mode, a *kernel stack* for that process is used, rather than the user stack belonging to that process. The kernel stack for the process immediately follows the user structure: The kernel stack and the user structure together compose the *system data segment* for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure A.6 illustrates how the process structure is used to find the various parts of a process.
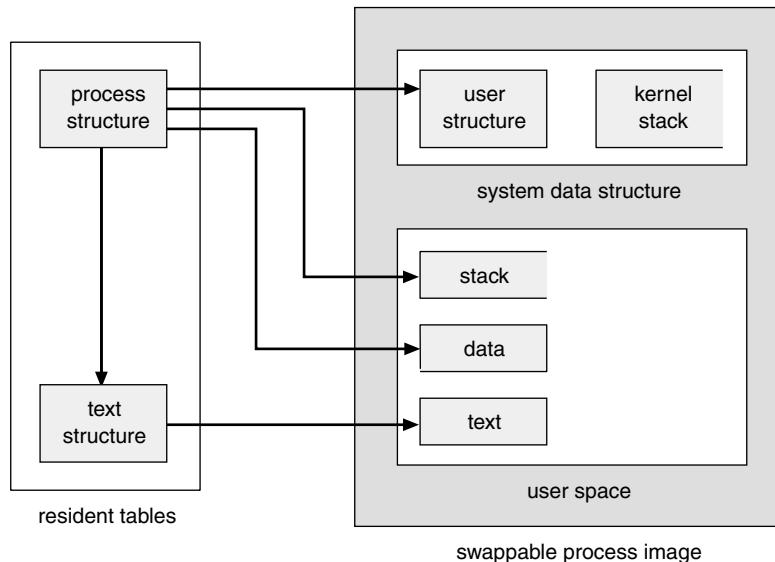
The **fork** system call allocates a new process structure (with a new process identifier) for the child process, and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text; the appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling, and most similar properties of a process.

The **vfork** system call does *not* copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this

system call is by a shell to execute a command and to wait for its completion. The parent process uses **vfork** to produce the child process. Because the child process wishes to use an **execve** immediately to change its virtual address space completely, there is no need for a complete copy of the parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the **vfork** and the **execve**. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls **vfork** until the child either calls **execve** or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, **vfork** can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the **execve** occurs. An alternative is to share all pages by duplicating the page table, but to mark the entries of both page tables as *copy-on-write*. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An **execve** system call creates no new process or user structure; rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an **execve**). Most signal-handling properties are preserved, but arrangements to

Figure A.6   Finding parts of a process using the process structure.

call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

A.5.2 CPU Scheduling

CPU scheduling in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than "pzero" and cannot be killed by signals. Ordinary user processes have positive priorities and thus are all less likely to be run than are any system process, although user processes can set precedence over one another through the *nice* command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa, so there is negative feedback in CPU scheduling and it is difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. FreeBSD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the *timeout* mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval; the subroutine to be called in this case causes the rescheduling and then resubmits a *timeout* to call itself again. The priority recomputation is also timed by a subroutine that resubmits a *timeout* for itself.

There is no preemption of one process by another in the kernel. A process may relinquish the CPU because it is waiting on I/O or because its time slice has expired. When a process chooses to relinquish the CPU, it goes to sleep on an *event*. The kernel primitive used for this purpose is called *sleep* (not to be confused with the user-level library routine of the same name). It takes an argument, which is by convention the address of a kernel data structure related to an *event* that the process wants to occur before that process is awakened. When the event occurs, the system process that knows about it calls *wakeup* with the address corresponding to the event, and *all* processes that had done a *sleep* on the same address are put in the ready queue to be run.

For example, a process waiting for disk I/O to complete will *sleep* on the address of the buffer header corresponding to the data being transferred. When the interrupt routine for the disk driver notes that the transfer is complete, it calls *wakeup* on the buffer header. The interrupt uses the kernel stack for whatever process happened to be running at the time, and the *wakeup* is done from that system process.

The process that actually does run is chosen by the scheduler. *Sleep* takes a second argument, which is the scheduling priority to be used for this purpose.

This priority argument, if less than "pzero," also prevents the process from being awakened prematurely by some exceptional event, such as a *signal*.

When a signal is generated, it is left pending until the system half of the affected process next runs. This event usually happens soon, since the signal normally causes the process to be awakened if the process has been waiting for some other condition.

There is no memory associated with events, and the caller of the routine that does a *sleep* on an event must be prepared to deal with a premature return, including the possibility that the reason for waiting has vanished.

There are *race conditions* involved in the event mechanism. If a process decides (because of checking a flag in memory, for instance) to sleep on an event, and the event occurs before the process can execute the primitive that does the actual sleep on the event, the process sleeping may then sleep forever. We prevent this situation by raising the hardware processor priority during the critical section so that no interrupts can occur, and thus only the process desiring the event can run until it is sleeping. Hardware processor priority is used in this manner to protect critical regions throughout the kernel, and is the greatest obstacle to porting UNIX to multiple processor machines. However, this problem has not prevented such ports from being done repeatedly.

Many processes such as text editors are I/O bound and usually will be scheduled mainly on the basis of waiting for I/O. Experience suggests that the UNIX scheduler performs best with I/O-bound jobs, as can be observed when there are several CPU-bound jobs, such as text formatters or language interpreters, running.

What has been referred to here as *CPU scheduling* corresponds closely to the *short-term scheduling* of Chapter 4, although the negative-feedback property of the priority scheme provides some long-term scheduling in that it largely determines the long-term *job mix*. Medium-term scheduling is done by the swapping mechanism described in Section A.6.

A.6   ■   Memory Management

Much of UNIX's early development was done on a PDP-11. The PDP-11 has only eight segments in its virtual address space, and each of these are at most 8192 bytes. The larger machines, such as the PDP-11/70, allow separate instruction and address spaces, which effectively double the address space and number of segments, but this address space is still relatively small. In addition, the kernel was even more severely constrained due to dedication of one data segment to interrupt vectors, another to point at the per-process system data segment, and yet another for the UNIBUS (system I/O bus) registers. Further, on the smaller PDP-11s, total physical memory was limited to 256K. The total memory resources were insufficient to justify or support complex memory-management algorithms. Thus, UNIX swapped entire process memory images.

A.6.1 Paging

Berkeley introduced paging to UNIX with 3BSD. VAX 4.2BSD is a demand-paged virtual-memory system. External fragmentation of memory is eliminated by paging. (There is, of course, internal fragmentation, but it is negligible with a reasonably small page size.) Swapping can be kept to a minimum because more jobs can be kept in main memory, because paging allows execution with only parts of each process in memory.

Demand paging is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

There are a few optimizations. If the page needed is still in the page table for the process, but has been marked invalid by the page-replacement process, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the list of free frames. When most processes are started, many of their pages are prepaged and are put on the free list for recovery by this mechanism. Arrangements may also be made for a process to have no prepaging on startup, but that is seldom done, as it results in more page-fault overhead, being closer to pure demand paging. FreeBSD implements page coloring with paging queues. The queues are arranged according to the size of the processor's L1 and L2 caches and when a new page needs to be allocated, FreeBSD tries to get one that is optimally aligned for the cache.

If the page has to be fetched from disk, it must be locked in memory for the duration of the transfer. This locking ensures that the page will not be selected for page replacement. Once the page is fetched and mapped properly, it must remain locked if raw physical I/O is being done on it.

The *page-replacement* algorithm is more interesting. 4.2BSD uses a modification of the *second chance* (clock) algorithm described in Section 10.4.5. The map of all nonkernel main memory (the *core map* or *cmap*) is swept linearly and repeatedly by a software *clock hand*. When the clock hand reaches a given frame, if the frame is marked as in use by some software condition (for example, physical I/O is in progress using it), or the frame is already free, the frame is left untouched, and the clock hand sweeps to the next frame. Otherwise, the corresponding text or process page-table entry for this frame is located. If the entry is already invalid, the frame is added to the free list; otherwise, the page-table entry is made invalid but reclaimable (that is, if it does not get paged out by the next time it is wanted, it can just be made valid again).

BSD Tahoe added support for systems which do implement the reference bit. On such systems, one pass of the clock hand turns the reference bit off, and a second pass places those pages whose reference bits remain off onto the free list for replacement. Of course, if the page is dirty, it must first be written to disk before being added to the free list. Pageouts are done in clusters to improve performance.

There are checks to make sure that the number of valid data pages for a process does not fall too low, and to keep the paging device from being flooded with requests. There is also a mechanism by which a process may limit the amount of main memory it uses.

The LRU clock hand scheme is implemented in the *pagedaemon,* which is process 2 (remember that the *swapper* is process 0, and *init* is process 1). This process spends most of its time sleeping, but a check is done several times per second (scheduled by a *timeout*) to see if action is necessary; if it is, process 2 is awakened. Whenever the number of free frames falls below a threshold, *lotsfree,* the *pagedaemon* is awakened; thus, if there is always a large amount of free memory, the *pagedaemon* imposes no load on the system, because it never runs.

The sweep of the clock hand each time the *pagedaemon* process is awakened (that is, the number of frames scanned, which is usually more than the number paged out), is determined both by the number of frames lacking to reach *lotsfree* and by the number of frames that the *scheduler* has determined are needed for various reasons (the more frames needed, the longer the sweep). If the number of frames free rises to *lotsfree* before the expected sweep is completed, the hand stops and the *pagedaemon* process sleeps. The parameters that determine the range of the clock-hand sweep are determined at system startup according to the amount of main memory, such that *pagedaemon* does not use more than 10 percent of all CPU time.

If the *scheduler* decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved. This swapping usually happens only if several conditions are met: load average is high, free memory has fallen below a low limit, *minfree;* and the average memory available over recent time is less than a desirable amount, *desfree,* where *lotsfree > desfree > minfree*. In other words, only a chronic shortage of memory with several processes trying to run will cause swapping, and even then free memory has to be extremely low at the moment. (An excessive paging rate or a need for memory by the kernel itself may also enter into the calculations, in rare cases.) Processes may be swapped by the *scheduler,* of course, for other reasons (such as simply for not running for a long time).

The parameter *lotsfree* is usually one-quarter of the memory in the map that the clock hand sweeps, and *desfree* and *minfree* are usually the same across different systems, but are limited to fractions of available memory. FreeBSD dynamically adjusts its paging queues so these virtual memory parameters will rarely need to be adjusted. *Minfree* pages must be kept free in order to supply any pages that might be needed at interupt time.

Every process' text segment is by default shared and read-only. This scheme is practical with paging, because there is no external fragmentation, and the swap space gained by sharing more than offsets the negligible amount of overhead involved, as the kernel virtual space is large.

CPU scheduling, memory swapping, and paging interact: the lower the priority of a process, the more likely that its pages will be paged out and the more likely that it will be swapped in its entirety. The age preferences in choosing processes to swap guard against thrashing, but paging does so more effectively. Ideally, processes will not be swapped out unless they are idle, because each process will need only a small working set of pages in main memory at any one time, and the *pagedaemon* will reclaim unused pages for use by other processes.

The amount of memory the process will need is some fraction of that process' total virtual size, up to one-half if that process has been swapped out for a long time.

A.7 ■ File System

The UNIX file system supports two main objects: files and directories. Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

A.7.1 Blocks and Fragments

Most of the file system is taken up by *data blocks,* which contain whatever the users have put in their files. Let us consider how these data blocks are stored on the disk.

The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file systems usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1BSD file system was limited to a 1024-byte (1K) block.

The 4.2BSD solution is to use *two* block sizes for files which have no indirect blocks: all the blocks of a file are of a large *block* size (such as 8K), except the last. The last block is an appropriate multiple of a smaller *fragment* size (for example, 1024) to fill out the file. Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

The *block* and *fragment* sizes are set during file-system creation according to the intended use of the file system: If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block-to-fragment ratio of 8:1, and a minimum block size of 4K, so typical choices are 4096:512 for the former case and 8192:1024 for the latter.

Suppose data are written to a file in transfer sizes of 1K bytes, and the block and fragment sizes of the file system are 4K and 512 bytes. The file system will allocate a 1K fragment to contain the data from the first transfer. The next transfer will cause a new 2K fragment to be allocated. The data from the original

fragment must be copied into this new fragment, followed by the second 1K transfer. The allocation routines do attempt to find the required space on the disk immediately following the existing fragment so that no copying is necessary, but, if they cannot do so, up to seven copies may be required before the fragment becomes a block. Provisions have been made for programs to discover the block size for a file so that transfers of that size can be made, to avoid fragment recopying.

A.7.2 Inodes

A file is represented by an *inode* (Figure 11.7). An inode is a record that stores most of the information about a specific file on the disk. The name *inode* (pronounced *EYE node*) is derived from "index node" and was originally spelled "i-node"; the hyphen fell out of use over the years. The term is also sometimes spelled "I node."

The inode contains the user and group identifiers of the file, the times of the last file modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character device, block device, or socket). In addition, the inode contains 15 pointers to the disk blocks containing the data contents of the file. The first 12 of these pointers point to *direct blocks*; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (no more than 12 blocks) can be referenced immediately, because a copy of the inode is kept in main memory while a file is open. If the block size is 4K, then up to 48K of data may be accessed directly from the inode.

The next three pointers in the inode point to *indirect blocks*. If the file is large enough to use indirect blocks, the indirect blocks are each of the major block size; the fragment size applies to only data blocks. The first indirect block pointer is the address of a *single indirect block*. The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data. Then, there is a *double-indirect-block pointer,* the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a *triple indirect block;* however, there is no need for it. The minimum block size for a file system in 4.2BSD is 4K, so files with as many as 2^{32} bytes will use only double, not triple, indirection. That is, as each block pointer takes 4 bytes, we have 49,152 bytes accessible in direct blocks, 4,194,304 bytes accessible by a single indirection, and 4,294,967,296 bytes reachable through double indirection, for a total of 4,299,210,752 bytes, which is larger than 2^{32} bytes. The number 2^{32} is significant because the file offset in the file structure in main memory is kept in a 32-bit word. Files therefore cannot be larger than 2^{32} bytes. Since file pointers are signed integers (for seeking backward and forward in a file), the actual maximum file size is 2^{32-1} bytes. Two gigabytes is large enough for most purposes.

