Operating System Concepts

PART ONE: OVERVIEW

Chapter 1: Introduction

- An operating system is a program that manages the computer hardware
- provides a basis for application programs
- acts as an intermediary between computer-user and hardware
- provides an environment within which other programs can do work
- **Objectives:**
    - To provide a grand tour of the major components of operating system.
    - To describe the basic organization of the computer.

What Operating Systems Do

- Computer system divided into 4 components:
    - **Hardware** – provides basic computing resources
        - CPU, memory, I/O devices
    - **Operating system**
        - Controls and coordinates use of hardware among various applications and users
    - **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
        - Word processors, compilers, web browsers, database systems, video games
    - **Users**
        - People, machines, other computers
- **Hardware**, consisting out of: Central Processing Unit (CPU); Memory; Input/Output (I/O) devices, provides the basic computing resources for the system.
- **Application programs** define the ways in which these resources are used to solve users' computing problems.
- The **operating system** controls the hardware and coordinates its use among the various application programs.
- Can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system.
- The operating system from two view points:
    - User View
    - System View

User View

- Users view varies according to interface used.
- Some operating systems are designed for **ease of use** with some attention paid to performance and none paid to resource allocation.

    - These systems are designed for the single-user experience.
- Some operating systems are designed to maximize resource utilization to assure that all available CPU time, memory, and I/O are used efficiently and no individual user takes more than his share.
    - These are multi-user systems where terminals are connected to mainframe or minicomputers.
    - users share resources and may exchange information.
- In some cases users sit at workstations connected to networks of other workstations and servers.
    - These systems have dedicated resources such as networking and servers.
    - These operating systems compromise between individual usability and resource utilization.

System View

- The program that is most intimately involved with the hardware.
- The operating system is a resource allocator.
- The following resources may be required to solve a problem:
    - CPU time
    - memory space
    - file-storage space
    - I/O devices
    - etc.
- The operating system acts as the manager of these resources.
- A different view of an operating system emphasizes the need to control the various I/O devices and user programs. The operating system as a control program.
    - A control program manages the execution of user programs to prevent errors and improper use of the computer.
    - It is especially concerned with the operation and control of I/O devices.

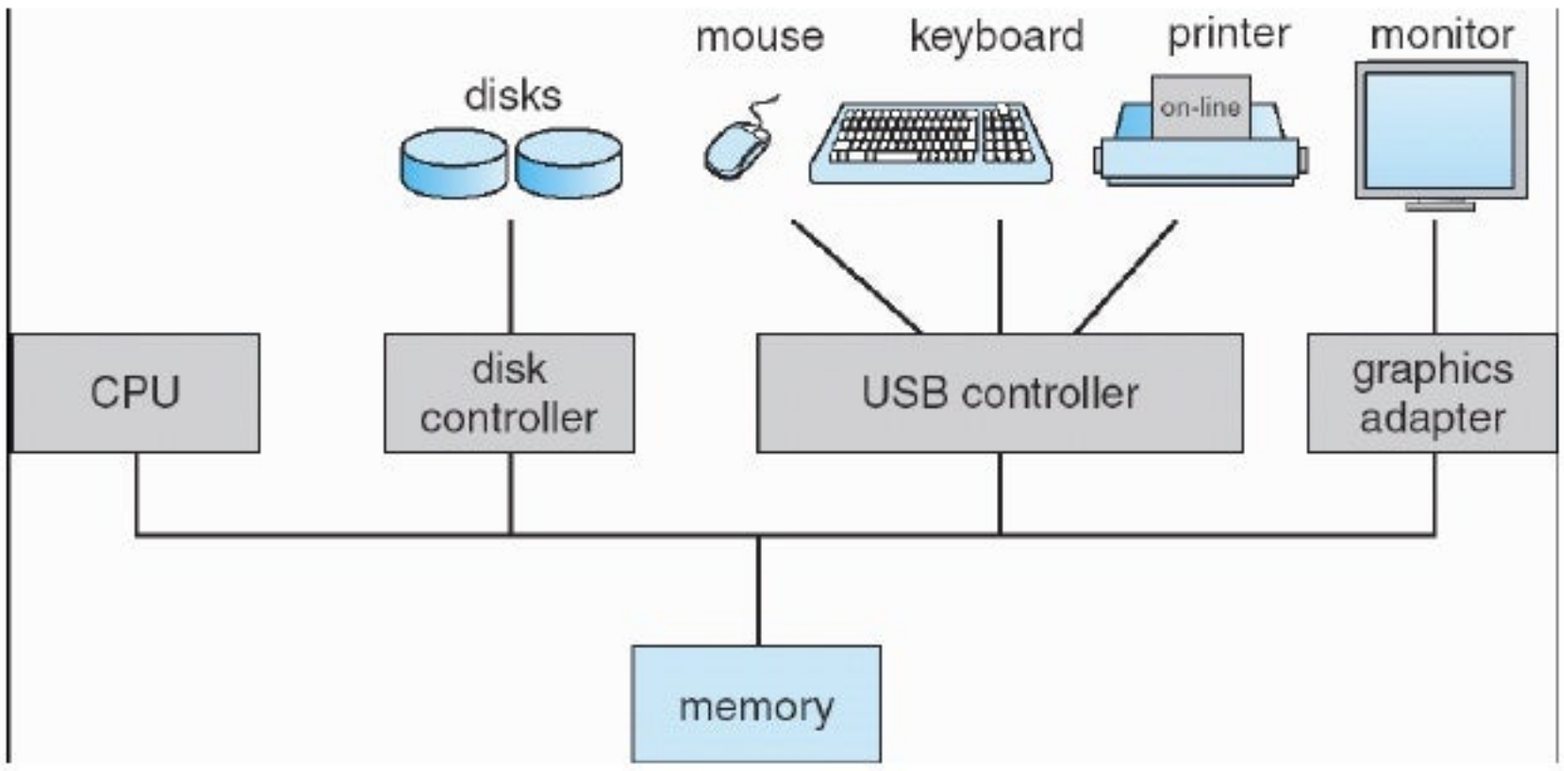
Defining Operating Systems

- There is no real definition for an Operating System.
- The goal of an operating system is to execute programs and to make solving user problems easier.
- The computer hardware is constructed toward this goal.
- Because hardware alone is not easy to use, application programs are developed.
- These programs require common operations, such as controlling I/O.
- These common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.
- The definition we use here is as follows:
    - The operating system is the one program running at all times on the computer - usually called the kernel.
- Along with the kernel there are two other types of programs:
    - **System programs:** associated with the operating system but not part of the kernel.
    - **Application programs:** include all programs not associated with the operation of the system.

Computer-System Organization

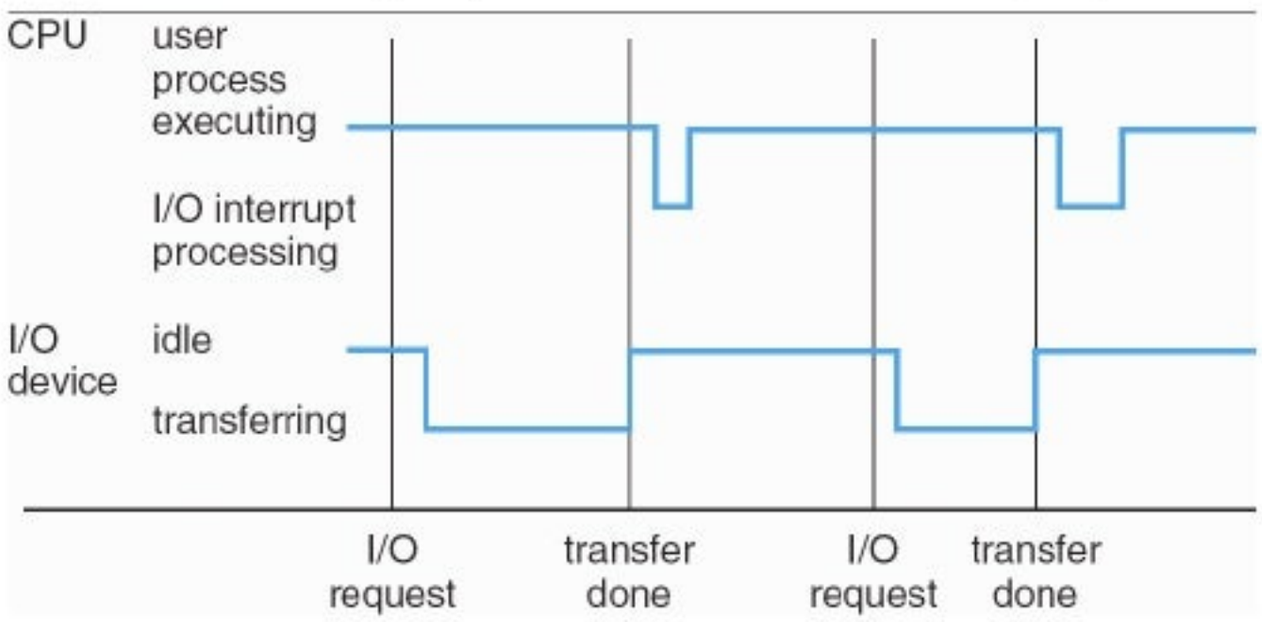
- Computer-system operation

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



Computer-System Operation

- For a computer to start running it needs an initial program to run at boot time.
  - This initial program or **bootstrap** program tends to be simple.
  - It is stored in ROM or EEPROM and is known as firmware within the computer hardware.
  - It initializes all aspects of the system.
  - The bootstrap must know how to load the operating system. To accomplish this the bootstrap program must locate and load the operating system kernel into memory.
- The occurrence of an event is usually signaled by an **interrupt** from either hardware or software.
  - Hardware trigger an interrupt by sending a **signal** to the CPU.
  - Software may trigger an interrupt by executing a special operation called a **system call or monitor call**.
  - Look at fig 1.3 p.9 for a timeline of the interrupt operation.

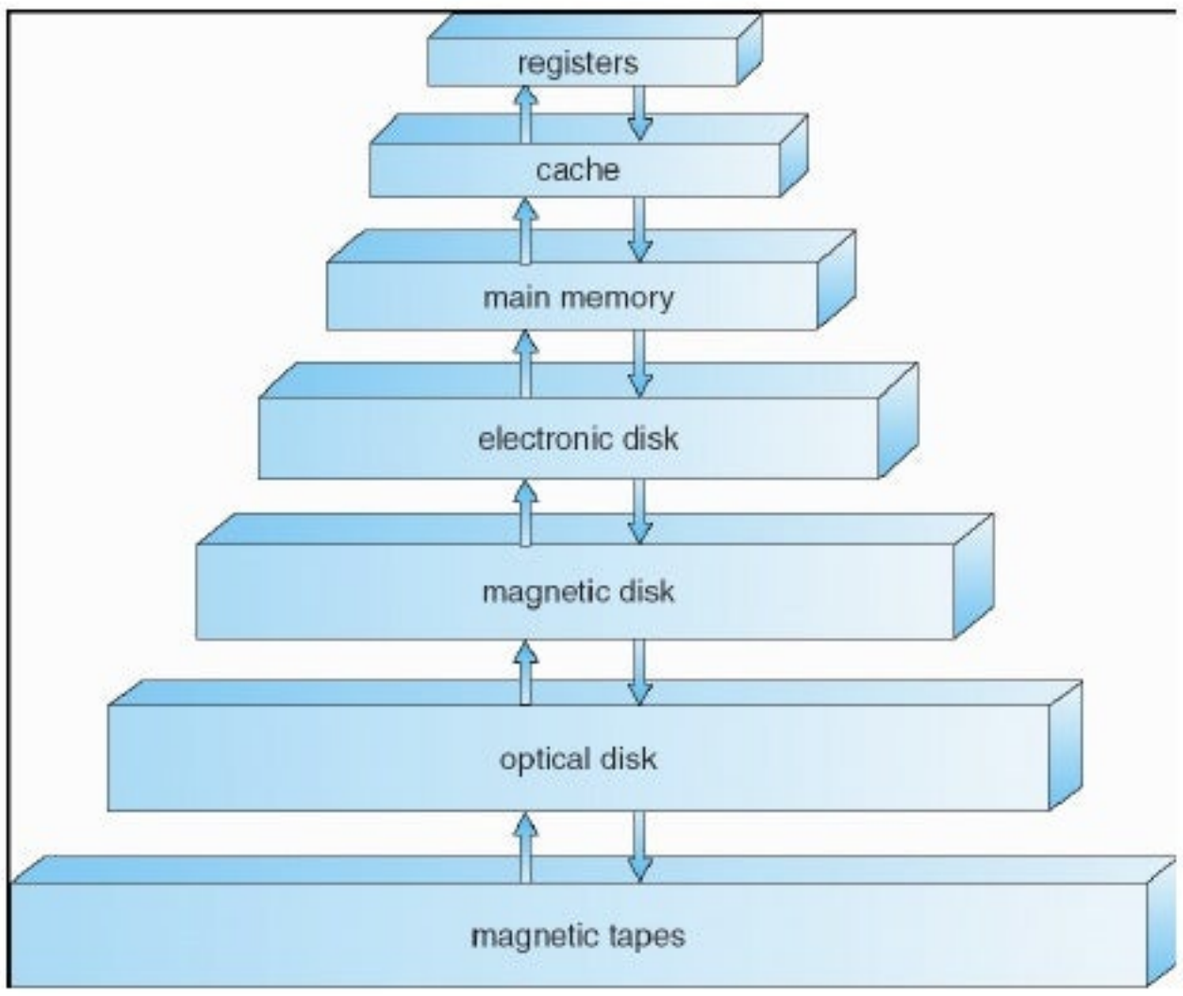


- Since only a predefined number of interrupts are possible, a table of pointers to interrupt routines is used to increase speed.
- The table of interrupt pointers is stored in low memory.
- These locations keep the addresses of the interrupt service routines for the various devices.
- This array or interrupt vector is then indexed by a unique device number. This number is given with the interrupt request to provide the address of the interrupt service routine for the interrupting device.

- The CPU and device controllers (each in charge of a certain type of device) are connected to shared memory through a common bus
- The CPU and device controllers can execute concurrently, competing for memory cycles
- A memory controller synchronizes access to the memory
- Bootstrap program = a simple initial program that runs when the computer is powered up, and transfers control to the OS
- Modern OSs are interrupt driven: If there is nothing for the OS to do, it will wait for something to happen
- Events are almost always signaled by an interrupt or a trap:
- Hardware interrupts usually occur by sending a signal to the CPU
- Software interrupts usually occur by executing a system call
- Trap = a software-generated interrupt caused by an error / a request from the program that an OS service be performed

Storage Structure

- General purpose computers run their programs from random-access memory (RAM) called main memory.
    - Main memory is implemented using **dynamic random-access memory (DRAM)** technology.
- Interaction with memory is achieved through a sequence of load and store instructions to specific memory addresses.
    - Load instruction moves a word from main memory to an internal register within the CPU.
    - Store instruction moves content of a register to main memory.
    - The CPU automatically loads instructions from main memory for execution.
- Instruction-execution cycle as executed by von Neumann architecture system:
    - Fetch instruction from memory and stores instruction in the instruction register.
    - Decodes instruction and may cause operands to be fetched from memory and store in some internal register.
    - After instruction on operands executed, result is stored back in memory.
- The memory unit only sees a stream of memory addresses; it doesn't know they are generated.
    - We are interested only in the sequence of memory addresses generated by the running program.
- Ideally we want programs and data to reside in main memory permanently, but it is not possible for the following two reasons:
    - Main memory is to small to store all needed programs and data permanently.
    - Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.
- For this reason most computer systems provide **secondary storage** as an extension of main memory.
    - The main requirement of **secondary storage** is that it must **hold large quantities** of data.
    - Most common secondary storage device is magnetic disk which provide storage for both programs and data.
    - There are other types of secondary storage systems of which the speed, cost, size, and volatility differ.
    - Look at fig 1.4 p.11 for the storage hierarchy.



  - *Caching*–copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage.
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy
- Movement between levels of storage hierarchy can be explicit or implicit

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

I/O Structure

- Each device controller is in charge of a specific type of device
- A SCSI (small computer-systems interface) controller can have 7 or more devices attached to it
- A device controller maintains some buffer storage and a set of special-purpose registers

- It moves data between the peripherals and its buffer storage
- I/O interrupts
    - Starting an I/O operation:
        - The CPU loads the appropriate registers in the device controller
        - The device controller examines the contents of these registers to see what action to take
        - Once the transfer of data is complete, the device controller informs the CPU that it has finished, by triggering an interrupt
- Synchronous I/O: Control is returned to the user process at I/O completion
    - To wait for I/O completion, some machines have a 'wait' instruction, while others have a wait loop: 'Loop:jmp Loop'
    - Advantage: The OS knows which device is interrupting
    - Disadvantage: No concurrent I/O operations to many devices
    - Disadvantage: No overlapping useful computation with I/O
- Asynchronous I/O: Control is returned to the user process without waiting for I/O to complete
    - A device-status table is used to keep track of I/O devices
    - Each table entry shows the device's type, address, & state
    - If other processes request a busy device, the OS maintains a wait queue
    - When an interrupt occurs, the OS determines which I/O device caused the interrupt and indexes the table to determine the status of the device, and modifies it
    - Advantage: increased system efficiency
- DMA structure
    - DMA is used for high-speed I/O devicesA program or the OS requests a data transfer
    - The OS finds a buffer for the transfer
    - A device driver sets the DMA controller registers to use appropriate source & destination addresses
    - The DMA controller is instructed to start the I/O operation
    - During the data transfer, the CPU can perform other tasks
    - The DMA controller 'steals' memory cycles from the CPU (which slows down CPU execution)
- The DMA controller interrupts the CPU when the transfer has been completed
- The device controller transfers a block of data directly to / from its own buffer storage to memory, with no CPU intervention
- There is no need for causing an interrupt to the CPU
- The basic operation of the CPU is the same:

Computer-System Architecture

Single-Processor Systems

- On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes

Multiprocessor Systems

- Several processors share the bus, clock, memory, peripherals…

- 3 main advantages:
    - Increased throughput
        - More processors get more work done in less time
    - Economy of scale
        - You save money because peripherals, storage, & power are shared
    - Increased reliability
    - Failure of one processor won't halt the system
- Graceful degradation = continuing to provide service proportional to the level of surviving hardware
- *Tandem system*
    - 2 identical processors (primary + backup) are connected by a bus
    - 2 copies are kept of each process, and the state information of each job is copied to the backup at fixed checkpoints
    - If a failure is detected, the backup copy is activated and restarted from the most recent checkpoint
    - Expensive, because of hardware duplication
- *Symmetric multiprocessing (SMP)*
    - Used by the most common multiple-processor systems
    - Each processor runs an identical copy of the OS, and these copies communicate with one another as needed
    - Processes and resources are shared dynamically among processors
    - Advantage of SMP: many processes can run simultaneously without causing a significant deterioration of performance
    - Disadvantage of SMP: Since the CPUs are separate, one may be idle while another is overloaded, resulting in inefficiencies
- *Asymmetric multiprocessing*
    - Each processor is assigned a specific task
    - A master processor controls the system and the others either look to the master for instruction or have predefined tasks
    - Master-slave relationship: The master processor schedules & allocates work to the slave processors
    - As processors become less expensive and more powerful, extra OS functions are off-loaded to slave processors (back ends)
        - E.g. you could add a microprocessor with its own memory to manage a disk system, to relieve the main CPU

Cluster Systems

- Multiple CPUs on two / more individual systems coupled together
- Clustering is usually performed to provide high availability
- A layer of cluster software runs on the cluster nodes
- Each node can monitor the others, so if the monitored machine fails, the monitoring one can take over
- *Asymmetric clustering*
    - A hot standby host machine and one running the applications

    - The hot standby host just monitors the active server
    - If that server fails, the hot standby host à active server
- *Symmetric mode*
    - Two / more hosts run applications and monitor each other
    - More efficient mode, as it uses all the available hardware
    - Requires that more than one application be available to run
- *Parallel clusters*
    - Allow multiple hosts to access same data on shared storage
    - Most clusters don't allow shared access to data on the disk
    - Distributed file systems must provide access control and locking
    - DLM = Distributed Lock Manager
    - Global clusters: Machines could be anywhere in the world
    - Storage Area Networks: Hosts are connected to a shared storage

Operating System Structure

- Multiprogramming
    - Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute
    - The OS keeps several jobs in memory and begins to execute one of them until it has to wait for a task (like an I/O operation), when it switches to and executes another job
    - Job scheduling = deciding which jobs to bring into memory if there is not enough room for all of them
    - CPU scheduling = deciding which job to choose if several are ready to run at the same time
- Time-sharing (multitasking) systems
    - Like multiprogramming, but *interactive* instead of batch!
    - Interactive computer system: direct communication between user & system, where the user expects immediate results
    - Time-sharing: many users can share the computer simultaneously
    - The CPU switches among multiple jobs so frequently, that users can interact with each program while it is running
    - CPU scheduling and multiprogramming provide each user with a small portion of a time-shared computer
    - Process = a program that's loaded into memory and executing
    - For good response times, jobs may have to be swapped in & out of main memory to disk (now serving as a backing store for memory)
    - Virtual memory = a technique that allows the execution of a job that may not be completely in memory
    - Advantage of VM: programs can be larger than physical memory
    - Time-sharing systems must also provide a file system
    - The file system resides on a collection of disks, so disk management must be provided
    - Concurrent execution needs sophisticated CPU-scheduling schemes

- The system must provide mechanisms for job synchronization & communication and ensure that jobs don't get stuck in a deadlock

Chapter 2 :System Structures

- **Objectives:**
  - To describe the services an operating system provides to users, processes, and other systems.
  - To discuss the various ways of structuring an operating system.
  - To explain how operating systems are installed and customized and how they boot.
- We can view an operating system from several vantage points.
  - One view focuses on the services that the system provides.
  - Another on the interface that it makes available to users and programmers.
  - And thirdly on the components and their interconnections.
- Here we look at the viewpoint from the users, programmers and the operating-system designers.

Operating-System Services

- Operating system provides environment for execution of programs.
- Operating systems provides services to programs and users that use those programs.
- We identify common classes of services for all operating systems.
- **One set of operating-system services provide functions helpful to the user:**
  - **User interface**
    - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - **Program execution**
    - The system must be able to load a program into memory and run it
    - The program must be able to end its execution, (ab)normally
  - **I/O operations**
    - For specific devices, special functions may be desired (e.g. to rewind a tape drive or to blank a CRT screen)
    - For efficiency and protection, users can't control I/O devices directly, so the OS must provide a means to do I/O
  - **File-system manipulation**
    - Programs need to read, write, create, and delete files
  - **Communications**
    - Communications between processes may be implemented via shared memory, or by the technique of message passing, in which packets of information are moved between processes by the OS
  - **Error detection**
    - Errors may occur in the hardware, I/O devices, user programs…
    - For each type of error, the OS should take appropriate action

        - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- **Another set of operating-system functions exists to ensure the efficient operation of the system itself:**
    - **Resource allocation**
        - When multiple users are logged on, resources must be allocated
        - Some resources have a special allocation code, whereas others have a general request & release code
    - **Accounting**
        - You can keep track of which users use how many & which resources
        - Usage statistics can be valuable if you want to reconfigure the system to improve computing services
    - **Protection and Security**
        - Concurrent processes shouldn't interfere with one another
        - **Protection** involves ensuring that all access to system resources is controlled
        - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
        - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link

User Operating-System Interface

- Two ways that users interface with the operating system:
    - Command Interpreter (Command-line interface)
    - Graphical User Interface (GUI)

Command Interpreter (Command-line interface)

- Main function of command interpreter is to get and execute the next user-specified command.
- Many of the commands are used to manipulate, create, copy, print, execute, etc. files.
- Two general ways to implement these commands:
    - Command interpreter self contains code to execute command;
    - Commands are implemented through system programs.

Graphical User Interface (GUI)

- No entering of commands but the use of a mouse-based window-and-menu system characterized by a **desktop** metaphor.
- The mouse is used to move a pointer to the position of an icon that represents a file, program or folder and by clicking on it the program is invoked.

System Calls

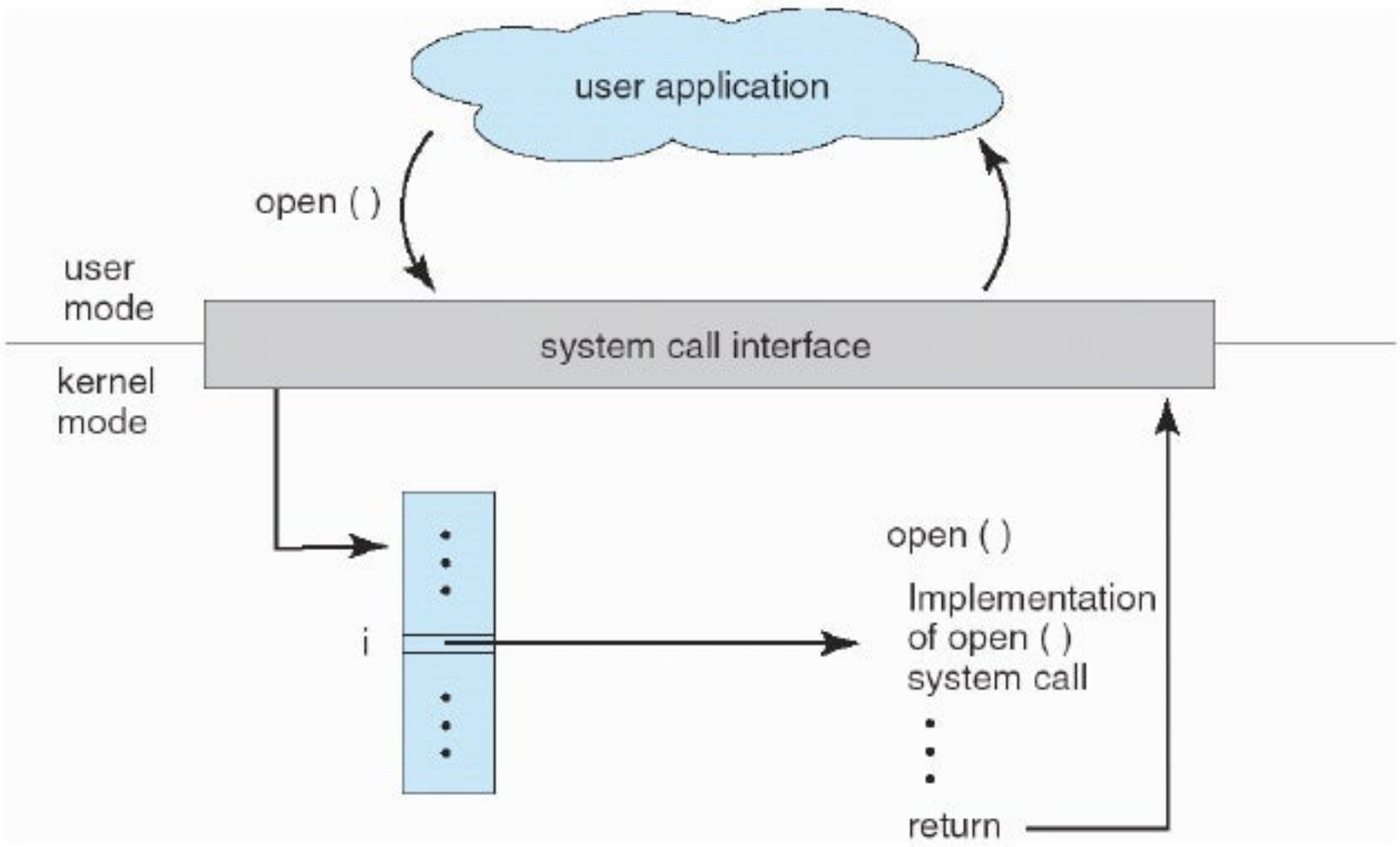
- **System calls** provide an interface to the services made available by an operating system.
- Look at figure 2.4 p.56 TB for an example of a sequence of system calls.
- Application developers design programs according to an application programming interface (API).
    - The API defines a set of functions that are available to an application programmer.
    - This includes the parameters passed to functions and the return values the programmer can accept.

- Win32 API, POSIX API and Java API are the three most common API's.
- The functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

- Benefits of programming according to an API:
  - Program portability;
  - Actual system calls are more detailed and more difficult to work with than the API available to the programmer.

- **Run-time support** system (set of functions built into libraries included with a compiler) provides a **system-call interface** that serves as link to system calls made available by the Operating System.
  - System-call interface intercepts function calls in the API and invokes necessary system calls within the operating system.

- The relationship between API, system-interface and the operating system shown in fig 2.6 p.58 TB.



- System calls occur in different ways on different computers.
- Three general methods used to pass parameters to the operating system.
  - Via registers;
  - Using a block or table in memory and the address is passed as a parameter in a register;
  - The use of a stack is also possible where parameters are pushed onto a stack and popped off the stack by the operating system.
- The block or stack methods do not limit the number or length of parameters being passed.

Types of System Calls

- There are six major categories each with the following types of system calls:
  - **Process control**:
  - **File manipulation**:
  - **Device manipulation**:
  - **Information maintenance**:
  - **Communications**:
  - **Protection**:

Process Control

- A program needs to be able to end execution normally (**end**) or abnormally (**abort**).
    - When abort a memory dump is written to disk for use with a **debugger** program to find the program.
    - The operating system must transfer control to the next invoking command interpreter.
        - Command interpreter then reads next command.
        - In interactive system the command interpreter simply continues with next command.
        - In GUI system a pop-up window will request action from user.
        - In batch system the command interpreter terminates whole job and continues with next job.
            1. Batch systems make use of **control card** system.
            2. If an error occur in execution an error level is assigned.
            3. The error level can be used by command interpreter or other program to determine next action.
- A process might want to execute or load another program.
    - This allows flexibility for the user by enabling the user to execute more than one program at a time.
    - Also to allow existing programs to execute new programs and thus allowing further flexibility.
- The question is where does control goes after such a new program terminates.
    - If control returns to the existing program when the new program terminates a memory image of the existing program should be saved. (Mechanism to call another program)
- If both programs runs concurrently a multiprogramming environment exists.
- We must be able to control the execution of a job or process.
    - The priority;
    - maximum allowable execution time;
    - terminate process;
    - etc...
- We must be able to wait for processes to complete certain actions (wait time / wait event).
    - When action completed a signal is sent to inform the operating system (signal event).
- System locks are also implemented when data is shared between processes to ensure data integrity (acquire lock / release lock).
- Examples of these process control system calls from p. 62 - 64 TB.
- **System calls**:
    - end, abort;
    - load execute;
    - create process, terminate process;
    - get process attributes, set process attributes;
    - wait for time;
    - wait event, signal event;
    - allocate and free memory.

File Management

- These system calls deal with files.
- A file needs to be created then opened for use.
- After the file was read from or written to, the file needs to be closed to indicate that it is no longer in use.
- We need to be able to read and write the attributes of such a file.
- Some operating systems also can move and copy files.
- **System calls**:
    - create file, delete file;
    - open, close;
    - read, write, reposition;
    - get file attributes, set file attributes.
    - move, copy

Device Management

- Resources are needed by processes to execute.
- Examples of resources:
    - main memory;
    - disk drives;
    - access to files;
    - etc;
- If resources available they can be granted and control is returned to user process otherwise the process will have to wait for resources.
- In multi-user environments devices should be locked by a particular user to prevent devices contention and deadlocks.
- Some are physical devices and others are abstract or virtual devices.
- A devices also have to be opened for use and closed after use.
    - Many devices are viewed similar as files and in some operating systems these devices and files are combined.
    - Some system calls are used on files and devices.
    - Even though the devices and files are viewed similarly, their underlying system calls are dissimilar in many cases.
- **System calls**:
    - request device, release device;
    - read, write, reposition;
    - get device attributes, set device attributes;
    - logically attach or detach devices.

Information Maintenance

- System calls to transfer information between user program and operating system.
- Information like:
    - time and date;

    - version number;
    - number of concurrent users;
    - free memory or disk space;
    - etc.
- Debugging information needed for program debugging is also provided in most cases.
- **System calls**:
    - get time or date, set time or date;
    - get system data, set system data;
    - get process, file, or device attributes;
    - get process, file, or device attributes.

Communication

- Two common models of interprocess communication:
    - message-passing model;
    - shared-memory model,
- **Message-passing model**:
    - Useful for transferring small amounts of data.
    - Easier to implement.
    - Communicating processes exchanges messages with one another to transfer information.
    - Messages are exchanged between processes either directly or indirectly through common mailbox.
    - explanation p.65 - 66 TB (NB!!!)

- **Shared-memory model**:
    - Deliver greater speed of communication if communication takes place in the same computer.
    - Greater risk on protection and synchronization problems.
    - Processes use **shared memory create** and **shared memory attach** system calls to create and gain of memory owned by other processes.
    - Two processes agree to remove the restriction that only one process can access a particular part of the memory. This is done to facilitate interprocess communication by sharing the memory.
    - This data is not controlled by the operating system but by the communicating processes.
    - Data can be shared by reading and writing the data to this shared areas.
    - The synchronization of this data is also handled by the processes.
    - explanation p.66 TB (NB!!!)
- **System calls**:
    - create, delete communication connection;
    - send, receive messages;
    - transfer status information;
    - attach or detach remote devices.

Protection

- Provides a mechanism for controlling access to the resources provided by the system.
    - Especially with the Internet and networks, protection is very important.

System calls:

- get file security status, set file security status;
- allow user, deny user;
- set file security group;

System Programs

- **System programs** also known as **system utilities** provide a convenient environment for program development and execution.
- **Divided into the following categories: (P.67 TB give definitions)**
    - File management
        - Programs create, delete, copy, rename, print, dump, list, and generally manipulate files & directories
    - Status information
        - Some programs ask the system for the date, time, disk space, number of users, or similar status information
    - File modification
        - Several text editors may be available to create & modify the content of files stored on disk / tape
    - Programming-language support
        - Compilers, assemblers, and interpreters are provided
    - Program loading and execution
        - The system may provide absolute loaders, re-locatable loaders, linkage editors, and overlay loaders
        - Debugging systems are also needed
    - Communications
        - These programs allow users to send messages to one another's screens, browse the web, send email…
- Read the part on application programs op p. 67 - 68 TB.