A.7.3 Directories

There is no distinction between plain files and directories at this level of implementation; directory contents are kept in data blocks, and directories are represented by an inode in the same way as plain files. Only the inode type field distinguishes between plain files and directories. Plain files are not assumed to have a structure, however, whereas directories have a specific structure. In Version 7, file names were limited to 14 characters, so directories were a list of 16-byte entries: 2 bytes for an inode number and 14 bytes for a file name.

In FreeBSD , file names are of variable length, up to 255 bytes, so directory entries are also of variable length. Each entry contains first the length of the entry, then the file name and the inode number. This variable-length entry makes the directory management and search routines more complex, but greatly improves the ability of users to choose meaningful names for their files and directories, with no practical limit on the length of the name.
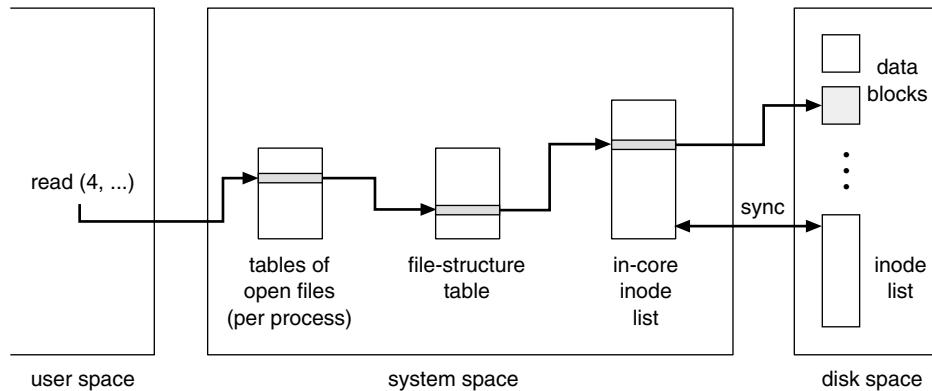
The first two names in every directory are "." and "..". New directory entries are added to the directory in the first space available, generally after the existing files. A linear search is used.

The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file. Thus, the kernel has to map the supplied user path name to an inode. The directories are used for this mapping.

First, a starting directory is determined. If the first character of the path name is "/", the starting directory is the root directory. If the path name starts with any character other than a slash, the starting directory is the current directory of the current process. The starting directory is checked for proper file type and access permissions, and an error is returned if necessary. The inode of the starting directory is always available.

The next element of the path name, up to the next "/", or to the end of the path name, is a file name. The starting directory is searched for this name, and an error is returned if the name is not found. If there is yet another element in the path name, the current inode must refer to a directory, and an error is returned if it does not, or if access is denied. This directory is searched as was the previous one. This process continues until the end of the path name is reached and the desired inode is returned. This step-by-step process is needed because at any directory a mount point (or symbolic link, see below) may be encountered, causing the translation to move to a different directory structure for continuation.

Hard links are simply directory entries like any other. We handle symbolic links for the most part by starting the search over with the path name taken from the contents of the symbolic link. We prevent infinite loops by counting the number of symbolic links encountered during a path-name search and returning an error when a limit (eight) is exceeded.

Figure A.7   File-system control blocks.

Nondisk files (such as devices) do not have data blocks allocated on the disk. The kernel notices these file types (as indicated in the inode) and calls appropriate drivers to handle I/O for them.

Once the inode is found by, for instance, the **open** system call, a *file structure* is allocated to point to the inode. The file descriptor given to the user refers to this file structure. FreeBSD has a *directory name cache* to hold recent directory-to-inode translations which greatly increases file system performance.

A.7.4 Mapping of a File Descriptor to an Inode

System calls that refer to open files indicate the file by passing a file descriptor as an argument. The file descriptor is used by the kernel to index a table of open files for the current process. Each entry of the table contains a pointer to a file structure. This file structure in turn points to the inode; see Figure A.7. The open file table has a fixed length which is only settable at boot time. Therefore, there is a fixed limit on the number of concurrently open files in a system.

The **read** and **write** system calls do not take a position in the file as an argument. Rather, the kernel keeps a *file offset,* which is updated by an appropriate amount after each **read** or **write** according to the number of data actually transferred. The offset can be set directly by the **lseek** system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Because more than one process may open the same file, and each such process needs its own offset for the file, keeping the offset in the inode is inappropriate. Thus, the file structure is used to contain the offset.

File structures are inherited by the child process after a **fork,** so several processes may share the *same* offset location for a file.

The *inode structure* pointed to by the file structure is an in-core copy of the inode on the disk. The in-core inode has a few extra fields, such as a reference

count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it. When a count becomes zero, the entry is no longer needed and may be reclaimed and reused.

A.7.5 Disk Structures
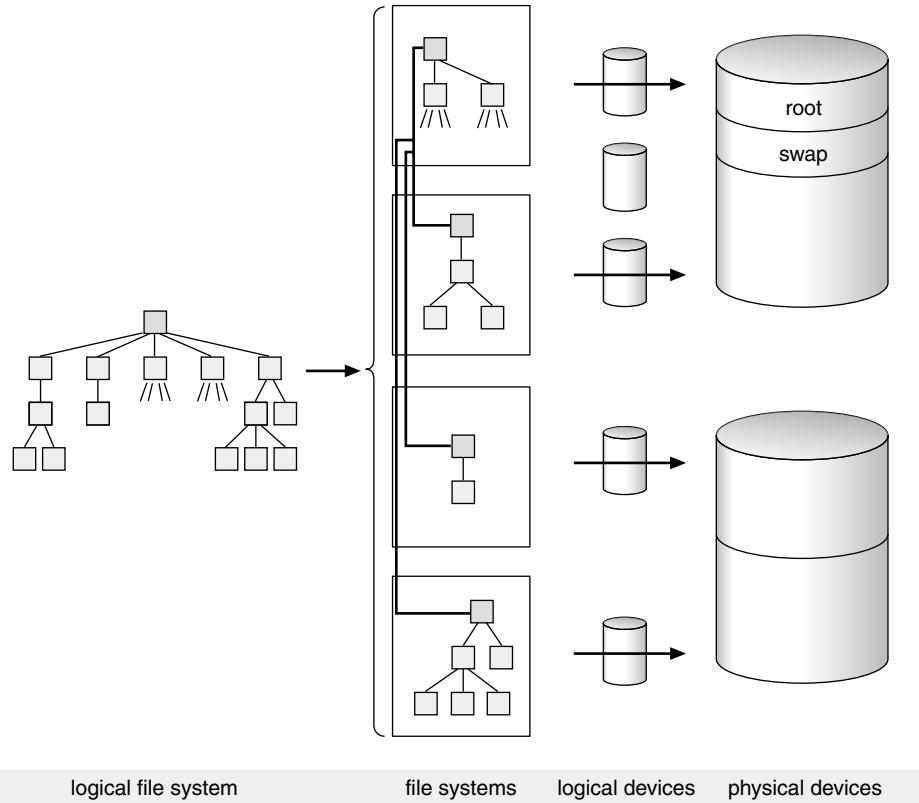
The file system that the user sees is supported by data on a mass storage device —usually, a disk. The user ordinarily knows of only one file system, but this one logical file system may actually consist of several *physical* file systems, each on a different device. Because device characteristics differ, each separate hardware device defines its own physical file system. In fact, it is generally desirable to partition large physical devices, such as disks, into multiple *logical* devices. Each logical device defines a physical file system. Figure A.8 illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices. The sizes and locations of these partitions were coded into device drivers in earlier systems, but are maintained on the disk by FreeBSD.

Partitioning a physical device into multiple file systems has several benefits. Different file systems can support different uses. Although most partitions would be used by the file system, at least one will be necessary for a swap area for the virtual-memory software. Reliability is improved, because software damage is generally limited to only one file system. We can improve efficiency by varying the file-system parameters (such as the block and fragment sizes) for each partition. Also, separate file systems prevent one program from using all available space for a large file, because files cannot be split across file systems. Finally, disk backups are done per partition, and it is faster to search a backup tape for a file if the partition is smaller. Restoring the full partition from tape is also faster.

The actual number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. One file system, the *root file system,* is always available. Other file systems may be *mounted*—that is, integrated into the directory hierarchy of the root file system.

A bit in the inode structure indicates that the inode has a file system mounted on it. A reference to this file causes the *mount table* to be searched to find the device number of the mounted device. The device number is used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a path-name element is ".." and the directory being searched is the root directory of a file system that is mounted, the mount table is searched to find the inode it is mounted on, and that inode is used.

Each file system is a separate system resource and represents a set of files. The first sector on the logical device is the *boot block*, possibly containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5K. A system needs only one partition containing boot-block data, but duplicates may be installed via privileged programs by

Figure A.8   Mapping of a logical file system to physical devices.

the systems manager, to allow booting when the primary copy is damaged. The *superblock* contains static parameters of the file system. These parameters include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies.

A.7.6 Implementations

The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user. The file system was changed between Version 6 and Version 7, and again between Version 7 and 4BSD. For Version 7, the size of inodes doubled, the maximum file and file-system sizes increased, and the details of free-list handling and superblock information changed. At that time also, *seek* (with a 16-bit offset) became *lseek* (with a 32-bit offset), to allow specification of offsets in larger files, but few other changes were visible outside the kernel.

In 4.0BSD, the size of blocks used in the file system was increased from 512 bytes to 1024 bytes. Although this increased size produced increased internal fragmentation on the disk, it doubled throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

4.2BSD added the Berkeley Fast File System, which increased speed, and was accompanied by new features. Symbolic links required new system calls. Long file names necessitated the new directory system calls to traverse the now-complex internal directory structure. Finally, the **truncate** calls were added. The Fast File System was a success, and is now found in most implementations of UNIX. Its performance is made possible by its layout and allocation policies, which we discuss next. In Section 12.4.4, we discussed changes made in SunOS to further increase disk throughput.
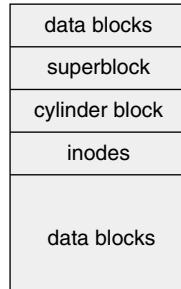
A.7.7 Layout and Allocation Policies

The kernel uses a *<logical device number, inode number>* pair to identify a file. The logical device number defines the file system involved. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of the logical device, with the data blocks following the inodes. The *inode number* is effectively just an index into this array.

With the Version 7 file system, a block of a file can be anywhere on the disk between the end of the inode array and the end of the file system. Free blocks are kept in a linked list in the superblock. Blocks are pushed onto the front of the free list, and are removed from the front as needed to serve new files or to extend existing files. Thus, the blocks of a file may be arbitrarily far from both the inode and one another. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. We can reverse this process only by reinitializing and restoring the entire file system, which is not a convenient task to perform. This process was described in Section 12.7.2.

Another difficulty is that the reliability of the file system is suspect. For speed, the superblock of each mounted file system is kept in memory. Keeping the superblock in memory allows the kernel to access a superblock quickly, especially for using the free list. Every 30 seconds, the superblock is written to the disk, to keep the in-core and disk copies synchronized (by the update program, using the **sync** system call). However, it is not uncommon for system bugs or hardware failures to cause a system crash, which destroys the in-core superblock between updates to the disk. Then, the free list on disk does not reflect accurately the state of the disk; to reconstruct it, we must perform a lengthy examination of all blocks in the file system. Note that this problem still remains in the new file system.

The 4.2BSD file-system implementation is radically different from that of Version 7. This reimplementation was done primarily to improve efficiency

| data blocks |
| superblock |
| cylinder block |
| inodes |
| data blocks |

Figure A.9   4.3BSD cylinder group.

and robustness, and most such changes are invisible outside the kernel. There were other changes introduced at the same time, such as symbolic links and long file names (up to 255 characters), that are visible at both the system-call and the user levels. Most of the changes required for these features were not in the kernel, however, but rather were in the programs that use them.

Space allocation is especially different. The major new concept in FreeBSD is the *cylinder group*. The cylinder group was introduced to allow localization of the blocks in a file. Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks (Figure A.9).

The superblock is identical in each cylinder group, so that it can be recovered from any one of them in the event of disk corruption. The *cylinder block* contains dynamic parameters of the particular cylinder group. These include a bit map of free data blocks and fragments, and a bit map of free inodes. Statistics on recent progress of the allocation strategies are also kept here.

The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the cylinder group. If it were, the header information for every cylinder group might be on the same disk platter; a single disk head crash could wipe out all of them. Therefore, each cylinder group has its header information at a different offset from the beginning of the group.

It is common for the directory-listing command *ls* to read all the inodes of every file in a directory, making it desirable for all such inodes to be close together on the disk. For this reason, the inode for a file is usually allocated from the same cylinder group as is the inode of the file's parent directory. Not everything can be localized, however, so an inode for a new directory is put in a *different* cylinder group from that of its parent directory. The cylinder group chosen for such a new directory inode is that with the greatest number of unused inodes.

To reduce disk head seeks involved in accessing the data blocks of a file, we allocate blocks from the same cylinder group as often as possible. Because

a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (such as 2 megabytes) has further block allocation redirected to a different cylinder group, the new group being chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected (at each megabyte) to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long head seeks involved in accessing a large file is kept small.

There are two levels of disk-block-allocation routines. The global policy routines select a desired disk block according to the considerations already discussed. The local policy routines use the specific information recorded in the cylinder blocks to choose a block near the one requested. If the requested block is not in use, it is returned. Otherwise, the block rotationally closest to the one requested in the same cylinder, or a block in a different cylinder but in the same cylinder group, is returned. If there are no more blocks in the cylinder group, a quadratic rehash is done among all the other cylinder groups to find a block; if that fails, an exhaustive search is done. If enough free space (typically 10 percent) is left in the file system, blocks usually are found where desired, the quadratic rehash and exhaustive search are not used, and performance of the file system does not degrade with use.

Because of the increased efficiency of the Fast File System, typical disks are now utilized at 30 percent of their raw transfer capacity. This percentage is a marked improvement over that realized with the Version 7 file system, which used about 3 percent of the bandwidth.

BSD Tahoe introduced the Fat Fast File System, which allows the number of inodes per cylinder group, the number of cylinders per cylinder group, and the number of distinguished rotational positions to be set when the file system is created. FreeBSD used to set these parameters according to the disk hardware type.

A.8 ■ I/O System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, the file system presents a simple consistent storage facility (the file) independent of the underlying disk hardware. In UNIX, the peculiarities of I/O devices are also hidden from the bulk of the kernel itself by the *I/O system*. The I/O system consists of a buffer caching system, general device driver code, and drivers for specific hardware devices. Only the device driver knows the peculiarities of a specific device. The major parts of the I/O system are diagrammed in Figure A.10.

There are three main kinds of I/O in FreeBSD : block devices, character devices, and the *socket* interface. The socket interface, together with its protocols and network interfaces, will be treated in Section 4.6.1.

Block devices include disks and tapes. Their distinguishing characteristic is that they are directly addressable in a fixed block size—usually, 512 bytes. A block-device driver is required to isolate details of tracks, cylinders, and so on, from the rest of the kernel. Block devices are accessible directly through appropriate device special files (such as */dev/rp0*), but are more commonly accessed indirectly through the file system. In either case, transfers are buffered through the *block buffer cache,* which has a profound effect on efficiency.

Character devices include terminals and line printers, but also almost everything else (except network interfaces) that does not use the block buffer cache. For instance, */dev/mem* is an interface to physical main memory, and */dev/null* is a bottomless sink for data and an endless source of end-of-file markers. Some devices, such as high-speed graphics interfaces, may have their own buffers or may always do I/O directly into the user's data space; because they do not use the block buffer cache, they are classed as character devices.

Terminals and terminal-like devices use *C-lists,* which are buffers smaller than those of the block buffer cache.

Block devices and *character* devices are the two main device classes. Device drivers are accessed by one of two arrays of entry points. One array is for block devices; the other is for character devices. A device is distinguished by a class (block or character) and a *device number*. The device number consists of two parts. The *major device number* is used to index the array for character or block devices to find entries into the appropriate device driver. The *minor device number* is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class, and by its use of common buffering systems. This segregation is important for portability, and also for system configuration.

system-call interface to the kernel					
socket	plain file	cooked block interface	raw block interface	raw tty interface	cooked TTY
protocols	file system				line discipline
network interface	block-device driver			character-device driver	
the hardware					

Figure A.10   4.3BSD kernel I/O structure.

A.8.1 Block Buffer Cache

The block devices use a block buffer cache. The buffer cache consists of a number of buffer headers, each of which can point to a piece of physical memory, as well as to a device number and a block number on the device. The buffer headers for blocks not currently in use are kept in several linked lists, one each for

- Buffers recently used, linked in LRU order (the LRU list)

- Buffers not recently used, or without valid contents (the AGE list)

- EMPTY buffers with no physical memory associated with them

The buffers in these lists are also hashed by device and block number for search efficiency.

When a block is wanted from a device (a read), the cache is searched. If the block is found, it is used, and no I/O transfer is necessary. If it is not found, a buffer is chosen from the AGE list, or the LRU list if AGE is empty. Then the device number and block number associated with it are updated, memory is found for it if necessary, and the new data are transferred into it from the device. If there are no empty buffers, the LRU buffer is written to its device (if it is modified) and the buffer is reused.

On a write, if the block in question is already in the buffer cache, the new data are put in the buffer (overwriting any previous data), the buffer header is marked to indicate the buffer has been modified, and no I/O is immediately necessary. The data will be written when the buffer is needed for other data. If the block is not found in the buffer cache, an empty buffer is chosen (as with a read) and a transfer is done to this buffer.

Writes are periodically forced for dirty buffer blocks to minimize potential file-system inconsistencies after a crash.

The number of data in a buffer in FreeBSD is variable, up to a maximum over all file systems, usually 8K. The minimum size is the paging-cluster size, usually 1024 bytes. Buffers are page-cluster aligned, and any page cluster may be mapped into only one buffer at a time, just as any disk block may be mapped into only one buffer at a time. The EMPTY list holds buffer headers which are used if a physical memory block of 8K is split to hold multiple, smaller blocks. Headers are needed for these blocks and are retrieved from EMPTY.

The number of data in a buffer may grow as a user process writes more data following those already in the buffer. When this increase in the data occurs, a new buffer large enough to hold all the data is allocated, and the original data are copied into it, followed by the new data. If a buffer shrinks, a buffer is taken off the empty queue, excess pages are put in it, and that buffer is released to be written to disk.

Some devices, such as magnetic tapes, require blocks to be written in a certain order, so facilities are provided to force a synchronous write of buffers

to these devices, in the correct order. Directory blocks are also written synchronously, to forestall crash inconsistencies. Consider the chaos that could occur if many changes were made to a directory, but the directory entries themselves were not updated.

The size of the buffer cache can have a profound effect on the performance of a system, because, if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low. FreeBSD optimizes the buffer cache by continually adjusting the amount of memory used by programs and the disk cache.

There are some interesting interactions among the buffer cache, the file system, and the disk drivers. When data are written to a disk file, they are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the buffer cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

A.8.2 Raw Device Interfaces

Almost every block device also has a character interface, and these are called *raw device interfaces*. Such an interface differs from the *block interface* in that the block buffer cache is bypassed.

Each disk driver maintains a queue of pending transfers. Each record in the queue specifies whether it is a read or a write, a main memory address for the transfer (usually in 512-byte increments), a device address for the transfer (usually the address of a disk sector), and a transfer size (in sectors). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process' virtual address space. This mapping is what a raw disk interface, for instance, does. Unbuffered transfers directly to or from a user's virtual address space are thus allowed. The size of the transfer is limited by the physical devices, some of which require an even number of bytes.

The kernel accomplishes transfers for swapping and paging simply by putting the appropriate request on the queue for the appropriate device. No special swapping or paging device driver is needed.

The 4.2BSD file-system implementation was actually written and largely tested as a user process that used a raw disk interface, before the code was moved into the kernel. In an interesting about-face, the Mach operating system has no file system per se. File systems can be implemented as user-level tasks.

A.8.3 C-Lists

Terminal drivers use a character buffering system, which keeps small blocks of characters (usually 28 bytes) in linked lists. There are routines to enqueue and dequeue characters for such lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for any given terminal line.

A **write** system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.

Input is similarly interrupt driven. Terminal drivers typically support *two* input queues, however, and conversion from the first (raw queue) to the other (canonical queue) is triggered by the interrupt routine putting an end-of-line character on the raw queue. The process doing a read on the device is then awakened, and its system phase does the conversion; the characters thus put on the canonical queue are then available to be returned to the user process by the read.

It is also possible to have the device driver bypass the canonical queue and return characters directly from the raw queue. This mode of operation is known as *raw mode*. Full-screen editors, and other programs that need to react to every keystroke, use this mode.

A.9 ■ Interprocess Communication

Many tasks can be accomplished in isolated processes, but many others require interprocess communication. Isolated computing systems have long served for many applications, but networking is increasingly important. With the increasing use of personal workstations, resource sharing is becoming more common. Interprocess communication has not traditionally been one of UNIX's strong points.

A.9.1 Sockets

The *pipe* (discussed in Section A.4.3) is the IPC mechanism most characteristic of UNIX. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the **pipe** system call. Its size is fixed, and when a process attempts to write to a full pipe, the process is suspended. Once all data previously written into the pipe have been read out, writing continues at the beginning of the file (pipes are not true circular buffers). One benefit of the small size (usually 4096 bytes) of pipes is that pipe data are seldom actually written to disk; they usually are kept in memory by the normal block buffer cache.

In FreeBSD , pipes are implemented as a special case of the *socket* mechanism. The socket mechanism provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities. Even on the same machine, a pipe can be used only by two processes related through use of the **fork** system call. The socket mechanism can be used by unrelated processes.

A socket is an endpoint of communication. A socket in use usually has an *address* bound to it. The nature of the address depends on the *communication domain* of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same *address format*. A single socket can communicate in only one domain.

The three domains currently implemented in FreeBSD are the UNIX domain (AF_UNIX), the Internet domain (AF_INET), and the XEROX Network Services (NS) domain (AF_NS). The address format of the UNIX domain is ordinary filesystem path names, such as */alpha/beta/gamma*. Processes communicating in the Internet domain use DARPA Internet communications protocols (such as TCP/IP) and Internet addresses, which consist of a 32-bit host number and a 32-bit port number (representing a rendezvous point on the host).

There are several *socket types,* which represent classes of services. Each type may or may not be implemented in any communication domain. If a type is implemented in a given domain, it may be implemented by one or more protocols, which may be selected by the user:

- **Stream sockets:** These sockets provide reliable, duplex, sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by the TCP protocol. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.

- **Sequenced packet sockets:** These sockets provide data streams like those of stream sockets, except that record boundaries are provided. This type is used in the XEROX AF_NS protocol.

- **Datagram sockets:** These sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by the UDP protocol.

- **Reliably delivered message sockets:** These sockets transfer messages that are guaranteed to arrive, and that otherwise are like the messages transferred using datagram sockets. This type is currently unsupported.

- **Raw sockets:** These sockets allow direct access by processes to the protocols that support the other socket types. The protocols accessible include not

only the uppermost ones, but also lower-level protocols. For example, in the Internet domain, it is possible to reach TCP, IP beneath that, or an Ethernet protocol beneath that. This capability is useful for developing new protocols.

The socket facility has a set of system calls specific to it. The **socket** system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned by the call is a small integer called a *socket descriptor,* which is in the same name space as file descriptors. The socket descriptor indexes the array of open "files" in the *u* structure in the kernel, and has a file structure allocated for it. The FreeBSD file structure may point to a *socket* structure instead of to an inode. In this case, certain socket information (such as the socket's type, message count, and the data in its input and output queues) is kept directly in the socket structure.

For another process to address a socket, the socket must have a name. A name is bound to a socket by the **bind** system call, which takes the socket descriptor, a pointer to the name, and the length of the name as a byte string. The contents and length of the byte string depend on the address format. The **connect** system call is used to initiate a connection. The arguments are syntactically the same as those for **bind;** the socket descriptor represents the local socket and the address is that of the foreign socket to which the attempt to connect is made.

Many processes that communicate using the socket IPC follow the *client–server model*. In this model, the *server* process provides a *service* to the *client* process. When the service is available, the server process listens on a well-known address, and the client process uses **connect,** as described previously, to reach the server.

A server process uses **socket** to create a socket and **bind** to bind the well-known address of its service to that socket. Then, it uses the **listen** system call to tell the kernel that it is ready to accept connections from clients, and to specify how many pending connections the kernel should queue until the server can service them. Finally, the server uses the **accept** system call to accept individual connections. Both **listen** and **accept** take as an argument the socket descriptor of the original socket. **Accept** returns a new socket descriptor corresponding to the new connection; the original socket descriptor is still open for further connections. The server usually uses **fork** to produce a new process after the **accept** to service the client while the original server process continues to listen for more connections.

There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an **accept**.

When a connection for a socket type such as a stream socket is established, the addresses of both endpoints are known and no further addressing informa-

tion is needed to transfer data. The ordinary **read** and **write** system calls may then be used to transfer data.

The simplest way to terminate a connection and to destroy the associated socket is to use the **close** system call on its socket descriptor. We may also wish to terminate only one direction of communication of a duplex connection; the **shutdown** system call can be used for this purpose.

Some socket types, such as datagram sockets, do not support connections; instead, their sockets exchange datagrams that must be addressed individually. The system calls **sendto** and **recvfrom** are used for such connections. Both take as arguments a socket descriptor, a buffer pointer and the length, and an address-buffer pointer and length. The address buffer contains the address to send to for **sendto** and is filled in with the address of the datagram just received by **recvfrom**. The number of data actually transferred is returned by both system calls.

The **select** system call can be used to multiplex data transfers on several file descriptors and/or socket descriptors. It can even be used to allow one server process to listen for client connections for many services and to **fork** a process for each connection as the connection is made. The server does a **socket, bind,** and **listen** for each service, and then does a **select** on all the socket descriptors. When **select** indicates activity on a descriptor, the server does an **accept** on it and forks a process on the new descriptor returned by **accept,** leaving the parent process to do a **select** again.

A.9.2 Network Support

Almost all current UNIX systems support the UUCP network facilities, which are mostly used over dial-up telephone lines to support the UUCP mail network and the USENET news network. These are, however, rudimentary networking facilities, as they do not support even remote login, much less remote procedure call or distributed file systems. These facilities are also almost completely implemented as user processes, and are not part of the operating system proper.

FreeBSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces. The framework in the kernel to support this is intended to facilitate the implementation of further protocols, and all protocols are accessible via the socket interface. The first version of the code was written by Rob Gurwitz of BBN as an add-on package for 4.1BSD.

The International Standards Organization's (ISO) Open System Interconnection (OSI) Reference Model for networking prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer, or with the protocol-protocol interface of a protocol in the layer immediately above or below in the same system. The ISO networking model is implemented in FreeBSD Reno and 4.4BSD.

The FreeBSD networking implementation, and to a certain extent the *socket* facility, is more oriented toward the ARPANET Reference Model (ARM). The ARPANET in its original form served as a proof of concept for many networking concepts, such as packet switching and protocol layering. The ARPANET was retired in 1988 because the hardware that supported it was no longer state of the art. Its successors, such as the NSFNET and the Internet, are even larger, and serve as a communications utility for researchers and as a testbed for Internet gateway research. The ARM predates the ISO model; the ISO model was in large part inspired by the ARPANET research.

Although the ISO model is often interpreted as requiring a limit of one protocol communicating per layer, the ARM allows several protocols in the same layer. There are only four protocol layers in the ARM, plus

- **Process/Applications:** This layer subsumes the application, presentation, and session layers of the ISO model. Such user-level programs as the File Transfer Protocol (FTP) and Telnet (remote login) exist at this level.

- **Host–Host:** This layer corresponds to ISO's transport and the top part of its network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO transport protocol, and IP performs the addressing functions of the ISO network layer.

- **Network Interface:** This layer spans the lower part of the ISO network layer and all of the data-link layer. The protocols involved here depend on the physical network type. The ARPANET uses the IMP-Host protocols, whereas an Ethernet uses Ethernet protocols.

- **Network Hardware:** The ARM is primarily concerned with software, so there is no explicit network hardware layer; however, any actual network will have hardware corresponding to the ISO physical layer.

The networking framework in FreeBSD is more generalized than is either the ISO model or the ARM, although it is most closely related to the ARM; see Figure A.11.

User processes communicate with network protocols (and thus with other processes on other machines) via the *socket* facility, which corresponds to the ISO Session layer, as it is responsible for setting up and controlling communications.

Sockets are supported by *protocols*—possibly by several, layered one on another. A protocol may provide services such as reliable delivery, suppression of duplicate transmissions, flow control, or addressing, depending on the socket type being supported and the services required by any higher protocols.