Operating-System Design and Implementation

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

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

Design Goals

- Firstly define goals and specification.
    - E.g. Convenience, reliability, speed, flexibility, efficiency…

Mechanisms and Policies

- Mechanisms determine how to do something
- Policies determine what will be done
- Separating policy and mechanism is important for flexibility
- Policies change over time; mechanisms should be general

Implementation

- OS's are nowadays written in higher-level languages like C / C++
- Advantages of higher-level languages: faster development and the OS is easier to port (i.e. move to other hardware)
- Disadvantages of higher-level languages: reduced speed and increased storage requirements

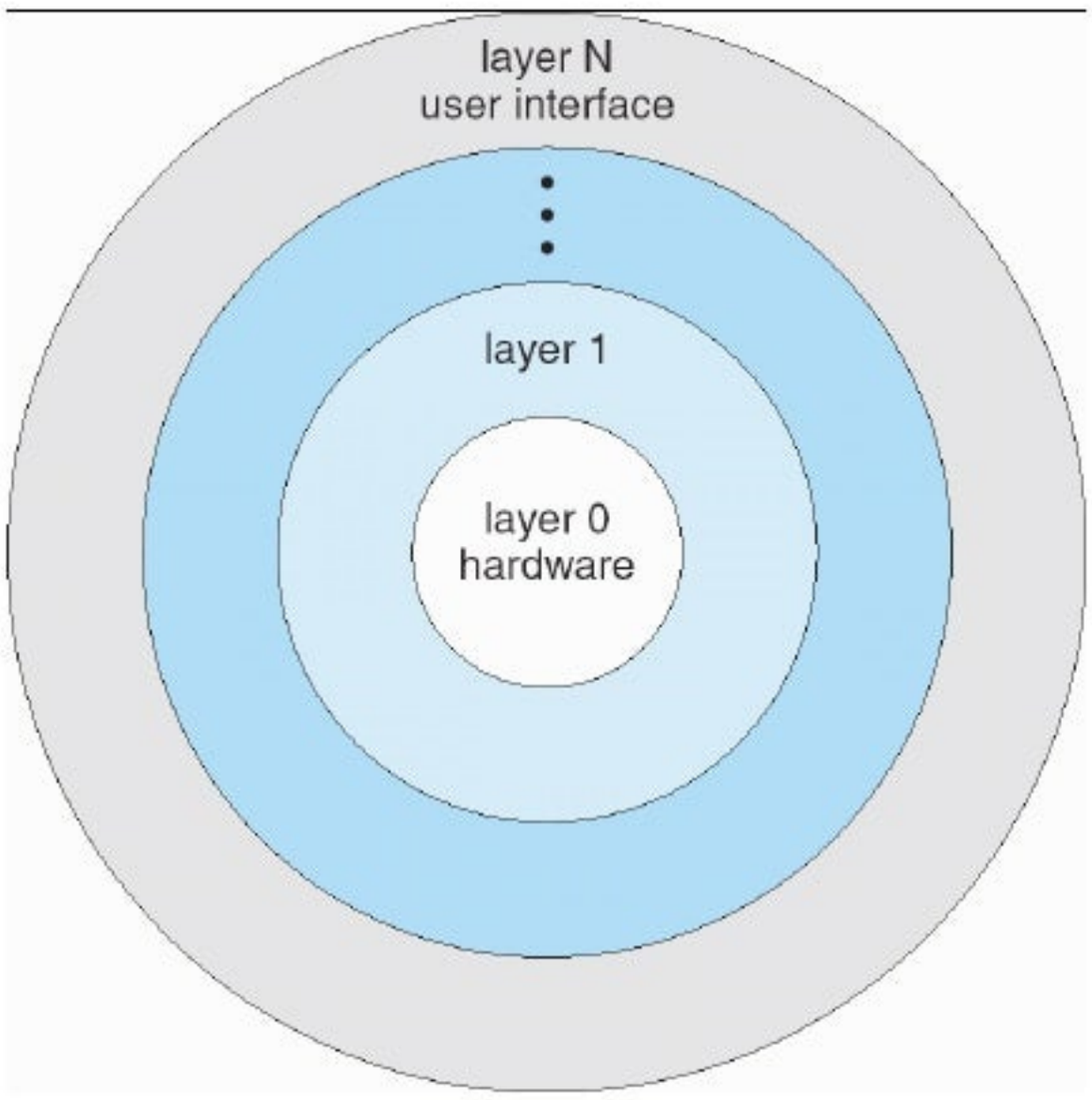
Operating-System Structure

Simple Structure

- MS-DOS and UNIX started as small, simple, limited systems

Layered Approach

- The OS is broken into layers: lowest = hardware, highest = GUI
- A typical layer has routines that can be invoked by higher ones
- Advantage: modularity (which simplifies debugging)
- A layer doesn't need to know how lower-level layer operations are implemented, only what they do
- Problems:
    - Layers can use only lower ones so they must be well defined
    - Layered implementations are less efficient than other types
- Nowadays fewer layers with more functionality are being designed

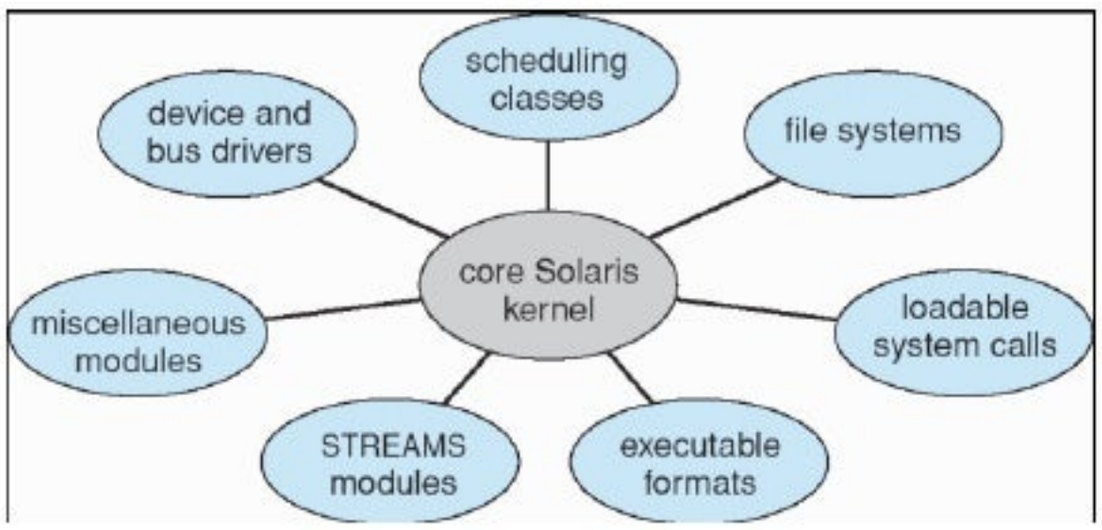


Microkernels

- Microkernel approach: all nonessential components are removed from the kernel and are implemented as system & user programs
- The smaller kernel provides minimal process & memory management
- Advantages:
    - Ease of extending the OS (new services are added to the user space and don't modify the kernel)
    - The OS is easier to port from 1 hardware design to another
    - More reliability: a failed user service won't affect the OS
- Main function of the microkernel: to provide a communication facility between the client program and the various services
- E.g. If the client program wants to access a file, it must interact with the file server indirectly through the microkernel
- QNX is a real-time OS that is based upon the microkernel design
- Windows NT uses a hybrid structure

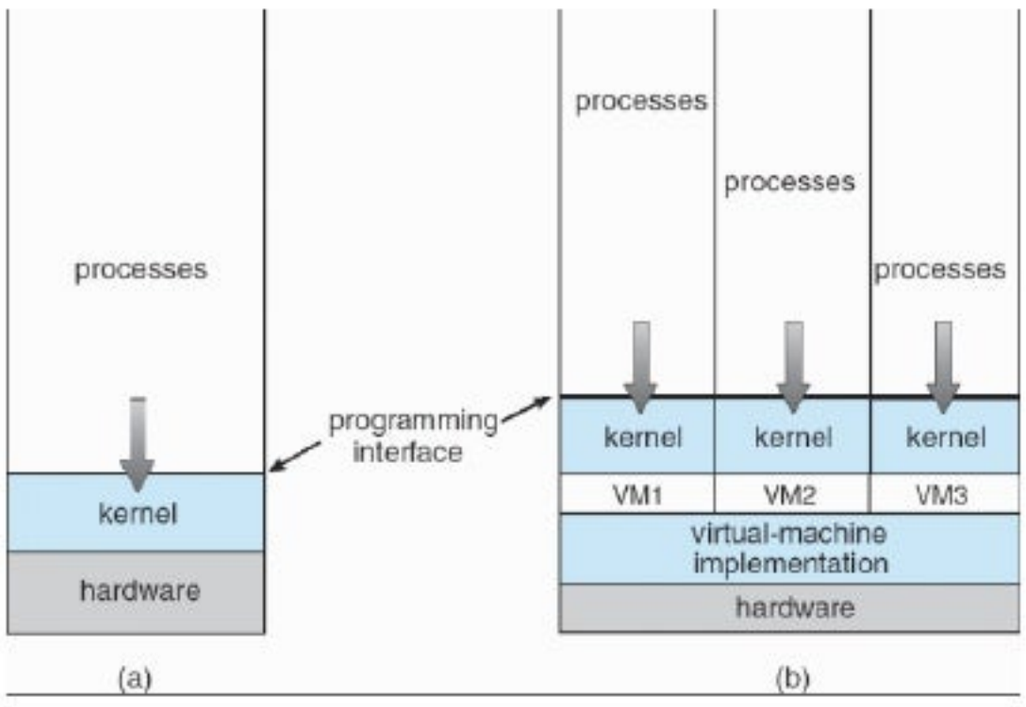
Modules

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



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
- 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



Non-virtual Machine Virtual Machine

- 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

History

Benefits

Simulation

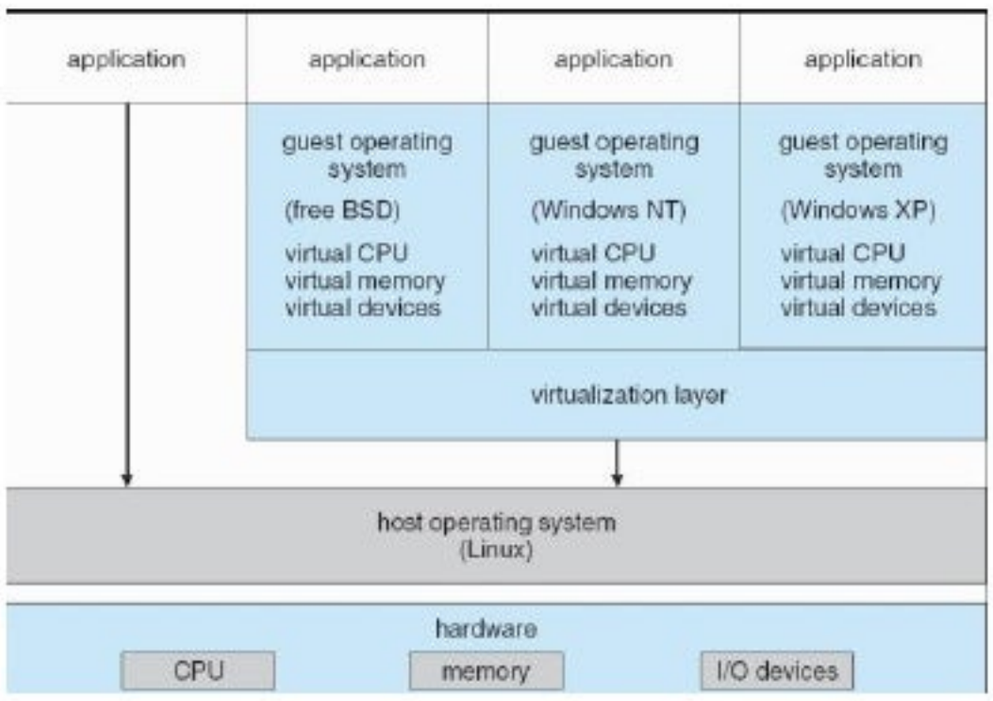
Para-virtualization

Implementation

Examples

VMware

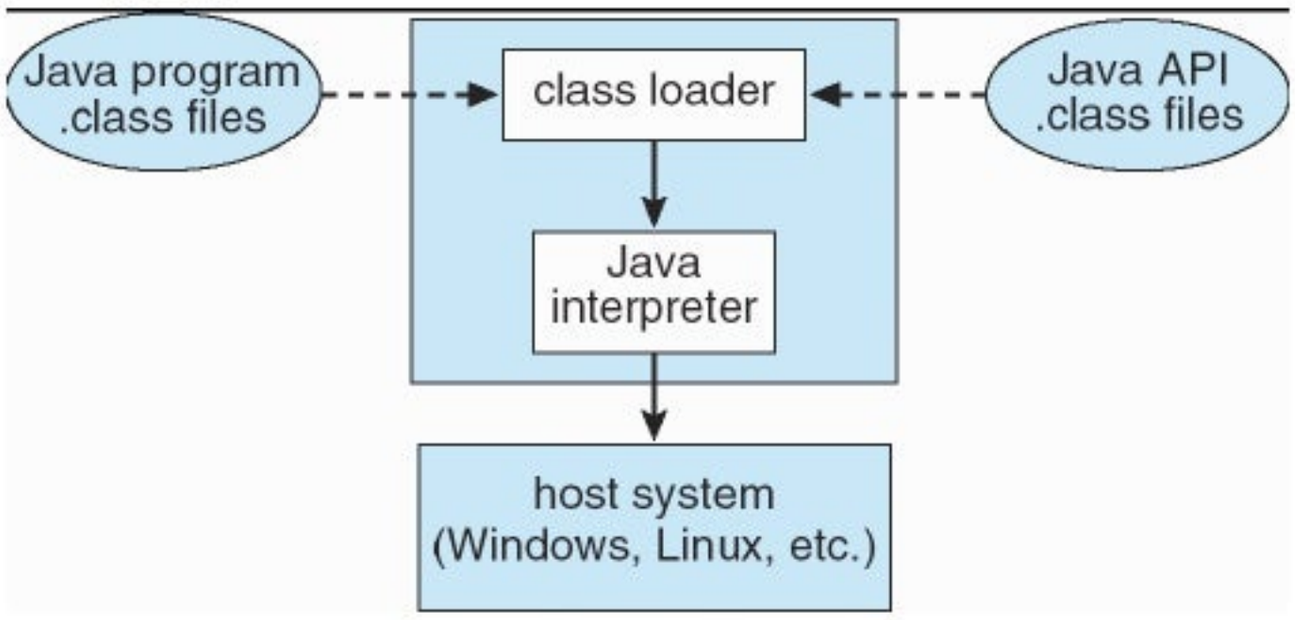
- VMware Architecture



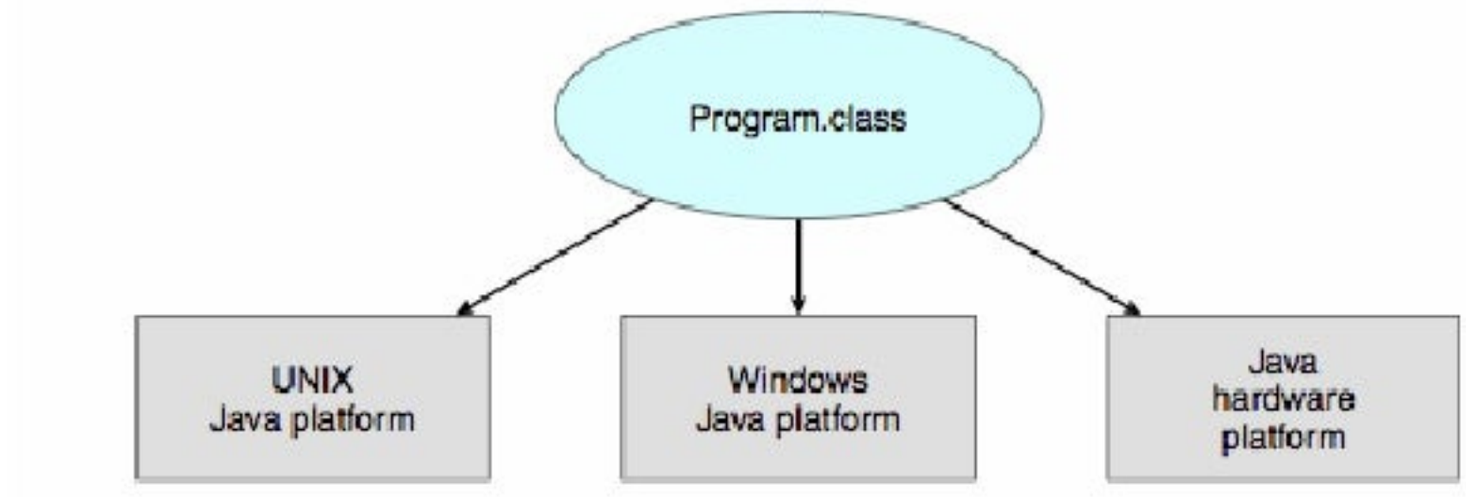
The Java Virtual Machine

Java consists of:

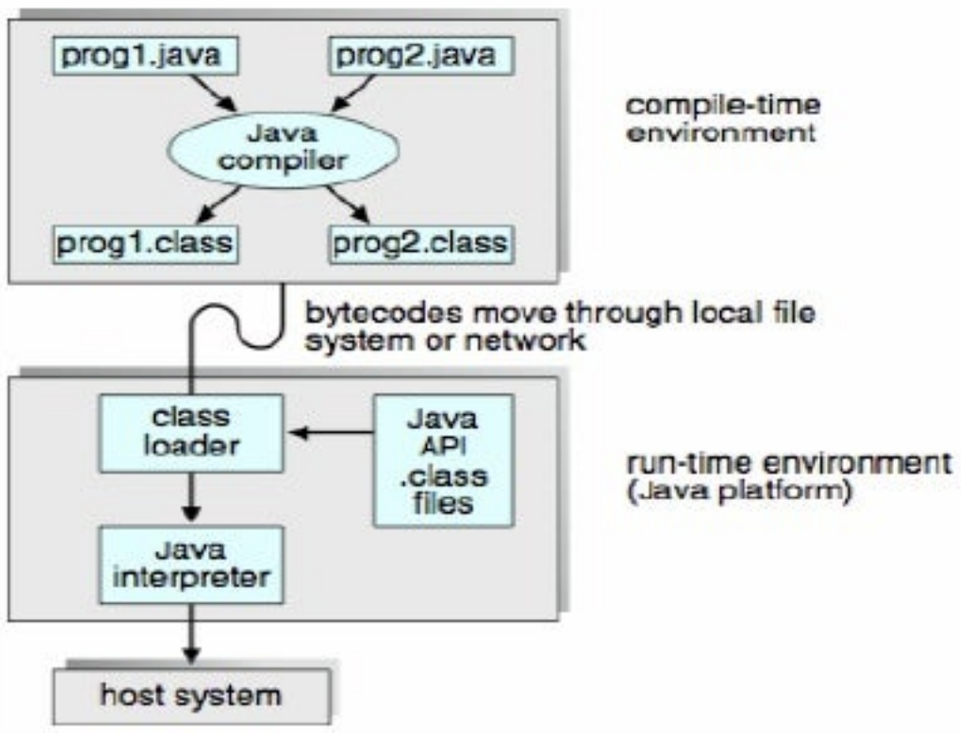
- Programming language specification
- Application programming interface (API)
- Virtual machine specification



- Java portability across platforms



- Java Development Environment



Operating-System Debugging

Failure Analysis

Performance Tuning

DTrace

Operating-System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN = configuring a system for each specific computer site
- The SYSGEN program must determine (from a file / operator):
    1. What CPU will be used
    2. How will boot disk be formatted
    3. How much memory is available
    4. What devices are available
    5. What OS options are desired
- A system administrator can use the above info to modify a copy of the source code of the OS
- The system description can cause the creation of tables and the selection of modules from a pre-compiled library. These modules are linked together to form the generated OS
- A system that is completely table driven can be constructed, which is how most modern OS's are constructed

System Boot

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

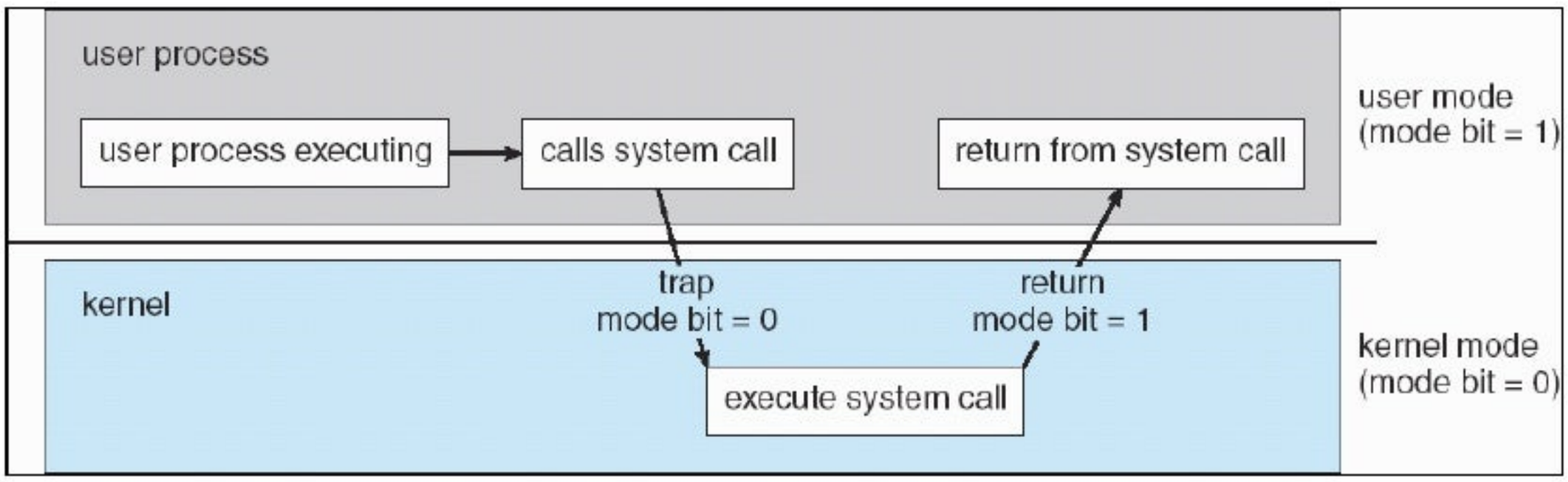
Summary

Operating System Operations

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

Dual-Mode Operation

- The OS and other programs & their data must be protected from any malfunctioning program
- You need two separate modes of operation: user & monitor mode
- A mode bit is added to the hardware to indicate the current mode: monitor: 0 (task executed on behalf of the OS) or user: 1 (task executed on behalf of the user)
- At system boot time, the hardware starts in monitor mode
- The OS loads and starts user processes in user mode
- When a trap / interrupt occurs, it switches to monitor mode
- Dual mode protects the OS from errant users, and errant users from one another
- This protection is accomplished by designating some machine instructions that may cause harm as privileged instructions
- Privileged instructions can only be executed in monitor mode
- If an attempt is made to execute a privileged instruction in user mode, the hardware traps it to the OS
- System call = a request by the user executing a privileged instruction, to ask the OS to do tasks that only it should do



Timer

- timer ensures that control is always returned to the OS, and prevents user programs from getting stuck in infinite loops
- The timer can be set to interrupt the computer after a while
- A variable timer has a fixed-rate clock and a counter
- The OS sets the counter, which decrements when the clock ticks
- When the counter reaches 0, an interrupt occurs
- The timer can be used to:
    - prevent a program from running too long
    - compute the current time
    - implement time sharing
- Timer to prevent infinite loop / process hogging resources
    - Set interrupt after specific period
    - Operating system decrements counter
    - When counter zero generate an interrupt
    - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Process Management

- A process needs resources (CPU, memory, files…) to do a task
- These resources are either given to the process when it is created, or allocated to it while it is running
- A **program** is a *passive* entity
- A **process** is an *active* entity, with a program counter giving the next instruction to execute
- The execution of a process must be sequential
- Process termination requires reclaim of any reusable resources
- **Single-threaded** process has **one program counter** specifying location of next instruction to execute
    - Process executes instructions sequentially, one at a time, until completion
- **Multi-threaded** process has **one program counter per thread** specifying location of next instruction to execute in each thread
- Typically system has many processes, some user, some operating system (kernel) running concurrently on one or more CPUs

        - Concurrency by multiplexing the CPUs among the processes / threads
- Processes can execute concurrently by multiplexing the CPU
- In connection with process management, the OS is responsible for
    - Scheduling processes and threads on the CPUs
    - Creating and deleting both user & system processes
    - Suspending and resuming processes
    - Providing mechanisms for process synchronization
    - Providing mechanisms for process communication
    - Providing mechanisms for deadlock handling

PART TWO: PROCESS MANAGEMENT

- A process can be thought as a program in execution.
    - A process will need resources - such as CPU time, memory, files, and I/O devices - to accomplish its task.
    - These resources are allocated to the process either when it is created or while it is executed.
- A process is the unit of work in most systems.
- Systems consist of a collection of processes:
    - Operating-system processes execute system code
    - User processes execute user code
- All these processes may execute concurrently.
- Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.
- The operating system is responsible for the following activities in connection with process and thread management:
    - The creation and deletion of both user and system processes;
    - The scheduling of processes;
    - and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Chapter 3: Process Concept

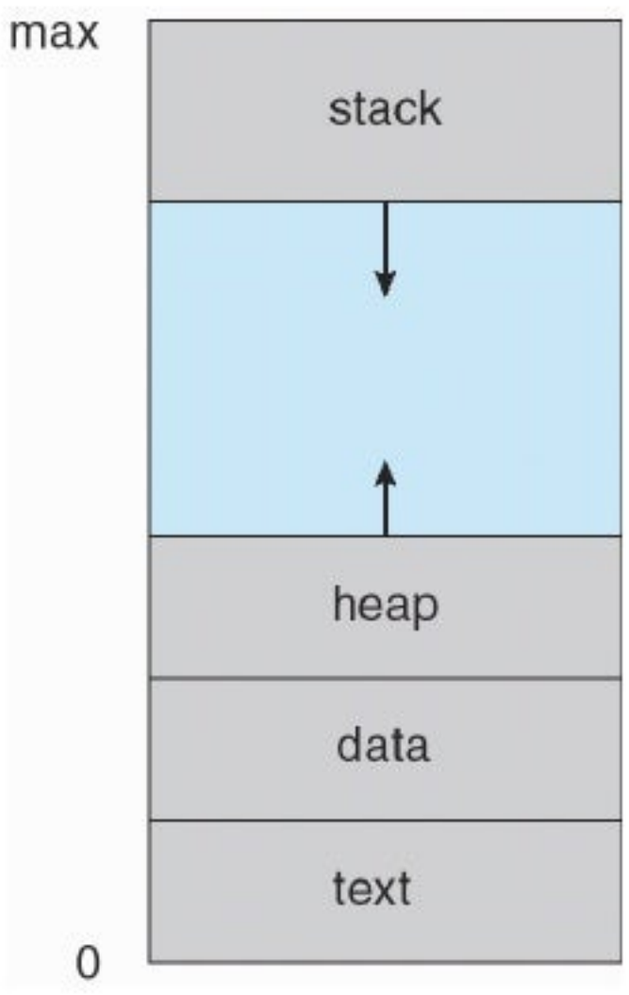
- **Objectives**:
    - To introduce the notion of a process - a program in execution, which forms the basis of all computation.
    - To describe the various features of processes, including scheduling, creation and termination, and communication.
    - To describe communication in client-server systems.

Process Concepts

- 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

The Process

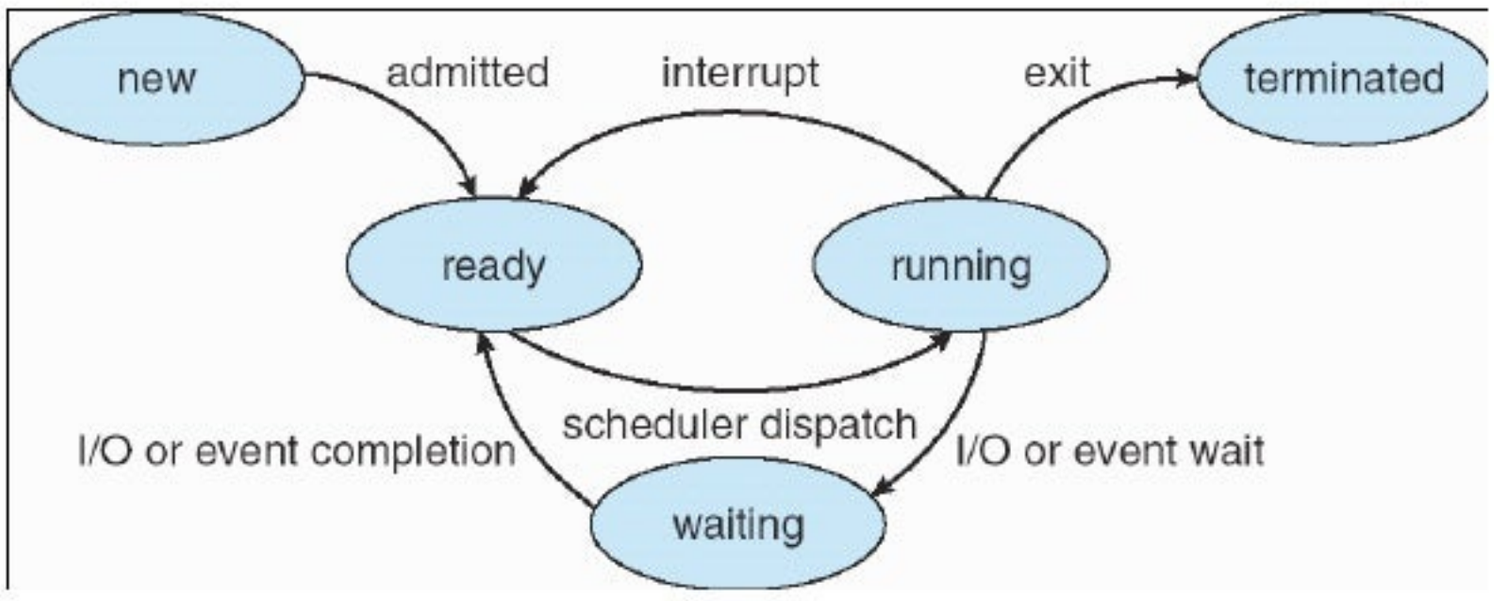
- Process = an active entity, with a program counter (to indicate the current activity), process stack (with temporary data), and a data section (with global variables)
- Text section = the program code
- If you run many copies of a program, each is a separate process (The text sections are equivalent, but the data sections vary)
- **Process**–a program in execution; process execution must progress in sequential fashion
- A process includes:
    - program counter
    - stack
    - data section
- A process in memory



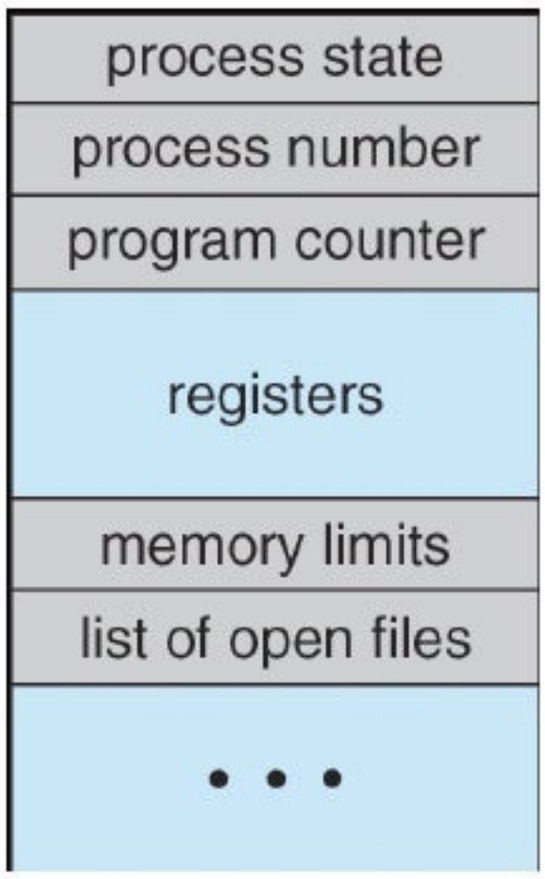
- p.102 give description of stack, heap, data and text areas

Process State

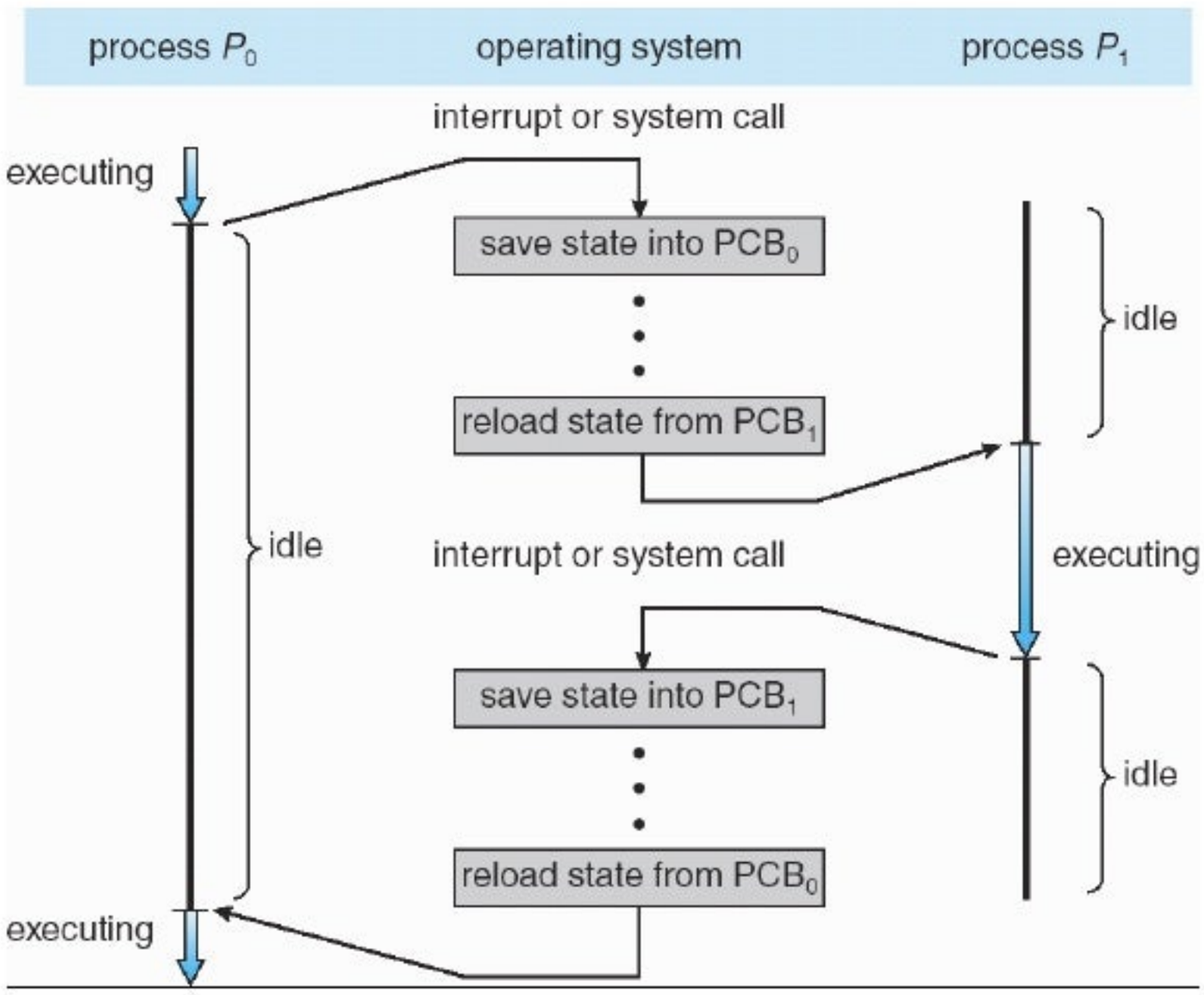
- Each process may be in one of the following states:
    - New (Process is being created)
    - Running (Instructions are being executed)
    - Waiting (Process is waiting for an event, e.g. I/O)
    - Ready (Process is waiting to be assigned to a processor)
    - Terminated (Process has finished execution)
- Only one process can be *running* on any processor at any instant



Process Control Block



- Contains information associated with a specific process:
    1. Process state (as above)
    2. Program counter (indicating the next instruction's address)
    3. CPU registers (Info must be saved when an interrupt occurs)



- CPU-scheduling info (includes process priority, pointers…)
- Memory-management info (includes value of base registers…)
- Accounting info (includes amount of CPU time used…)
- I/O status info (includes a list of I/O devices allocated…)

Threads

- Many OS's allow processes to perform more than one task at a time

Process Scheduling

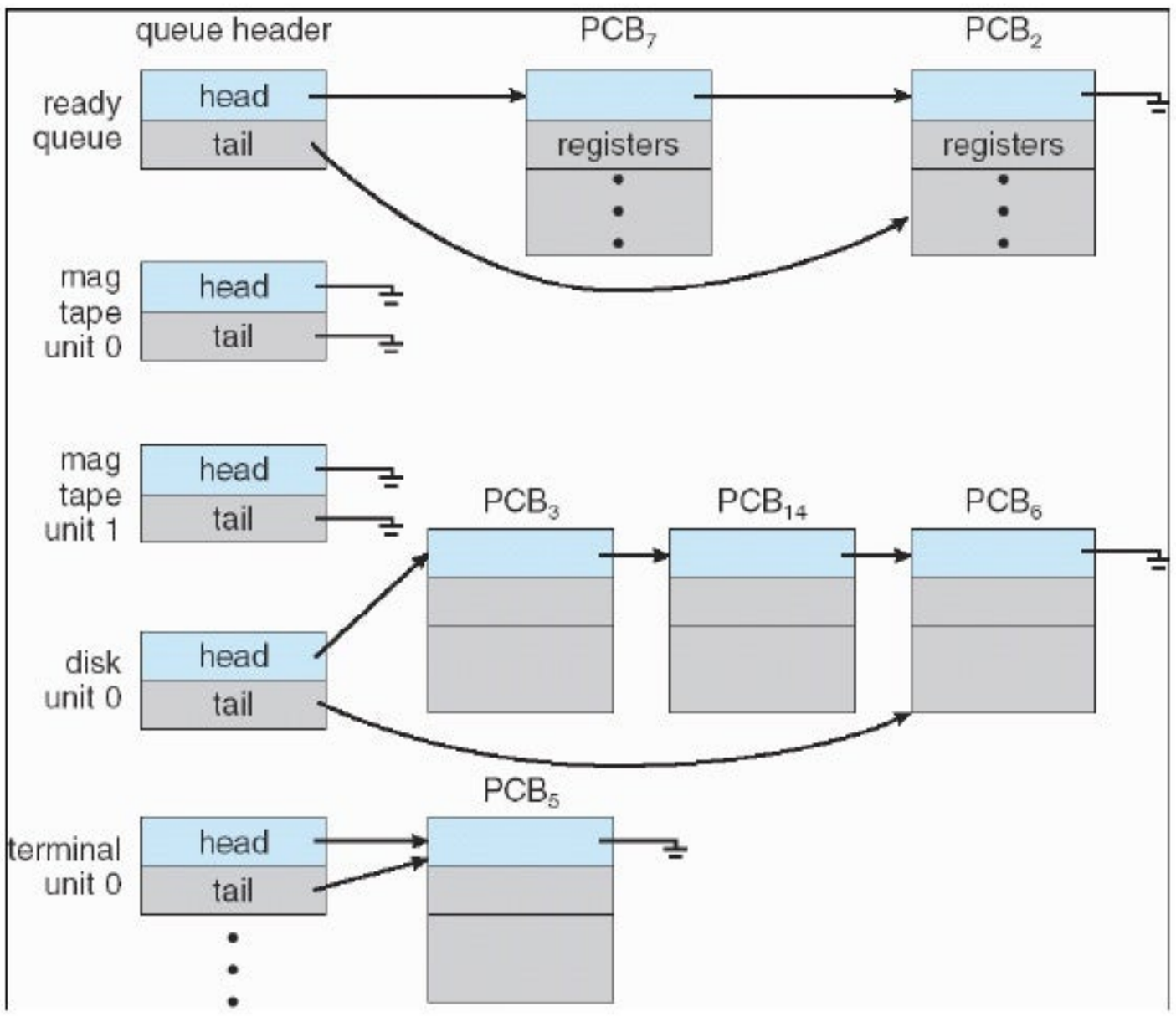
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization
    - The objective of time sharing is to switch the CPU among processes so frequently that the user can interact with each program while it is running
    - To meet this objectives, the process scheduler selects an available process for execution on the CPU
    - For single-processor system, there will never be more than one running process
    - If more than one process, it will have to wait until CPU is free and can be rescheduled

Scheduling Queues

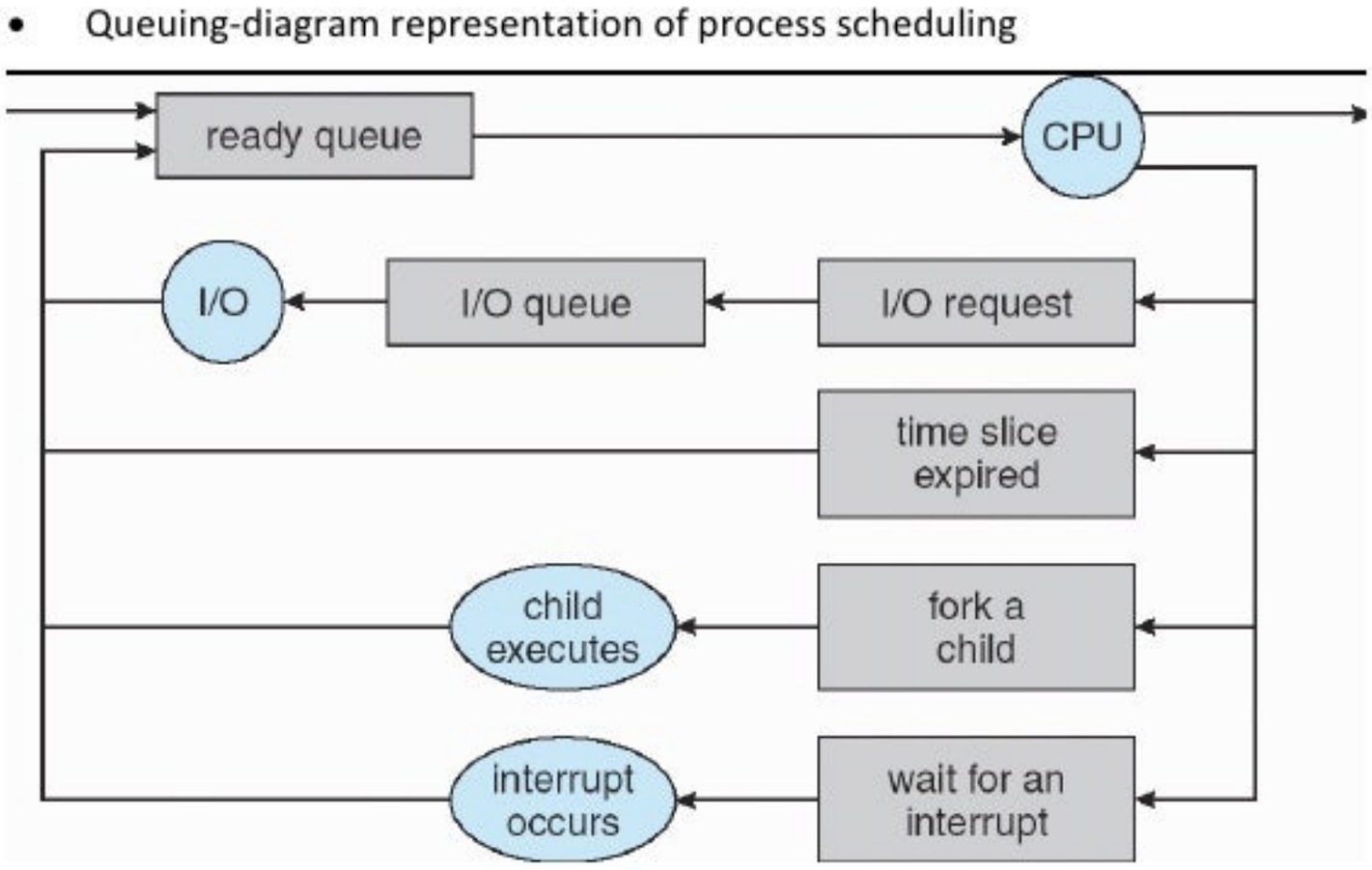
- As processes enter the system, they are put into a **job queue**
- Processes in memory, waiting to execute, are in the **ready queue**
- A ready queue header contains pointers to the fist & last PCBs in the list, each of which has a pointer to the next PCB
- **Device queue** = the list of processes waiting for an I/O device
- After a process in the ready queue is selected for execution…
    - it could issue an I/O request and be put in the I/O queue

- it could create a sub-process and wait for its termination
- it could be interrupted and go to the ready queue

- Processes migrate among the various queues



- Queuing-diagram representation of process scheduling



Schedulers

- A process migrates between the various scheduling queues throughout its lifetime
- The appropriate scheduler selects processes from these queues
- In a batch system, more processes are submitted than can be executed immediately
    - These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution
- The **long-term scheduler / job scheduler** selects processes from this pool and loads them into memory for execution

- The **short-term scheduler** / **CPU scheduler** selects from among the processes that are ready to execute, and allocates the CPU to it
- The main **difference** between these two schedulers is the **frequency of execution** (short-term = more frequent)
- The degree of multiprogramming (= the number of processes in memory) is controlled by the long-term scheduler
- 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
- The long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes for good performance
- Some time-sharing systems have a **medium-term scheduler**:
  - It **removes processes from memory** and thus **reduces the degree of multiprogramming**
  - Later, the process can be reintroduced into memory and its execution can be continued where it left off (= Swapping)
  - Swapping may be necessary to improve the process mix, or because memory needs to be freed up

Context Switch

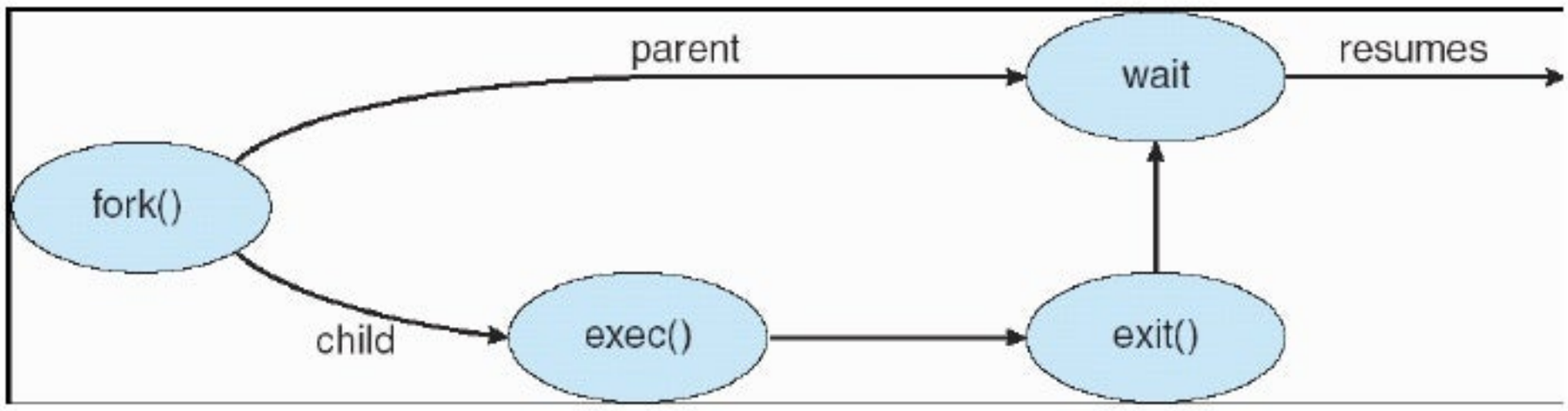
- Context switch = saving the state of the old process and switching the CPU to another process
- The context of a process is represented in the PCB of a process
- (It includes the value of the CPU registers, process state, and memory-management information)
- Context-switch time is pure overhead, because the system does no useful work while switching
- Context-switch time is highly dependent on hardware support (e.g. some processors provide multiple sets of registers)

Operations on Processes

Process Creation

- Parent process = the creating process
- Children = new processes created by parent ones
- Sub-processes may…
  - get resources directly from the OS
  - be constrained to a subset of the parent's resources (This prevents sub-processes from overloading the system)
- When child processes are created, they may obtain initialization data from the parent process (in addition to resources)
- Execution possibilities when a process creates a new one:
  - The parent continues to execute concurrently with children
  - The parent waits until some / all children have terminated
- Address space possibilities when a process creates a new one:
  - The child process is a duplicate of the parent
  - The child process has a program loaded into it
- **UNIX example**
  - **fork** system call creates new process

- **exec** system call used after a **fork** to replace the process' memory space with a new program




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

int main()
{
pid_t pid;

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

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

- **Windows example**

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

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

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

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

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

- **Java example**

```java
import java.io.*;

public class OSProcess
{
 public static void main(String[] args) throws IOException {
  if (args.length != 1) {
   System.err.println("Usage: java OSProcess <command>");
   System.exit(0);
  }

  // args[0] is the command
  ProcessBuilder pb = new ProcessBuilder(args[0]);
  Process proc = pb.start();

  // obtain the input stream
  InputStream is = proc.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);

  // read what is returned by the command
  String line;
  while ( (line = br.readLine()) != null)
    System.out.println(line);

  br.close();
 }
}
```

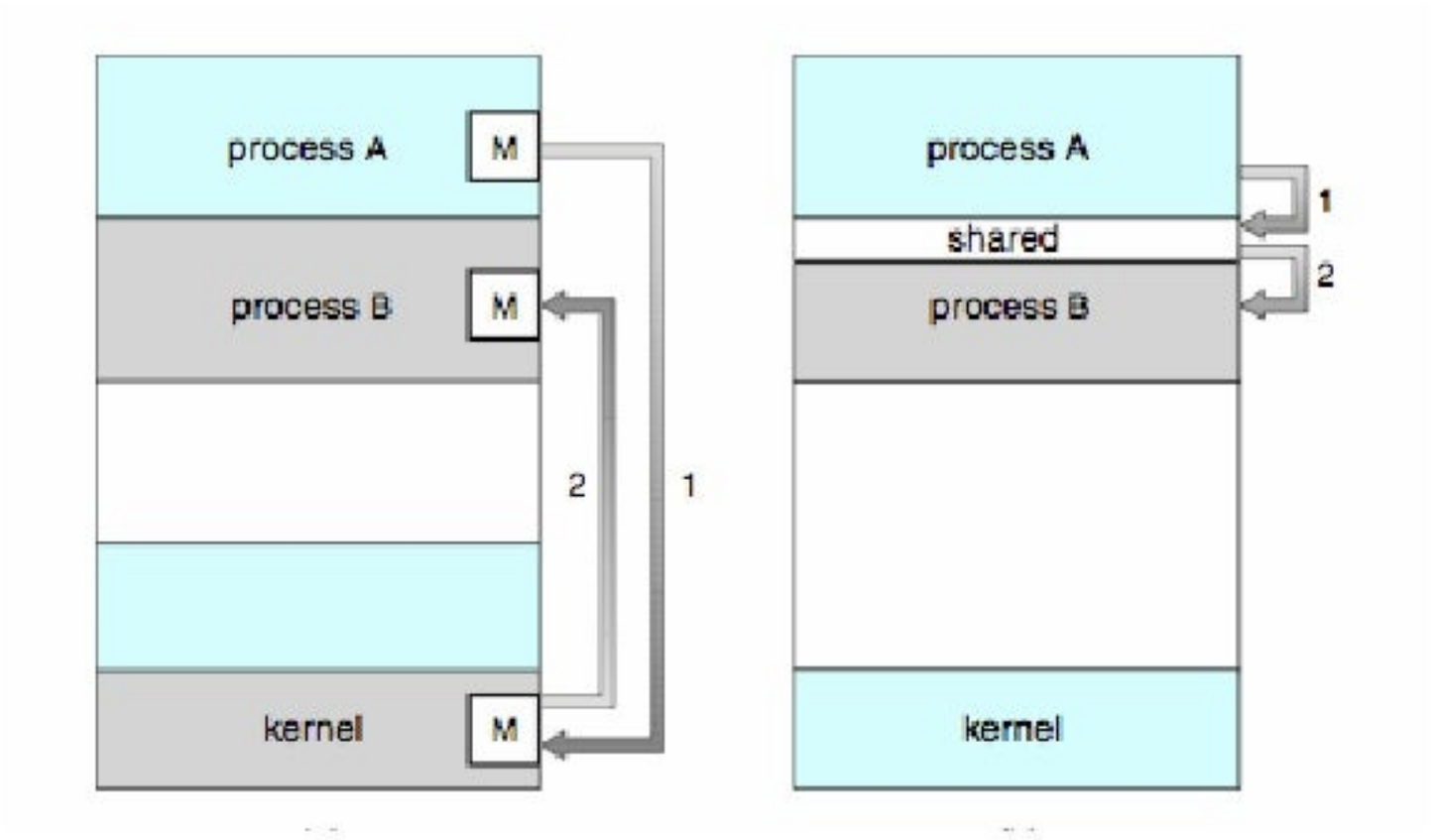
Process Termination

- A process terminates after it executes its final statement
- At that point the process may return data to its parent process
- All the process' resources (memory, open files, I/O buffers) are de-allocated by the OS
- A parent process can cause its child processes to terminate
- Parents therefore need to know the identities of their children
- Reasons why a parent may terminate execution of children:
    - If the child exceeds its usage of some resources
    - If the task assigned to the child is no longer required
    - If the parent is exiting, and the OS won't allow a child to continue if its parent terminates (Cascading termination)

Interprocess Communication (IPC)

- Independent process: can't affect / be affected by the other processes (E.g. processes that don't share data with other ones)
- Cooperating process: can affect / be affected by the other processes (E.g. processes that share data with other ones)
- Reasons for providing an environment that allows cooperation:
    - Information sharing: Several users may be interested in the same file
    - Computation speedup: A task can be broken into subtasks to run faster
    - Modularity: Functions can be divided into separate processes
    - Convenience: An individual user may want to work on many tasks
- There are two fundamental models of interprocess communication:
    - Shared memory
    - message passing

Message Passing **Shared Memory**



Shared-Memory Systems

- With a shared memory environment, processes share a common buffer pool, and the code for implementing the buffer must be written explicitly by the application programmer

- **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

Message-Passing Systems

- The function of a **message system** is to allow processes to communicate with one another without resorting to shared data
- Messages sent by a process can be of a fixed / variable size:
    - Fixed size:
        - Straightforward system-level implementation
        - Programming task is more difficult
    - Variable size:
        - Complex system-level implementation
        - Programming task is simpler
- A communication link must exist between processes to communicate
- Methods for logically implementing a link:
    - Direct or indirect communication
    - Symmetric or asymmetric communication
    - Automatic or explicit buffering
- Message passing 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)

Naming

Direct communication	**Indirect communication**
• Each process must explicitly name the recipient / sender	• Messages are sent to / received from mailboxes (ports)

Properties of a communication link: • A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate • A link is associated with exactly two processes • Exactly one link exists between each pair of processes	Properties of a communication link: • A link is established between a pair of processes only if both members have a shared mailbox • A link may be associated with more than two processes • A number of different links may exist between each pair of communicating processes
Symmetric addressing: • Both sender and receiver processes must name the other to communicate	Mailbox owned by a process: • The owner can only receive, and the user can only send • The mailbox disappears when its owner process terminates
Asymmetric addressing: • Only the sender names the recipient; the recipient needn't name the sender	Mailbox owned by the OS: • The OS must provide a mechanism that allows a process to: * Create a new mailbox * Send & receive messages via it * Delete a mailbox

Synchronization

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

Synchronous message passing (blocking)	**Asynchronous passing (non-blocking)**
Blocking send: • The sending process is blocked until the message is received by the receiving process or by the mailbox.	*Non-blocking send:* • The sending process sends the message and resumes operation.
Blocking receive:	*Non-blocking receive:*

• The receiver blocks until a message is available.	• The receiver retrieves either a valid message or a null.

- Different combinations of send and receive are possible
- Rendezvous = when both the send and receive are blocking
- Look at NB!!! p.122 TB

Buffering

- Messages exchanged by processes reside in a temporary queue
- Such a queue can be implemented in three ways:
  - **Zero capacity**
    - The queue has maximum length 0, so the link can't have any messages waiting in it
    - The sender must block until the recipient receives the message
  - **Bounded capacity**
    - The queue has finite length n (i.e. max n messages)
    - If the queue is not full when a new message is sent, it is placed in the queue
    - If the link is full, the sender must block until space is available in the queue
  - **Unbounded capacity**
    - The queue has potentially infinite length
    - Any number of messages can wait in it
    - The sender never blocks

Examples of IPC Systems

An Example: POSIX Shared Memory

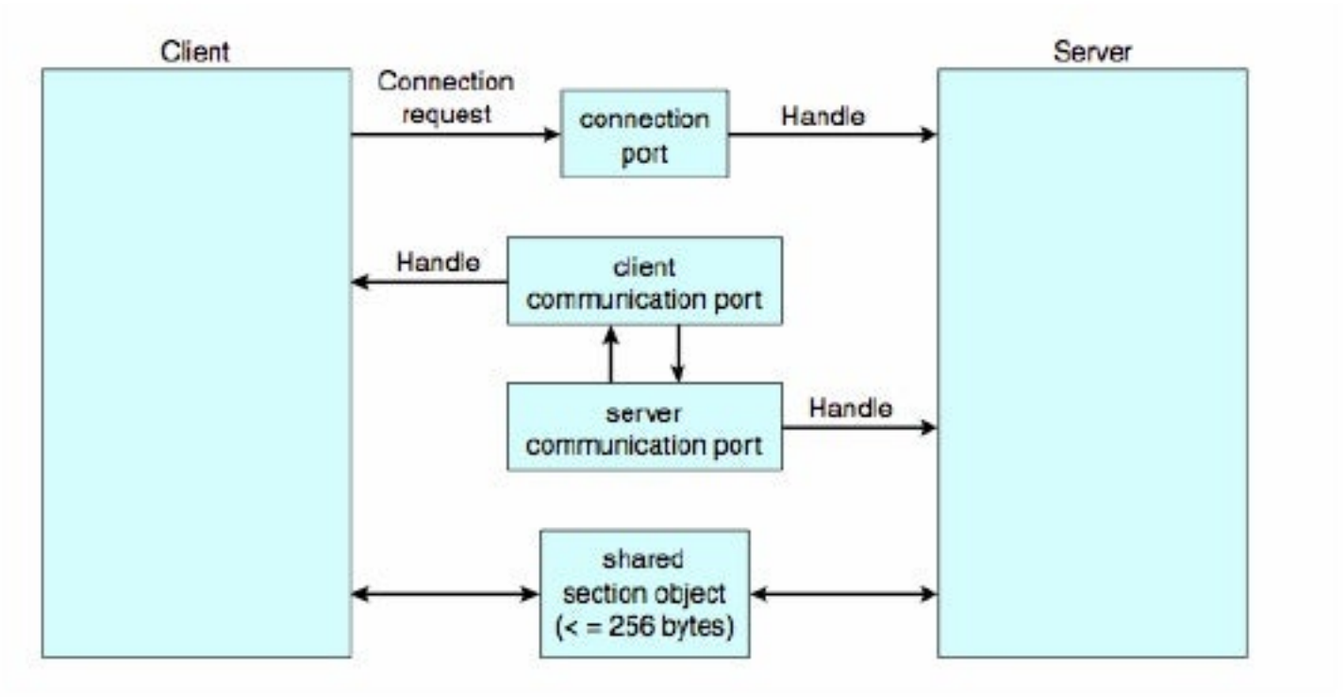
- p.123 - 124

An Example: Mach

- p.124 - 126

An Example: Windows XP

- p.127 - 128

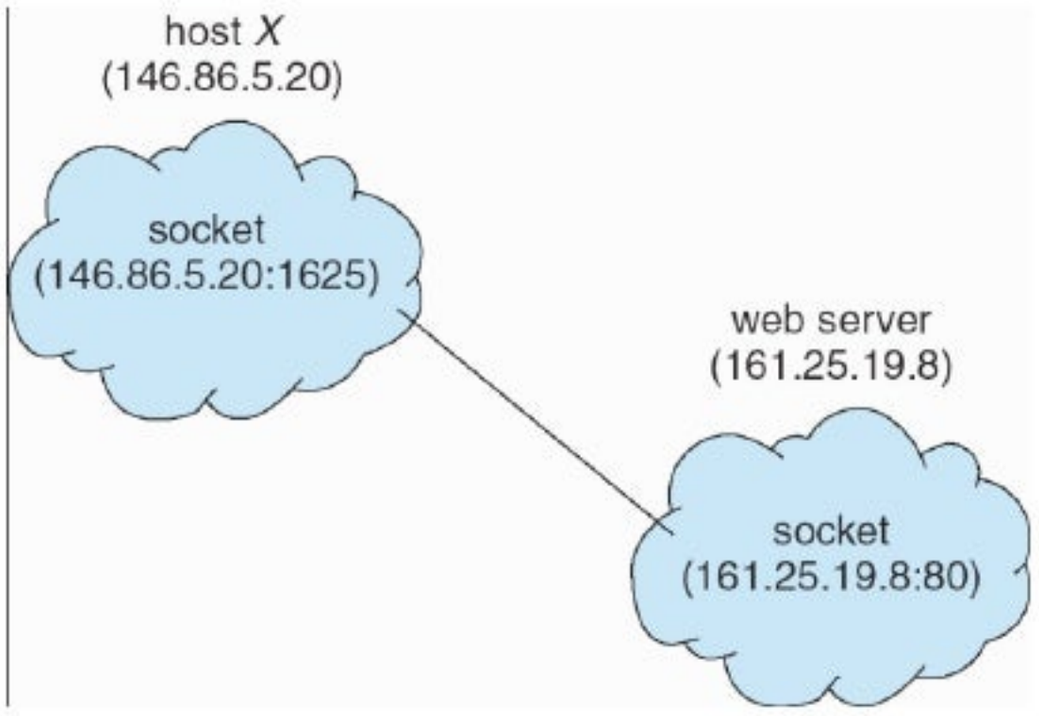


Communication in Client-Server Systems

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

Sockets

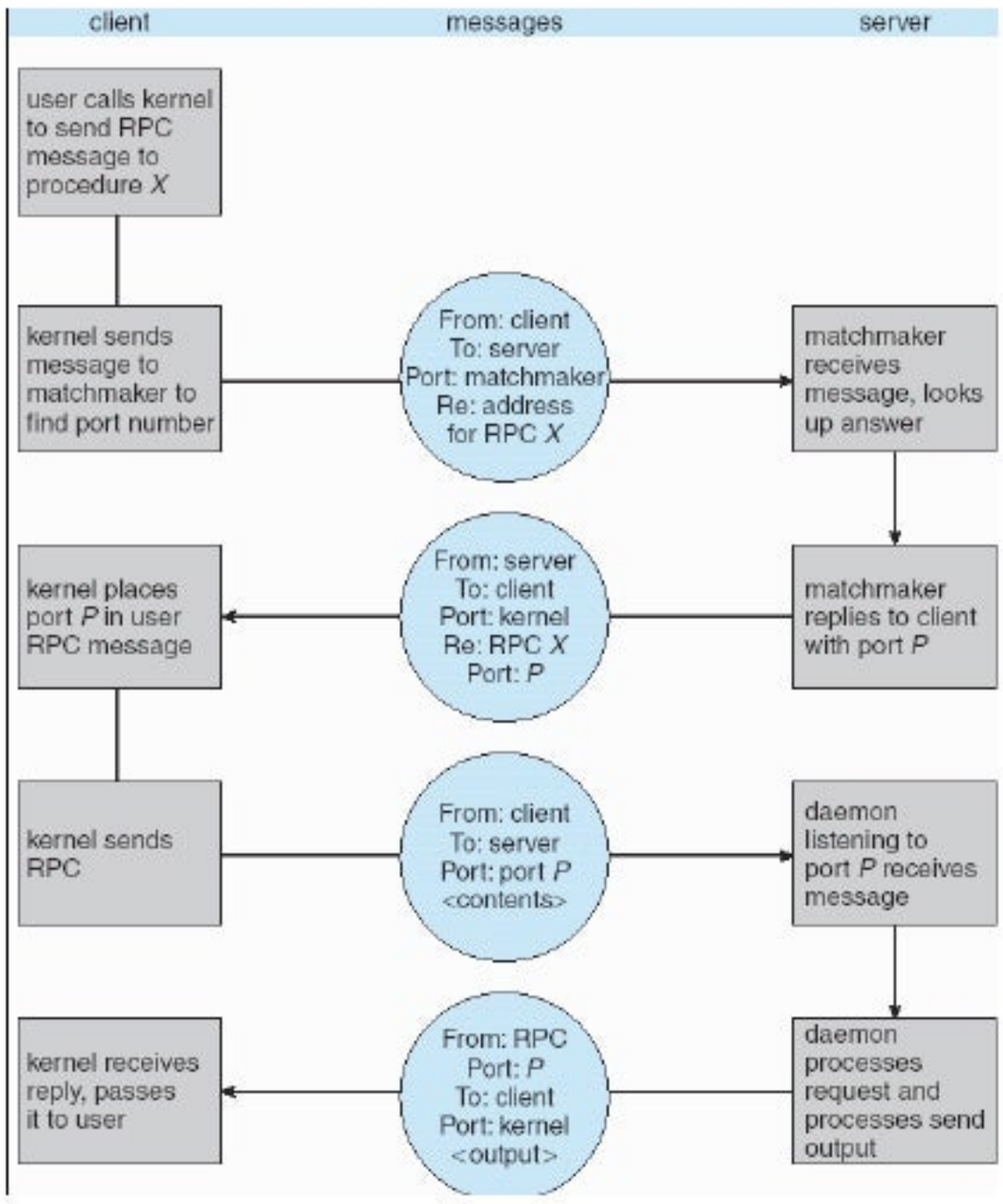
- **Socket** = an endpoint for communication
- A pair of processes communicating over a network employs a pair of sockets - one for each process
- A socket is identified by an IP address together with a port no
    - The socket **161.25.19.8:1625**refers to port **1625**on host **161.25.19.8**
- In general, sockets use a client-server architecture
- The server waits for incoming requests by listening to a port
- Once a request is received, the server accepts a connection from the client socket to complete the connection
- Servers implementing specific services (like telnet, ftp, http) listen to well-known ports (below 1024)
- When a client process initiates a request for a connection, it is assigned a port by the host computer (a no greater than 1024)
- The connection consists of a unique pair of sockets
- Communication using sockets is considered low-level
- RPCs and RMI are higher-level methods of communication

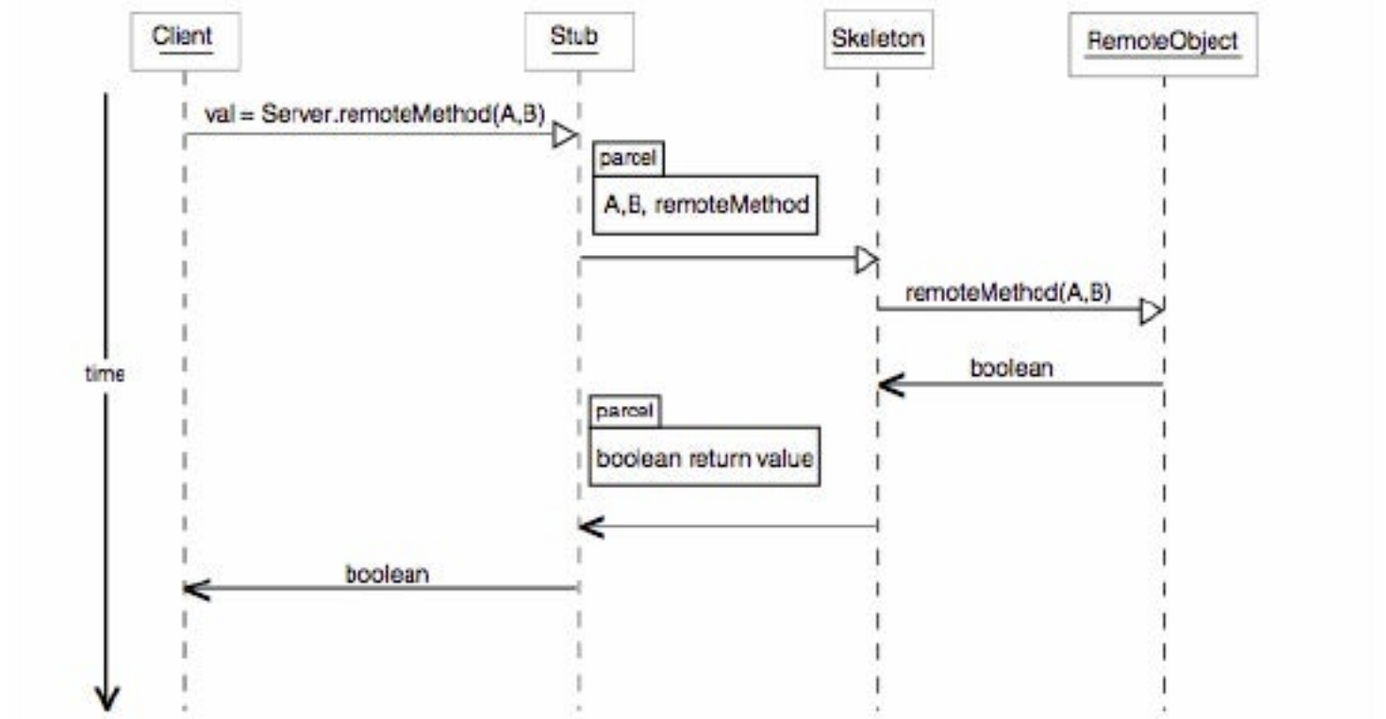


Remote Procedure Calls

- Messages exchanged for RPC communication are well structured
- They are addressed to an RPC daemon listening to a port on the remote system, and contain an identifier of the function to execute and the parameters to pass to that function
- The function is executed and any output is sent back to the requester in a separate message
- A port is a number included at the start of a message packet
- A system can have many ports within one network address
- If a remote process needs a service, it addresses its messages to the proper port
- The RPC system provides a **stub (client-side proxy for actual procedure)** on the client side, to hide the details of how the communication takes place

    - When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided
    - This stub locates the port on the server and marshals (=packs the parameters into a form for the network) the parameters
    - The stub then transmits a message to the server using message passing
    - A similar stub on the server side receives this message and invokes the procedure on the server
    - If necessary, return values are passed back to the client
- Many RPC systems define a machine-independent representation of data (because systems could be big-endian / little-endian)
- External data representation (XDR) is one such representation:
    - On the client side, parameter marshalling involves converting the machine-dependent data into XDR before going to the server
    - On the server side, the XDR data is unmarshalled and converted into the machine-dependent representation for the server.
- Two approaches for binding client & server:
    - The binding information may be predetermined, in the form of fixed port addresses
    - Binding can be done dynamically by a rendezvous mechanism (also called a matchmaker daemon)





Pipes

- A pipe act as a conduit allowing two processes to communicate
- In implementing a pipe four issues need to be considered:
    - Does the pipe allow unidirectional communication or unidirectional communication?
    - If two-way communication is allowed, is it half or full duplex?
    - Must a relationship exist between the communicating processes? (parent-child concept)
    - Can pipes communicate over a network, or must the communicating processes reside on the same machine?

Ordinary Pipes

- Allow communication between parent and child process
    - Make use of producer-consumer concept
    - Producer writes to write end of the write-end of the pipe
    - Consumer reads from the read-end of the pipe
- Named anonymous pipes on Windows
- Ordinary pipes cease to exist as soon as processes terminate communication
- Unidirectional

Named Pipes

- More powerful than ordinary pipes
- Permit unrelated processes to communicate with one another
- Bidirectional, no parent child relationship needed

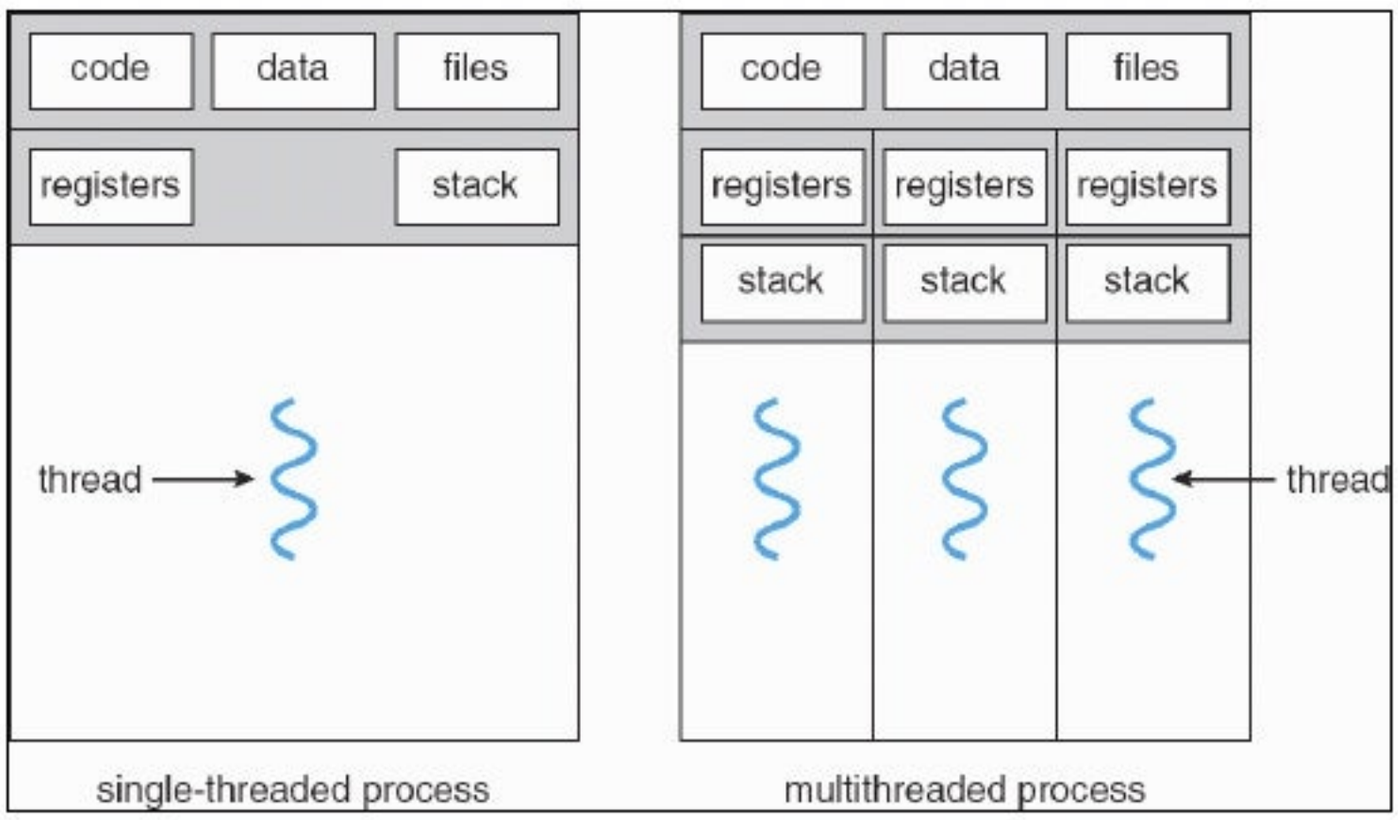
Summary

Chapter 4: Multithreaded Programming

- **Objectives**:
    - To introduce the notion of a thread - a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
    - To discuss the APIs for the Pthreads, Win32, and Java thread libraries.
    - To examine issues related to multithreaded programming.

Overview

- A thread is a **flow of control within** a **process**
- A **multithreaded process** contains several different flows of control within the same address space
- A traditional (heavyweight) process has one thread of control
- A thread / lightweight process (LWP) = a unit of **CPU utilization**
- It comprises a thread ID, program counter, register set, & stack
- It shares with other threads belonging to the same process its code section, data section, and other OS resources
- If a process has multiple threads of control, it can perform more than one task at a time
- Look at fig 4.1 p.153 TB



- User-level threads are threads that are visible to a programmer and are unknown to the kernel
- OS kernel supports and manages kernel-level threads

Motivation

- It is more efficient to have multithreading than many processes
- RPC servers are typically multithreaded
    - When a server receives a message, it services it with a separate thread
    - This lets the server service several concurrent requests

Benefits

- **Responsiveness**:
    - A program can continue running even if part of it is busy
- **Resource sharing**:
    - Threads share the memory and resources of their process
- **Economy**:
    - Allocating memory and resources for processes is costly (time)
- **Scalability**:
    - Utilization of multiprocessor architectures
    - Each thread runs on a separate CPU, increasing concurrency / parallelism

Multicore Programming

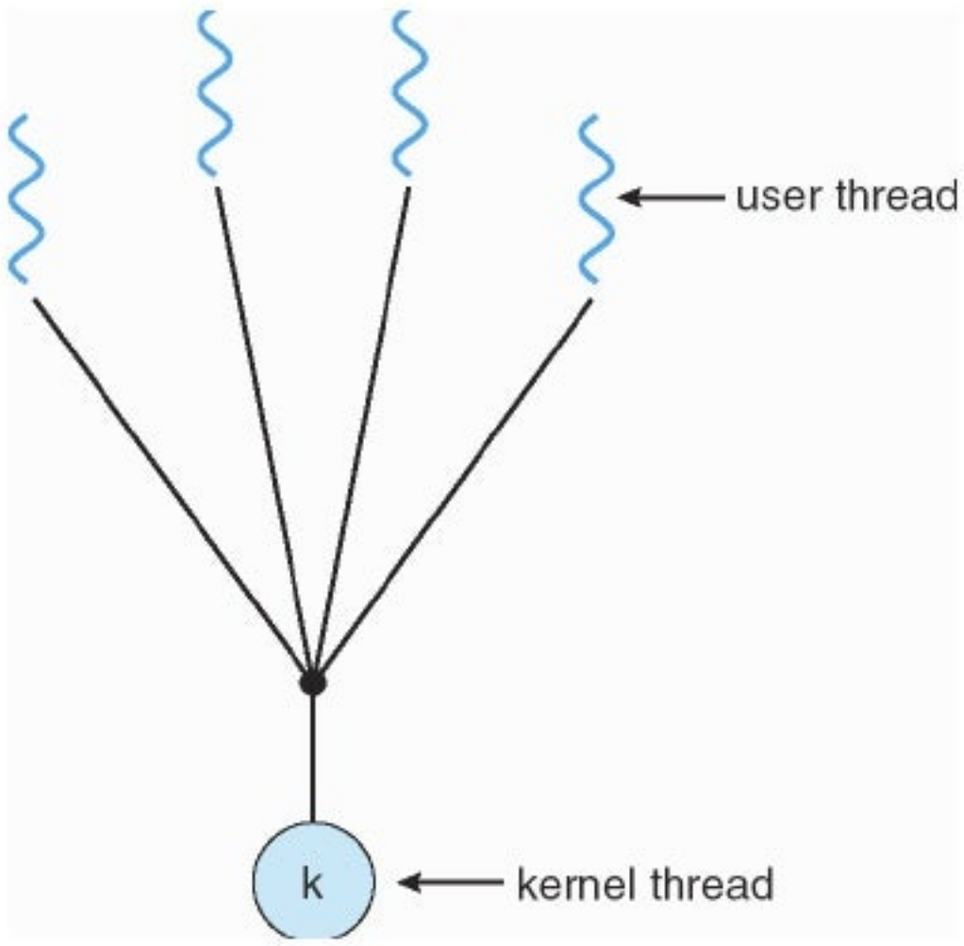
- p.156 - 157 TB
- Provides a mechanism for more efficient use of multiple cores and improved concurrency
- On a system with multiple cores the processes run concurrently since the system can assign a separate thread to each core
- Five areas that present challenges in programming multicore systems:
    - **Dividing activities:**
        - Areas of applications to be divided into different tasks
    - **Balance:**
        - Tasks must perform equal work of equal value, else CPU time is wasted
    - **Data splitting:**
        - Data accessed and manipulated must be divided to run on separate cores
    - **Data dependency**:
        - If data between cores depends on each other, execution must be synchronized
    - **Testing and debugging**:
        - More difficult to test and debug than single-threaded execution

Multithreading Models

User threads (Many-to-One)	**Kernel threads (One-to-One)**
Implemented by a thread library at the user level	Supported directly by the OS
The library provides support for thread creation, scheduling, and management with no support from the OS kernel	The kernel performs thread creation, scheduling, and management in kernel space
Faster to create & manage because the kernel is unaware of user threads and doesn't intervene	Slower to create & manage than user threads because thread management is done by the OS
Disadvantage: If the kernel is single- threaded, then any user-level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application	Since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. In a multiprocessor environment, the kernel can schedule threads on different processors.