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. There is little restriction in the general framework on what protocols may communicate with what other

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation			
session		sockets	sock_stream
transport	host–host	protocol	TCP
network			IP
data link			
	network interface	network interfaces	Ethernet driver
hardware			
	network hardware	network hardware	interlan controller

Figure A.11   Network reference models and layering.

protocols, or on how many protocols may be layered on top of one another. The user process may, by means of the raw socket type, directly access any layer of protocol from the uppermost used to support one of the other socket types, such as streams, down to a raw network interface. This capability is used by routing processes and also for new protocol development.

There tends to be one *network-interface* driver per network controller type. The network interface is responsible for handling characteristics specific to the local network being addressed. This arrangement ensures that the protocols using the interface do not need to be concerned with these characteristics.

The functions of the network interface depend largely on the *network hardware,* which is whatever is necessary for the network to which it is connected. Some networks may support reliable transmission at this level, but most do not. Some networks provide broadcast addressing, but many do not.

The socket facility and the networking framework use a common set of memory buffers, or *mbuf*s. These are intermediate in size between the large buffers used by the block I/O system and the C-lists used by character devices. An *mbuf* is 128 bytes long, 112 bytes of which may be used for data; the rest is used for pointers to link the *mbuf* into queues and for indicators of how much of the data area is actually in use.

Data are ordinarily passed between layers (socket-protocol, protocol-protocol, or protocol-network interface) in *mbuf*s. This ability to pass the buffers containing the data eliminates some data copying, but there is still frequently a need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data that occupy an area the size of the memory-management page. Thus, it is possible for the data of an *mbuf* to reside not in the *mbuf* itself, but rather elsewhere in memory. There is an *mbuf* page table for this purpose, as well as a pool of pages dedicated to *mbuf* use.

A.10 ■ Summary

The early advantages of UNIX were that this system was written in a high-level language, was distributed in source form, and had provided powerful operating-system primitives on an inexpensive platform. These advantages led to UNIX's popularity at educational, research, and government institutions, and eventually in the commercial world. This popularity first produced many strains of UNIX with variant and improved facilities.

UNIX provides a file system with tree-structured directories. Files are supported by the kernel as unstructured sequences of bytes. Direct access and sequential access are supported through system calls and library routines.

Files are stored as an array of fixed-size data blocks with perhaps a trailing fragment. The data blocks are found by pointers in the inode. Directory entries point to inodes. Disk space is allocated from cylinder groups to minimize head movement and to improve performance.

UNIX is a multiprogrammed system. Processes can easily create new processes with the **fork** system call. Processes can communicate with pipes or, more generally, sockets. They may be grouped into jobs that may be controlled with signals.

Processes are represented by two structures: the process structure and the user structure. CPU scheduling is a priority algorithm with dynamically computed priorities that reduces to round-robin scheduling in the extreme case.

FreeBSD memory management is swapping supported by paging. A *pagedaemon* process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

Page and file I/O uses a block buffer cache to minimize the amount of actual I/O. Terminal devices use a separate character buffering system.

Networking support is one of the most important features in FreeBSD. The socket concept provides the programming mechanism to access other processes, even across a network. Sockets provide an interface to several sets of protocols.

■ Exercises

A.1 How were the design goals of UNIX different from those of other operating systems during the early stages of UNIX development?

A.2 Why are there many different versions of UNIX currently available? In what ways is this diversity an advantage to UNIX? In what ways is it a disadvantage?

A.3 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

A.4 In what circumstances is the system-call sequence **fork execve** most appropriate? When is **vfork** preferable?

A.5 Does FreeBSD give scheduling priority to I/O or CPU-bound processes? For what reason does it differentiate between these categories, and why is one given priority over the other? How does it know which of these categories fits a given process?

A.6 Early UNIX systems used swapping for memory management, whereas FreeBSD uses paging and swapping. Discuss the advantages and disadvantages of the two memory methods.

A.7 Describe the modifications to a file system that the FreeBSD makes when a process requests the creation of a new file */tmp/foo* and writes to that file sequentially until the file size reaches 20K.

A.8 Directory blocks in FreeBSD are written synchronously when they are changed. Consider what would happen if they were written asynchronously. Describe the state of the file system if a crash occurred after all the files in a directory were deleted but before the directory entry was updated on disk.

A.9 Describe the process that is needed to recreate the free list after a crash in 4.1BSD.

A.10 What effects on system performance would the following changes to FreeBSD have? Explain your answers.

    a. Clustering disk I/O into larger chunks

    b. Implementing and using shared memory to pass data between processes, rather than using RPC or sockets

    c. Using the ISO seven-layer networking model, rather than the ARM network model

A.11 What socket type should be used to implement an intercomputer file-transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.

Bibliographical Notes

The best general description of the distinctive features of UNIX is still that presented by Ritchie and Thompson [1974]. Much of the history of UNIX was given in Ritchie [1979] . A critique of UNIX was offered by Blair et al. [1985]. The two main modern versions of UNIX are 4.3BSD and System V. System V internals were described at length in Bach [1987]. The authoritative treatment of the design and implementation of 4.3BSD is that by Leffler et al. [1989].

Possibly the best book on general programming under UNIX, especially on the use of the shell and facilities such as *yacc* and *sed*, is that by Kernighan and Pike [1984]. Systems programming was covered by Stevens [1992]. Another text of interest is Bourne [1983]. The programming language of choice under UNIX is C Kernighan and Ritchie [1988]. C is also the system's implementation language. The Bourne shell was described in Bourne [1978]. The Korn shell was described in Korn [1983].

The set of documentation that comes with UNIX systems is called the *UNIX Programmer's Manual* (UPM) and is traditionally organized in two volumes. Volume 1 contains short entries for every command, system call, and subroutine package in the system, and is also available on-line via the *man* command. Volume 2, *Supplementary Documents* (usually divided into Volumes 2A and 2B for convenience of binding), contains assorted papers relevant to the system and manuals for those commands or packages too complex to describe in one or two pages. Berkeley systems add Volume 2C to contain documents concerning Berkeley-specific features.

The Version 7 file system was described in Thompson [1978], and the 4.2BSD file system was described in McKusick et al. [1984]. A crash-resistant UNIX file system was described by Anyanwu and Marshall [1986]. The basic reference for process-management is Thompson [1978]. The 3BSD memory-management system was described in Babaoglu and Joy [1969], and some 4.3BSD memory-management developments were described in McKusick and Karels [1988]. The I/O system was described in Thompson [1978].

A description of the UNIX operating-system security was given by Grampp and Morris [1984] and by Wood and Kochan [1985].

Two useful papers on communications under 4.2BSD are those by [Leffler et al. 1978, 1983] , both in UPM Volume 2C.

The *ISO Reference Model* was given in [ISO 1981] . The *ARPANET Reference Model* was set forth in Cerf and Cain [1983]. The Internet and its protocols were described in Comer [2000], Comer and Stevens [1991] and Comer and Stevens [1993]. UNIX network programming was described thoroughly in Stevens [1997] and Stevens [1998]. The general state of networks was given in Quarterman [1990].

There are many useful papers in the two special issues of *The Bell System Technical Journal* on UNIX [BSTJ 1978, BSTJ 1984]. Other papers of interest have appeared at various USENIX conferences and are available in the proceedings of those conferences, as well as in the USENIX-refereed journal, *Computer Systems*.

Several textbooks describing variants of the UNIX system are those by Holt [1983], discussing the Tunis operating system; [Comer 1984, 1987] , discussing the Xinu operating system; and Tanenbaum and Woodhull [1997], describing the Minix operating system.

FreeBSD is described in The freebsd Handbook FreeBSD [1999] and may be downloaded from http://www.freebsd.org/.

Appendix B

THE MACH SYSTEM

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Its multiprocessing support is also exceedingly flexible, ranging from shared memory systems to systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one to thousands of processors. In addition, Mach is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach is better able to satisfy the needs of the masses than the others are because it offers full compatibility with UNIX 4.3BSD. As such, it provides a unique opportunity for us to compare two functionally similar, but internally dissimilar, operating systems. The order and contents of the presentation of Mach is different from that of UNIX to reflect the differing emphasis of the two systems. There is no section on the user interface, because that component is similar in 4.3BSD when running the BSD server. As we shall see, Mach provides the ability to layer emulation of other operating systems as well, and they can even run concurrently.
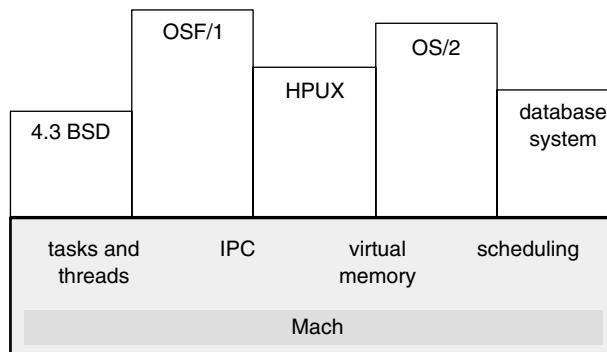
B.1 ■ History

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number of novel oper-

ating system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture that made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system, task and thread management) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components being replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provides compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach make the kernels in these releases larger than the corresponding BSD kernels. Mach 3 (Figure B.1) moves the BSD code outside of the kernel, leaving a much smaller *microkernel*. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows replacement of BSD with another operating system, or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the *virtual-machine* concept, but the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. As of Release 3.0, Mach

Figure B.1   Mach 3 structure.

became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines, and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled into the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The initial release of OSF/1 occurred a year later, and now competes with UNIX System V, Release 4, the operating system of choice among *UNIX International (UI)* members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the *NeXT* workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a future operating-system release, and research on Mach continues at CMU and OSF, and elsewhere.

B.2 ▪ Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD compatible and, in addition, excels in the following areas.

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and No Remote Memory Access (NORMA)

- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors

- Simplified kernel structure, with a small number of abstractions; in turn these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach

- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally

- Integrated memory management and interprocess communication, to provide both efficient communication of large numbers of data, as well as communication-based memory management

- Heterogeneous system support, to make Mach widely available and interoperable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities

- Easy portability to a wide class of uniprocessors

- An extensive library of utilities and applications

- The ability to combine utilities easily via pipes

Of course, BSD was seen as having several drawbacks that need to be redressed:

- A kernel that has become the repository of many redundant features—and that consequently is difficult to manage and modify

- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries; for instance, because the kernel was designed for uniprocessors, it has no provisions for locking code or data that other processors might be using

- Too many fundamental abstractions, providing too many similar, competing means to accomplish the same task

It should be clear that the development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing uni- and multiprocessor architectures, and can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current instantiation, the Mach system is usually as efficient as are other major versions of UNIX when performing similar tasks.

B.3   ■   System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel, but to make what is there powerful enough that all other features can be implemented at user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. By limiting all data operations to one mechanism, Mach is able to provide systemwide protection to its users by protecting the communications

mechanism. Optimizing this one communications path can result in significant performance gains, and is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an *object-oriented* system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The *port* mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system, and are as follows:
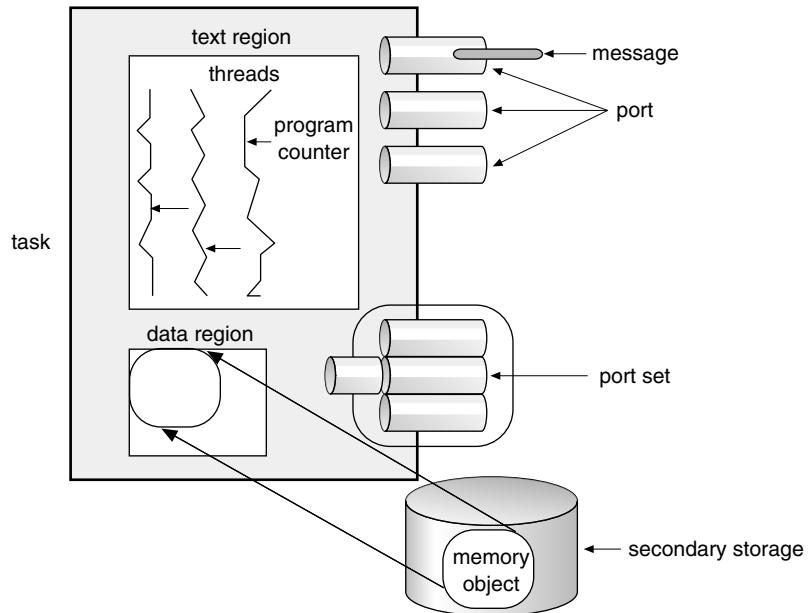
- A **task** is an execution environment that provides the basic unit of resource allocation. A task consists of a virtual address space and protected access to system resources via ports. A task may contain one or more threads.

- A **thread** is the basic unit of execution, and must run in the context of a task (which provides the address space). All threads within a task share the tasks' resources (ports, memory, and so on). There is no notion of a "process" in Mach. Rather, a traditional process would be implemented as a task with a single thread of control.

- A **port** is the basic object reference mechanism in Mach, and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or *port rights*; a task must have a port right to send a message to a port. The programmer invokes an operation on an object by *sending* a message to a port associated with that object. The object being represented by a port *receive*s the messages.

- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set, and thus service multiple ports. Each received message identifies the individual port (within the set) that it was received from; the receiver can use this to identify the object referred to by the message.

- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects; for each object, it may contain

the actual data or a pointer to out-of-line data. Port rights are passed in messages; passing port rights in messages is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)

- A **memory object** is a source of memory; tasks may access it by mapping portions (or the entire object) into their address spaces. The object may be managed by a user-mode *external memory manager*. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is one example.

Figure B.2 illustrates these abstractions, which we shall elaborate in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed

Figure B.2   Mach's basic abstractions.

and parallel programming, but also helps in the implementation of the kernel itself.

Mach connects memory management and communication (IPC) by allowing each to be used in the implementation of the other. Memory management is based on the use of *memory objects*. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, *pagein, pageout*) on the object. Because IPC is used, memory objects may reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing. Where possible, Mach passes messages by moving pointers to shared memory objects, rather than by copying the object itself.