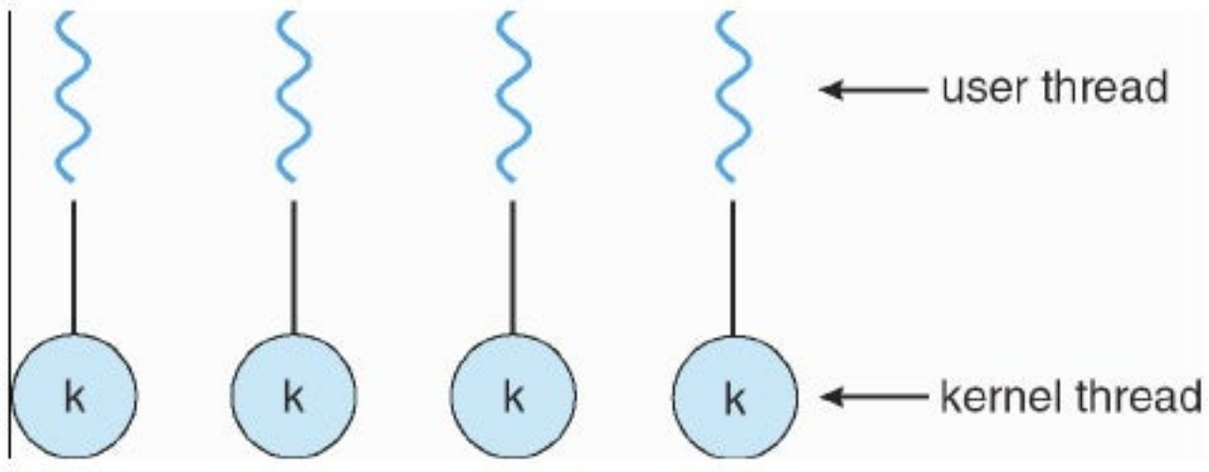
Many-to-One Model

- Many user-level threads are mapped to one kernel thread
- Thread management is done in user space, so it is **efficient**
- The entire process will block if a thread makes a blocking call
- Multiple threads can’t run in parallel on multiprocessors



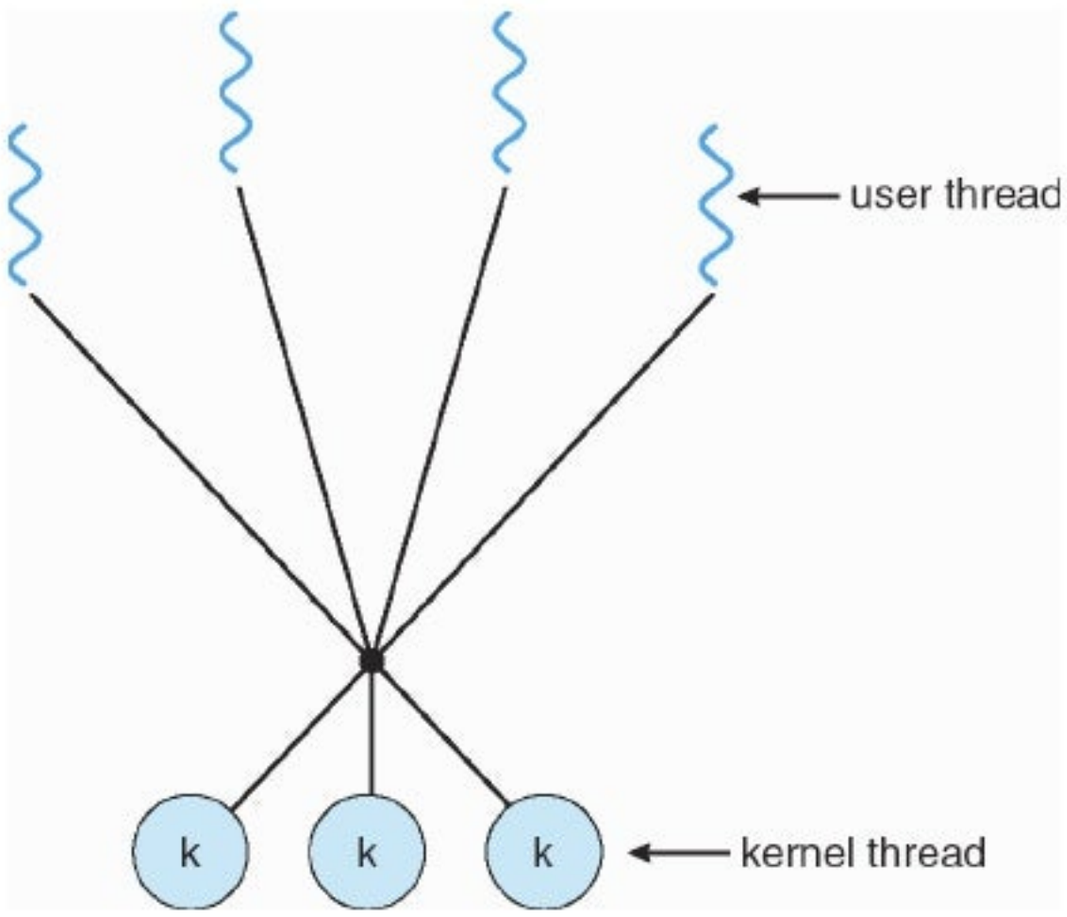
One-to-One Model

- Each user thread is mapped to a kernel thread
- **More concurrency** than the many-to-one model because another thread can run when a thread makes a blocking system call
- Multiple threads can run in parallel on multiprocessors
- Disadvantage: creating a user thread requires creating the corresponding kernel thread (This **overhead** burdens performance)

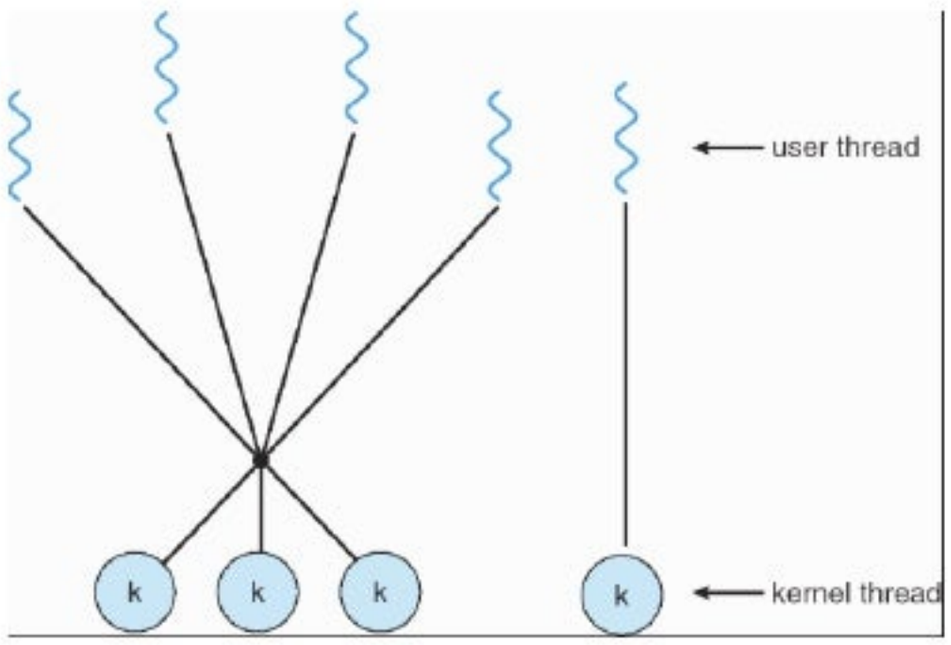


Many-to-Many Model

- Many user-level threads are multiplexed to a smaller / equal number of kernel threads
- Developers can create as many user threads as necessary
- The kernel threads can run in parallel on a multiprocessor
- When a thread performs a blocking system call, the kernel can schedule another thread for execution



- A variation on the Many-to-Many Model is the two level-model:
    - Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Thread Libraries

- Thread libraries provide the application programmer with an API for creating and managing threads
- Three main thread libraries in use today:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

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, Linux, Mac OS X)

Win32 Threads

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
    1. Implementing the Runnable interface

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

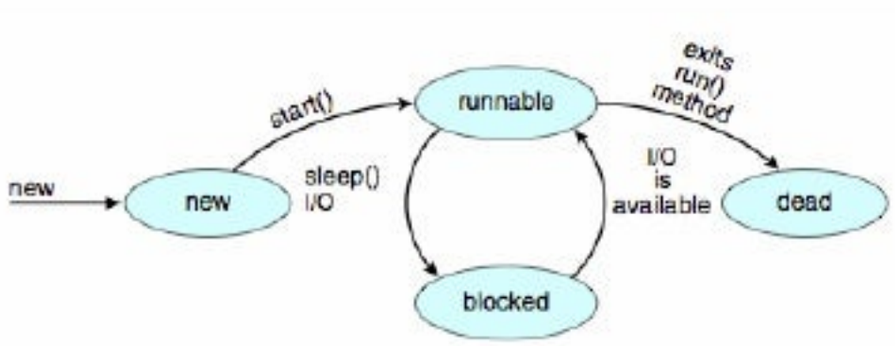
- Sample program:

```
class MutableInteger
{
  private int value;
  public int getValue() {
   return value;
  }
  public void setValue(int value) {
   this.value = value;
  }
}

class Summation implements Runnable
{
  private int upper;
  private MutableInteger sumValue;
  public Summation(int upper, MutableInteger sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }
  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setValue(sum);
  }
}
```

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

- Java thread states:



Threading Issues

- Here we discuss issues to consider with multithreaded programs

The fork() and exec() System Calls

- fork() system call: used to create a **separate**, **duplicate** process
    - Some versions of fork() duplicate all threads
        - If exec() won't be called afterwards
    - Other versions duplicate only the thread that invoked fork()
        - If exec() is called immediately after forking
- exec() system call: the parameter used will **replace** the process
    - All threads will also be replaced

Cancellation

- **Thread cancellation** is the task of terminating a thread before it has completed.
- Target thread = the thread that is to be canceled
- Cancellation of target threads occur in two different scenarios:

Asynchronous cancellation	**Deferred cancellation**
One thread immediately terminates the target thread	The target thread can periodically check if it should terminate
Canceling a thread may not free a necessary system- wide resource	Cancellation occurs only when the target thread checks if it should be canceled. (Cancellation points)

- Deferred cancellation in Java
    - Interrupting a thread

```
Thread thrd = new Thread(new InterruptibleThread());
thrd.start();
. . .
thrd.interrupt();
```

- Deferred cancellation in Java
    - Checking interruption status

```
class InterruptibleThread implements Runnable
{
    /**
     * This thread will continue to run as long
     * as it is not interrupted.
     */
    public void run() {
        while (true) {
            /**
             * do some work for awhile
             * . . .
             */

            if (Thread.currentThread().isInterrupted()) {
                System.out.println("I'm interrupted!");
                break;
            }
        }
        // clean up and terminate
    }
}
```

Signal Handling

- A **signal is used in UNIX to** notify a process that a particular event has occurred
- All signals follow this pattern:
    - A signal is generated by the occurrence of a certain event
    - A generated signal is delivered to a process
    - Once delivered, the signal must be handled
- A **signal handler** is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled
- Delivering signals in multithreaded programs, the **options** are:
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process
    - Deliver the signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process
- Synchronous signals are delivered to the *same* process that performed the operation causing the signal (E.g. / by 0)
- Asynchronous signals are generated by an event external to a running process (E.g. user terminating a process with <ctrl><c>)
- Every signal must be handled by one of two possible handlers:
    - A default signal handler
        - Run by the kernel when handling the signal
    - A user-defined signal handler
        - Overrides the default signal handler

Single-threaded programs	Multithreaded programs
Straightforward signal handling	Complicated signal handling
Signals are always delivered to a process	Which thread should the signal be delivered to?

- The method for delivering a signal depends on the signal type:
    - **Synchronous** signals need to be delivered to the thread that generated the signal, not to other threads in the process
    - It is not clear what to do with **asynchronous** signals
        - Signals need to be handled only once, so they're usually delivered to the 1st thread not blocking them

Thread Pools

- The idea is to create a number of threads at process startup and place them into a pool, where they sit and wait for work
    - When a server receives a request, it awakens a thread from this pool
    - If one is available the request is passed to it for service

    - Once the service is completed, the thread returns to the pool and wait for more work
- **Benefits** of thread pools:
    - It is **faster** to service a request with an existing thread
    - A thread pool **limits the number of threads** that exist
- Potential **problems** with a multithreaded server:
    - It takes **time** to create a thread before servicing a request
    - Unlimited threads could **exhaust** system resources (CPU time)
- Thread pools are a **solution to these problems**:
    - At process startup, several threads are created and placed into a pool, where they sit and wait for work
    - When a server receives a request, it awakens a thread from this pool, passing it the request to service
    - When the thread finishes its service it returns to the pool

Thread-Specific Data

- Threads belonging to a process **share the data** of the process
- Sometimes, each thread might need its own copy of certain data
    - E.g. Transactions in different threads may each be assigned a unique identifier
- Thread-specific data in Java

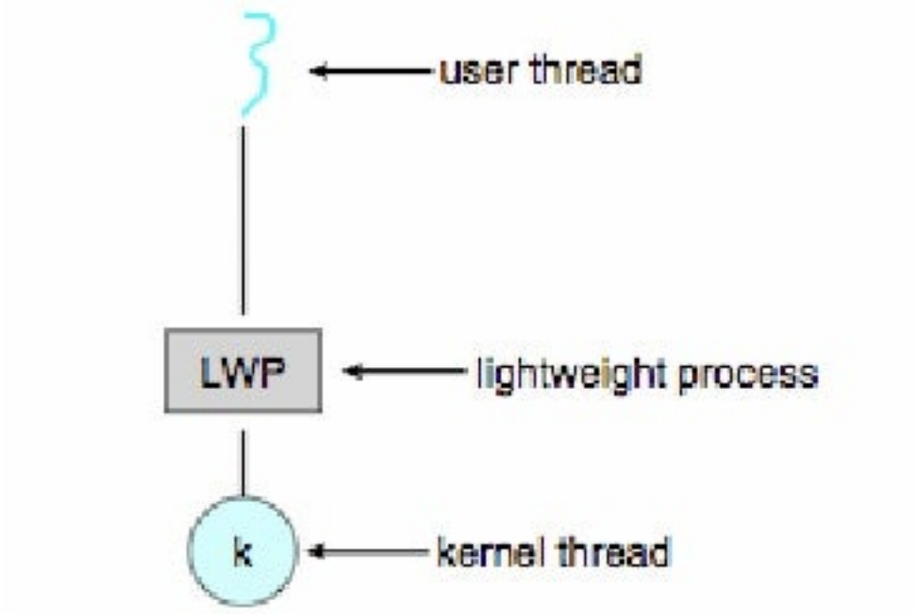
```
class Service
{
   private static ThreadLocal errorCode =
     new ThreadLocal();

   public static void transaction() {
      try {
         /**
          * some operation where an error may occur
          . . .
          */
      }
      catch (Exception e) {
          errorCode.set(e);
      }
   }

   /**
    * get the error code for this transaction
    */
   public static Object getErrorCode() {
      return errorCode.get();
   }
}
```

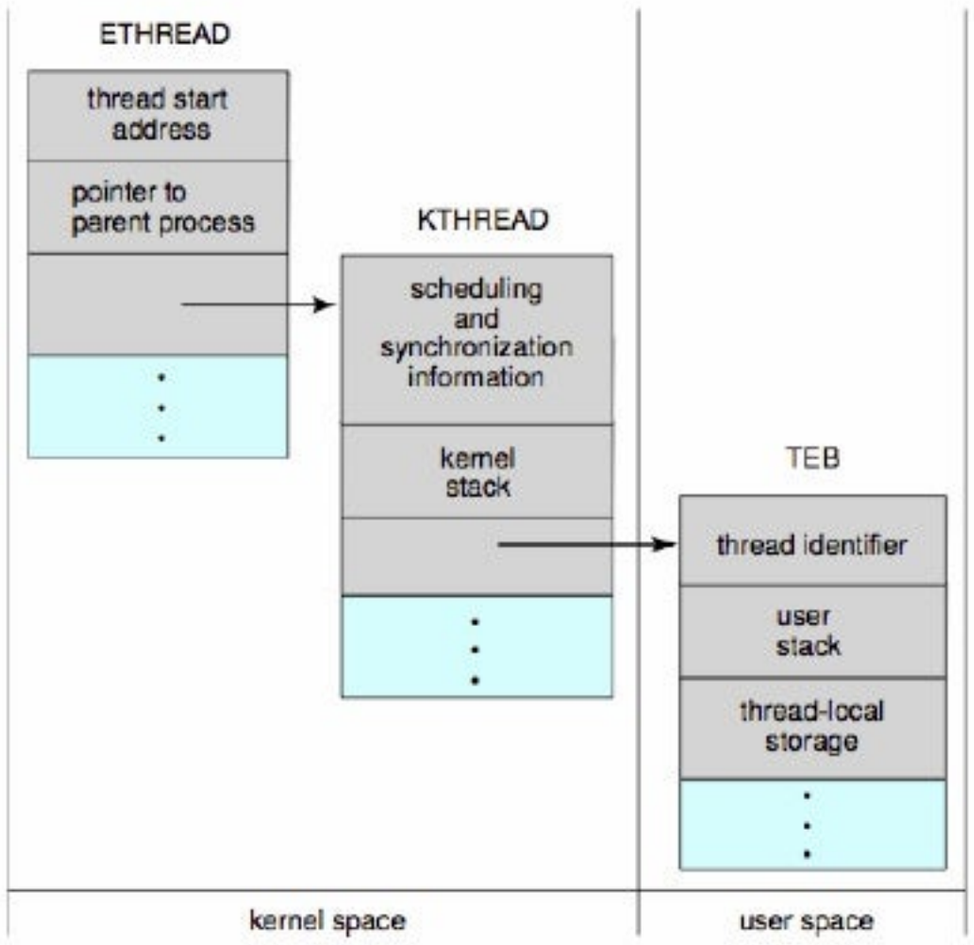
Scheduler Activations

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



Operating-System Examples

- **Windows XP threads**
  - Implements the one-to-one mapping
  - Each thread contains
    1. A thread id
    2. Register set
    3. Separate user and kernel stacks
    4. Private data storage area
  - The register set, stacks, and private storage area are known as the **context** of the threads



- **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)

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

Summary

Chapter 5: Process (CPU) Scheduling

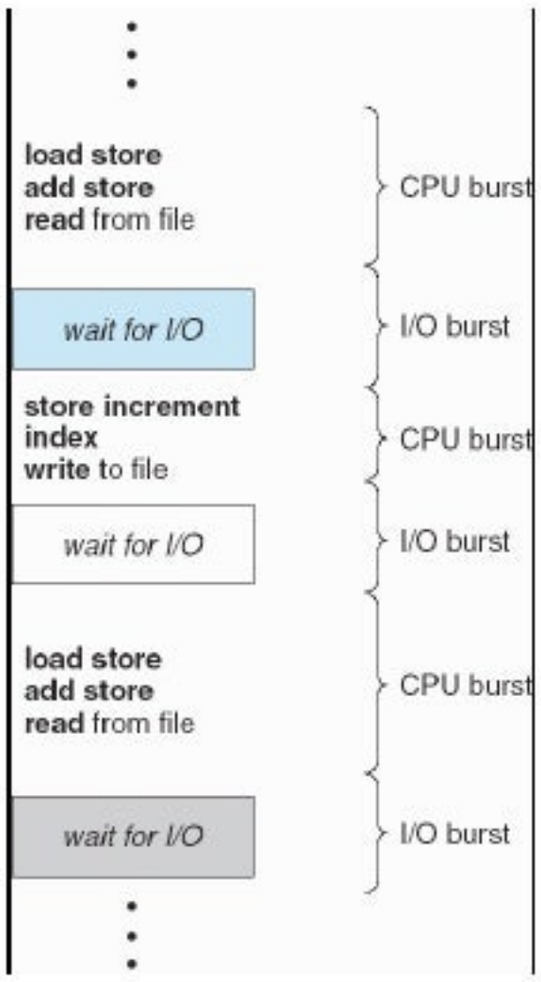
- Here we look at basic CPU-scheduling concepts and present several CPU-scheduling algorithms.
- We also consider the problem of selecting an algorithm for a particular system.
- **Objectives**:
  - To introduce CPU scheduling, which is the basis for multi-programmed operating systems.
  - To describe various CPU-scheduling algorithms.
  - To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.
- The terms process scheduling and thread scheduling are often used interchangeably

Basic Concepts

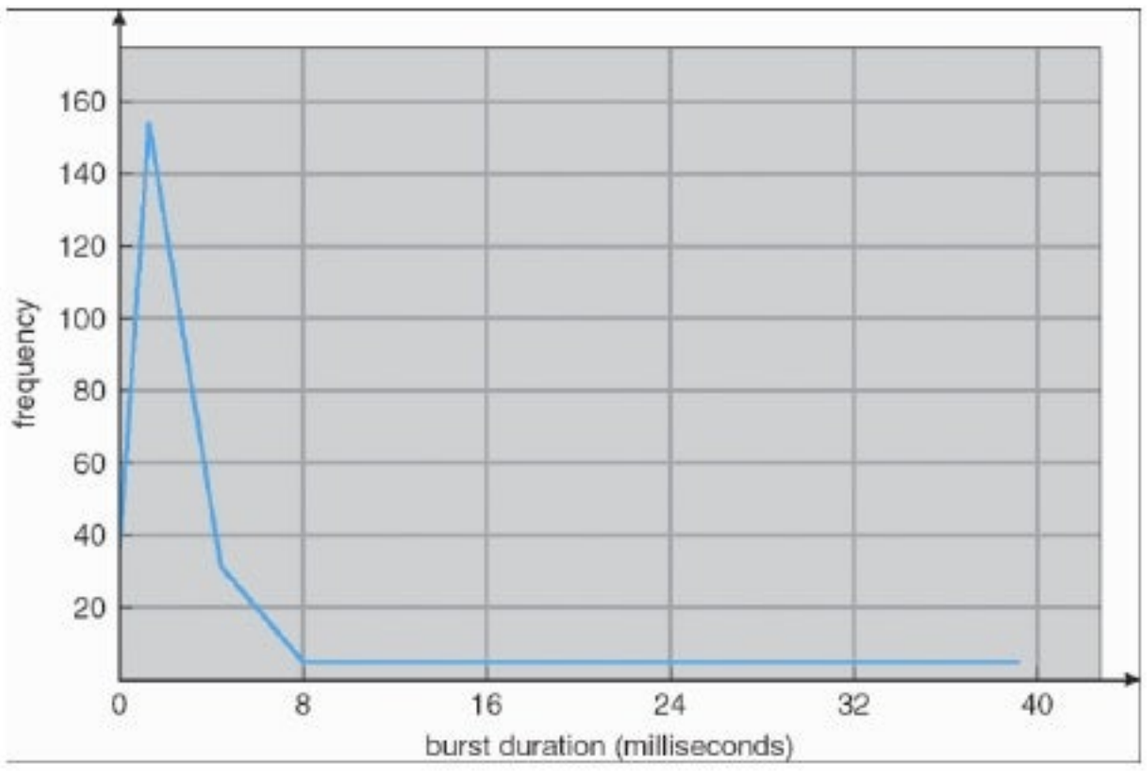
- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it
  - The CPU is allocated to the selected process by the dispatcher
- In a uni-processor system, only one process may run at a time; any other process must wait until the CPU is rescheduled
- The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

CPU-I/O Burst Cycle

- Process execution comprises a cycle of CPU execution & I/O wait
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminate execution



Histogram of CPU-burst times:



- An I/O-bound program typically has many short CPU bursts
- A CPU-bound program might have a few long CPU bursts
- These are important points to keep in mind for the selection of an appropriate CPU-scheduling algorithm

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- The short-term scheduler selects a process in the ready queue when the CPU becomes idle
- The ready queue could be a FIFO / priority queue, tree, list…
- The records in the queues are generally process control blocks (PCBs) of the processes

Preemptive Scheduling

- Circumstances under which CPU scheduling decisions take place:
  - When a process switches from the running state to the waiting state (E.g. I/O request) (1)
  - When a process switches from the running state to the ready state (E.g. when an interrupt occurs) (2)
  - When a process switches from the waiting state to the ready state (E.g. completion of I/O) (3)

    - When a process terminates (4)
- Non-preemptive/cooperative scheduling
    - Processes are allowed to **run to completion**
    - When scheduling takes place under circumstances 1 & 4
    - There is no choice in terms of scheduling
- Preemptive scheduling
    - Processes that are runnable may be **temporarily suspended**
    - There is a scheduling choice in circumstances 2 & 3
    - Problem: if one process is busy updating data and it is preempted for the second process to run, if the second process reads that data, it could be inconsistent

Dispatcher

- A component involved in the CPU scheduling function
- The dispatcher is the module that **gives control** of the CPU to the process selected by the short-term scheduler
- This function involves:
    - Switching context
    - Switching user mode
    - Jumping to the proper location in the user program to restart that program
- The dispatcher should be as fast as possible, given that it is invoked during every process switch
- Dispatch latency = the time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- Different CPU-scheduling algorithms have different properties and the choice of a particular algorithm may favor one class of process over another
- **Criteria** to compare CPU-scheduling algorithms:
    - **CPU utilization**
        - CPU utilization should range from 40% - 90%
    - **Throughput**
        - The number of processes completed per time unit
    - **Turnaround time**
        - The time interval from process submission to completion
        - Formula: Time of completion – Time of submission
        - Formula: CPU burst time + Waiting time (includes I/O)
    - **Waiting time**
        - The sum of the periods spent waiting in the ready queue
        - Formula: Turnaround time – CPU burst time
    - **Response time**
        - The amount of time it takes to start responding, but not the time it takes to output that response
- We want to maximize CPU utilization, and minimize turnaround, waiting & response time

Scheduling Algorithms

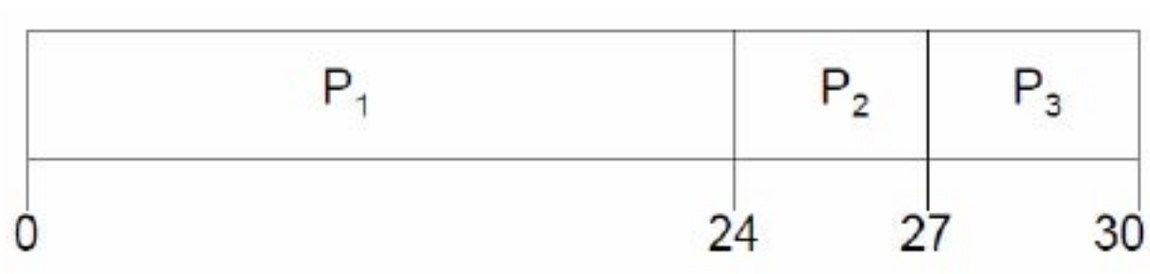
- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU
- There are many different CPU-scheduling algorithms. Here we describe several of them.

First-Come, First-Served (FCFS) Scheduling

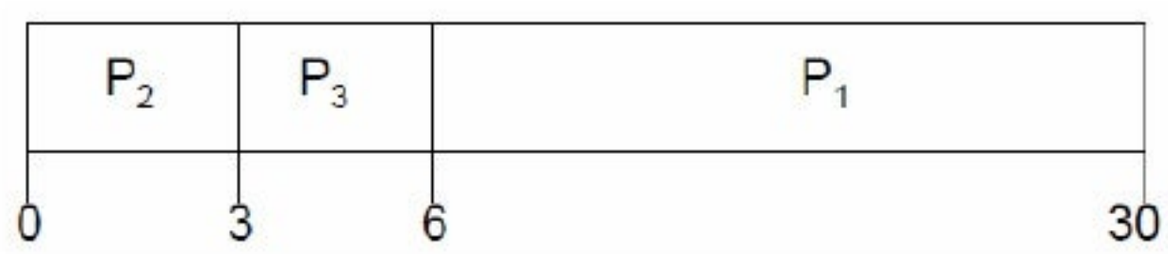
- The process that requests the CPU first is allocated the CPU 1st
- The PCB of a process is linked onto the tail of the ready queue
- When the CPU is free, it gets the process at the queue's head
- The average waiting time is generally **not minimal**
- Convoy effect = when processes wait for a big one to get off
- Non-preemptive (a process keeps the CPU until it releases it)
- Not good for time-sharing systems, where each user needs to get a share of the CPU at regular intervals
- **Example**:

Process	**Burst Time**
P1	24
P2	3
P3	3

  - Suppose that the processes arrive in the order: *P1*, *P2*, *P3*
  - The Gantt Chart for the schedule is:



  - Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
- Suppose that the processes arrive in the order *P2*, *P3*, *P1*
  - The Gantt chart for the schedule is:



  - Waiting time for *P1* = 6;*P2* = 0; *P3* = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Much better than previous case
  - *Convoy effect* short process behind long process

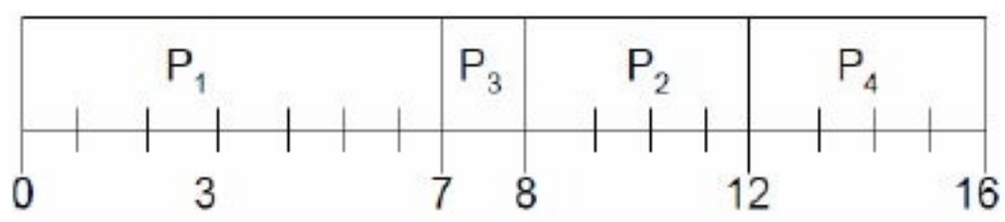
Shortest-Job-First (SJF) Scheduling

- The CPU is assigned the process with the shortest next CPU burst
- If two processes have the same length, FCFS scheduling is used

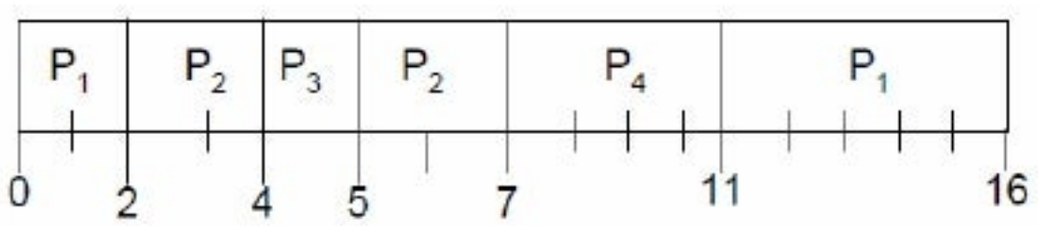
- The difficulty is knowing the length of the next CPU request
- For **long-term scheduling** in a **batch** system, we can use the process time limit specified by the user, as the 'length'
- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
- We can, however, try to predict the length of the next CPU burst
- The SJF algorithm may be either **preemptive** or **non-preemptive**
    - **Preemptive SJF algorithm**:
        - If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted
        - aka Shortest-Remaining-Time-First (SRTF) scheduling
    - **Non-preemptive SJF algorithm**:
        - The current process is allowed to finish its CPU burst
- SJF has the **minimum average waiting time** for a set of processes
- **Example**:

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

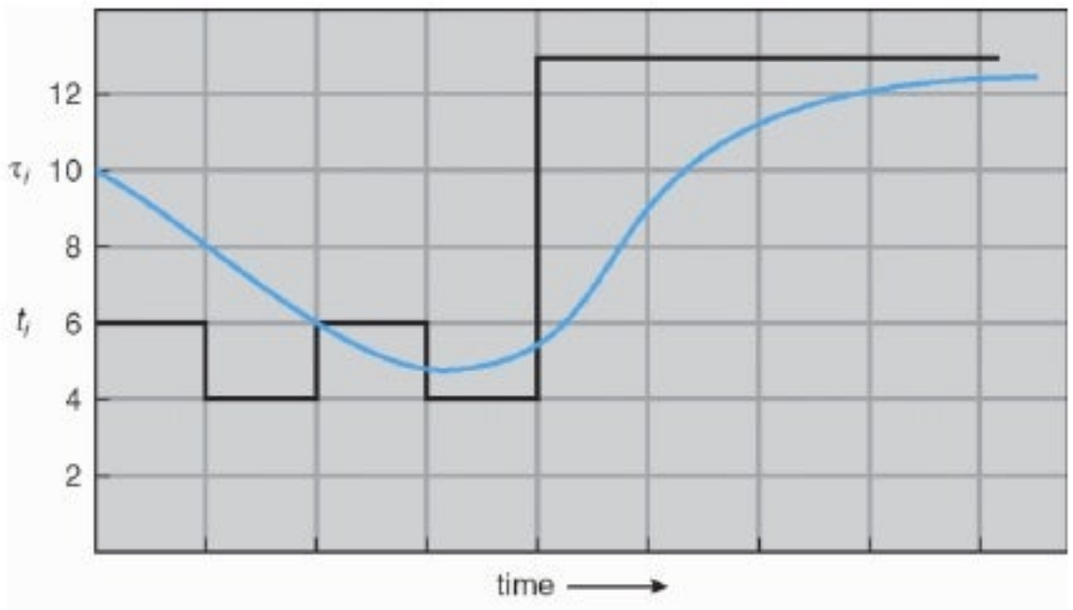
    - **SJF (non-preemptive)**



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



        - Average waiting time = (9 + 1 + 0 +2)/4 = 3
- **Determining the length of the next CPU burst**:
    - Can only estimate the length
    - Can be done by using the length of previous CPU bursts, using exponential averaging
    - Formula on p.191 top



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**:

α =0

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

α =1

- $\tau_{n+1} = \alpha\, t_n$
- Only the actual last CPU burst counts

If we expand the formula, we get:

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

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

Priority Scheduling

- Each process gets a priority (Highest priority = executed first)
  - **Preemptive** priority scheduling
    - The CPU is preempted if the priority of the newly arrived process is higher than the priority of the current one
  - **Non-preemptive** priority scheduling
    - The new process is put at the **head** of the ready queue
- Equal-priority processes are scheduled in FCFS order
  - **Internally-defined** priorities
    - Use some measurable quantity to compute the priority
    - E.g. time limits, memory requirements, no. of open files…
  - **Externally-defined** priorities
    - Set by criteria that are external to the OS
    - E.g. the importance of a process, political factors…
- **Problem**:
  - Indefinite blocking (starvation), where low-priority processes are left waiting indefinitely for the CPU

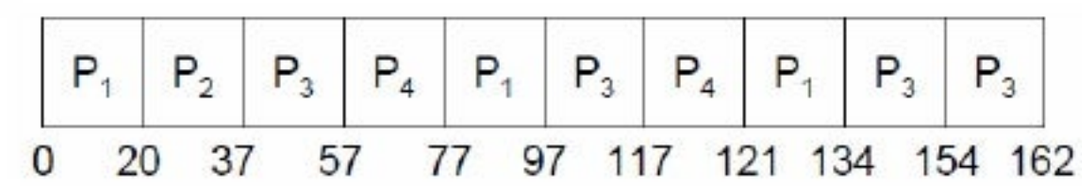
- **Solution**:
    - Aging (a technique of gradually increasing the priority of processes that wait in the system for a long time)