IPC tends to involve considerable system overhead and is generally less efficient than is communication accomplished through shared memory, for intrasystem messages. Because Mach is a message-based kernel, it is important that message handling be carried out efficiently. Most of the inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (if the message is intracomputer) or low network transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual-memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. *Virtual copy*, or *copy-on-write,* techniques are used to avoid or delay the actual copying of the data. There are several advantages to this approach:

- Increased flexibility in memory management to user programs

- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers

- Improved performance over UNIX message passing

- Easier task migration; because ports are location independent, a task and all its ports can be moved from one machine to another; all tasks that previously communicated with the moved task can continue to do so because they reference a task by only its ports and communicate via messages to these ports

We shall detail the operation of process management, IPC, and memory management. Then, we shall discuss Mach's chameleonlike ability to support multiple operating-system interfaces.

B.4 ■ Process Management

A *task* can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more *thread*s executing in it.

B.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a **fork** system call produces a new UNIX process, Mach creates a new task to emulate this behavior. The new task's memory is a duplicate of the parent's address space, as dictated by the *inheritance attributes* of the parent's memory. The new task contains one thread, which is started at the same point as the creating **fork** call in the parent. Threads and tasks may also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. They also allow efficient use of parallel computing resources. Rather than having one process on each processor (with the corresponding performance penalty and operating-system overhead), a task may have its threads spread among parallel processors. Threads also add efficiency to user-level programs. For instance, in UNIX, an entire process must wait when a page fault occurs, or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (see Chapter 5), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Chapter 5.

At the user level, threads may be in one of two states.

- **Running:** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).

- **Suspended:** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This primitives provision is consistent with Mach's philosophy of

providing minimum yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A *suspend count* is kept for each thread. This count allows multiple suspend calls to be executed on a thread, and only when an equal number of resume calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a start call to be executed before a stop call (the suspend count would become negative), these routines cannot be used to synchronize shared data access. However, *wait* and *signal* operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. We discuss this method in Section B.5.

B.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the *C Threads* package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C Threads is one of the major influences of the POSIX P Threads standard, which many operating systems are being modified to support. As a result there are strong similarities between the C Threads and P Threads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.

- Destroy the calling thread, and return a value to the creating thread.

- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX **wait** system calls.

- Yield use of a processor, signaling that the scheduler may run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as were discussed in Chapter 7. The routines associated with mutual exclusion are these:

- The routine *mutex_alloc* dynamically creates a mutex variable.

- The routine *mutex_free* deallocates a dynamically created mutex variable.

- The routine *mutex_lock* locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is not guaranteed by the C Threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.

- The routine *mutex_unlock* unlocks a mutex variable, much like the typical *signal* operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of *condition variables*, which can be used to implement a *condition critical region* or a *monitor,* as was described in Chapter 7. A condition variable is associated with a mutex variable, and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine *condition_alloc* dynamically allocates a condition variable.

- The routine *condition_free* deletes a dynamically created condition variable allocated as result of *condition_alloc*.

- The routine *condition_wait* unlocks the associated mutex variable, and blocks the thread until a *condition_signal* is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A *condition_signal* does not guarantee that the condition still holds when the unblocked thread finally returns from its *condition_wait* call, so the awakened thread must loop, executing the *condition_wait* routine until it is unblocked and the condition holds.

   As an example of the C Threads routines, consider the bounded-buffer synchronization problem of Section 7.5.1. The producer and consumer are represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full, and to block the consumer thread if the buffer is empty. Although this program normally would be written in the C language on a Mach system, we shall use the familiar Pascal-like syntax of previous chapters for clarity. As in Chapter 7, we assume that the buffer consists of n slots, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The *empty* and *full* semaphores count the number of empty and full buffers, respectively. The semaphore *empty* is initialized to the value n; the semaphore *full* is initialized to the value 0. The condition variable *nonempty* is true while the buffer has items in it, and *nonfull* is true if the buffer has an empty slot.

   The first step includes the allocation of the mutex and condition variables:

   mutex_alloc(*mutex*); **condition_alloc**(*nonempty, nonfull*);

```
repeat
            ...
    produce an item into nextp
            ...
    mutex_lock(mutex);
    while(full)
        condition_wait(nonfull, mutex);
            ...
    add nextp to buffer
            ...
    condition_signal(nonempty);
    mutex_unlock(mutex);
until false;
```

Figure B.3   The structure of the producer process.

The code for the producer thread is shown in Figure B.3; the code for the consumer thread is shown in Figure B.4. When the program terminates, the mutex and condition variables need to be deallocated:

mutex_free(*mutex*); **condition_free**(*nonempty, nonfull*);

B.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than are its process-based relatives. There are generally more threads in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult, and is a relatively new area

```
repeat
    mutex_lock(mutex);
    while(empty)
        condition_wait(nonempty, mutex);
            ...
    remove an item from the buffer to nextc
            ...
    condition_signal(nonfull);
    mutex_unlock(mutex);
            ...
    consume the item in nextc
            ...
until false;
```

Figure B.4   The structure of the consumer process.

of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of there being a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues (like most other objects in Mach) are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because there may be fewer runable threads than there are available processors, for instance. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, there are still complications to be considered. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a call must be issued by a thread to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. There are many other internal thread states that are used by the scheduler to control thread execution.

B.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid

redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the "victim") and that of the handler, and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD *signal* package, as described later in this section.

Disruptions to normal program execution come in two varieties: internally generated *exceptions* and external *interrupts*. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support. This facility is also useful for other reasons, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas debuggers use per-task handling. It makes little sense to try to debug only one thread, or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only other difference between the two types of exceptions lies in their inheritance from a parent task. Taskwide exception-handling facilities are passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited, and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task, and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a *raise* RPC message being sent to the handler.

- The victim then calls a routine to wait until the exception is handled.

- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.

- The handler performs its function according to the type of exception. The handler's action involves *clearing* the exception, causing the victim to resume, or *terminating* the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine

in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by zero), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however! A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel. Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving these exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code. In this manner, all hardware exceptions begin in a uniform way—as exceptions RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system—as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

B.5 ■ Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes, and between hosts with fixed, global names (internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are *ports* and *messages*. Almost everything in Mach is an object, and all objects are addressed via their communications ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task, as discussed later. Mach ensures security by requiring that message senders and receivers have *rights*. A right consists of a port name and a capability (send or receive) on that port, and is much like a capability in object-oriented systems. There can be only one task with receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object, and obtains the access rights to that port. Rights can be given out by the creator of the object (including the kernel), and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right and the sender loses it. A task may allocate ports to allow access to any objects it owns, or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

B.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the send, wait for a slot to become available in the queue, or have the kernel deliver the message for it.

There are several system calls to provide the port functionality:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.

- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed and all other tasks with send rights are, potentially, notified.

- Get the current status of a task's port.

- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation (or terminates).
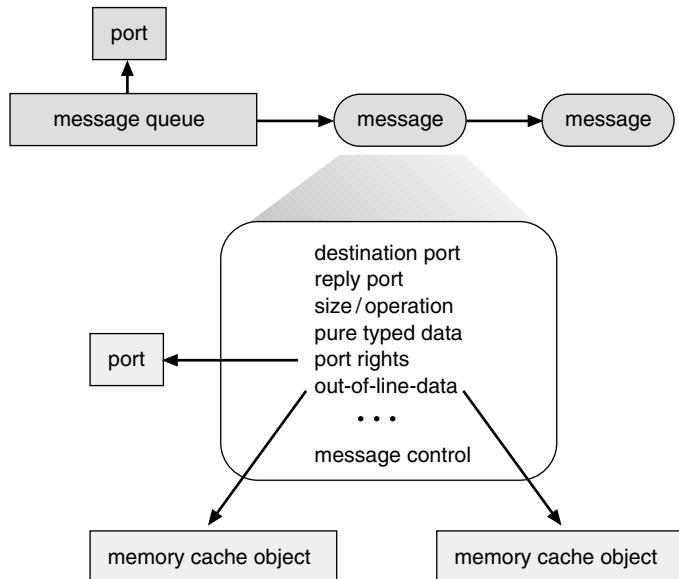
When a task is created, the kernel creates several ports for it. The function *task_self* returns the name of the port that represents the task in calls to the kernel. For instance, for a task to allocate a new port, it would call *port_allocate* with *task_self* as the name of the task that will own the port. Thread creation results in a similar *thread_self* thread kernel port. This scheme is similar to the

standard process-id concept found in UNIX. Another port created for a task is returned by *task_notify*, and is the name of the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into *port set*s. This facility is useful if one thread is to service requests coming in on multiple ports (for example, for multiple objects). A port may be a member of at most one port set at a time, and, if a port is in a set, it may not be used directly to receive messages. Instead, the message will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3BSD **select** system call, but they are more efficient.

B.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (see Figure B.5). The data in the message (*in-line* data) were limited to less than 8K in Mach 2.5 systems, but Mach 3.0 has no limit. Any data exceeding that limit must be sent in multiple messages, or more likely via reference by a pointer in a message (*out-of-line* data, as we shall describe shortly). Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver

Figure B.5   Mach messages.

can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address, or by forwarding it for processing to the NetMsgServer, as we shall explain.

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, as a pointer to data in the sender's address space would be invalid in the receiver's—especially if the sender and receiver reside on different systems! Generally, systems send messages by copying the data from the sender to the receiver. Because this technique can be inefficient, especially in the case of large messages, Mach optimizes this procedure. The data referenced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy, and makes message passing efficient. In essence, message passing is implemented via virtual-memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region of memory into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be *sparse*, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure B.6.

B.5.3 The NetMsgServer

For a message to be sent between computers, the destination of a message must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example, the port number for services based on TCP or UDP). One of Mach's tenets is that all objects within the system are location independent, and that the location is transparent to the user. This tenet requires Mach to provide location-
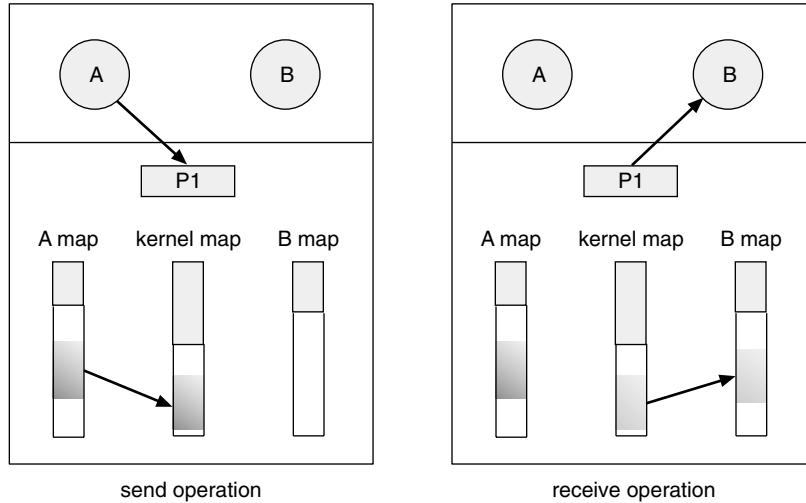
Figure B.6   Mach message transfer.

transparent naming and transport to extend IPC across multiple computers. This naming and transport are performed by the *Network Message Server* or NetMsgServer, a user-level capability-based networking daemon that forwards messages between hosts.   It also provides a primitive networkwide name service that allows tasks to register ports for lookup by tasks on any other computer in the network.   Mach ports can be transferred only in messages, and messages must be sent to ports; the primitive name service solves the problem of transferring the first port that allows tasks on different computers to exchange messages.   Subsequent IPC interactions are fully transparent; the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages, and arranges for the appropriate transfers.   The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel's computer.   Mach's kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer; the notion of a NetMsgServer is protocol-independent, and NetMsgServers have been built that use various protocols.   Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel's IPC to send the message to the correct destination task. The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer
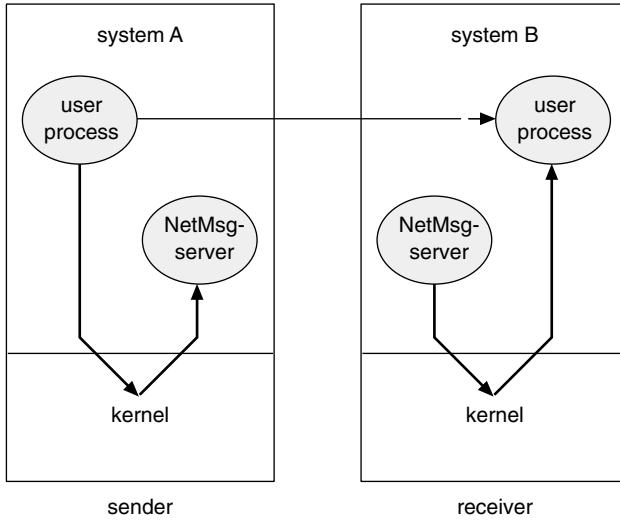
on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently to the original port; this procedure is one example of how the NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way to send between systems data that are formatted in a way that is understandable by both the sender and receiver. Unfortunately, computers vary the format in which they store types of data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B then presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a *msg_receive* call. This sequence of events is shown in Figure B.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem of Mach 3.0 implements functionality similar to the NetMsgServer directly in the Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of NORMA IPC does not exclude use of the NetMsgServer; the NetMsgServer can still be used to provide MACH IPC service over networks that link a NORMA multiprocessor to other computers. In addition to NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system, and the ability for a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor; the multiprocessor behaves like one large system, rather than like an assemblage of smaller systems (for both users and applications).
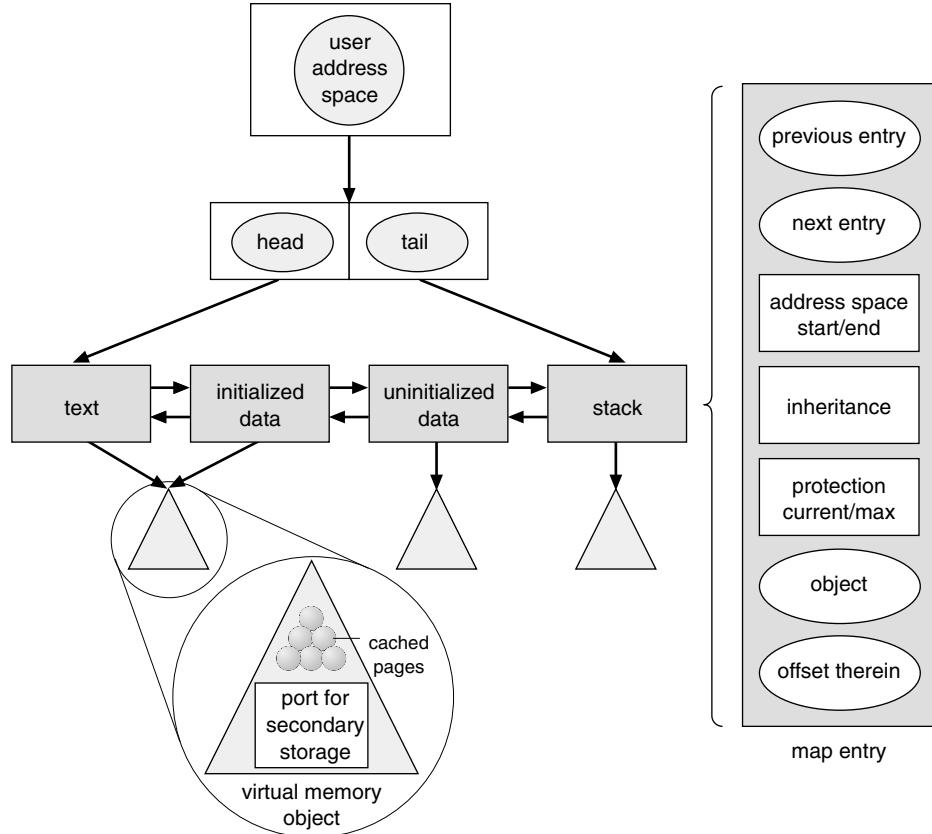
Figure B.7   Network IPC forwarding by NetMsgServer.

B.5.4 Synchronization Through IPC

The IPC mechanism is extremely flexible, and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable, and may have *n* messages sent to it for *n* resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available; otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation *wait*, and sending is equivalent to *signal*. This method can be used for synchronizing semaphore operations among threads in the same task, but cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon may be written that implements the same method.

B.6 ■ Memory Management

Given the object-oriented nature of Mach, it is not surprising that a principle abstraction in Mach is the *memory object*. Memory objects are used to manage secondary storage, and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure B.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual-memory pager found in

Figure B.8   Mach virtual memory task address map.

other operating systems. In contrast to the traditional approach of having the kernel provide management of secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated with it, and may be manipulated by messages being sent to its port. Memory objects—unlike the memory-management routines in monolithic, traditional kernels—allow easy experimentation with new memory-manipulation algorithms.

B.6.1 Basic Structure

The virtual address space of a task is generally *sparse*, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual-memory address is used to provide the threads with access to the message. As new items are mapped

or removed from the address space, holes of unallocated memory appear in the address space.

Mach makes no attempt to compress the address space, although a task may fail (crash) if it has no room for a requested region in its address space. Given that address spaces are 4 gigabytes or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4 gigabyte address space for each task, especially one with holes in it, would use excessive amounts of memory (1 megabyte or more). The key to sparse address spaces is that page-table space is used for only currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual-memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual-memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (discussed in Section B.6.2). Virtual-memory objects are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

B.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual-memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept of a memory object being created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained by only the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

There are several circumstances in which user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory

manager could fail to perform pageout. Mach itself also needs a memory manager to take care of its memory needs. For these cases, Mach provides a *default memory manager*. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard filesystem or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the *pageout daemon*. A paging algorithm based on FIFO with second chance (Section 10.4.5) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again), and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the *vm_map* system call. Included in this system call is a port which identifies the object, and the memory manager which is responsible for the region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines (tasks on a single machine share a single copy of a mapped memory object). Consider a situation in which tasks on two different machines attempt to modify the same page of an object concurrently. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page, and that it could merge the modifications successfully at some future time). Note that most

external memory managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first *vm_map* call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the *memory_manager_init* routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a *control port* and a *name port*. The control port is used by the memory manager to provide data to the kernel (for example, pages to be made resident). Name ports are used throughout Mach. They do not receive messages, but rather are used simply as a point of reference and comparison. Finally, the memory object must respond to a *memory_manager_init* call with a *memory_object_set_attributes* call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

There are several kernel calls that are needed to support external memory managers. The *vm_map* call has already been discussed in the paragraph above. There are also commands to get and set attributes and to provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. There are also several calls to allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed *memory_object_init* and others. When a thread causes a page fault on a memory object's page, the kernel sends a *memory_object_data_request* to the memory object's port on behalf of the faulting thread. The thread is placed in wait state until the memory manager either returns the page in a *memory_object_data_provided* call, or returns an appropriate error to the kernel. Any of the pages that have been modified, or any *precious pages* that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via *memory_object_data_write*. Precious pages are pages that may not have been modified, but that cannot be discarded as they otherwise would, because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

Again, there are several other calls for locking, protection information and modification, and the other details with which all virtual memory systems must deal.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select

the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel's replacement victim if it knows of a better candidate (for instance, MRU page replacement).

- It may monitor the memory object it is backing, and request pages to be paged out before the memory usage invokes Mach's pageout daemon.

- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we shall show in Section B.6.3.

- It can control the order of operations on secondary storage, to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.

- It can control mapped file access.

B.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task's memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX **fork** system call.

It is obviously difficult for tasks on multiple machines to share memory, and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be *inherited* by its children, and which are to be readable–writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task's address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the same location concurrently). This coordination may be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share

a section of memory, the tasks may use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels; the XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory, and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels, and sophisticated copy-on-write optimizations.

B.7 ■ Programmer Interface

There are several levels at which a programmer may work within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3BSD system-call interface. Version 2.5 includes most of 4.3BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality, but in theory could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the library are redirected to the appropriate server. Servers can be multithreaded for improved efficiency. BSD and OSF/1 are implemented as single multi-threaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task, and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control will decrease the efficiency of Mach, this decrease is somewhat ameliorated by the ability for multiple threads to be executing BSD-like code concurrently.

At the next higher programming level is the *C Threads* package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section B.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C Threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C Threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called *MIG*. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types and procedures), and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

B.8 ■ Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

The Mach operating system was designed with the following three critical goals in mind:

- Emulate 4.3BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.

- Have a modern operating system that supports many memory models, and parallel and distributed computing.

- Design a kernel that is simpler and easier to modify than is 4.3BSD.

As we have shown in this chapter, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3BSD code has been rewritten to provide the same 4.3 functionality, but to use the Mach primitives. This change allows the 4.3BSD support code to run in user space on a Mach 3.0 system.

Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communications method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual-memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual-memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions may be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual-machine systems.

■ Exercises

B.1 What three features of Mach make it appropriate for distributed processing?

B.2 Name two ways that port sets are useful in implementing parallel programs.

B.3 Consider an application that maintains a database of information, and provides facilities for other tasks to add, delete, and query the database. Give three configurations of ports, threads, and message types that could be used to implement this system. Which is the best? Explain your answer.

B.4 Give the outline of a task that would migrate subtasks (tasks it creates) to other systems. Include information about how it would decide when to migrate tasks, which tasks to migrate, and how the migration would take place.

B.5 Name two types of applications for which you would use the MIG package.

B.6 Why would someone use the low-level system calls, instead of the C Threads package?

B.7 Why are external memory managers not able to replace the internal page-replacement algorithms? What information would need to be made available to the external managers for them to make page-replacement decisions? Why would providing this information violate the principle behind the external managers?

B.8 Why is it difficult to implement mutual exclusion and condition variables in an environment where like-CPUs do not share any memory? What approach and mechanism could be used to make such features available on a NORMA system?

B.9 What are the advantages to rewriting the 4.3BSD code as an external, user-level library, rather than leaving it as part of the Mach kernel? Are there any disadvantages? Explain your answer.

Bibliographical Notes

The Accent operating system was described by Rashid and Robertson [1981]. An historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by Rashid [1986]. General discussions concerning the Mach model are offered by Tevanian and Smith [1989].

Accetta et al. [1986] presented an overview of the original design of Mach. The Mach scheduler was described in detail by Tevanian et al. [1987a] and Black [1990]. An early version of the Mach shared memory and memory-mapping system was presented by Tevanian et al. [1987b].

The most current description of the C Threads package appears in Cooper and Draves [1987]; MIG was described by Draves et al. [1989]. An overview of these packages' functionality and a general introduction to programming in Mach was presented by Walmer and Thompson [1989] and Boykin et al. [1993].

Black et al. [1988] discussed the Mach exception-handling facility. A multithreaded debugger based on this mechanism was described in Caswell and Black [1989].

A series of talks about Mach sponsored by the OSF UNIX consortium is available on videotape from OSF. Topics include an overview, threads, networking, memory management, many internal details, and some example implementations of Mach. The slides from these talks were given in [OSF 1989].

On systems where USENET News is available (most educational institutions in the United States, and some overseas), the news group *comp.os.mach* is used to exchange information on the Mach project and its components.

An overview of the microkernel structure of Mach 3.0, complete with performance analysis of Mach 2.5 and 3.0 compared to other systems, was given

in Black et al. [1992]. Details of the kernel internals and interfaces of Mach 3.0 were provided in Loepere [1992]. Tanenbaum [1992] presented a comparison of Mach and Amoeba. Discussions concerning parallelization in Mach and 4.3BSD are offered by Boykin and Langerman [1990].

Ongoing research was presented at USENIX Mach and Micro-kernel Symposia [USENIX 1990, USENIX 1991, and USENIX 1992b]. Active research areas include virtual memory, real time, and security [McNamee and Armstrong 1990].

Credits

Appendix C

THE NACHOS SYSTEM

By Thomas E. Anderson

University of California, Berkeley

I hear and I forget, I see and I remember,

I do and I understand.

> –Chinese proverb

A good way to gain a deeper understanding of modern operating-system concepts is to get your hands dirty—to take apart the code for an operating system to see how it works at a low level, to build significant pieces of the operating system yourself, and to observe the effects of your work. An operating-system course project provides this opportunity to see how you can use basic concepts to solve real-world problems. Course projects can also be valuable in many other areas of computer science, from compilers and databases to graphics and robotics. But a project is particularly important for operating systems, where many of the concepts are best learned by example and experimentation.

That is why we created *Nachos,* an instructional operating system intended for use as the course project for an undergraduate or first-year graduate course in operating systems. Nachos includes code for a simple but complete working operating system, a machine simulator that allows it to be used in a normal UNIX workstation environment, and a set of sample assignments. Nachos

lets anyone explore all the major components of a modern operating system described in this book, from threads and process synchronization, to file systems, to multiprogramming, to virtual memory, to networking. The assignments ask you to design and implement a significant piece of functionality in each of these areas.

Nachos is distributed without charge. It currently runs on both Digital Equipment Corporation MIPS UNIX workstations and Sun SPARC workstations; ports to other machines are in progress. See Section C.4 to learn how to obtain a copy of Nachos.

Here, we give an overview of the Nachos operating system and the machine simulator, and describe our experiences with the example assignments. Of necessity, Nachos is evolving continually, because the field of operating systems is evolving continually. Thus, we can give only a snapshot of Nachos; in Section C.4 we explain how to obtain more up to date information.

C.1 ■ Overview

Many of the earliest operating-system course projects were designed in response to the development of UNIX in the mid-1970s. Earlier operating systems, such as MULTICS and OS/360, were far too complicated for an undergraduate to understand, much less to modify, in one semester.

Even UNIX itself is too complicated for that purpose, but UNIX showed that the core of an operating system can be written in only a few dozen pages, with a few simple but powerful interfaces. However, recent advances in operating systems, hardware architecture, and software engineering have caused many operating-systems projects developed over the past two decades to become out-of-date. Networking and distributed applications are now commonplace. Threads are crucial for the construction of both operating systems and higher-level concurrent applications. And the cost–performance tradeoffs among memory, CPU speed, and secondary storage are now different from those imposed by core memory, discrete logic, magnetic drums, and card readers.

Nachos is intended to help people learn about these modern systems concepts. Nachos illustrates and takes advantage of modern operating-systems technology, such as threads and remote procedure calls; recent hardware advances, such as RISCs and the prevalence of memory hierarchies; and modern software-design techniques, such as protocol layering and object-oriented programming.

In designing Nachos, we faced constantly the tradeoff between simplicity and realism in choosing what code to provide as part of the baseline system, and what to leave for the assignments. We believe that a course project must achieve a careful balance among the time that students spend reading code, that they spend designing and implementing, and that they spend learning new concepts. At one extreme, we could have provided nothing but bare

hardware, leaving a *tabula rasa* for students to build an entire operating system from scratch. This approach is impractical, given the scope of topics to cover. At the other extreme, starting with code that is too realistic would make it easy to lose sight of key ideas in a forest of details.

Our approach was to build the simplest possible implementation for each subsystem of Nachos; this provides a working example—albeit an overly simplistic one—of the operation of each component of an operating system. The baseline Nachos operating-system kernel includes a thread manager, a file system, the ability to run user programs, and a simple network mailbox. As a result of our emphasis on simplicity, the baseline kernel comprises about 2500 lines of code, about one-half of which are devoted to interface descriptions and comments. (The hardware simulator takes up another 2500 lines, but you do not need to understand the details of its operation to do the assignments.) It is thus practical to read, understand, and modify Nachos during a single semester course. By contrast, building a project around a system like UNIX would add realism, but the UNIX 4.3BSD file system *by itself*, even excluding the device drivers, comprises roughly 5000 lines of code. Since a typical course will spend only about 2 to 3 weeks of the semester on file systems, size makes UNIX impractical as a basis for an undergraduate operating-system course project.

We have found that the baseline Nachos kernel can demystify a number of operating-system concepts that are difficult to understand in the abstract. Simply reading and walking through the execution of the baseline system can answer numerous questions about how an operating system works at a low level, such as:
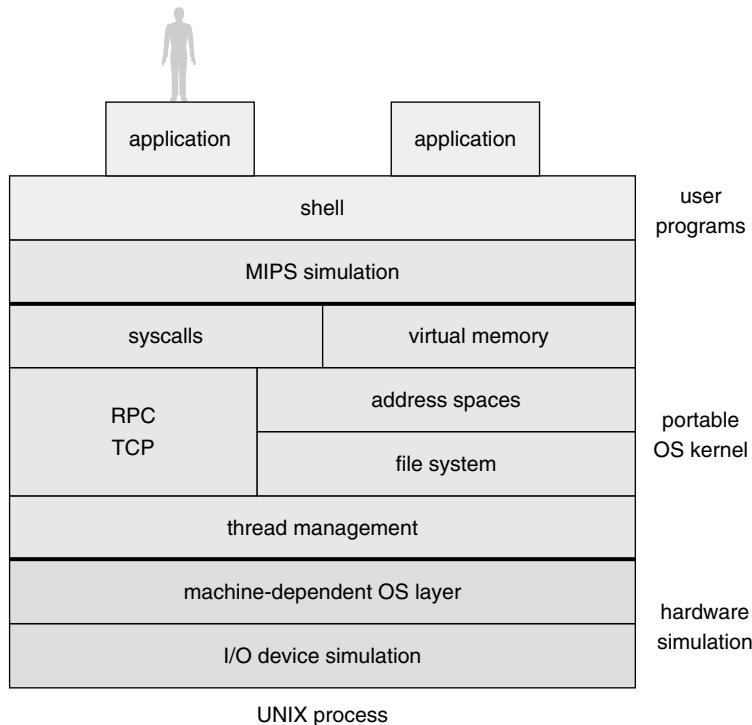
- How do all the pieces of an operating system fit together?

- How does the operating system start a thread? How does it start a process?

- What happens when one thread context switches to another thread?

- How do interrupts interact with the implementation of critical sections?

- What happens on a system call? What happens on a page fault?

- How does address translation work?

- Which data structures in a file system are on disk, and which are in memory?

- What data need to be written to disk when a user creates a file?

- How does the operating system interface with I/O devices?

- What does it mean to build one layer of a network protocol on another?

Of course, reading code by itself can be a boring and pointless exercise; we addressed this problem by keeping the code as simple as possible, and by

designing assignments that modify the system in fundamental ways. Because we start with working code, the assignments can focus on the more interesting aspects of operating-system design, where tradeoffs exist and there is no single right answer.

C.2 ■ Nachos Software Structure

Before we discuss the sample assignments in detail, we first outline the structure of the Nachos software. Figure C.1 illustrates how the major pieces in Nachos fit together. Like many earlier instructional operating systems, Nachos runs on a *simulation* of real hardware. Originally, when operating-system projects were first being developed in the 1970s and early 1980s, the reason for using a simulator was to make better use of scarce hardware resources. Without a simulator, each student would need her own physical machine to test new versions of the kernel. Now that personal computers are commonplace, is there still a reason to develop an operating system on a simulator, rather than on physical hardware?

Figure C.1   How the major pieces in Nachos fit together.

We believe that the answer is yes, because using a simulator makes debugging easier. On real hardware, operating-system behavior is nondeterministic; depending on the precise timing of interrupts, the operating system may take one path through its code or another. Synchronization can help to make operating-system behavior more predictable, but what if we have a bug in our synchronization code such that two threads can access the same data structure at the same time? The kernel may behave correctly most of the time, yet crash occasionally. Without being able to repeat the behavior that led to the crash, however, it would be difficult to find the root cause of the problem. Running on a simulator, rather than on real hardware, allows us to make system behavior repeatable. Of course, debugging nonrepeatable execution sequences is part of life for professional operating-system engineers, but it did not seem advisable for us to make this experience part of anyone's introduction to operating systems.

Running on simulated hardware has other advantages. During debugging, it is important to be able to make a change to the system quickly, to recompile, and to test the change to see whether it fixed the problem. Using a simulator reduces the time required for this edit–compile–debug cycle, because otherwise the entire system has to be rebooted to test a new version of the kernel. Moreover, normal debugging tools do not work on operating-system kernels, because, for example, if the kernel stops at a breakpoint, the debugger cannot use the kernel to print the prompt for the next debugging command. In practice, debugging an operating-system kernel on real hardware requires *two* machines: one to run the kernel under test, and the other to run the debugger. For these reasons, many commercial operating-system development projects now routinely use simulators to speed development.

One approach would be to simulate the entire workstation hardware, including fetching, decoding, and executing each kernel- or user-mode instruction in turn. Instead, we take a shortcut for performance. The Nachos kernel code executes in native mode as a normal (debuggable) UNIX process linked with the hardware simulator. The simulator surrounds the kernel code, making it appear as though it is running on real hardware. Whenever the kernel code accesses an I/O device—such as a clock chip, a disk, a network controller, or a console—the simulator is invoked to perform the I/O activity. For instance, the simulator implements disk I/O using UNIX file routines; it implements network packet transfer via UNIX sockets.

In addition, we simulate each instruction executed in user mode. Whenever the kernel gives up control to run application code, the simulator fetches each application instruction in turn, checks for page faults or other exceptions, and then simulates its execution. When an application page fault or hardware interrupt occurs, the simulator passes control back to the kernel for processing, as the hardware would in a real system.

Thus, in Nachos, user applications, the operating-system kernel, and the hardware simulator run together in a normal UNIX process. The UNIX process

thus represents a single workstation running Nachos. The Nachos kernel, however, is written as though it were running on real hardware. In fact, we could port the Nachos kernel to a physical machine simply by replacing the hardware simulation with real hardware and a few machine-dependent device-driver routines.

Nachos is different from earlier systems in several significant ways:

1. We can run normal compiled C programs on the Nachos kernel, because we simulate a standard, well-documented, instruction set (MIPS R2/3000 integer instructions) for user-mode programs. In the past, operating-system projects typically simulated their own ad hoc instruction set, requiring user programs to be written in a special-purpose assembly language. However, because the R2/3000 is a RISC, it is straightforward to simulate its instruction set. In all, the MIPS simulation code is only about 10 pages long.

2. We simulate accurately the behavior of a network of workstations, each running a copy of Nachos. We connect Nachos "machines," each running as a UNIX process, via UNIX sockets, simulating a local-area network. A thread on one "machine" can then send a packet to a thread running on a different "machine"; of course, both are simulated on the same physical hardware.

3. The simulation is deterministic, and kernel behavior is reproducible. Instead of using UNIX signals to simulate asynchronous devices such as the disk and the timer, Nachos maintains a simulated time that is incremented whenever a user program executes an instruction and whenever a call is made to certain low-level kernel routines. Interrupt handlers are then invoked when the simulated time reaches the appropriate point. At present, the precise timing of network packet delivery is not reproducible, although this limitation may be fixed in later versions of Nachos.

4. The simulation is randomizable to add unpredictable, but repeatable, behavior to the kernel thread scheduler. Our goal was to make it easy to test kernel behavior given different interleavings of the execution of concurrent threads. Simulated time is incremented whenever interrupts are enabled within the kernel (in other words, whenever any low-level synchronization routine, such as semaphore P or V, is called); after a random interval of simulated time, the scheduler will cause the current thread to be time sliced. As another example, the network simulation randomly chooses which packets to drop. Provided that the initial seed to the random number generator is the same, however, the behavior of the system is repeatable.

5. We hide the hardware simulation from the rest of Nachos via a machine-dependent interface layer. For example, we define an abstract disk that accepts requests to read and write disk sectors and provides an interrupt handler to be called on request completion. The details of the disk simulator are hidden behind this abstraction, in much the same way that

disk-device–specific details are isolated in a normal operating system. One advantage to using a machine-dependent interface layer is to make clear which portions of Nachos can be modified (the kernel and the applications) and which portions are off-limits (the hardware simulation—at least until you take a computer-architecture course).

C.3 ■ Sample Assignments

Nachos contains five major components, each the focus of one assignment given during the semester: thread management and synchronization, the file system, user-level multiprogramming support, the virtual-memory system, and networking. Each assignment is designed to build on previous ones; for instance, every part of Nachos uses thread primitives for managing concurrency. This design reflects part of the charm of developing operating systems: You get to use what you build. It is easy, however, to change the assignments or to do them in a different order.

In Sections C.3.1 through C.3.5, we discuss each of the five assignments in turn, describing what hardware-simulation facilities and operating-system structures we provide, and what we ask you to implement. Of course, because Nachos is continuing to evolve, what is described here is a snapshot of what is available at the time of printing. Section C.4 explains how to obtain more up-to-date information.

The assignments are intended to be of roughly equal size, each taking approximately 3 weeks of a 15-week (semester) course, assuming that two people work together on each. The file-system assignment is the most difficult of the five; the multiprogramming assignment is the least difficult. Faculty who have used Nachos say that they find it useful to spend 1/2 to 1 hour per week discussing the assignments. We have found it useful for faculty to conduct a design review with each pair of students the week before each assignment is due.

Nachos is intended to encourage a quantitative approach to operating-system design. Frequently, the choice of how to implement an operating-system function reduces to a tradeoff between simplicity and performance. Making informed decisions about tradeoffs is one of the key tasks to learn in an undergraduate operating-system course. The Nachos hardware simulation reflects current hardware performance characteristics (except that kernel execution time is estimated, rather than being measured directly). The assignments exploit this feature by asking that you explain and optimize the performance of your implementations on simple benchmarks.

The Nachos kernel and simulator are implemented in a subset of C++. Object-oriented programming is becoming more popular, and it is a natural idiom for stressing the importance of modularity and clean interfaces in building systems. Unfortunately, C++ is a complicated language; thus, to simplify matters, we omitted certain aspects from standard C++: derived classes, oper-

ator and function overloading, C++ streams, and generics. We also kept inlines to a minimum. The Nachos distribution includes a short primer to help people who know C to learn our subset of C++; we have found that our students pick up this subset quickly.

C.3.1 Thread Management

The first assignment introduces the concepts of threads and concurrency. The baseline Nachos kernel provides a basic working thread system and an implementation of semaphores; the assignment is to implement Mesa-style locks and condition variables using semaphores, and then to implement solutions to a number of concurrency problems using these synchronization primitives.

In much the same way as understanding pointers can be difficult for beginning programmers, understanding concurrency requires a conceptual leap. We believe that a good way to learn about concurrency is to take a hands-on approach. Nachos helps to teach concurrency in two ways. First, thread management in Nachos is *explicit*: it is possible to trace, literally statement by statement, what happens during a context switch from one thread to another, from the perspectives of an outside observer and of the threads involved. We believe that this experience is crucial to demystifying concurrency. Precisely because C and C++ allow nothing to be swept under the carpet, concurrency may be easier to understand (although more difficult to use) in these programming languages than in those explicitly designed for concurrency, such as Ada or Modula-3.

Second, a working thread system, like that in Nachos, provides a chance to practice writing, and testing, concurrent programs. Even experienced programmers find it difficult to think concurrently. When we first used Nachos, we omitted many of the practice problems that we now include in the assignment, thinking that students would see enough concurrency in the rest of the project. Later, we realized that many students were still making concurrency errors even in the final phase of the project.

Our primary goal in building the baseline thread system was simplicity, to reduce the effort required to trace through the thread system's behavior. The implementation takes a total of about 10 pages of C++ and one page of MIPS assembly code. For simplicity, thread scheduling is normally nonpreemptive, but to emphasize the importance of critical sections and synchronization, we have a command-line option that causes threads to be time sliced at "random," but repeatable, points in the program. Concurrent programs are correct only if they work when a context switch can happen at any time.

C.3.2 File Systems

Real file systems can be complex artifacts. The UNIX file system, for example, has at least three levels of indirection—the per-process file-descriptor table, the

system wide open-file table, and the in-core inode table—before you even get to disk blocks. As a result, to build a file system that is simple enough to read and understand in a couple of weeks, we were forced to make some difficult choices about where to sacrifice realism.

We provide a basic working file system, stripped of as much functionality as possible. Although the file system has an interface similar to that of UNIX (cast in terms of C++ objects), it also has many significant limitations with respect to commercial file systems: there is no synchronization (only one thread at a time can access the file system), files have a very small maximum size, files have a fixed size once created, there is no caching or buffering of file data, the file name space is completely flat (there is no hierarchical directory structure), and there is no attempt to provide robustness across machine and disk crashes. As a result, the basic file system takes only about 15 pages of code.

The assignment is (1) to correct some of these limitations, and (2) to improve the performance of the resulting file system. We list a few possible optimizations, such as caching and disk scheduling, but part of the exercise is to decide which solutions are the most cost effective.

At the hardware level, we provide a disk simulator, which accepts read sector and write sector requests and signals the completion of an operation via an interrupt. The disk data are stored in a UNIX file; read and write sector operations are performed using normal UNIX file reads and writes. After the UNIX file is updated, we calculate how long the simulated disk operation should have taken (from the track and sector of the request), and set an interrupt to occur that far in the future. Read and write sector requests (emulating hardware) return immediately; higher-level software is responsible for waiting until the interrupt occurs.

We made several mistakes in developing the Nachos file system. In our first attempt, the file system was much more realistic than the current one, but it also took more than four times as much code. We were forced to rewrite it to cut it down to code that could be read and understood quickly. When we handed out this simpler file system, we did not provide sufficient code for it to be working completely; we left out file read and file write to be written as part of the assignment. Although these functions are fairly straightforward to implement, the fact that the code did not work meant that students had difficulty understanding how each of the pieces of the file system fit with the others.

We also initially gave students the option of which limitation to fix; we found that students learned the most from fixing the first four listed. In particular, the students who chose to implement a hierarchical directory structure found that, although it was conceptually simple, the implementation required a relatively large amount of code.

Finally, many modern file systems include some form of write-ahead logging or log structure, simplifying crash recovery. The assignment now completely ignores this issue, but we are currently looking at ways to do crash

recovery by adding simple write-ahead logging code to the baseline Nachos file system. As it stands, the choice of whether or not to address crash recovery is simply a tradeoff. In the limited amount of time available, we ask students to focus on how basic file systems work, how the file abstraction allows disk data layout to be changed radically without changing the file-system interface, and how caching can be used to improve I/O performance.

C.3.3 Multiprogramming

In the third assignment, we provide code to create a user address space, to load a Nachos file containing an executable image into user memory, and then to run the program. The initial code is restricted to running only a single user program at a time. The assignment is to expand this base to support multiprogramming, to implement a variety of system calls (such as UNIX **fork** and **exec**) as well as a user-level shell, and to optimize the performance of the resulting system on a mixed workload of I/O- and CPU-bound jobs.

Although we supply little Nachos kernel code as part of this assignment, the hardware simulation does require a fair amount of code. We simulate the entire MIPS R2/3000 integer instruction set and a simple single-level page-table translation scheme. (For this assignment, a program's entire virtual address space must be mapped into physical memory; true virtual memory is left for assignment 4.) In addition, we provide an abstraction that hides most of the details of the MIPS object-code format.

This assignment requires few conceptual leaps, but it does tie together the work of the previous two assignments, resulting in a usable—albeit limited —operating system. Because the simulator can run C programs, it is easy to write utility programs (such as the shell or UNIX **cat**) to exercise the system. (One overly ambitious student attempted unsuccessfully to port emacs.) The assignment illustrates that there is little difference between writing user code and writing operating-system kernel code, except that user code runs in its own address space, isolating the kernel from user errors.

One important topic that we chose to leave out (again, as a tradeoff against time constraints) is the trend toward a small-kernel operating-system structure, where pieces of the operating system are split off into user-level servers. Because of Nachos' modular design, it would be straightforward to move Nachos toward a small-kernel structure, except that (1) we have no symbolic debugging support for user programs, and (2) we would need a stub compiler to make it easy to make remote procedure calls across address spaces. One reason for adopting a micro-kernel design is that it is easier to develop and debug operating-system code as a user-level server than if the code is part of the kernel. Because Nachos runs as a UNIX process, the reverse is true: It is easier to develop and debug Nachos kernel code than application code running on top of Nachos.

C.3.4 Virtual Memory

Assignment 4 is to replace the simple memory-management system from the previous assignment with a true virtual-memory system—that is, one that presents to each user program the abstraction of an (almost) unlimited virtual-memory size by using main memory as a cache for the disk. We provide no new hardware or operating-system components for this assignment.

The assignment has three parts. The first part is to implement the mechanism for page-fault handling—the kernel must catch the page fault, find the needed page on disk, find a page frame in memory to hold the needed page (writing the old contents of the page frame to disk if the page frame is dirty), read the new page from disk into memory, adjust the page-table entry, and then resume the execution of the program. This mechanism can take advantage of the code written for the previous assignments: The backing store for an address space can be represented simply as a Nachos file, and synchronization is needed when multiple page faults occur concurrently.

The second part of the assignment is to devise a policy for managing the memory as a cache—for deciding which page to toss out when a new page frame is needed, in what circumstances (if any) to do read-ahead, when to write unused dirty pages back to disk, and how many pages to bring in before starting to run a program.

These policy questions can have a large effect on overall system performance, in part because of the large and increasing gap between CPU speed and disk latency—this gap has widened by two orders of magnitude in only the past decade. Unfortunately, the simplest policies often have unacceptable performance. So that realistic policies are encouraged, the third part of the assignment is to measure the performance of the paging system on a matrix multiply program where the matrices do not fit in memory. This workload is not meant to be representative of real-life paging behavior, but it is simple enough to illustrate the influence of policy changes on application performance. Further, the application illustrates several of the problems with caching: Small changes in the implementation can have a large effect on performance.

C.3.5 Networking

Although distributed systems have become increasingly important commercially, most instructional operating systems do not have a networking component. To address this omission, we chose the capstone of the project to be to write a significant and interesting distributed application.

At the hardware level, each UNIX process running Nachos represents a uniprocessor workstation. We simulate the behavior of a network of workstations by running multiple copies of Nachos, each in its own UNIX process, and by using UNIX sockets to pass network packets from one Nachos "machine" to another. The Nachos operating system can communicate with other systems

by sending packets into the simulated network; the transmission is accomplished by socket send and receive. The Nachos network provides unreliable transmission of limited-sized packets from machine to machine. The likelihood that any packet will be dropped can be set as a command-line option, as can the seed used to determine which packets are "randomly" chosen to be dropped. Packets are dropped but are never corrupted, so that checksums are not required.

To show how to use the network and, at the same time, to illustrate the benefits of layering, the Nachos kernel comes with a simple post-office protocol layered on top of the network. The post-office layer provides a set of mailboxes that route incoming packets to the appropriate waiting thread. Messages sent through the post office also contain a return address to be used for acknowledgments.

The assignment is first to provide reliable transmission of arbitrary-sized packets, and then to build a distributed application on top of that service. Supporting arbitrary-sized packets is straightforward—you need merely to split any large packet into fixed-sized pieces, to add fragment serial numbers, and to send the pieces one by one. Ensuring reliability is more interesting, requiring a careful analysis and design. To reduce the time required to do the assignment, we do not ask you to implement congestion control or window management, although of course these are important issues in protocol design.

The choice of how to complete the project is left open. We do make a few suggestions: multiuser UNIX talk, a distributed file system with caching, a process-migration facility, distributed virtual memory, a gateway protocol that is robust to machine crashes. Perhaps the most interesting application that a student built (that we know of) was a distributed version of the "battleship" game, with each player on a different machine. This application illustrated the role of distributed state, since each machine kept only its local view of the gameboard; it also exposed several performance problems in the hardware simulation, which we have since fixed.

Perhaps the biggest limitation of the current implementation is that we do not model network performance correctly, because we do not keep the timers on each of the Nachos machines synchronized with one another. We are currently working on fixing this problem, using distributed simulation techniques for efficiency. These techniques will allow us to make performance comparisons between alternate implementations of network protocols; they will also enable us to use the Nachos network as a simulation of a message-passing multiprocessor.

C.4 ■ Information on Obtaining a Copy of Nachos

You can obtain Nachos by anonymous ftp from the machine ftp.cs.berkeley.edu by following these steps:

1. Use UNIX ftp to access ftp.cs.berkeley.edu:

   ftp ftp.cs.berkeley.edu

2. You will get a login prompt. Type the word anonymous, and then use your e-mail address as the password.

   Name: anonymous
   Password: tea@cs.berkeley.edu (for example)

3. You are now in ftp. Move to the Nachos subdirectory.

   ftp> cd ucb/nachos

4. You must remember to turn on "binary" mode in ftp; unfortunately, if you forget to do so, when you fetch the Nachos file, it will be garbled without any kind of warning message. This error is one of the most common that people make in obtaining software using anonymous ftp.

   ftp> binary

5. You can now copy the compressed UNIX tar file containing the Nachos distribution to your machine. The software will automatically enroll you in a mailing list for announcements of new releases of Nachos; you can remove yourself from this list by sending e-mail to nachos@cs.berkeley.edu.

   ftp> get nachos.tar.Z

6. Exit the ftp program:

   ftp> quit

7. Decompress and detar to obtain the Nachos distribution. (If the decompress step fails, you probably forgot to set binary mode in ftp in step 4. You will need to start over.)

   uncompress nachos.tar.Z
   tar -xf nachos.tar

8. The preceding steps will produce several files, including the code for the baseline Nachos kernel, the hardware simulator, documentation on the sample assignments, and the C++ primer. There will also be a README file to get you started: It explains how to build the baseline system, how to print out documentation, and which machine architectures are currently supported.

<div align="center">cat README</div>

Mendel Rosenblum at Stanford has ported the Nachos kernel to run on Sun SPARC workstations, although user programs running on top of Nachos must still be compiled for the MIPS R2/3000 RISC processor. Ports to machines other than Digital Equipment Corporation MIPS UNIX workstations and Sun SPARC workstations are in progress. Up-to-date information on machine availability is included in the README file in the distribution. The machine dependence comes in two parts. First, the Nachos kernel runs just like normal application code on a UNIX workstation, but a small amount of assembly code is needed in the Nachos kernel to implement thread context switching. Second, Nachos simulates the instruction-by-instruction execution of user programs, to catch page faults and other exceptions. This simulation assumes the MIPS R2/3000 instruction set. To port Nachos to a new machine, we replace the kernel thread-switch code with machine-specific code, and rely on a C cross-compiler to generate MIPS object code for each user program. (A cross-compiler is a compiler that generates object code for one machine type while running on a different machine type.) Because we rely on a cross-compiler, we do not have to rewrite the instruction-set simulator for each port to a new machine. The SPARC version of Nachos, for instance, comes with instructions on how to cross-compile to MIPS on the SPARC.

Questions about Nachos and bug reports should be directed via e-mail to nachos@cs.berkeley.edu. Questions can also be posted to the alt.os.nachos newsgroup.

C.5 ■ Conclusions

Nachos is an instructional operating system designed to reflect recent advances in hardware and software technology, to illustrate modern operating-system concepts, and, more broadly, to help teach the design of complex computer systems. The Nachos kernel and sample assignments illustrate principles of computer-system design needed to understand the computer systems of today and of the future: concurrency and synchronization, caching and locality, the tradeoff between simplicity and performance, building reliability from unreliable components, dynamic scheduling, object-oriented programming, the power of a level of translation, protocol layering, and distributed computing.

Familiarity with these concepts is valuable, we believe, even for those people who do not end up working in operating-system development.

Bibliographical Notes

Wayne Christopher, Steve Procter, and Thomas Anderson (the author of this appendix) did the initial implementation of Nachos in January 1992. The first version was used for one term as the project for the undergraduate operating-systems course at The University of California at Berkeley. We then revised both the code and the assignments, releasing Nachos, Version 2 for public distribution in August 1992; Mendel Rosenblum ported Nachos to the Sun SPARC workstation. The second version is currently in use at several universities including Carnegie Mellon, Colorado State, Duke, Harvard, Stanford, State University of New York at Albany, University of Washington, and, of course, Berkeley; we have benefited tremendously from the suggestions and criticisms of our early users.

In designing the Nachos project, we have borrowed liberally from ideas found in other systems, including the TOY operating system project, originally developed by Ken Thompson while he was at Berkeley, and modified extensively by a collection of people since then; Tunis, developed by Rick Holt [Holt 1983]; and Minix, developed by Andy Tanenbaum [Tanenbaum 1987]. Lions [1977] was one of the first people to realize that the core of an operating system could be expressed in a few lines of code, and then used to teach people about operating systems. The instruction-set simulator used in Nachos is largely based on a MIPS simulator written by John Ousterhout.

We credit Lance Berc with inventing the acronym "Nachos" Not Another Completely Heuristic Operating System.

Credits

This Appendix is derived from Christopher/Procter/Anderson, "The Nachos Instructional Operating System," Proceedings of Winter USENIX, January 1993. Reprinted with permission of the authors.

Errata[*]
OPERATING SYSTEM CONCEPTS, SIXTH EDITION
Silberschatz, Galvin, and Gagne
December 10, 2001

To Be Corrected in 2nd Printing:

page 11, section 1.3, first sentence: *Personal Computers PCs $\Rightarrow$ Personal Computers (PCs)*

page 20, Figure 1.6 line 3 from the bottom: *"no software" should come before "compliers"*

page 64, fourth paragraph, line 4: *count $\Rightarrow$ cout*

page 76, Figure 3.7, third light block fix line breaks:
               signals terminal // handling $\Rightarrow$ signals // terminal handling
               swapping block I/O // system $\Rightarrow$ swapping // block I/O system

page 80, Figure 3.10: *add arrow to OS/2 application*
               POSIX application $\Rightarrow$ POSIX server

page 92, line 2: *Brinch-Hansen $\Rightarrow$ Brinch Hansen*

page 106, Figure 4.8, line 2: *void main $\Rightarrow$ main*

page 108, last paragraph, second line: *or by explicitly $\Rightarrow$ or explicitly*

page 109, fourth paragraph, last line: *items can be the buffer st $\Rightarrow$ items can be the buffer at*

page 114, Section 4.5.5, paragraph 2 next to last line: *RPC model a $\Rightarrow$ an RPC models a*

page 117, Section 4.6.1, first paragraph, third line: *A socket is made up $\Rightarrow$ A socket is identified*

page 140, line 14*: %d must be <= o\n $\Rightarrow$ %d must be >= o\n*

page 176, Figure 6.9: *CPU173 $\Rightarrow$ CPU 173*

page 196, last line: *number k !=0 $\Rightarrow$ number [k]!=0*

page 197, Figure 7.5, line 7: *(number [j, j] < number [i , i])) ; $\Rightarrow$ ((number [j], j) < (number [i] , i))) ;*

page 212, line 3 from the bottom: *region v when B do S; $\Rightarrow$ region v when (B) S;*

page 220, last line: *Brinch-Hansen $\Rightarrow$ Brinch Hansen*

page 241, paragraph 3, lines 1 and 2: *Brinch-Hansen $\Rightarrow$ Brinch Hansen*

page 291, line 8: *2^{36} bytes (or 64GB) $\Rightarrow$ 2^{44} bytes (or 16TB)*

page 299, Section 9.4.4.2, line 5: *he mapped page frame $\Rightarrow$ the mapped page frame*

---

[*] Errors reported by: Don Colton, Elaine Cheong, Jim McQuillan, Alison Pechenick, Greg Sims, Sekar Sundarraj

page 349, paragraph 3, line 2: *algorightm* $\Rightarrow$ *algorithm*

page 355, paragraph 2, line 2 from bottom: *checking and investigating* $\Rightarrow$ *clearing and investigating*

page 377, Figure 11.1, under function, line 1: *read to run* $\Rightarrow$ *ready to run;*
    under usual extension, cell 6: *rrf* $\Rightarrow$ *rtf*
    under usual extension, cell 7: *remove "mpeg, mov, rm"*
    under usual extension, cell 8: *arc, zip, tar* $\Rightarrow$ *ps, pdf, jpg*

page 381, Figure 11.3 caption: *remove extra period from end*

page 389, paragraph 3, line 3 from bottom: *(95, 95, NT, 2000)* $\Rightarrow$ *(95, 98, NT, 2000, XP)*

page 406, Section 11.6.4 last line: *creation date* $\Rightarrow$ *date of last modification*

page 407, line 1*: to block* $\Rightarrow$ *to order*

page 409, Bibliographical Notes, paragraph 3, line 2: *VAX VMS* $\Rightarrow$ *VAX/VMS*

page 415, paragraph 3, line 5: *system- wide* $\Rightarrow$ *system-wide*

page 415, last line: *index* $\Rightarrow$ *entry*

page 427, paragraph 3, last line: *non-varnil* $\Rightarrow$ *non-nil*

page 434, second paragraph, last line: *functional generality* $\Rightarrow$ *general functionality*

page 444, paragraph 4 line 2: /etc/vfstab $\Rightarrow$ */etc/vfstab*

page 523, paragraph 1, line 3: *switching disks is expensive, but switching tapes is not* $\Rightarrow$ *switching tapes is expensive, but switching disks is not*

page 635, bullet 5: *b3 is less than or equal to i* $\Rightarrow$ *b3 is greater than or equal to i*

page 810, in all 6 instances: *Brinch-Hansen* $\Rightarrow$ *Brinch Hansen*