Round-Robin Scheduling

- Designed especially for time-sharing systems
- Like FCFS scheduling, but with preemption
- A time quantum / time slice is defined (generally 10 – 100 ms)
- The ready queue is treated as a circular queue
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum
- The ready queue is kept as a FIFO queue of processes
- The CPU scheduler
    - picks the 1st process from the ready queue
    - sets a timer to interrupt after 1 time quantum, and
    - dispatches the process
- One of two things will then happen:
    - The process may have a CPU burst of less than 1 time quantum, and will release the CPU voluntarily
    - If the process has a CPU burst longer than 1 time quantum, the timer will go off and cause an interrupt to the OS. The process will then be put at the **tail** 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
- RR Performance depends heavily on the size of the time quantum
    - q large ⇒ FIFO
    - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high
- We want the time quantum to be large with respect to the context-switch time
- **Example of RR with time Quantum = 20:**

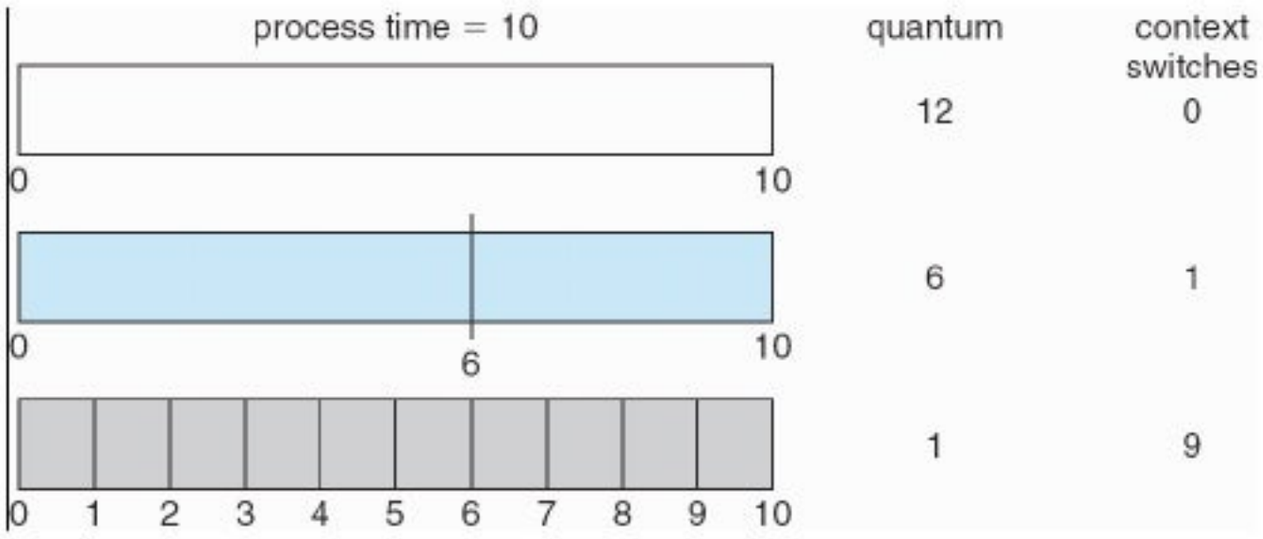
Process	Burst Time
P1	53
P2	17
P3	68
P4	24

    - The Gantt chart is:

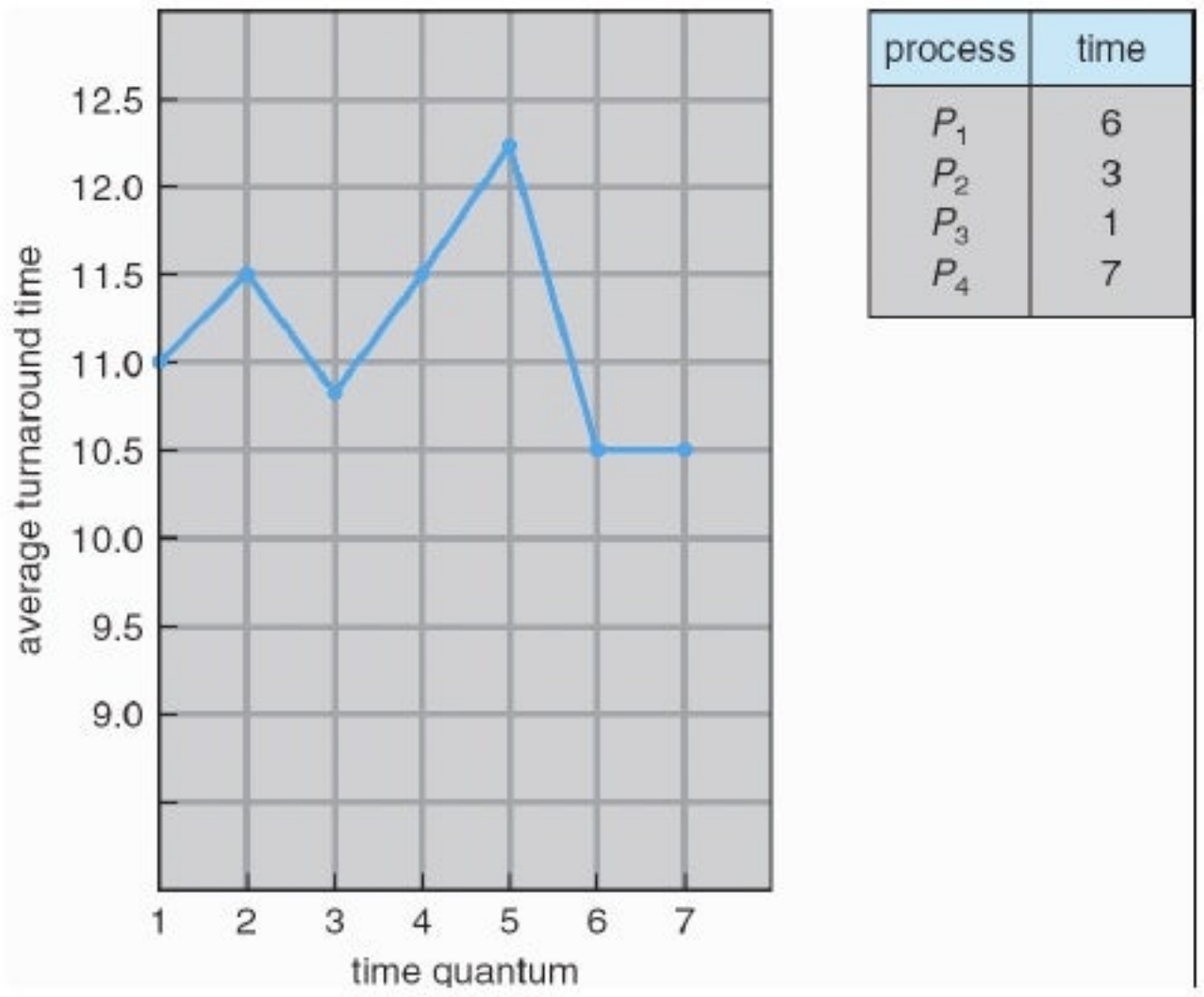


- Typically, **higher average turnaround** than SJF, but better *response*
- In software we need to consider the effect of context switching on the performance of RR scheduling
    - The larger the time quantum for a specific process time, the less time is spend on context switching

- The smaller the time quantum, more overhead is added for the purpose of context-switching
- **Example**: (This is on a per case situation)

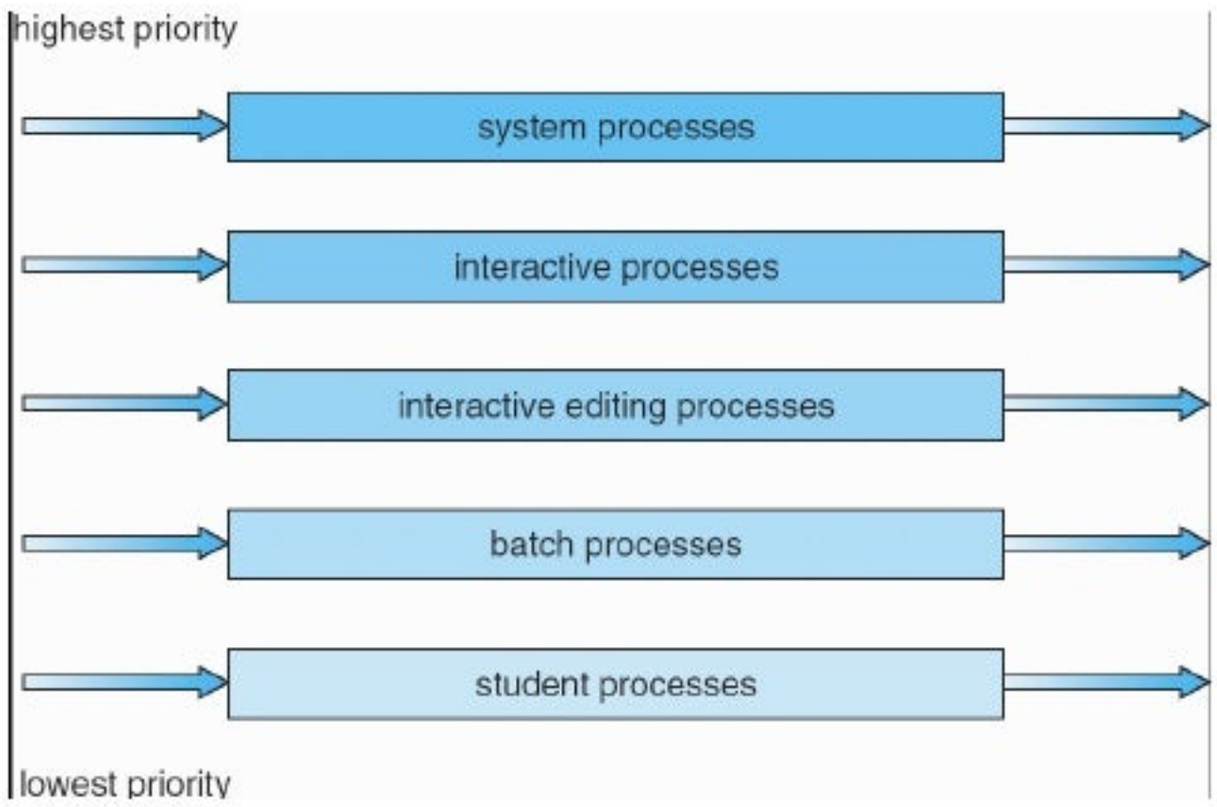


- Turnaround time also depends on the size of the time quantum:



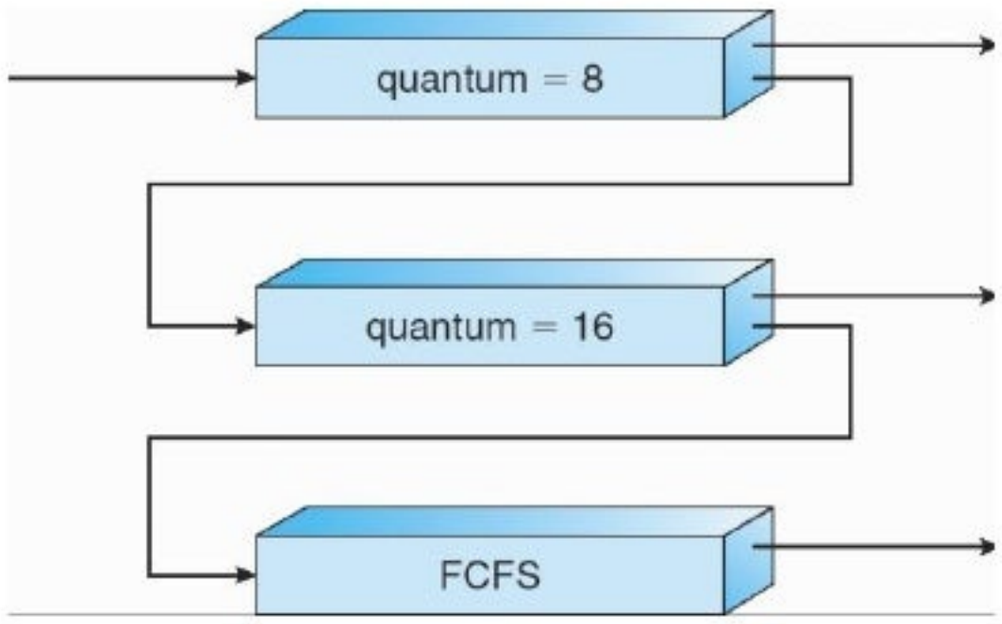
Multilevel Queue Scheduling

- For when processes can easily be classified into separate groups
- E.g. a common division is made between foreground (interactive) and background (batch) processes
- The ready queue is partitioned into several separate queues
- The processes are **permanently** assigned to one queue, based on a property like memory size, process priority, process type…
- Each queue has its own scheduling algorithm
- There must also be **scheduling among the queues**, which is commonly implemented as fixed-priority preemptive scheduling
  - 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 Feedback Queue Scheduling

- Processes may **move** between queues
- Processes with different CPU-burst characteristics are separated
- If a process uses too much CPU time, it will be moved to a lower-priority queue
- If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue (Aging prevents starvation)
- In general, a multilevel feedback queue scheduler is defined by the following parameters:
    - The number of queues
    - The scheduling algorithm for each queue
    - The method used to determine when to upgrade a process to a higher priority queue
    - The method used to determine when to demote a process to a lower-priority queue
    - The method used to determine which queue a process will enter when that process needs service
- This is the most general, but complex scheme
- **Example** of Multilevel Feedback Queue:
    - **Three queues**:
        - *Q*0 - RR with time quantum 8 milliseconds
        - *Q*1 - RR time quantum 16 milliseconds
        - *Q*2 - FCFS
    - **Scheduling**
        - A new job enters queue *Q0* 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:



Thread Scheduling

- p.199 TB
- On operating systems that support them, it is kernel-level threads - not processes - that are being scheduled by the operating system
- **Local Scheduling**
  - How the **threads library** decides which thread to put onto an available LWP
- **Global Scheduling**
  - How the **kernel** decides which kernel thread to run next

Contention Scope

- **Process-Contention scope**:
  - On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP
- **System-Contention scope**:
  - The process of deciding which kernel thread to schedule on the CPU

Pthread Scheduling

Sample of thread creation with Pthreads:

```c
int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
    }

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

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

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

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

   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
    - Typically each processor maintains its own private queue of processes (or threads) all of which are available to run
- *Load sharing*
- *Asymmetric multiprocessing*
    - Only one processor accesses the system data structures, alleviating the need for data sharing

Approaches to Multiple-Processor Scheduling

- We assume homogeneous processors (identical in functionality) and uniform memory access (UMA)
- If several identical processors are available, then load sharing can occur, with a **common ready queue**
- Processes in the queue are scheduled to any available processor
- One of two scheduling approaches may be used:
    - Each processor is **self-scheduling**, and selects a process from the common ready queue to execute
    - One processor is appointed as scheduler for the other processors, creating a **master-slave** structure
- Some systems carry the master-slave structure further by having all scheduling decisions, I/O processing, and other system activities handled by one single processor – the **master server**

- This **asymmetric multiprocessing** is simpler than **symmetric multiprocessing (SMP)**, because only one processor accesses the system data structures, alleviating the need for data sharing
- It isn't as efficient, because I/O processes may bottleneck on the one CPU that is performing all of the operations
- Typically, asymmetric multiprocessing is implemented 1st within an OS, and then upgraded to symmetric as the system evolves

Processor Affinity

- Processor affinity:
  - Migration of processes to another processor is avoided because of the cost of invalidating the process and repopulating the processor cache
- **Soft** affinity:
  - When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen
- **Hard** affinity:
  - When an OS have the ability to allow a process to specify that it is not to migrate to other processors

Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system
- Two **migration** approaches:
  - **Push** migration
    - A specific task checks the load on each processor and if it finds an imbalance it evenly distributes the load to less-busy processors
  - **Pull** migration
    - A idle processor pulls a waiting task from a busy processor

Multicore Processors

- Complicated scheduling issue

Virtualization and Scheduling

Operating System Examples

Algorithm Evaluation

- p.213 TB
- Deterministic modeling:
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queuing models
- Implementation

Deterministic Modeling

- A method that takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Simple; fast; exact numbers, so algorithms can be compared

- However, it requires exact numbers for input, and its answers apply to only those cases
- The main uses of deterministic modeling are in describing scheduling algorithms and providing examples
- Good if you're running the same programs over and over again
- Over many examples, deterministic modeling may indicate trends
- In general, deterministic modeling is too specific, and requires too much exact knowledge to be useful

Queuing Models

- You can determine the distribution of CPU and I/O bursts
- A formula describes the probability of a particular CPU burst
- The computer system is described as a network of servers
- Each server has a queue of waiting processes
- Knowing arrival & service rates, we can compute utilization, average queue length, wait time… (= queuing-network analysis)
- Limitations of queuing analysis:
    - The algorithms that can be handled are limited
    - The math of complicated algorithms can be hard to work with
    - It is necessary to make assumptions that may not be accurate
    - As a result, the computed results might not be accurate

Simulations

- Involve programming a model of the computer system
- Software data structures represent the major system components
- The simulator has a variable representing a clock
- As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the process, and the scheduler
- As the simulation executes, statistics that indicate algorithm performance are gathered and printed
- A random-number generator is programmed to generate processes, CPU-burst times… according to probability distributions
- The distributions may be defined mathematically or empirically
- If the distribution is to be defined empirically, measurements of the actual system under study are taken
- The results are used to define the actual distribution of events in the real system, and this distribution can then be used to drive the simulation
- Trace tapes can be used to record the sequence of actual events
- **Disadvantages**:
    - Simulations can be costly, requiring hours of computer time
    - Traced tapes can require large amounts of storage space
    - The design, coding, and debugging of the simulator can be a major task

Implementation

- The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the OS, and see how it works
- The major difficulty is the cost of this approach

- The environment in which the algorithm is used will change

Summary

PART THREE: PROCESS COORDINATION

Chapter 6: Synchronization

- Co-operating process = one that can affect / be affected by other processes.
- Co-operating processes may either directly share a logical address space (i.e. code & data) , or share data through files or messages through threads (ch4).
- Concurrent access to shared data can result in inconsistencies
- **Objectives**:
    1. To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
    2. To present both software and hardware solutions of the critical- section problem
    3. To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

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

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

    - **Consumer**:

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

- **Race condition**:
    - When the outcome of the execution depends on the particular **order** in which data access takes place
- **Example**:
    - count++ could be implemented as

        register1 = count

register1 = register1 + 1

count = register1

- count-- could be implemented as

  register2 = count

  register2 = register2 -1

  count = register2

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count{register1 = 5}

  S1: producer execute register1 = register1 + 1 {register1 = 6}

  S2: consumer execute register2 = count{register2 = 5}

  S3: consumer execute register2 = register2 -1{register2 = 4}

  S4: producer execute count = register1{count = 6 }

  S5: consumer execute count = register2{count = 4}

The Critical-Section Problem

- Critical section = a segment of code in which a process may be changing common variables, updating a table, writing a file, etc
  - **Entry section**
    - Requests permission to enter the critical section
  - **Critical section**
    - Mutually exclusive in time (no other process can execute in its critical section)
  - **Exit section**
    - Follows the critical section
  - **Remainder section**
- ***A solution to the critical-section problem must satisfy***:
  - **Mutual exclusion**
    - Only one process can be in its critical section
  - **Progress**
    - Only processes that are not in their remainder section can enter their critical section, and the selection of a process cannot be postponed indefinitely
  - **Bounded waiting**
    - There must be a bound 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 the request is granted

- **Structure of a typical process**:

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

Peterson's Solution

- This is an example of a software solution that can be used to prevent race conditions
- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
    - int turn;
    - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section
    - flag[i] = true implies that process Pi is ready!
- Algorithm for process Pi:

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

- To prove that this solution is correct we show that:
    - **Mutual exclusion** is preserved
    - The **progress requirement** is satisfied
    - The **bounded-waiting requirement** is met

Synchronization Hardware

- Hardware can also be used to solve the critical-section problem
- If in a uni-processor environment interrupts were disabled, no unexpected modifications would be made to shared variables
- **Disabling interrupts** in a multi-processor environment **isn't feasible**, so many machines provide special hardware instructions

- Instead, we can generally state that any solution to the critical-section problem requires a simple tool, a **lock**
    - Race conditions are prevented by requiring that critical regions be protected by locks

```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

- Modern machines provide special atomic hardware instructions
    - **Atomic = non-interruptible**
    - Either test memory word and set value
    - Or swap contents of two memory words
- These instructions allow us either to test & modify the content of a word, or to swap the contents of two words, atomically
    - TestAndSet

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

- NB characteristic: this instruction is executed atomically, so if two TestAndSet instructions are executed simultaneously (on different CPUs), they will be executed sequentially
- TestAndSet with **mutual exclusion**

```
do{
  while( TestAndSet( &lock ) ) ;
  // critical section
  lock = false;
  // remainder section
} while( true);
```

- lock is initialized to false
- Swap

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

```
}
```

- Swap with mutual-exclusion

```
do{
  key = true;
  while(key == true )
    swap(& lock, &key );
  // critical section
  lock = false;
  // remainder section
} while(true);
```

- lock is initialized to false
- Bounded-waiting mutual exclusion with TestAndSet

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

- Common data structures are
  - boolean waiting[n];
  - boolean lock;
  - Data structures initialized to false
- To **prove** that the mutual-exclusion requirements is met:
  - note that Pi can enter its critical section only if either waiting[i] == false or key == false
  - key can become false only if the TestAndSet() is executed
    - first process to execute TestAndSet() will find key == false, all others must wait

- waiting[i] can become false only if another process leaves its critical section
    - only one waiting[i] is set to false

- To **prove** the Progress requirement is met:
    - The mutual exclusion arguments apply, since they let a process that is waiting to enter its critical section proceed
- To **prove** the Bounded waiting requirement is met:
    - When a process leaves its critical section, it scans the waiting array in the cyclic ordering (i+1, i+2…, n-1, 0…, i-1) and designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section

Semaphores

- Semaphore = a **synchronization tool** used to control **access to shared variables** so that only one process may at any point in time **change** the value of the shared variable
- A semaphore S is an integer variable that is accessed only through two standard atomic operations: wait and signal

```
wait(s){
  while(s<=0) ;
   ;//no-op
s--;
 }
 signal(s){
  s++;
  }
```

- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly (atomic)

Usage

- Counting semaphores can allow n processes to access (e.g. a database) by initializing the semaphore to n
- Binary semaphores (with values 0 & 1) are simpler to implement
- N processes share a semaphore, mutex (*mut*ual *ex*clusion), initialized to 1
- Each process is organized as follows:

```
do {
 wait(mutex);
    // critical section
 Signal(mutex);
    // remainder section
} while (true);
```

- Example on p.235 mid

Implementation

- p.235 - p.238

- Disadvantage of these mutual-exclusion solutions: they all require **busy waiting** (i.e. processes trying to enter their critical sections must **loop continuously** in the entry code)
- This **wastes CPU cycles** that another process might be able to use
- This type of semaphore is also called a **spinlock** (because the process 'spins' while waiting for the lock)
- Advantage of a spinlock: **no context switch** is required when a process must wait on a lock (Useful for short periods)
- To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations so that rather than busy waiting, the process can **block itself**:
    - The process is placed into a waiting queue
    - The state of the process is switched to the waiting state
    - Control is transferred the CPU scheduler
    - The CPU scheduler selects another process to execute
- The critical aspect of semaphores is that they must be executed **atomically**, i.e. wait & signal operations can't execute together
- This (critical-section) problem can be solved in two ways:
    - In a uni-processor environment
        - Inhibit interrupts when the wait and signal operations execute
        - Only the current process executes, until interrupts are re-enabled and the scheduler regains control
    - In a multiprocessor environment
        - Inhibiting interrupts doesn't work
        - Use the hardware / software solutions described above

Deadlocks and Starvation

- Deadlock state = when every process in a set is waiting for an event that can be caused only by another process in the set
- Implementing a semaphore with a waiting queue may result in two processes each waiting for the other one to **signal**
- **Resource acquisition** and **release** are the events concerned here
- **Starvation (indefinite blocking)** = when processes wait indefinitely within the semaphore

Priority Inversion

- Priority inversion = when a high-priority process needs data currently being accessed by a lower-priority one

Classic Problem of Synchronization

Bounded-Buffer Problem

- There is a pool of n buffers, each capable of holding one item
- The mutex semaphore provides mutual exclusion for access to the buffer pool and is initialized to 1
- The empty & full semaphores count the no of empty & full buffers
- Symmetry: The producer produces full buffers for the consumer / the consumer produces empty buffers for the producer
- p.240 TB

The Readers-Writers Problem

- p.241 TB
- A data set is shared among a number of concurrent processes
    - **Readers**
        - only read the data set; they do not perform any updates
    - **Writers**
        - can both read and write
- Many readers can access shared data without problems
- Writers need **exclusive use** to shared objects
- First readers-writers problem:
    - Readers don't wait, unless a writer has permission
    - Problem: writers may starve if new readers keep appearing because the readers are granted shared access each time
- Second readers-writers problem:
    - If a writer is ready, no new readers may start reading
    - Problem: readers may starve
- Used to provide reader-writer locks on some systems
- The mode of lock needs to be specified
    - read access
    - write access
- Reader-writer locks most useful in following situations:
    - In applications where it is easy to identify which processes only read shared data and which processes only write shared data
    - In applications that have more readers than writers. This is because reader-writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock

The Dining-Philosophers Problem

- p.242 TB
- There are 5 philosophers with 5 chopsticks (semaphores)
- A philosopher is either eating (with two chopsticks) or thinking
- A philosopher tries to grab a chopstick by executing a wait operation on that semaphore, and releases the chopsticks by executing the signal operation on the appropriate semaphores
- The shared data are: semaphore chopstick[5]; where all the elements of chopstick are initialized to 1
- This solution guarantees that no two neighbors are eating simultaneously, but a deadlock will occur if all 5 philosophers become hungry simultaneously and grab their left chopstick
- Some remedies to the deadlock problem:
    - Allow **at most four** philosophers to be sitting simultaneously at the table
    - Allow a philosopher to pick up chopsticks **only if both are available** (He must pick them up in a critical section)

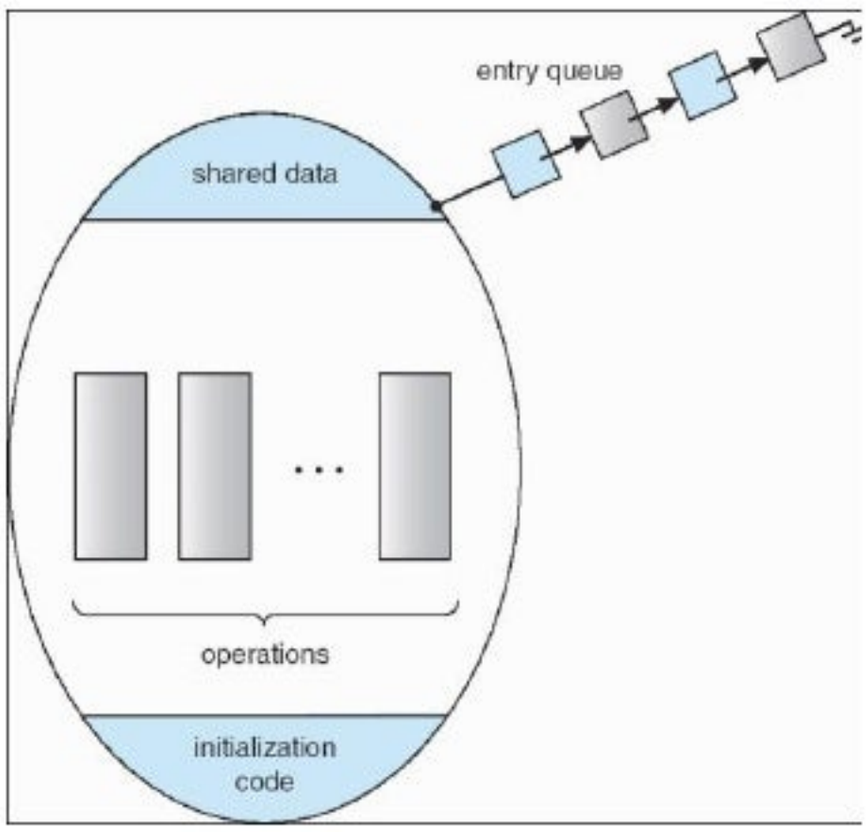
    - Use an **asymmetric solution**: An odd philosopher picks up his left chopstick first, and an even one the right one first
- A deadlock-free solution doesn't necessarily eliminate the possibility of starvation
- Monitors is a solution to the dining-philosophers problem

Monitors

- p.244-245 TB
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time
- Monitors are needed because if many programmers are using a semaphore and one programmer forgets to signal after the program has left the critical section, then the entire synchronization mechanism will end up in a deadlock
- Definition: A collection of procedures, variables, and data structures that are all grouped together in a module / package

Usage

- p.245-246 TB
- A monitor type is presents a set of programmer-defined operations that are provided to ensure mutual exclusion within the monitor
- Three distinguishing characteristics of a monitor:
    - It **encapsulates** its permanent variables
    - Procedures execute in **mutual exclusion**
    - Synchronization is provided via **condition variables**
- The monitor shouldn't access any non-local variables, and local variables shouldn't be accessible from outside the monitor
- Any process may call the monitor, but **only one** process at any point in time may be **executing inside** the monitor



Dining-Philosophers Solution Using Monitors

- p.248 - p.249

Implementing a Monitor Using Semaphores

Resuming Processes within a Monitor

Synchronization Examples

Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
    - Here we are concerned with changes to stable storage - disk
    - Transaction is series of read and write operations
    - Terminated by commit(transaction successful) or abort(transaction failed) operation
    - Aborted transaction must be rolled back to undo any changes it performed
- To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by transactions
    - Volatile storage - information stored here does not survive system crashes
        - Example: main memory, cache
    - Nonvolatile storage -Information usually survives crashes
        - Example: disk and tape
    - Stable storage - Information never lost
        - Not actually possible, so approximated via replication or RAID to devices with independent failure modes
- Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Write-ahead logging: Each log record describes a single operation of a transaction write and has these fields:
    - Transaction name (The unique name of the transaction)
    - Data item name (The unique name of the item written)
    - Old value (The value of the data prior to the write)
    - New value (The value that the data will have afterwards)
- Prior to a write being executed, the log records must be written onto stable storage
- Performance penalty: Two physical writes are required for each logical write requested, and more storage is needed
- Two procedures used by the recovery algorithm:
    - undo – restores the data to the old values
    - redo – sets the data to the new values
- The set of data updated by the transaction and their respective old and new values can be found in the log

Checkpoints

- When a failure occurs, we must consult the log to determine those transactions that need to be redone / undone
- Drawbacks to searching the entire log:
    - The search is time-consuming
    - Redoing data modifications causes recovery to take longer
- To reduce this overhead, the system performs checkpoints:
    - Output all log records onto stable storage
    - Output all modified data to the stable storage
    - Output a log record <checkpoint> onto stable storage
- The presence of a <checkpoint> record in the log allows streamlined recovery, since you search for the last checkpoint

Concurrent Atomic Transactions

- Serializability = when transactions are executed serially
    - Can be maintained by executing each transaction within a critical section
- All transactions could share a semaphore mutex, initialized to 1
    - When the transaction starts, it first executes wait
    - After the transaction commits / aborts, it executes signal
- This scheme ensures **atomicity** of all concurrently executing transactions, but is still too restrictive
- Concurrency-control algorithms to ensure serializability

Serializability

- **Serial schedule**: each transaction executes atomically
- **Example**:
    - Consider two data items A and B
    - Consider Transactions T0 and T1
    - Execute T0, T1atomically
    - Execution sequence called schedule
    - Atomically executed transaction order called serial schedule
    - For N transactions, there are N! valid serial schedules
- Schedule 1: T0 then T1

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

- **Non-serial schedule**: transactions overlap execution
    - Resulting execution not necessarily incorrect
- Consider schedule S, operations Oi, Oj
    - Conflict if access same data item, with at least one write
- If Oi, Oj consecutive and operations of different transactions & Oi and Oj don't conflict
    - Then S' with swapped order Oj Oi equivalent to S
- If S can become S' via swapping non conflicting operations
    - S is conflict serializable

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

Locking Protocol

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

Timestamp-Based Protocols

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

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Summary

Chapter 7: Deadlocks

- **Objectives:**
    - To develop a description of deadlocks, which prevents sets of concurrent processes from completing their tasks
    - To present a number of different methods for preventing or avoiding deadlocks in a computer system

System Model

- Computer resources are partitioned into several types (e.g. memory space, CPU cycles, files, I/O devices…)
- Each type consists of some number of identical instances (e.g. if you have two CPUs, the resource type CPU has two instances)
- If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request
- A process may utilize a resource in only this sequence: (p.284 TB)
    - **Request**:
        - The process requests the resource
        - If the request cannot be granted immediately (for example, if the resource is being used by other process), then the requesting process must **wait** it can acquire the resource
    - **Use**:
        - The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer)
    - **Release**:
        - The process releases the resource

Deadlock Characterization

- A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes

Necessary Conditions

- A deadlock situation can arise if **all** these situations hold **simultaneously**:
    - **Mutual exclusion**
        - At least one resource must be held in a non-sharable mode
    - **Hold and wait**
        - A process must hold at least one resource and be waiting

- **No preemption**
    - A resource can be released only voluntarily by a process
- **Circular wait**
    - In a set of waiting processes, all are waiting for a resource held by another process in the set

- All four conditions must hold for a deadlock to occur

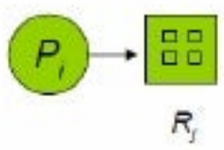
Resource-Allocation Graph

- p.287 - p.289 TB
- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph
- It consists of the following parts:
    - A set of vertices *V* and a set of edges *E*
        - V is partitioned into two types:
            - *P*= {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system
            - *R*= {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system
        - **request edge** - directed edge *P*1 →*Rj*
            - From a process to a resource
        - **assignment edge** - directed edge *Rj*→*Pi*
            - From a resource to a process

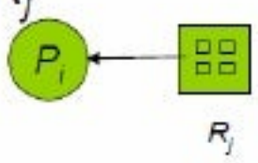
- Process

- Resource Type with 4 instances

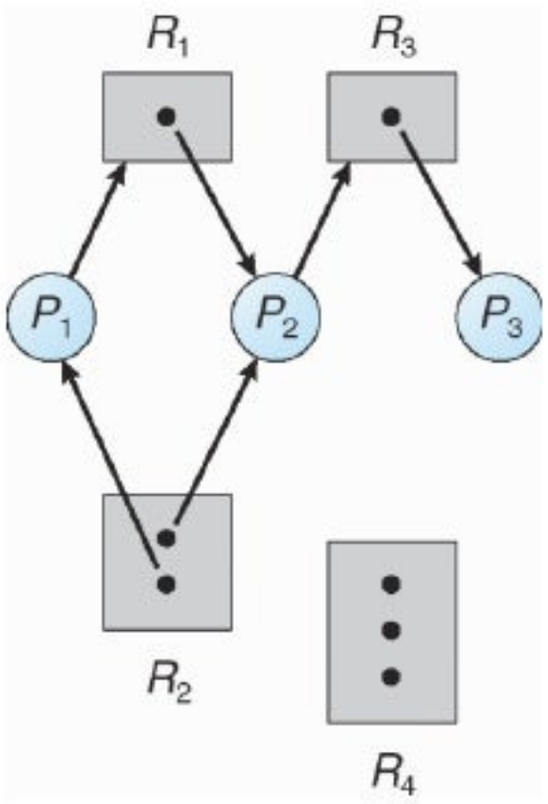
- P_i requests instance of R_j



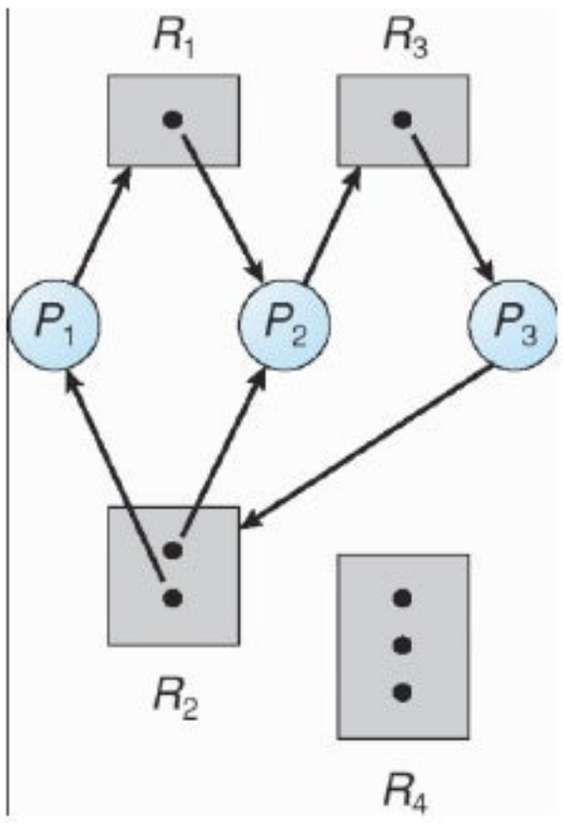
- P_i is holding an instance of R_j



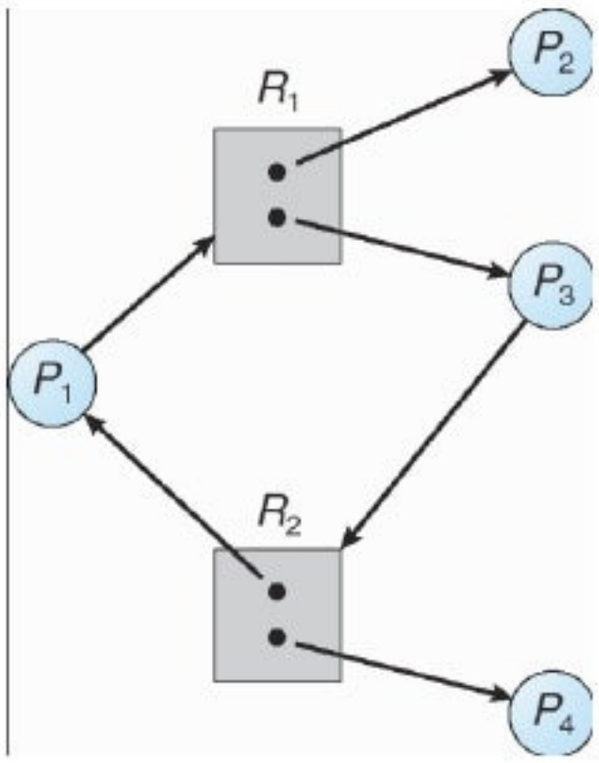
- **Example** of a Resource Allocation Graph:



- If a resource-allocation graph doesn't have a cycle, then the system is not in a deadlock state
- If there is a cycle, then the system may / may not be in a deadlock state
- Resource Allocation Graph with a deadlock:



- Graph with a Cycle but no deadlock:



- **Basic facts about Resource Allocation Graphs**:
  - If graph contains no cycles ⇒ no deadlock

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

Methods for Handling Deadlocks

- Three ways to deal with the deadlock problem:
    - Can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state
    - Can allow the system to enter a deadlock state, detect it, and recover
    - Can ignore the problem altogether and pretend that deadlock never occur in the system
- Make use of either deadlock prevention or deadlock avoidance to make sure that deadlocks never occur
- Deadlock **prevention**:
    - Provides a set of methods for ensuring that at least one of the necessary conditions can hold
- Deadlock **avoidance**:
    - Requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime

Deadlock Prevention

- Ensures that at least one of the necessary conditions can't hold

Mutual Exclusion

- Mutual-exclusion condition must hold for non sharable resources
    - e.g. a printer
- Sharable resources cannot be involved in a deadlock
    - e.g. read-only files

Hold and Wait

- To ensure this condition never occurs, when a process requests a resource it must not hold any other resources
- <u>Protocol # 1</u>
    - Each process requests and is allocated all its resources before it begins execution
- <u>Protocol # 2</u>
    - A process requests some resources only when it has none
    - Must release all resources first before can request new resources
- Disadvantages (for both protocols):
    - Resource utilization may be low, since many resources may be allocated but unused for a long period
    - Starvation is possible – a process that needs several popular resources may have to wait indefinitely

No Preemption

- <u>Protocol # 1</u>
    - If a process is holding some resources and requests another resource that can't be immediately allocated to it, then all resources currently being held are **implicitly released**

    - The process is restarted only when it can regain its old resources as well as the new ones that it is requesting
- Protocol # 2
    - If some requested resources are not available, check whether they are allocated to a process that is waiting for additional resources. If so, **preempt these resources** from the waiting process and allocate them to the requesting one
    - This protocol is often applied to resources whose state can be easily saved and restored later, like CPU registers

Circular Wait

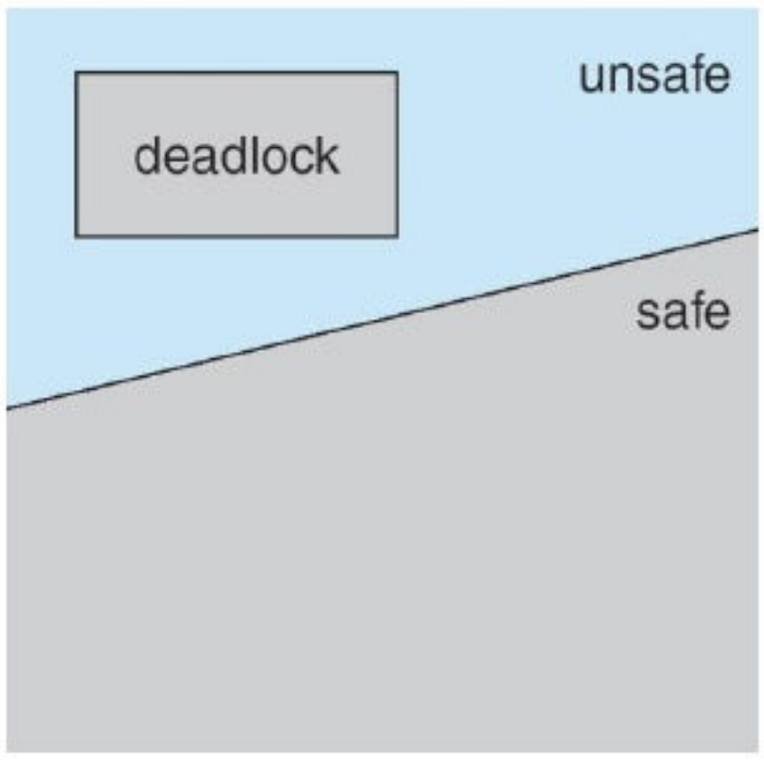
- Protocol # 1
    - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order
- Protocol # 2
    - Require that whenever a process requests an instance of a resource type, it has released resources with a lower no

Deadlock Avoidance

- The OS is given in advance additional info concerning which resources a process will request & use during its lifetime

Safe State

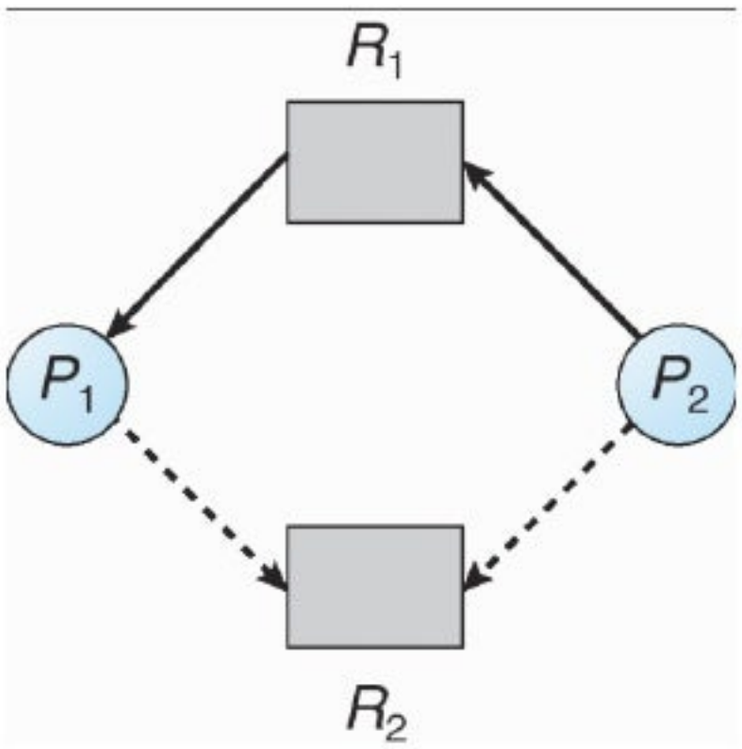
- A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock
- Safe sequence: <P1…Pn>, where all the resources that Pi may request are
    - Currently available, or
    - Held by a process Pj, where j < i
- If the resources are not immediately available, Pi can wait until all processes Pj, j < i, have completed
- When Pi finishes, Pi+1 may obtain its resources
- An unsafe state may (but not necessarily) lead to a deadlock
- Deadlocks are avoided by refusing any requests which lead to an unsafe state, so processes may wait for resources that are available, which may lead to sub-optimal resource utilization



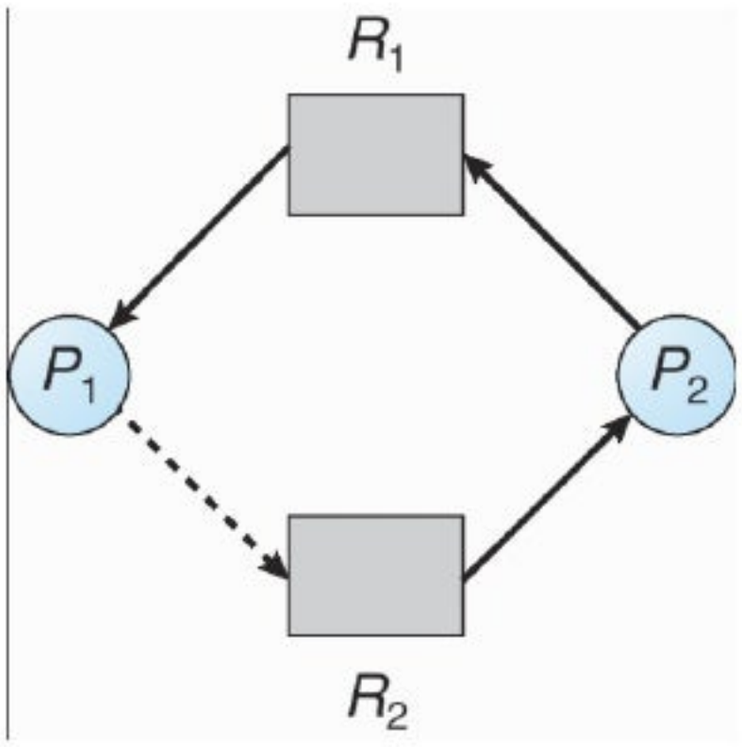
Resource-Allocation-Graph Algorithm

- *Claim edge Pi→Rj* indicated that process *Pj* may request resource *Rj*; represented by a dashed line

- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system
- Resource allocation graph:



- Unsafe state in resource allocation graph:



- Resource allocation graph algorithm:
  - Suppose that process *Pi* requests a resource *Rj*
  - The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- First use the Banker's **safety algorithm** to determine whether the system is currently in a safe state
- Then use the **resource-request algorithm** to check whether each of the given requests may be safely granted or not
- Each process has
  - Allocation vector: The no of each resource type allocated
  - Max vector: The maximum no of each resource to be used
  - Need vector: Outstanding resources (Max – Allocation)
- Available ('work') vector: Free resources over all processes

- Maximum resource vector: Allocation vectors + Available vector
- Finish vector: Indicates which processes are still running
- Step 1: Initialize the Finish vector to 0 (0 = false)
- Step 2: Search the array Need from the top to find a process needing fewer resources than those Available
- Step 3: Assume the process completes, and free its resources:
    - Add the resources to the Available vector
    - Subtract the resources from the Process' Allocation vector
    - Place 1 in the appropriate place in the Finish vector
- Continue until Finish contains only 1s

<u>Problems with the Banker's algorithm:</u>

- It requires a fixed number of resources to allocate
    - Resources may suddenly break down
    - Processes rarely know their max resource needs in advance
- It requires a fixed number of processes
    - The no of processes varies dynamically (users log in & out)

Safety Algorithm

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

  Work = Available

  Finish [*i*] =*false* for*i*= 0, 1, …, *n*-1.
- Find and *i* such that both: (2)

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

  (b) *Needi*≤*Work*

  If no such *i* exists, go to step 4.
- *Work= Work + Allocationi* *(3)*

  Finish[*i*] =*true*

  go to step 2
- If *Finish*[*i*] == true for all *i*, then the system is in a safe state (4)

Resource-Request Algorithm

- *Request*= request vector for process *Pi*. If *Requesti*[*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*.
- If *Requesti* ≤*Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim (1)
- If *Requesti*≤*Available*, go to step 3. Otherwise *Pi*must wait, since resources are not available (2)
- Pretend to allocate requested resources to *Pi*by modifying the state as follows: (3)

  Available= Available -Request;

  Allocationi= Allocationi+ Requesti;

  Needi=Needi-Requesti;

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

An Illustrative Example

- **Example of the 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	

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

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

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

- **Example: P1 Request (1,0,2)**:

Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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?

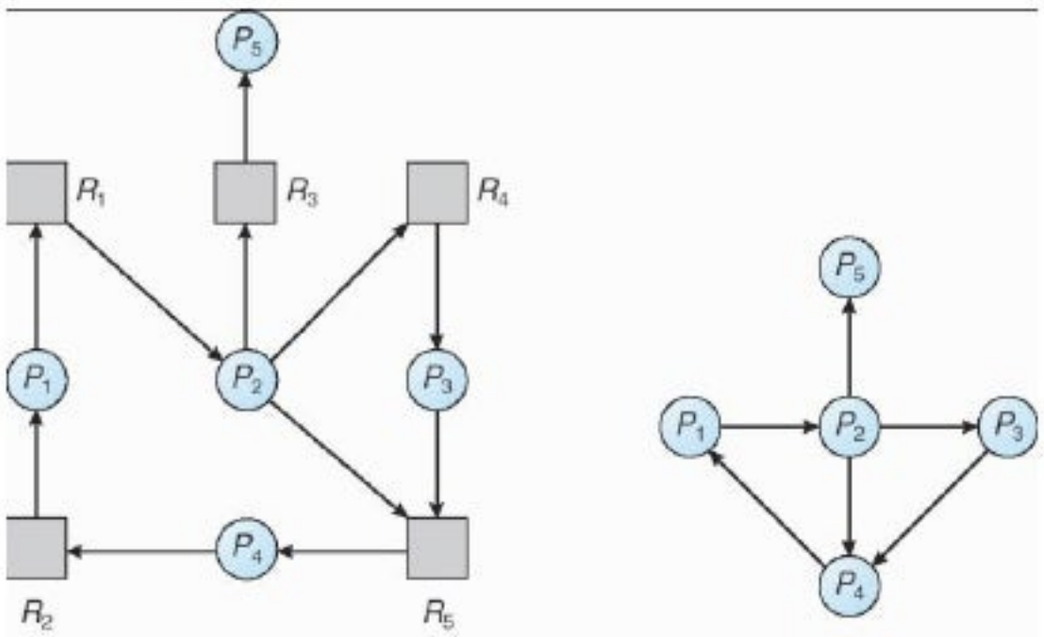
Deadlock Detection

- A detection algorithm determines if a deadlock has occurred
- An algorithm recovers from the deadlock

- **Advantage**:
    1. Processes don't need to indicate their needs beforehand
- **Disadvantages**:
    1. Detection-and-recovery schemes require **overhead**
    2. Potential **losses** inherent in recovering from a deadlock

Single Instance of Each Resource Type

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



Several Instances of a Resource Type

- Wait-for graph not applicable to a resource-allocation system with multiple instances of each resource type
- Here we make use of a deadlock detection algorithm which is applicable to such a system
- Data structures of the algorithm:
    - ***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* [*ij*] = *k*, then process *Pi* is requesting *k* more instances of resource type. *Rj*.
    - ***Detection algorithm:***
        - Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize: (1)

            (a) *Work* = *Available*

            (b) For *i* = 1,2, …,*n*, if *Allocationi* ≠0, then

            Finish[i] = false;otherwise, *Finish*[i] = *true*.
        - Find an index *i* such that both: (2)

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

(b) *Requesti ≤Work*

- If no such *i* exists, go to step 4
- *Work = Work + Allocationi* *(3)*

  Finish[*i*] = *true*

  go to step 2.
- If *Finish*[*i*] == false, for some *i*, 1 ≤*i*≤*n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then *Pi* is deadlocked (4)

Detection-Algorithm Usage

- The frequency of invoking the detection algorithm depends on:
  - How **often** a deadlock is likely to occur?
  - How **many processes** will be affected by deadlock when it happens?
- 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
- Every invocation of the algorithm adds to computation overhead

Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists,
  - The operator can deal with the deadlock manually
  - The system can recover from the deadlock automatically

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?

Resource Preemption

- **Selecting a victim**:
  - We must determine the order of preemption to minimize cost
  - Cost factors: no. of resources being held, time consumed…
- **Rollback**:
  - If we preempt a resource from a process, the process can't go on with normal execution because its missing a resource
  - We must roll back the process to a safe state & restart it

- **Starvation**:
    - In a system where victim selection is based primarily on cost factors, the same process may always be picked
    - To ensure a process can be picked only a small number of times, include the number of rollbacks in the cost factor

Summary

Memory Management

- For a program to be executed, it must be **mapped to absolute addresses** and loaded into memory
- As the program executes, it accesses program instructions and data from memory by generating these absolute addresses
- Eventually the program terminates and its memory space is declared available so the next program can be loaded & executed
- To improve CPU utilization, keep several programs in memory
- Selection of a **memory-management scheme** depends on many factors, especially the hardware design of the system
- The OS is responsible for these **memory-management** activities:
    - Keeping track of which parts of memory are currently being used and by whom
    - Deciding which processes are to be loaded into memory when memory space becomes available
    - Allocating and de-allocating memory space as needed

PART FOUR: MEMORY MANAGEMENT

Chapter 8: Memory-Management Strategies

Chapter Objectives:

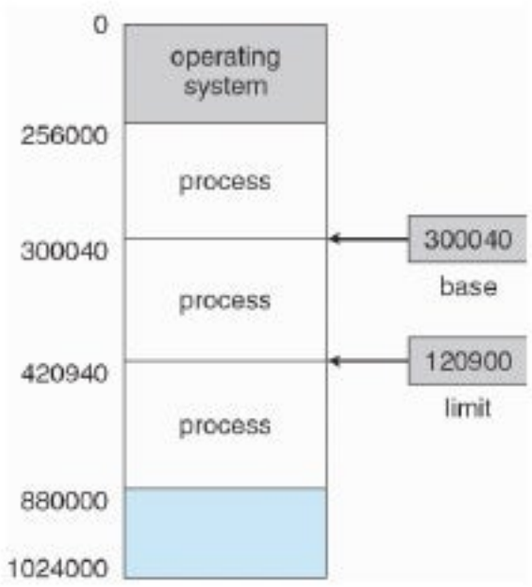
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

- The strategies in this chapter have all the same goal:
    - To keep many processes in memory simultaneously to allow multiprogramming
    - However, they require that an entire process be in memory before it can execute

Basic Hardware

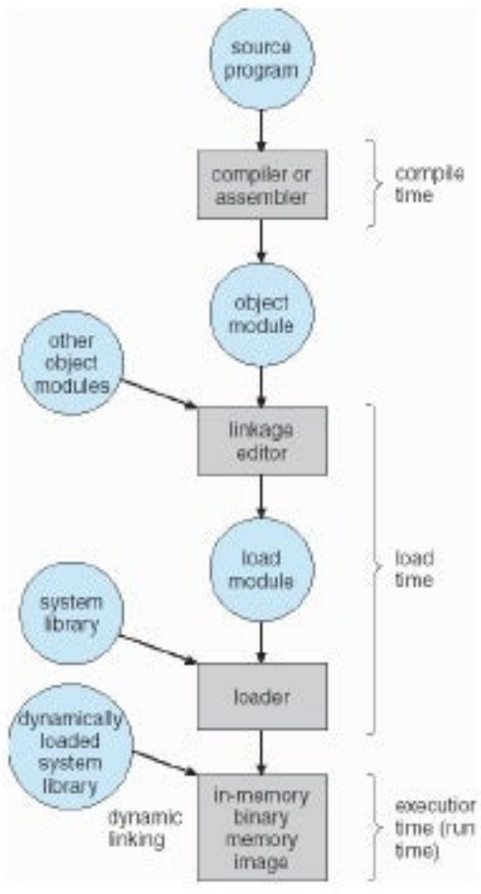
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
- A pair of **base** and **limit** registers define the **logical (virtual) address space**



- 420940 - 300040 = 120900 (logical address space)

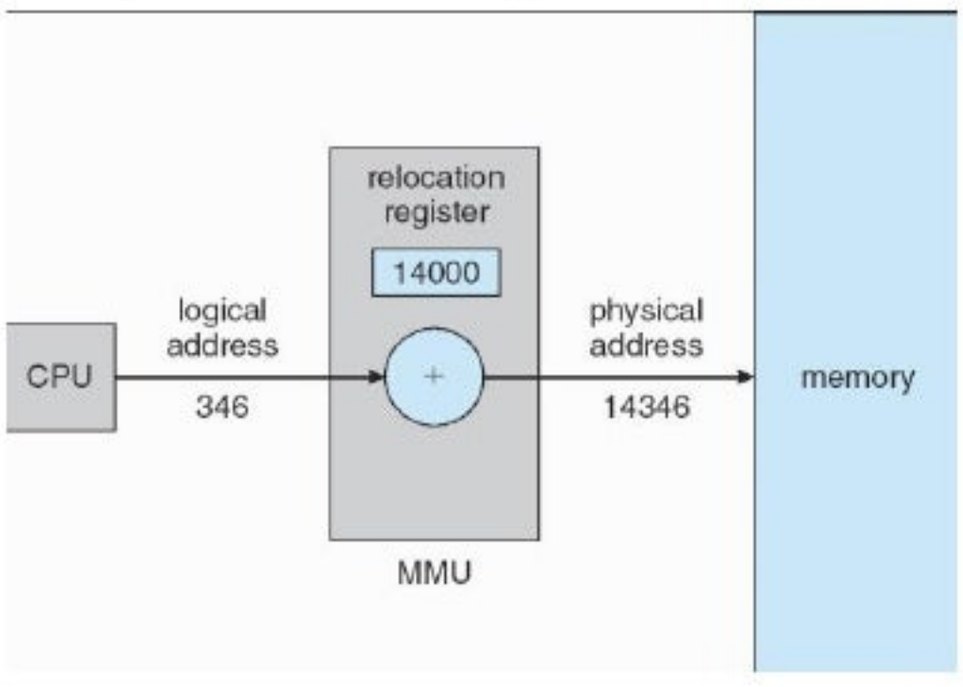
Address Binding

- Input queue = the collection of processes on the disk that is waiting to be brought into memory for execution
- Processes can normally reside in any part of the physical memory
- Addresses in the source program are generally symbolic (‘*count*’)
- A compiler binds these symbolic addresses to relocatable addresses (*’14 bytes from the beginning of this module’*)
- The linkage editor / loader binds these relocatable addresses to absolute addresses (*‘74014’*)
- 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)
- Steps a user program needs to go through (some optional) before being executed:



Logical versus Physical Address Space

- Logical address = one generated by the CPU
- Physical address = one seen by the memory unit, and loaded into the memory-address register of the memory
- The compile-time and load-time address-binding methods generate **identical** logical & physical addresses
- The execution-time address-binding scheme results in **differing** logical (= 'virtual') & physical addresses
- **Logical(virtual)-address space** = the set of all logical addresses generated by a program
- **Physical-address space** = the set of all physical addresses corresponding to these logical addresses
- Memory-management unit (MMU) = a hardware device that does the run-time mapping from virtual to physical addresses
- The **MMU**:
    - Hardware device that maps virtual to physical address
    - In MMU scheme, the value in the relocation (**base**) register is added to every address generated by a user process at the time it is sent to memory
    - The user program deals with *logical* addresses; it never sees the *real* physical addresses
- Dynamic relocation using a relocation register:



Dynamic Loading

- With dynamic loading, a routine is not loaded until it is called
- All routines are kept on disk in a relocatable load format
    - The main program is loaded into memory and is executed
    - When a routine needs to call another one, the calling routine first checks to see whether the other routine has been loaded
    - If not, the relocatable linking loader is called to load the desired routine into memory
    - Then, control is passed to the newly loaded routine
- **Advantage** of dynamic loading:
    - An unused routine is never loaded
- Dynamic loading doesn't require special support from the OS
- The user must design his programs to take advantage of this
- However, OS's may help the programmer by providing library routines to implement dynamic loading

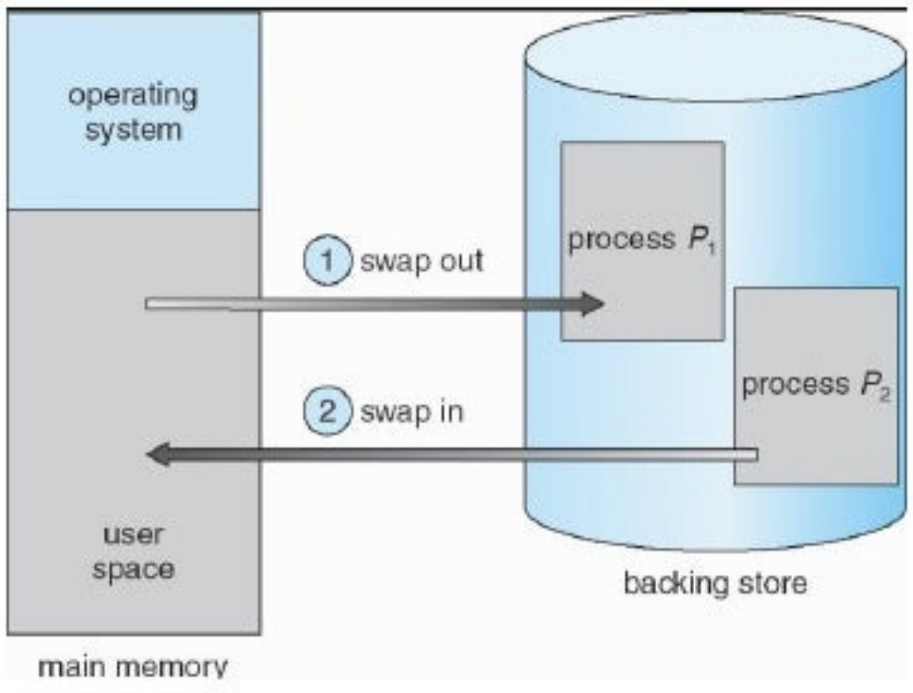
Dynamic Linking and Shared Libraries

- **Static linking**:
    - System language libraries are treated like any other object module and are combined by the loader into the binary program image
- ***Dynamic linking***:
    - Linking is postponed until execution time
    - Look at image in address binding!!! Also shows dynamic linking
- A **stub** is found in the image for each library-routine reference
    - **Stub**:
        - Code that indicates how to **locate** the memory-resident library routine, or how to **load** the library if the routine is not already in memory
- Either way, the stub replaces itself with the address of the routine, and executes the routine
- The next time that code segment is reached, the library routine is executed directly, with no cost for dynamic linking
- Under this scheme, all processes that use a language library execute only one copy of the library code
- Unlike dynamic loading, dynamic linking requires help from the OS:
    - If the processes in memory are protected from one another, then the OS is the only entity that can check to see whether the needed routine is in another process' memory space
- ***Shared libraries***:
    - A library may be replaced by a new version, and all programs that reference the library will automatically use the new one
- Version info is included in both program & library so that programs won't accidentally execute incompatible versions

Swapping

- p.322 - 324 TB
- 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**:
    - When a higher-priority process arrives a lower-priority process is swapped out, and then swapped back in when the higher-priority process finishes
- Major part of swap time is **transfer time**
    - Total transfer time is directly proportional to the amount of memory swapped
- Swapping requires a backing store (normally a fast disk)
- The backing store must be big enough to accommodate all copies of memory images for all users, and must provide direct access
- The system has a ready queue with all processes whose memory images are on the backing store or in memory and ready to run
    - The CPU scheduler calls the **dispatcher** before running a process

    - The dispatcher checks if the next process in queue is in memory
    - If not, and there is no free memory region, the dispatcher **swaps out** a process currently in memory and swaps in the desired one
    - It then reloads registers and transfers control to the process
- The context-switch time in such a swapping system is fairly high
- If we want to swap a process, it must be completely idle
- Schematic view of Swapping:

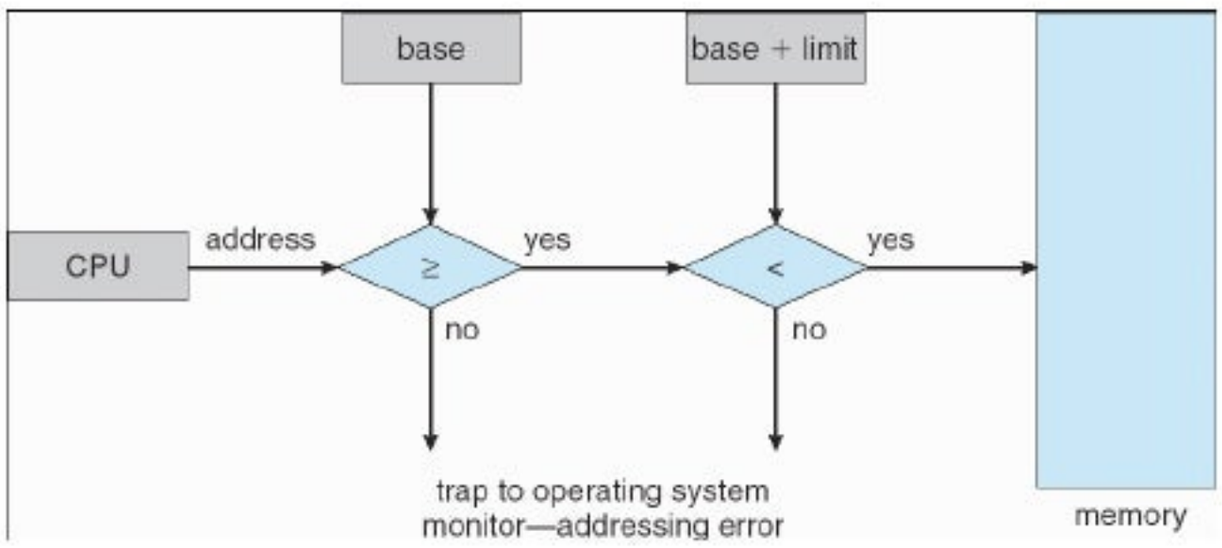


Contiguous Memory Allocation

- Memory is usually divided into two partitions:
    - One for the resident OS
    - One for the user processes
- The OS is usually placed in low or high memory (Normally low since interrupt vector is in low memory)
    - Affected by location of the interrupt vector
- **Contiguous memory allocation**:
    - Each process is contained in a single contiguous section of memory

Memory Mapping and Protection

- The OS must be protected from user processes, and user processes must be protected from one another
- Use a relocation register with a limit register for protection
- Relocation register contains the smallest **physical** address
- The limit register contains the range of **logical** addresses
- The memory-management unit (MMU) maps the logical address **dynamically** by adding the value in the relocation register
    - This mapped address is sent to memory
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation & limit registers
- Because every address generated by the CPU is checked against these registers, we can protect both the OS and the other users' programs & data from being modified by this running process
- HW address protection with base and limit registers:



- The relocation-register scheme provides an effective way to allow the **OS size to change** dynamically
- **Transient** OS code:
    - Code that comes & goes **as needed** to save memory space and overhead for unnecessary swapping

Memory Allocation

- A **simple method**: divide memory into fixed-sized **partitions**
    - Each partition may contain exactly one process
    - The degree of multiprogramming is bound by the no of partitions
        - When a partition is free, a process is selected from the input queue and is loaded into the free partition
        - When the process terminates the partition becomes available
    - (The above method is no longer in use)
- **Another method**: the OS keeps a table indicating which parts of memory are available and which are occupied
- Initially, all memory is available, and is considered as one large block of available memory, a **hole**
- When a process arrives, we search for a hole large enough
- If we find one, we allocate only as much memory as is needed
    - As processes enter the system, they are put into an input queue
    - When a process is allocated space, it is loaded into memory and can then compete for the CPU
    - When a process terminates, it releases its memory
- We have a list of **available block sizes** and the **input queue**
- The OS can order the input queue with a scheduling algorithm
- Memory is allocated to processes until, finally, there isn't a hole (block of memory) large enough to hold the next process
- The OS can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met
- In general, a set of holes of various sizes is scattered throughout memory at any given time
- When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process
- If the hole is too large, it is split up:
    - One part is allocated to the process, and the other is returned to the set of holes
- On process termination, the memory block returns to the hole set
- Adjacent holes are merged to form one larger hole

- Solutions to the dynamic storage allocation problem:
  - **First fit (Better and faster)**
    - Allocate the **first hole** that is **big enough**
  - **Best fit (Ok)**
    - Allocate the **smallest hole** that is **big enough**
  - **Worst fit (Not as good as the other two)**
    - Allocate the **largest hole**
- These algorithms suffer from **external fragmentation**:
  - Free memory space is broken into pieces as processes are loaded and removed

Fragmentation

- **External fragmentation**:
  - Exists when enough 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
- **Compaction** is a solution to the problem of external fragmentation
  - Free memory is shuffled together into one large block
  - Compaction is not always possible: if relocation is static and is done at assembly / load time, compaction cannot be done

As static relocation does not allow to change the content of memory.

- Simplest compaction algorithm:
  - Move all processes towards one end of memory, leaving one large hole of free memory (**Expensive, lots of overhead**)
- Another solution:
  - Permit the logical-address space of a process to be noncontiguous
  - Paging and segmentation allows this solution

Paging

- Permits the physical-address space of a process be **noncontiguous**
- Traditionally: support for paging has been handled by hardware
- Recent designs: the hardware & OS are closely integrated

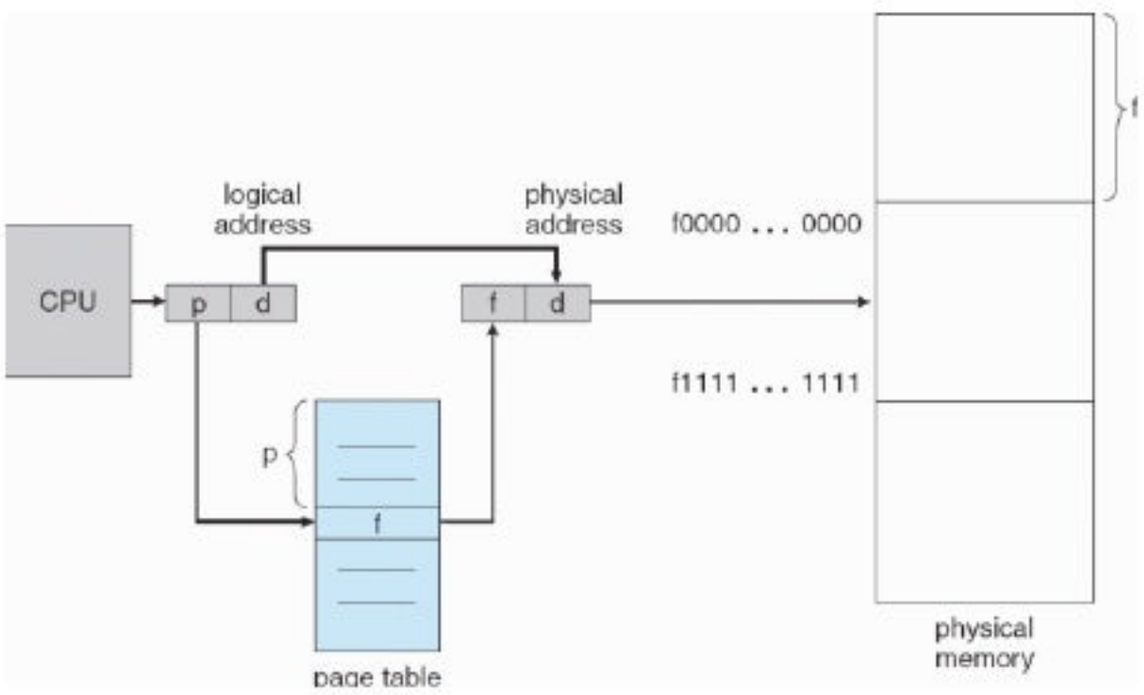
Basic Method

- **Physical memory** is broken into fixed-sized blocks called **frames**
- **Logical memory** is broken into same-sized blocks called **pages**
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store
- The backing store has blocks the same size as the memory frames
- Every address generated by the CPU is divided into 2 parts:
  - a page number (to index the page table)
  - a page offset

- The page table contains the base address of each page in memory
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit
- The page size, defined by the hardware, is usually a power of 2
- Paging schemes have no external, but some internal fragmentation
- Small page sizes mean less internal fragmentation
- However, there is less overhead involved as page size increases
- Also, disk I/O is more efficient with larger data transfers
    - When a process arrives in the system to be executed, its size, expressed in pages, is examined
    - (Noncontiguous) frames are allocated to each page of the process
- The frame table contains entries for each physical page frame, indicating which are allocated to which pages of which processes
- **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

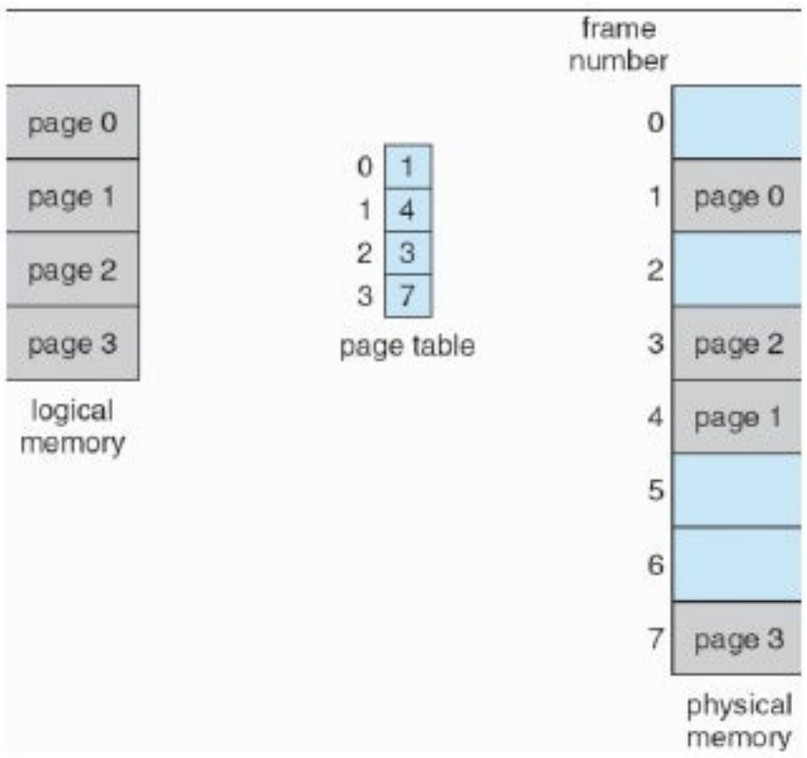
page number	page offset
p	*d*
m - n	*n*

    - For given logical address space 2^*m and page size* 2^n page
- An illustration of the hardware that supports paging:

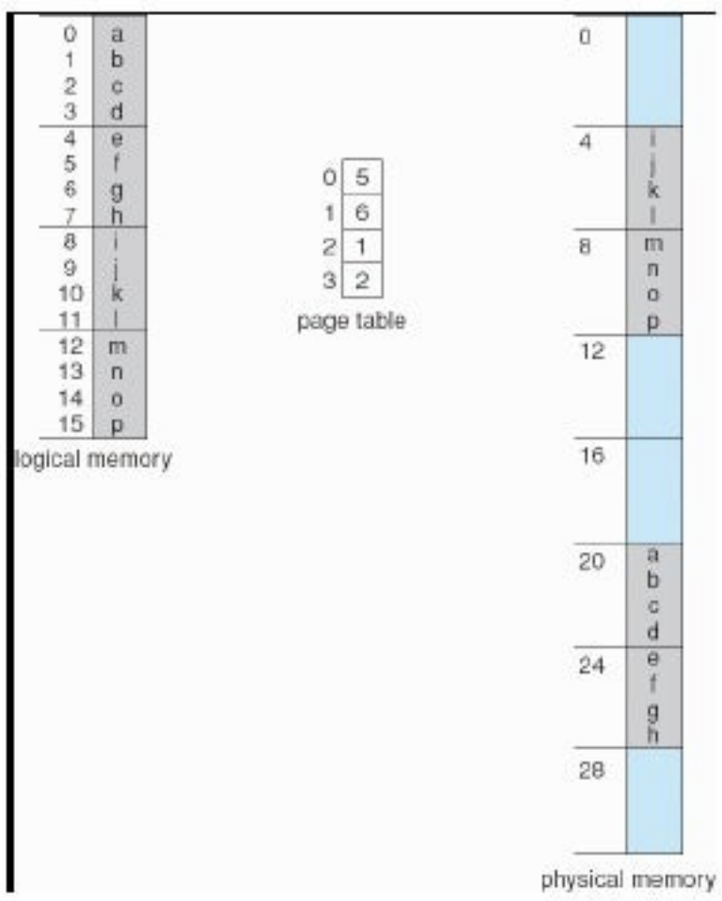


b

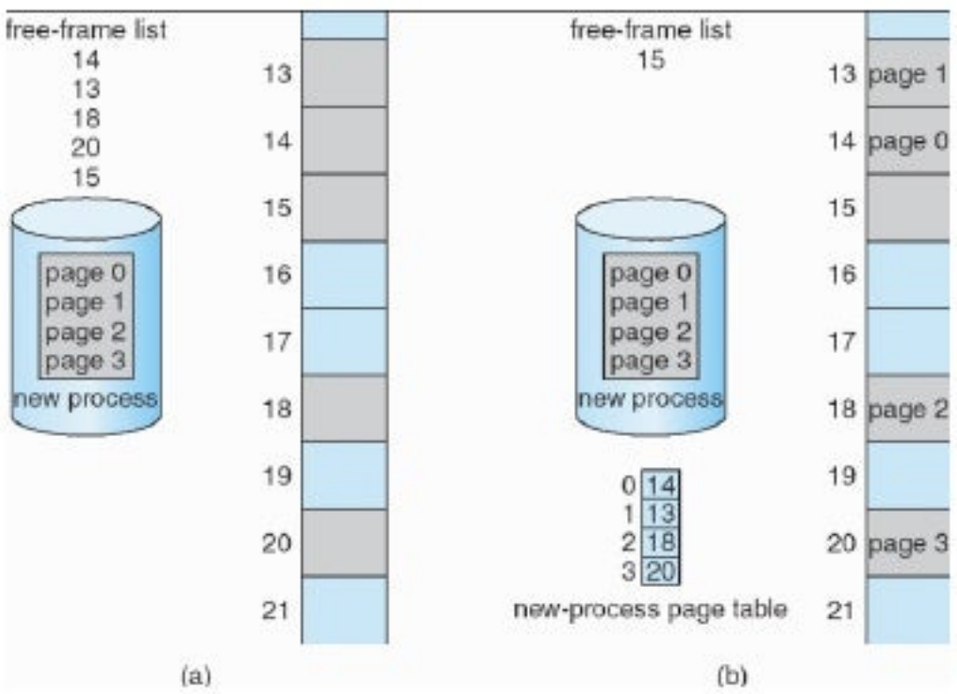
- The paging model of logical and physical memory:



- **Paging Example**: 32-bytee memory and 4-byte pages (p.330)



- With the arrival of new processes the following happens:



- a) Before allocation
- b) After allocation

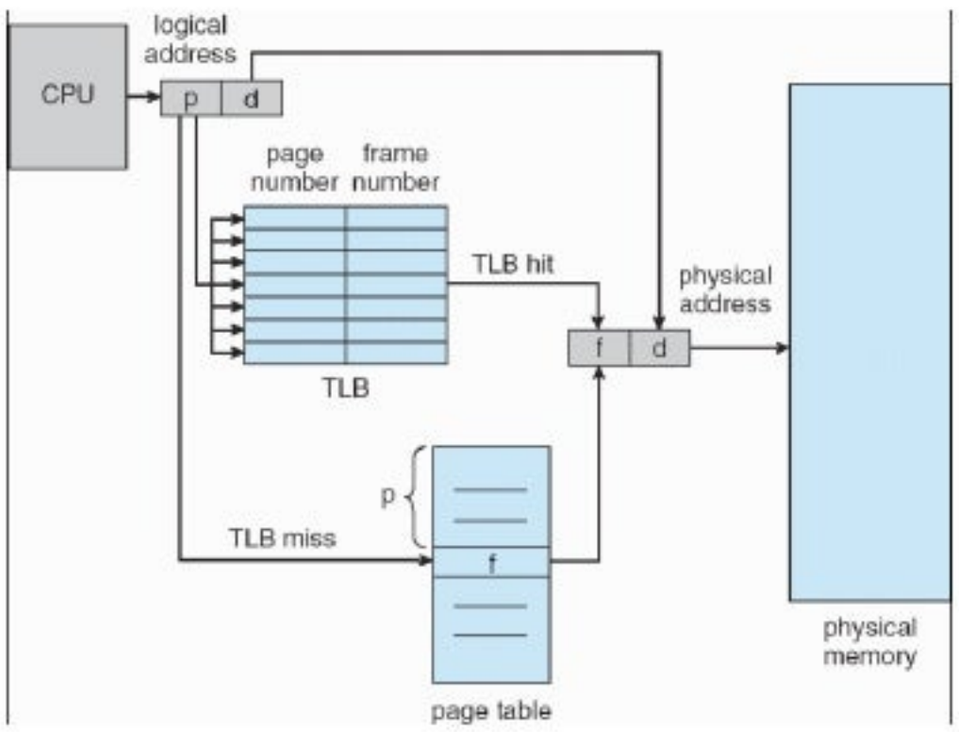
Hardware Support

- Most OS's store a **page table** for each process
- A pointer to the page table is stored in the PCB

- *Different ways for hardware implementation of the page table:*
  - The page table is implemented as a set of dedicated **registers**
    - The CPU dispatcher reloads these registers just like the others
    - Instructions to load / modify the page-table registers are privileged, so that only the OS can change the memory map
    - Disadvantage: works only if the page table is reasonably small
    - The page table is kept in memory, and a **page-table base register** (PTBR) points to the page table
    - Changing page tables requires changing only this one register, substantially reducing context-switch time
    - Disadvantage: two memory accesses are needed to access one byte
  - Use a small, fast-lookup hardware cache: the **translation look-aside buffer** (TLB)
    - The TLB is associative, high-speed memory
    - Each entry in the TLB consists of a key and a value
    - When the associative memory is presented with an item, it is compared with all keys simultaneously
    - If the item is found, the corresponding value field is returned
    - The search is fast, but the hardware is expensive
- The TLB is used with page tables in the following way:
  - When a logical address is generated by the CPU, its page number is presented to the TLB
  - If the page number is found, its frame number is immediately available and is used to access memory
  - If the page number is not in the TLB, a memory reference to the page table must be made
  - The obtained frame number can be used to access memory
  - If the TLB is full of entries, the OS must replace one
  - Some TLBs have wired down entries that can't be removed
  - Some TLBs store address-space identifiers (ASIDs) in each entry of the TLB, that uniquely identify each process and provide address space protection for that process

Process ka page number TBL ko dia jata hai suru mein phr agar use frame mil jae tou ok otherwise purane tarike se pagetable wala.

- Paging Hardware with TLB:



- The percentage of times that a particular page number is found in the TLB is called the hit ratio

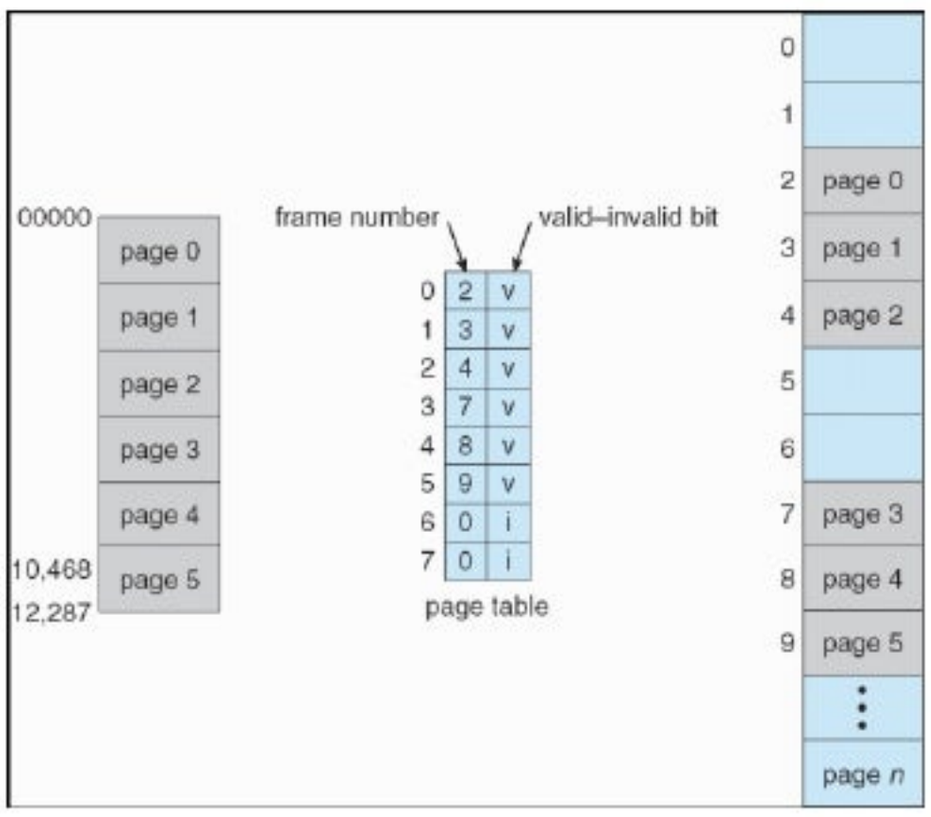
- **Effective access time**:
  - Associative Lookup = ε time unit
  - Assume memory cycle time is 1 microsecond
  - Hit ratio –percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
  - Hit ratio = ±
  - **Effective Access Time**(EAT)

    EAT = (1 + ε) α+ (2 + ε)(1 –α)

    = 2 + ε–α

Protection

- Memory protection is achieved by protection bits for each frame
- Normally, these bits are kept in the page table
- One bit can define a page to be read-write or read-only
- Every reference to memory goes through the page table to find the correct frame number, so the protection bits can be checked
- A valid-invalid bit is usually attached to each page table entry
  - 'Valid': the page is in the process' logical-address space
  - 'Invalid': the page is not in the process' logical-address space
- Illegal addresses are trapped by using the valid-invalid bit
- Valid (v) or Invalid (i) Bit in a Page Table:

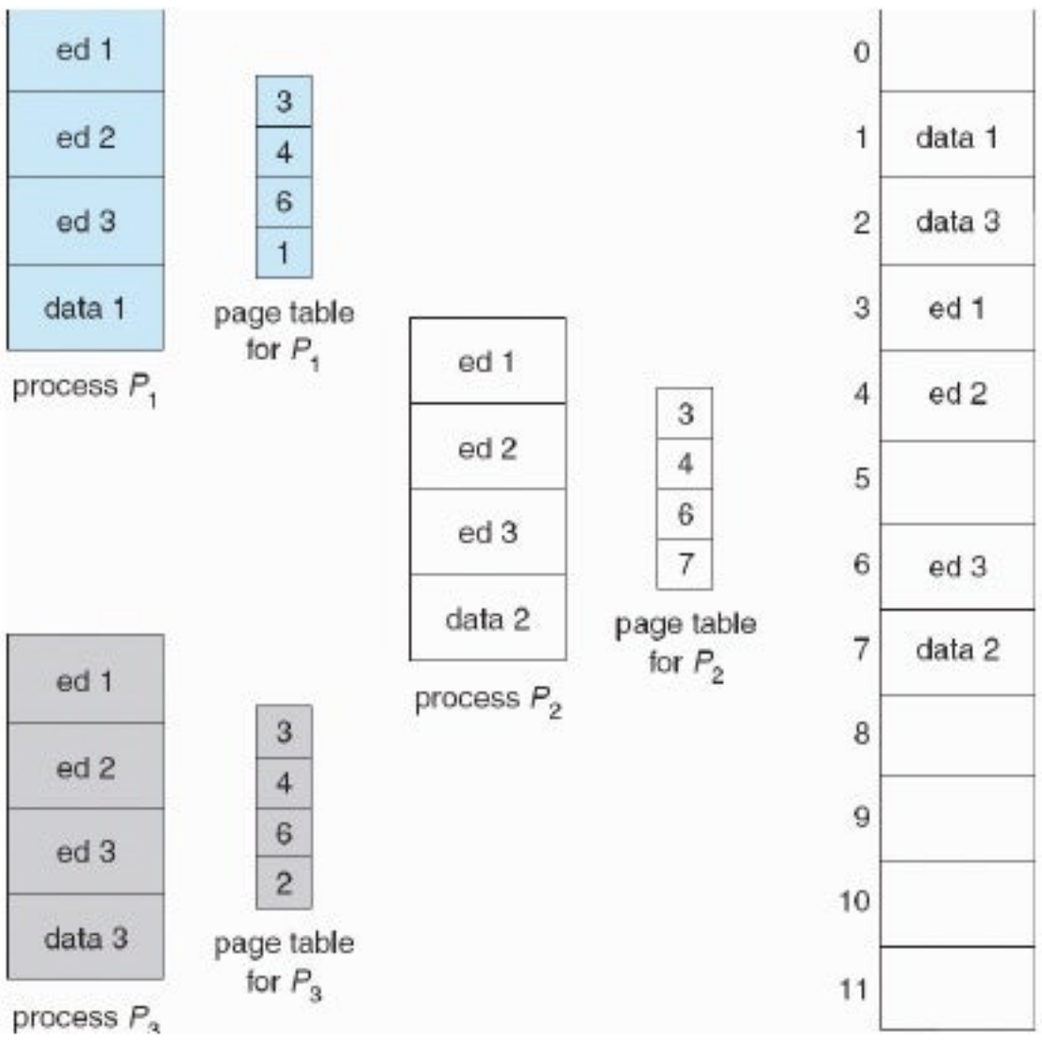


- Many processes use only a small fraction of the address space available to them, so it's wasteful to create a page table with entries for every page in the address range
- A page-table length register (PTLR) can indicate the size of the page table

Shared Pages

- Another advantage of paging: it is possible to **share common code**
- Reentrant code (pure code) = non-self-modifying code
- If the code is reentrant, then it never changes during execution
- Thus, two or more processes can execute the same code at once

- Each process has its own copy of registers and data storage to hold the data for the process' execution
- Only one copy of the editor needs to be kept in physical memory
- Each user's page table **maps onto the same physical copy** of the editor, but data pages are mapped onto different frames
- Systems that use inverted page tables have difficulty implementing shared memory
- **Shared code**
    - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
    - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
    - Each process keeps a separate copy of the code and data
    - The pages for the private code and data can appear anywhere in the logical address space
- Shared Pages Example:



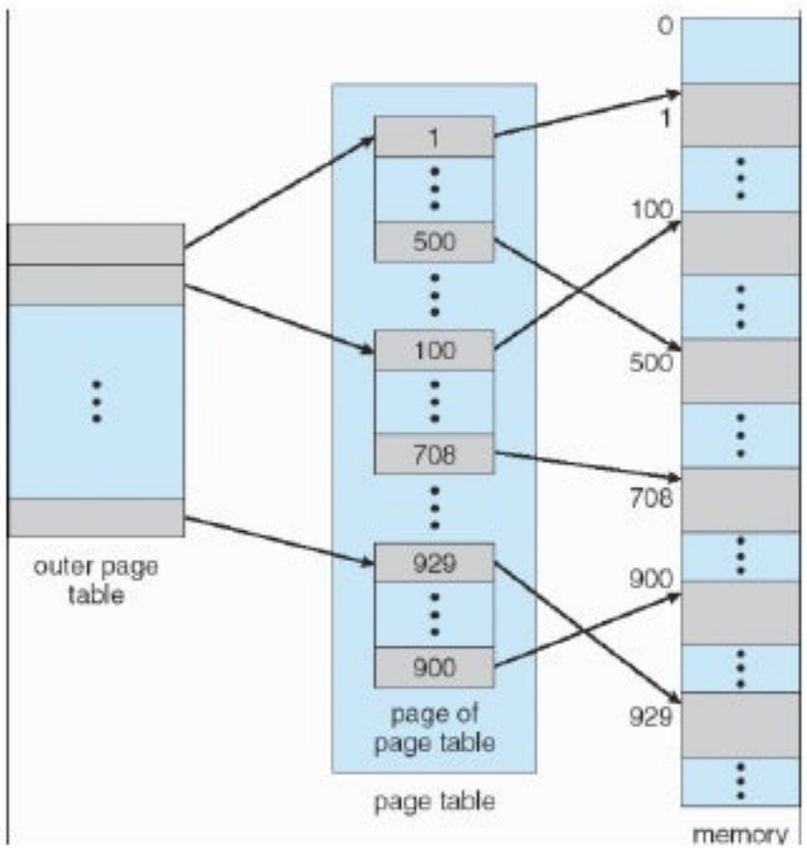
- ed1, 2, 3 are shared code pages, while each process has its own data page

Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Paging

- Two-level paging algorithm:
    - The page table is also paged
- Known as a forward-mapped page table because address translation works from the outer page table inwards
- Two-Level Page-Table Scheme:

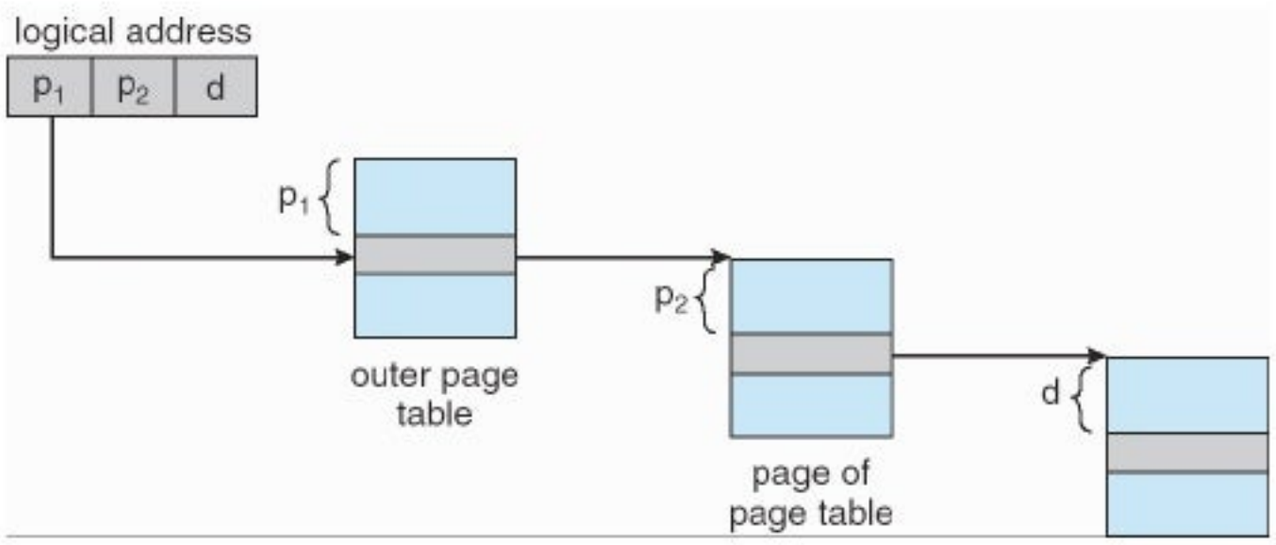


- Two-Level Paging **Example**:
  - A logical address (on 32-bit machine with 1K page size) is divided into:
    - a page number consisting of 22 bits
    - a page offset consisting of 10 bits
  - Since the page table is paged, the page number is further divided into:
    - a 12-bit page number
    -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

page number		page offset
p_i	p_2	d
12	10	10

    - where *pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table

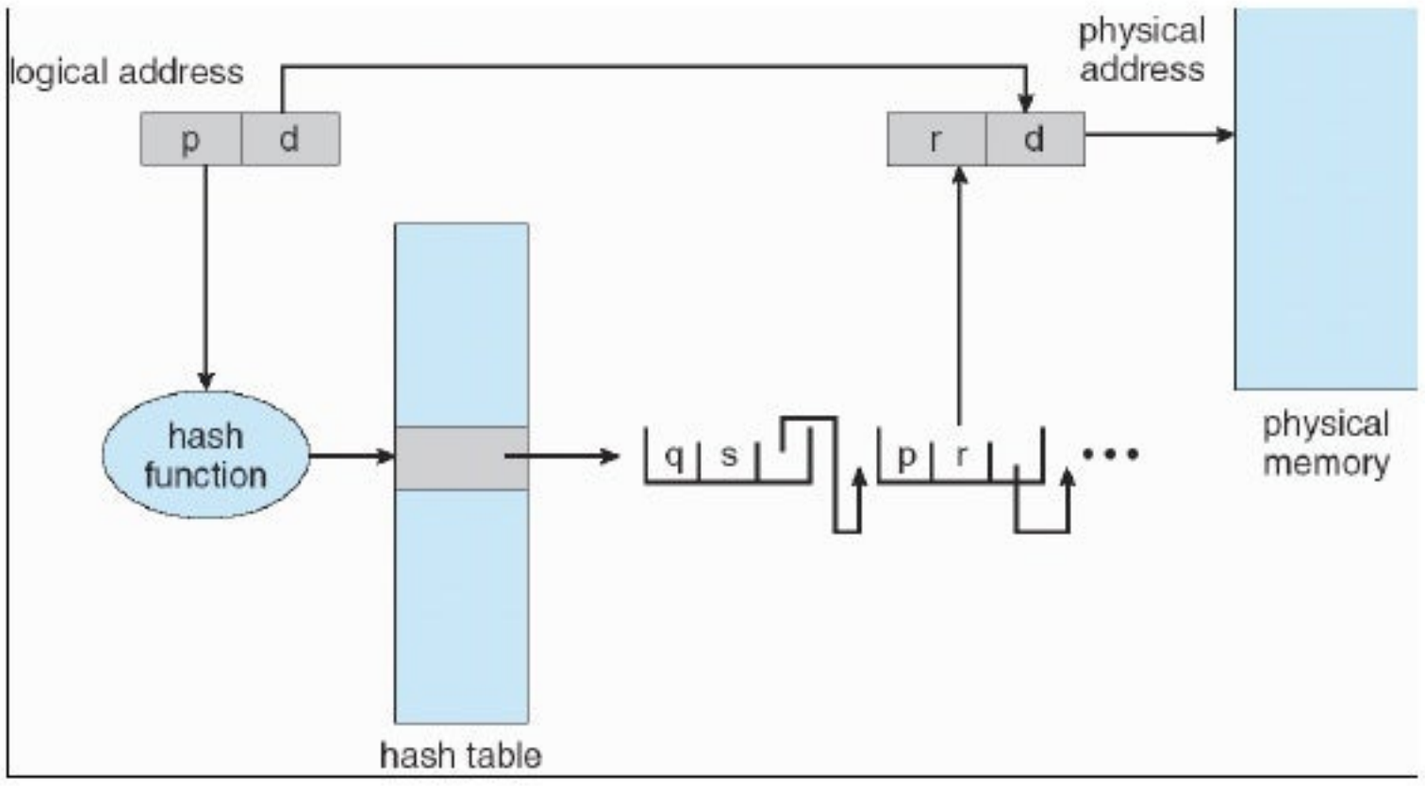
- Address-Translation Scheme:



Hashed Page Tables

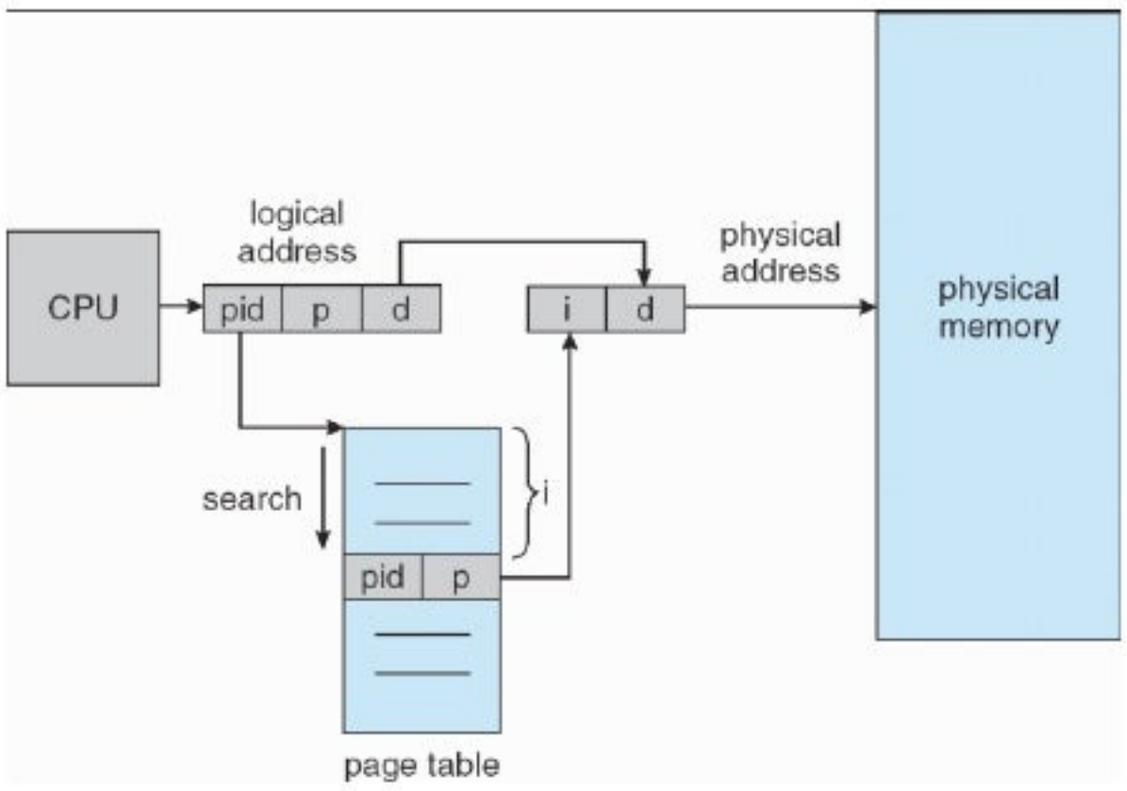
- Each entry in the hash table contains a linked list of elements that hash to the same location
- Each element consists of:
  - The **virtual page number**
  - The **value of the mapped page frame**

    - A **pointer to the next element** in the linked list
- The virtual page number is compared to field (a) in the first element in the linked list
- If there is a match, the corresponding page frame (field b) is used to form the desired physical address
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number
- Clustered page tables are similar to hashed page tables, except that each entry in the table refers to several pages
- Clustered page tables are particularly useful for sparse address spaces where memory references are noncontiguous and scattered throughout the address space



Inverted Page Tables

- 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



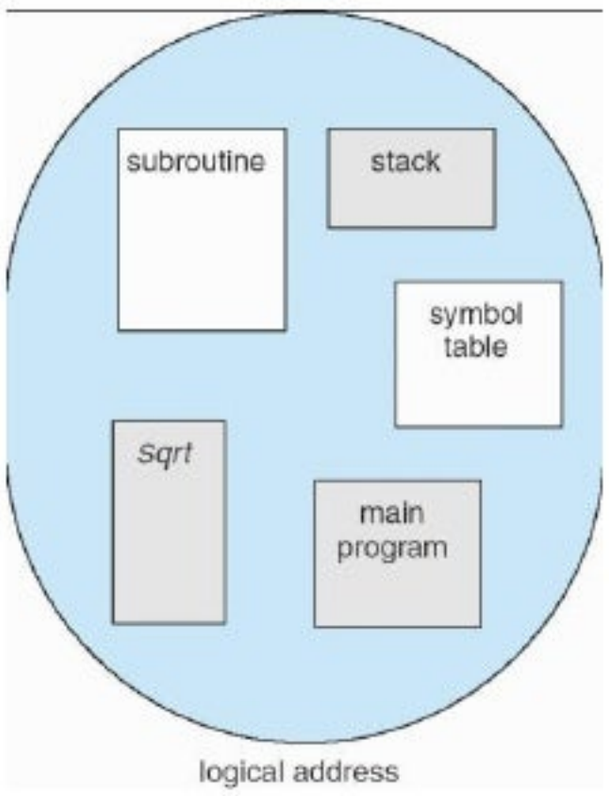
Segmentation

- Memory-management scheme that supports the **users' view** of memory
- A program is a collection of segments. A segment is a logical unit such as:
    - main program,

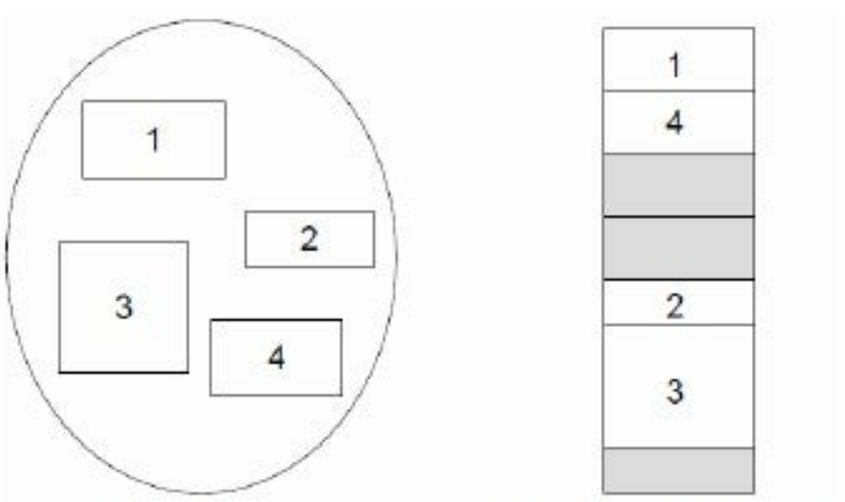
- procedure,
- function,
- method,
- object,
- local variables, global variables,
- common block,
- stack,
- symbol table, arrays

Basic Method

- Segmentation is a memory-management scheme that supports this user view memory
- A logical address space is a collection of segments
- The user's view of a Program:



- The user specifies each address by: a segment **name** and an **offset**
  - (Segments are *implemented* with numbers rather than names)
- Logical view of segmentation

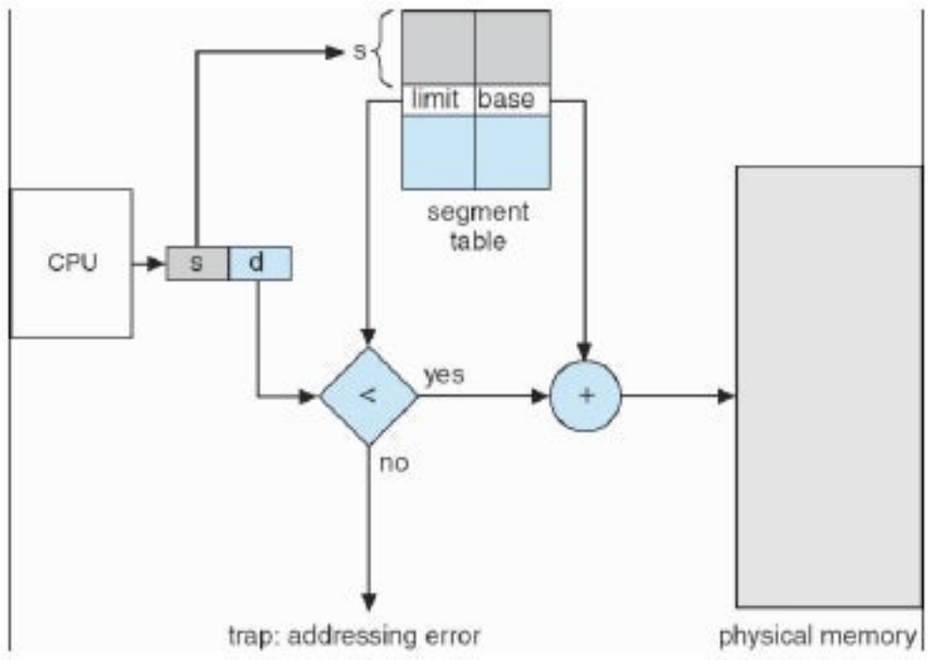


- When a program is compiled, segments are constructed for E.g.
  - The code
  - The global variables
  - The heap, from which memory is allocated
  - The stacks used by each thread

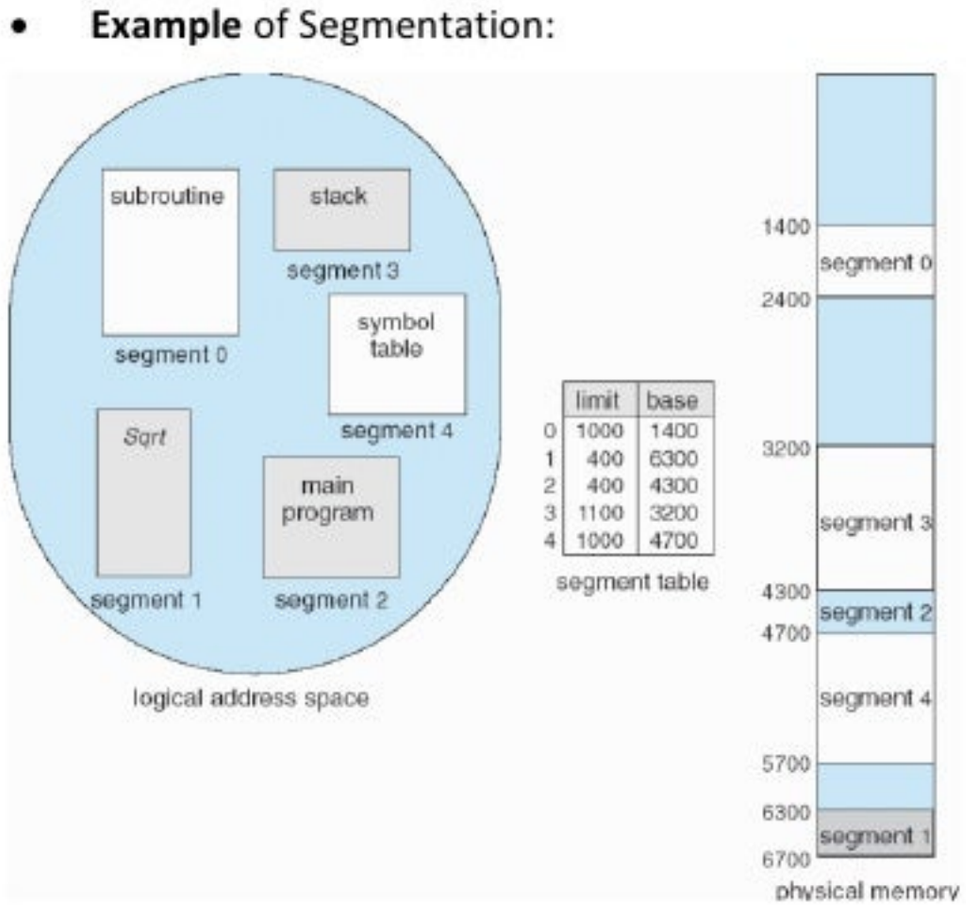
    - The procedure call stack, to store parameters
    - The code portion of each procedure or function
    - The local variables of each procedure and function
- The loader would take all these segments and assign them segment numbers

Hardware

- Although you can refer to objects by a two-dimensional address, the physical memory is still a one-dimensional sequence of bytes
- A segment table maps two-dimensional user-defined addresses into one-dimensional physical addresses
- Each entry of the table has a **segment base** and a **segment limit**



- **Example** of Segmentation:



Example: The Intel Pentium

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

Pentium Segmentation

Pentium Paging

Linux on Pentium Systems

Summary

Chapter 9: Virtual-Memory Management

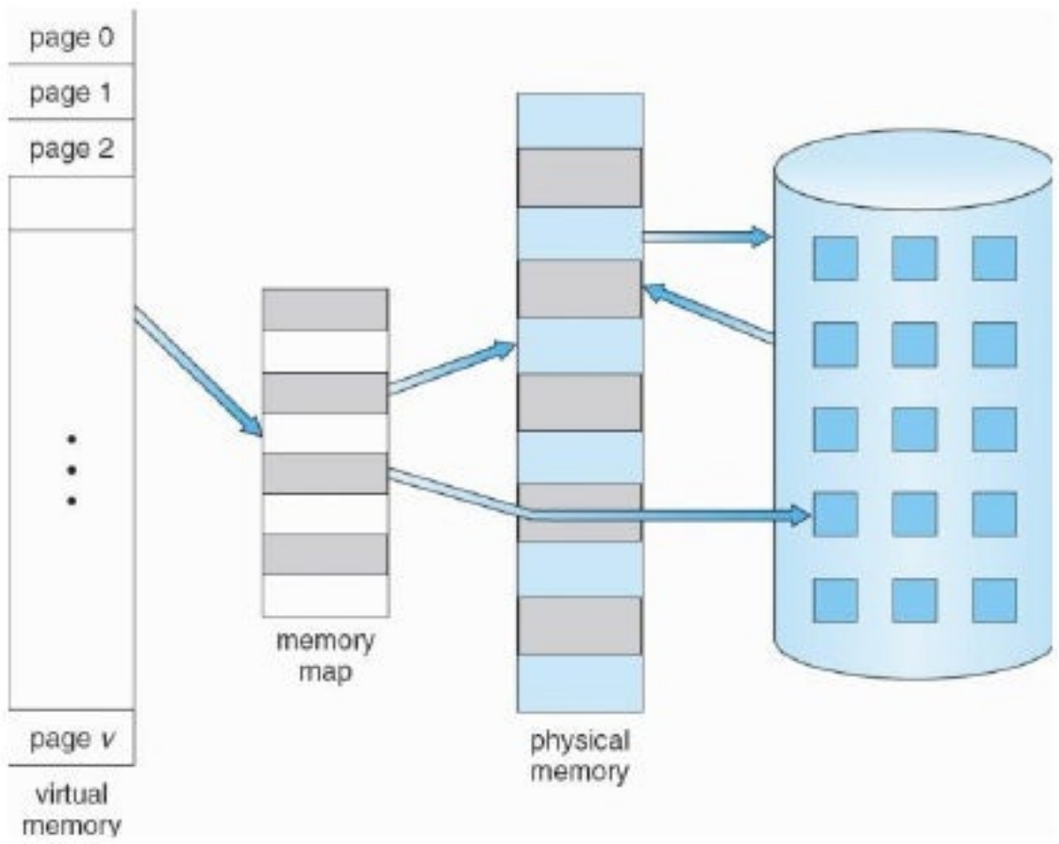
- Virtual memory is a technique that allows the execution of processes that are not completely in memory

Objectives:

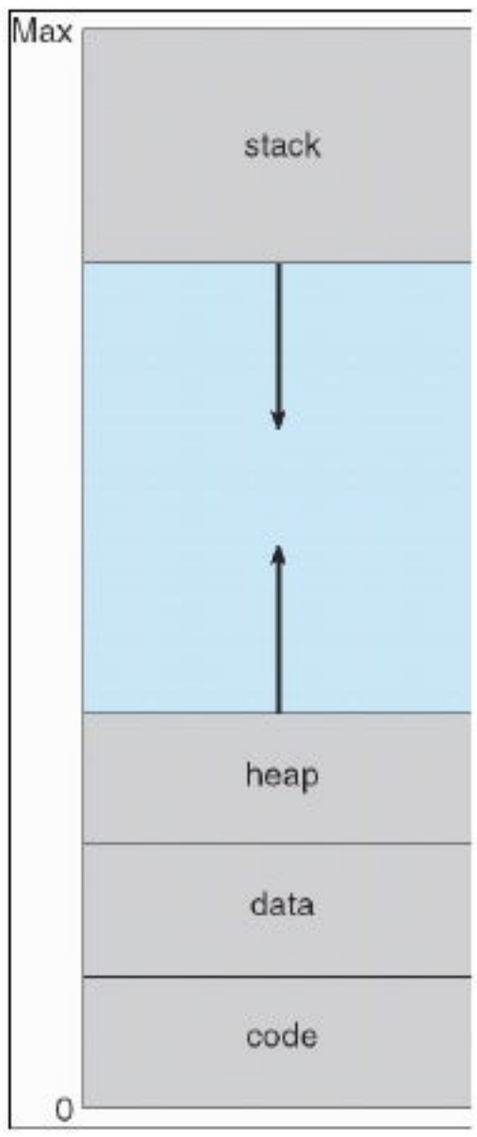
- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Background

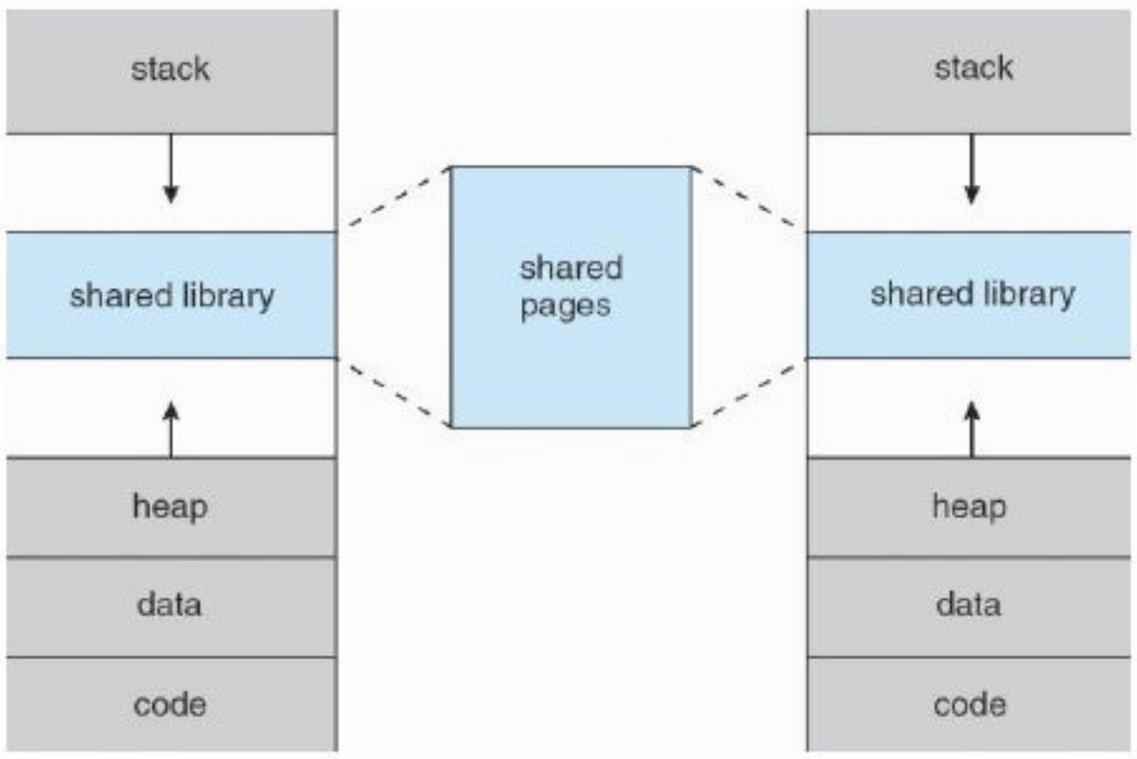
- In many cases, the entire program is not needed:
    - Unusual error conditions are almost never executed
    - Arrays & lists are often allocated more memory than needed
    - Certain options & features of a program may be used rarely
- Benefits of executing a program that is only partially in memory
    - **More programs** could be run at the same time
    - Programmers could write for a **large** virtual-address space and need no longer use overlays
    - **Less I/O** would be needed to load / swap programs into memory, so each user program would **run faster**
- **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 by:
    - Demand paging
    - Demand segmentation
- Diagram showing virtual memory that is larger than physical memory:



- The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory
    - Typically, this view is that a process begins at a certain logical address - say, address 0 - and exists in contiguous memory

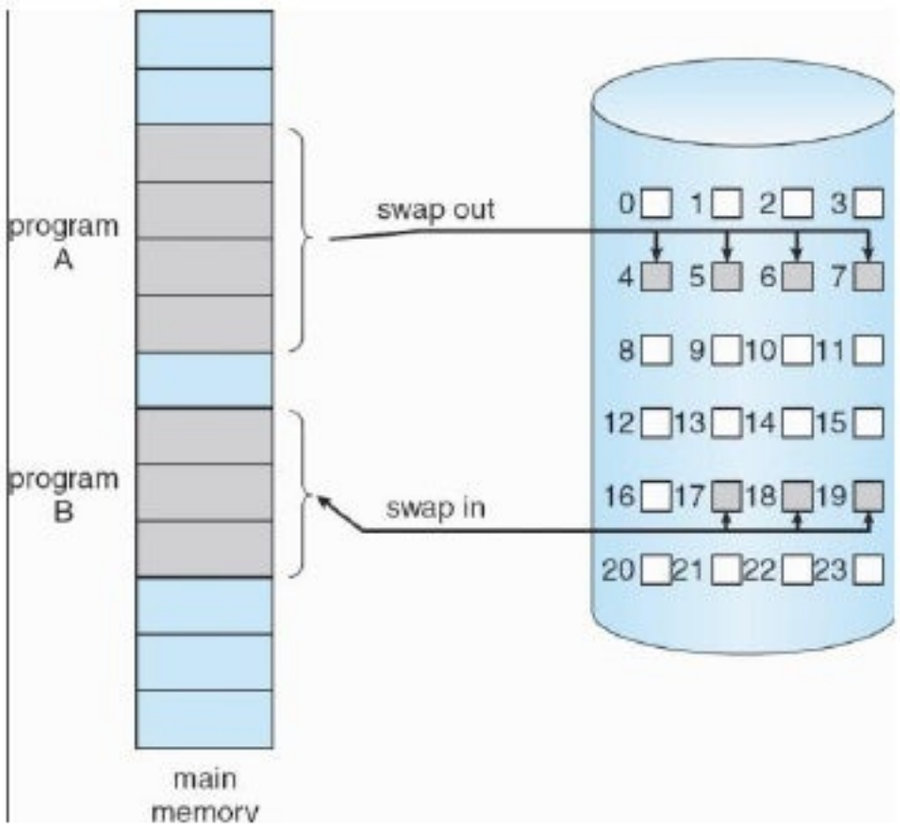


    - We allow the for the heap to grow upward in memory as it is used for dynamic memory allocation
- Virtual memory allows files and memory to be shared by two of more processes through page sharing
    - **Benefits**:
        - System libraries can be shared by several processes through mapping of the shared object into virtual address space
        - Virtual memory enables processes to share memory
            - From chapter 3 we learned that two or more processes can communicate through the use of shared memory
        - Virtual memory can allow pages to be shared during process creation with the fork() system call, speeding up process creation



Demand Paging

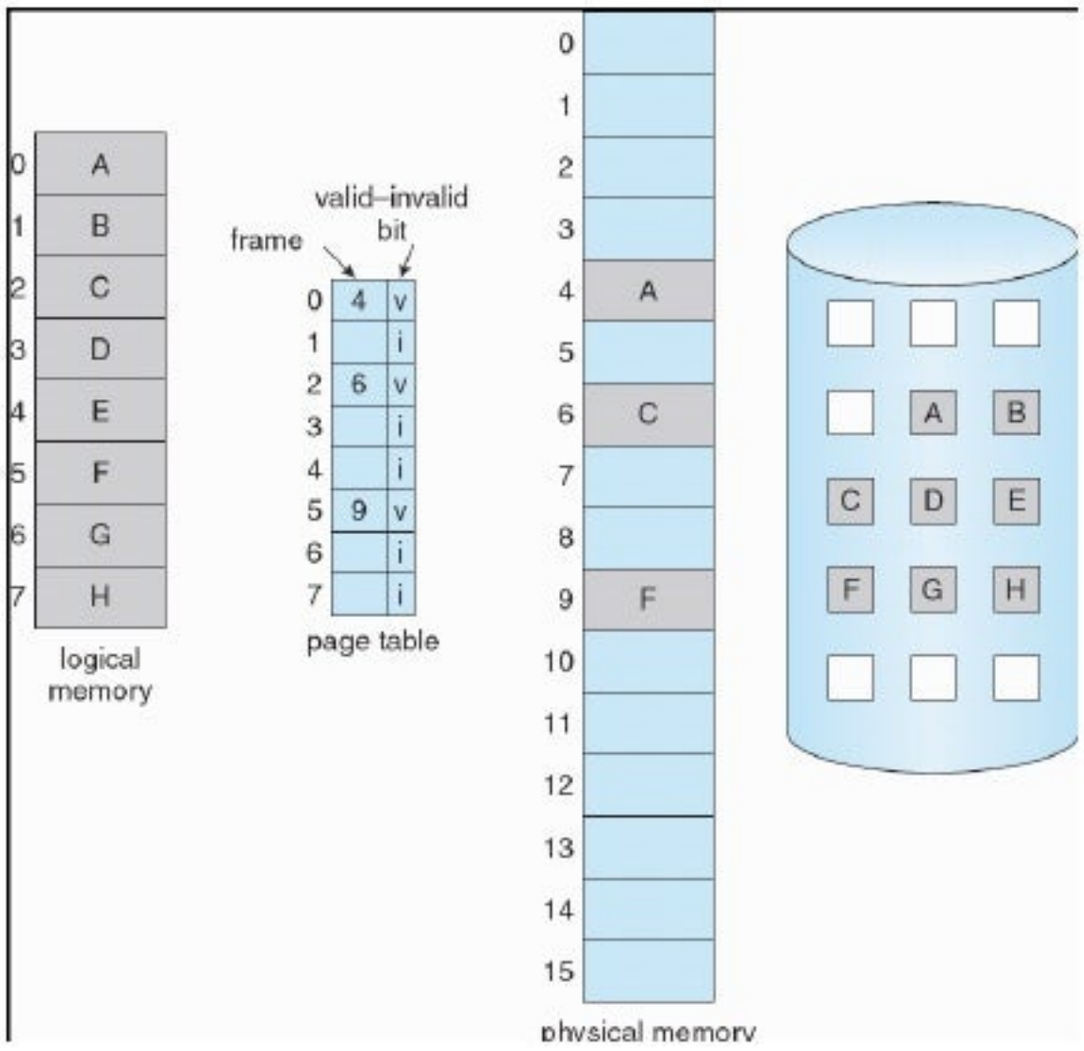
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper**-never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually on disk)



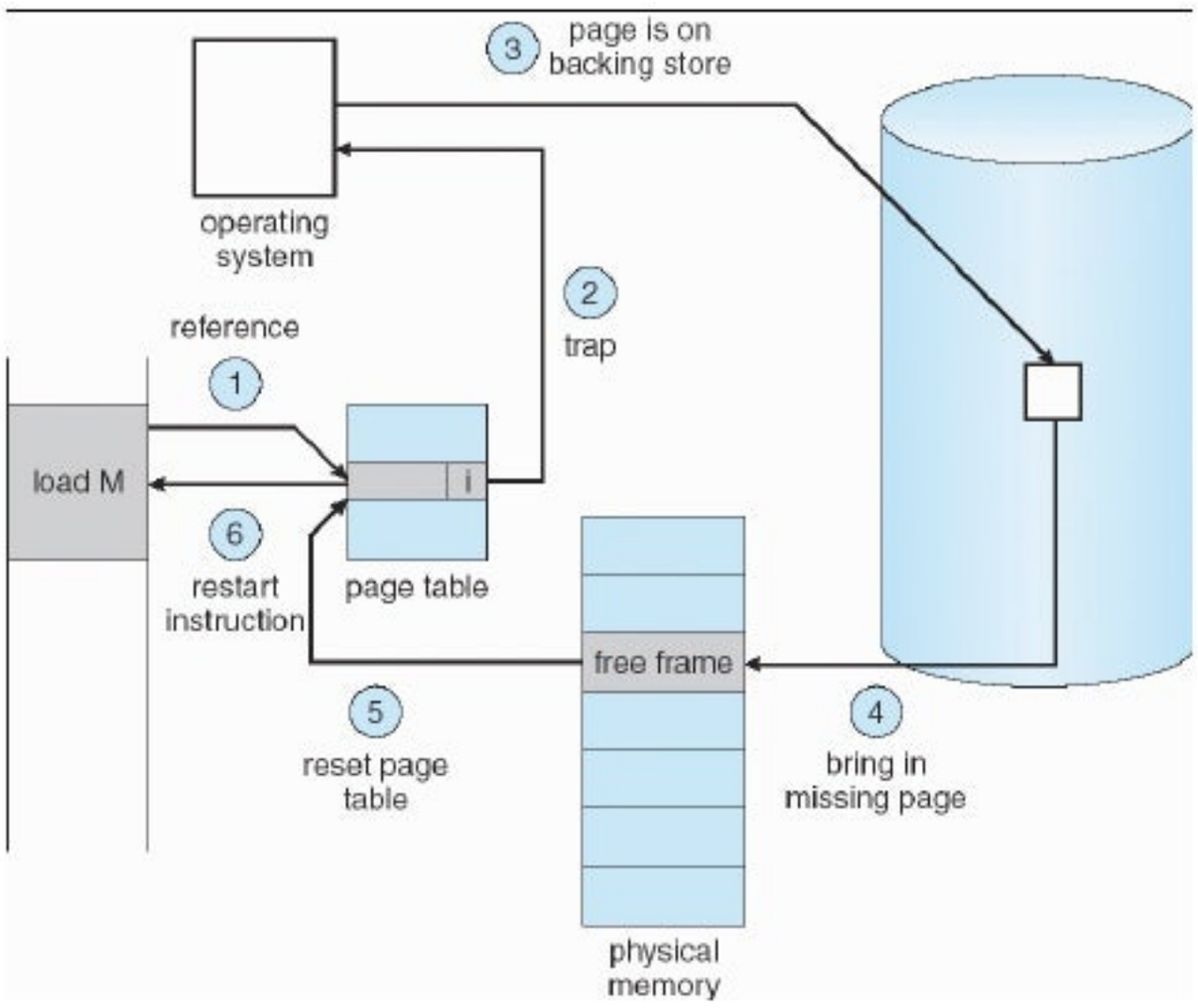
Basic Concepts

- Hardware support to distinguish pages in memory / pages on disk:
  - **Valid** bit:
    - The page is both legal and in memory

- **Invalid** bit:
  - The page is either not valid, or valid but currently on the disk



- The process executes and accesses pages that are memory resident
- If the process tries to access a page that was not brought into memory (i.e. one marked 'invalid'), a page-fault trap is caused
- Procedure for **handling page faults**:
  - Check an internal table to determine whether the reference was a valid / invalid memory access
  - Invalid reference terminate the process; if it was valid, but we have not yet brought in that page, page it in
  - Find a free frame (by taking one from the free-frame list)
  - Read the desired page into the newly allocated frame
  - Modify the internal table and the page table to indicate that the page is now in memory
  - Restart the instruction that was interrupted by the illegal address trap at the same place



- **Pure demand paging**:
    - Never bring pages into memory until required
- Some programs may access several new pages of memory with each instruction, causing multiple page faults and poor performance
- Programs tend to have locality of reference, so this results in reasonable performance from demand paging
- Hardware to support demand paging:
    - **Page table:**
        - Can mark an entry invalid through valid - invalid bit
    - **Secondary memory:**
        - Holds pages that are not present in main memory
        - Known as the swap device, and has a swap space (high-speed disk)
- Architectural software constraints:
    - Instructions must be able to be restarted after page faults

Performance of Demand Paging

- p365-367 TB
- Demand paging can significantly affect the performance of a computer system
    - We compute the effective access time for a demand-paged memory
- Page Fault Rate $0 \leq p \leq 1.0$
    - if $p = 0$ no page faults
    - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 -p) x memory access

  \+ p(page fault overhead

  \+ swap page out

+ swap page in

+ restart overhead)

- **Example**:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - EAT = (1 -p) x 200 + p (8 milliseconds)

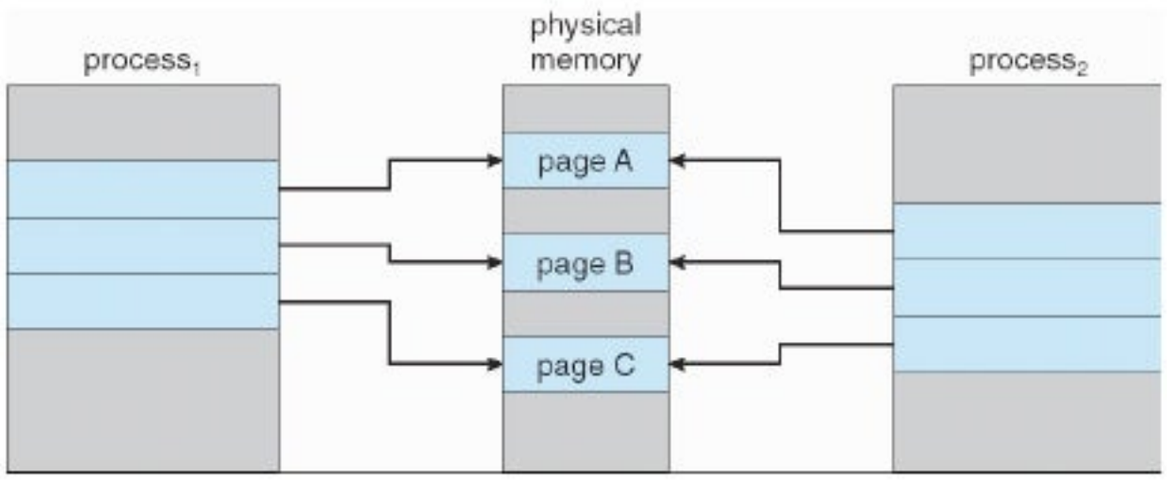
    = (1 -p x 200 + p x 8,000,000

    = 200 + p x 7,999,800
  - If one access out of 1,000 causes a page fault, then

    EAT = 8.2 microseconds.
  - This is a slowdown by a factor of 40!!

Copy-on-Write

- p.369 TB
- Virtual memory allows other benefits during **process creation**:
  - Copy-on-Write
  - Memory-Mapped Files (later)
- 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
- Before process 1 modifies page C

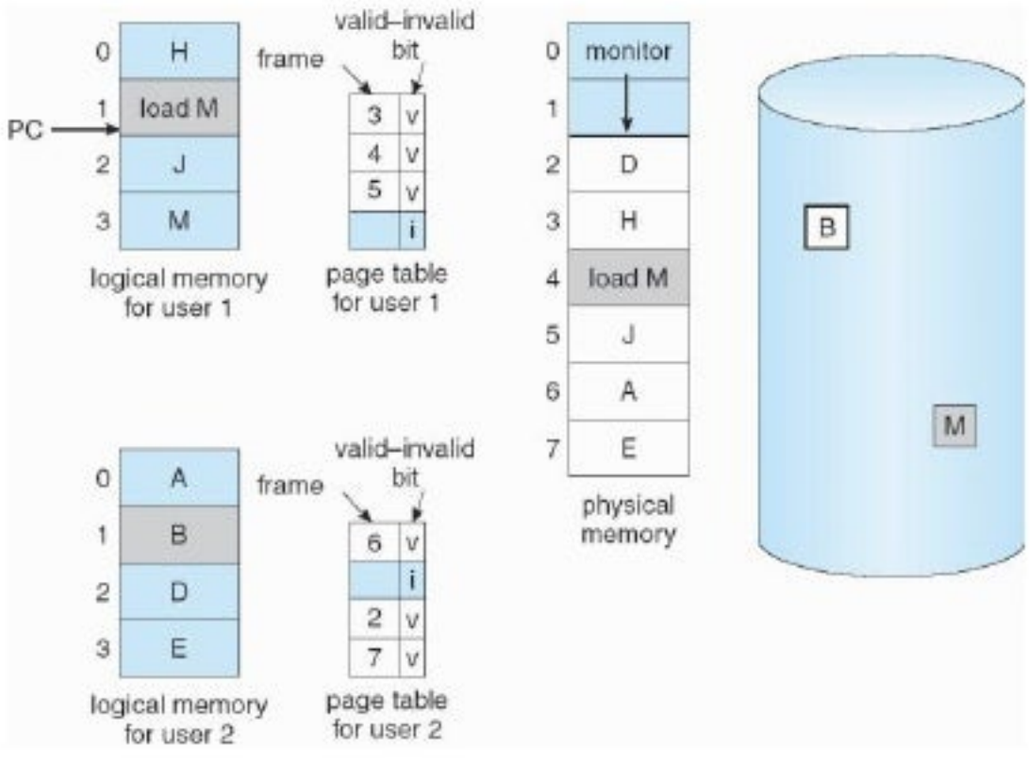


- After process 1 modifies page C there will also be a Copy of page C in physical memory (p.368 bottom)
- **If there is no free frame, the following happens**:
  - Page replacement - find some page in memory, but not really in use, swap it out
    - algorithm
    - performance - want an algorithm which will result in minimum number of page faults
  - Same page may be brought into memory several times

Page Replacement

- If we increase our degree of multiprogramming, we are over-allocating memory:
  - While a process is executing, a page fault occurs

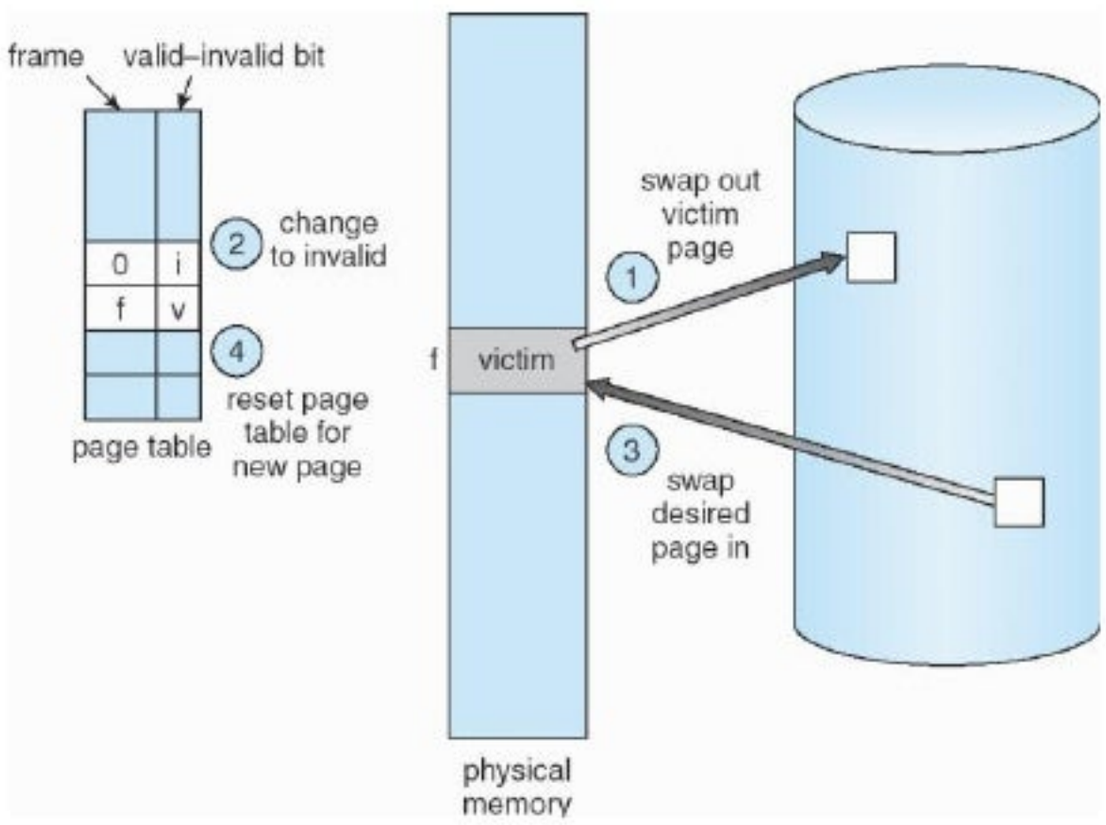
- The hardware traps to the OS, which checks its internal tables to see that this page fault is a genuine one
- The OS determines where the desired page is residing on disk, but then finds **no free frames** on the free- frame list
- The OS then could:
    - Terminate the user process (Not a good idea)
    - Swap out a process, freeing all its frames, and reducing the level of multiprogramming
    - Perform page replacement
- The need for page replacement arises:



- 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

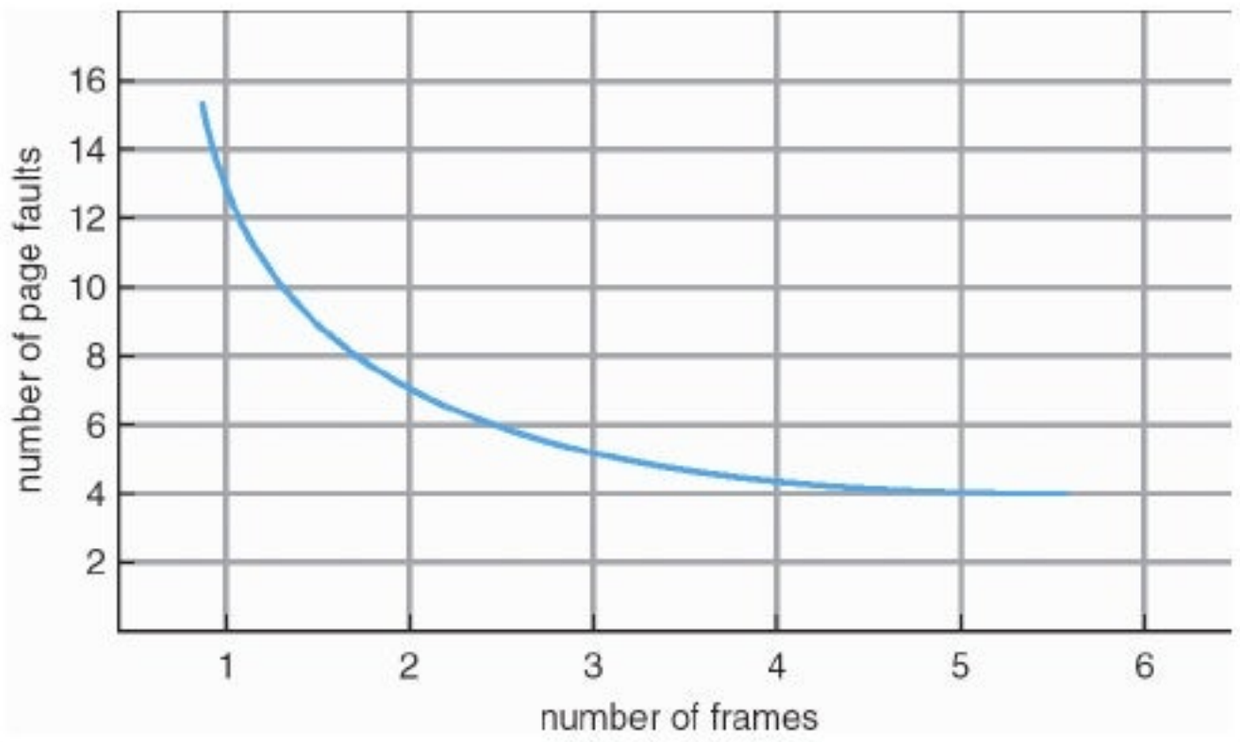
Basic Page Replacement

- Basic page replacement approach:
    - If no frame is free, we find one that is not being used and free it
- Page replacement takes the following steps:
    - Find the location of the desired page on the disk
    - Find a free frame:
        - If there is a free frame, use it, else
        - Select a **victim frame** with a page-replacement algorithm
        - Write the victim page to the disk and change the page & frame tables accordingly
    - Read the desired page into the (newly) free frame and change the page & frame tables
    - Restart the user process



- **Note**: if no frames are free, two page transfers (one out & one in) are required, which doubles the page-fault service time and increases the effective access time accordingly
- We can reduce this overhead by using a **modify / dirty bit**:
  - When a page is modified, its modify bit is set
  - If the bit is set, the page must be written to disk
  - If the bit is not set, you don't need to write it to disk since it is already there, which reduces I/O time
- We must solve two major problems to implement demand paging:
  - Develop a frame-allocation algorithm
    - If we have multiple processes in memory, we must decide how many frames to allocate to each process
  - Develop a page-replacement algorithm
    - When page replacement is required, we must select the frames that are to be replaced
- When selecting a particular algorithm, we want the one with the lowest page-fault rate
- To evaluate an algorithm, run it on a reference string (a string of memory references) and compute the number of page faults
  - 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
- You can **generate** reference strings or trace a **given system** and record the address of each memory reference
- Graph of page faults versus the number of frames:



FIFO Page Replacement

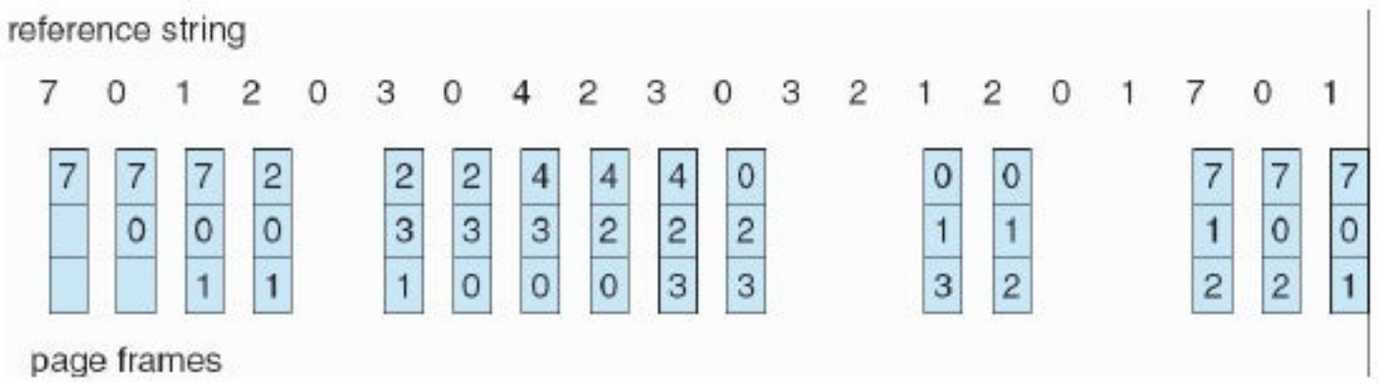
- p.373 TB
- The simplest page-replacement algorithm is the first-in, first-out (FIFO) algorithm
    - A FIFO replacement algorithm associates with each page the time when that page was brought into memory
    - When a page must be replaced, the oldest page is chosen
    - Notice it is not strictly necessary to record the time when a page is brought in
    - We can create a FIFO queue to hold all pages in memory
    - We replace the page at the head of the queue
    - When a page is brought into memory, we insert it at the tail of the queue
- Easy to understand and implement
- **Example**:
    - 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

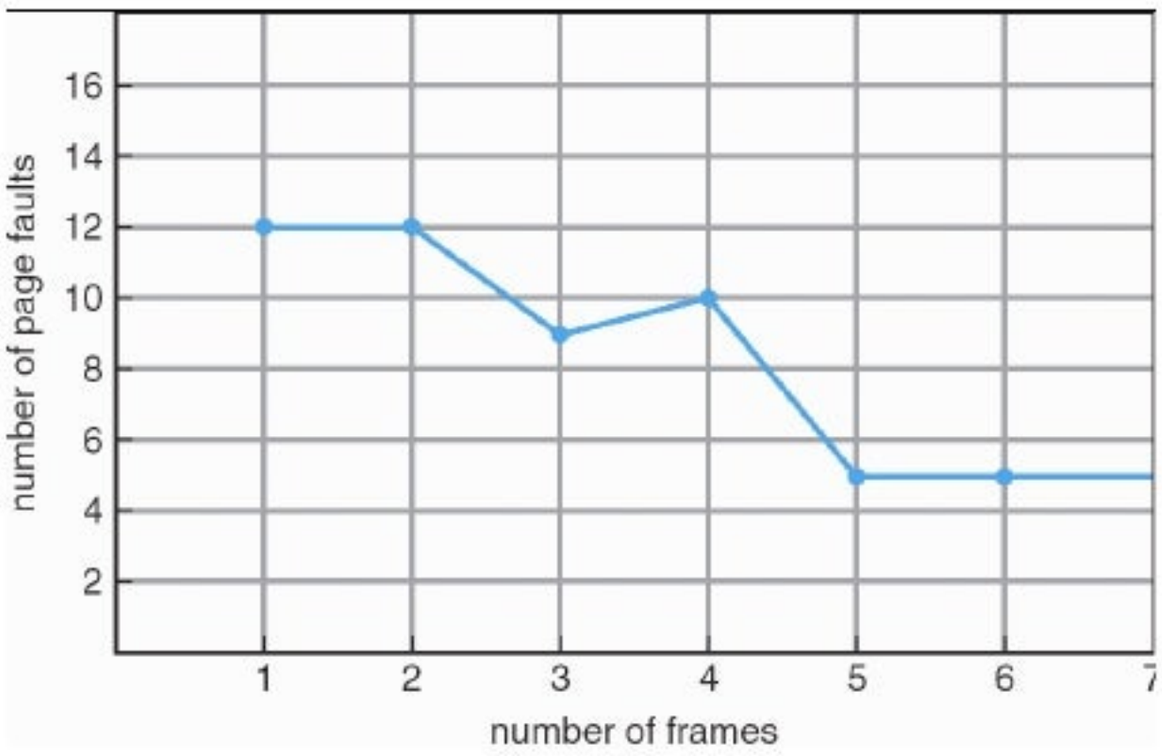
1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- FIFO page replacement algorithm:



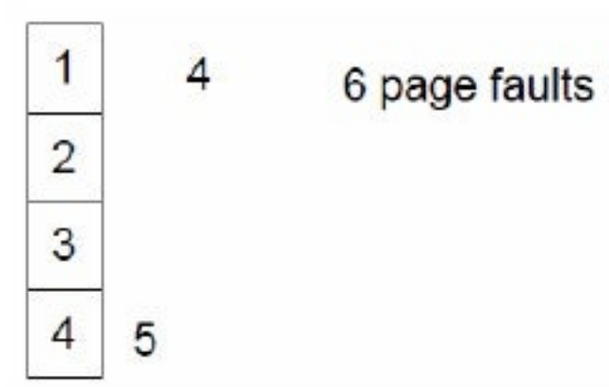
    - Yields 15 page faults
    -

- **Belady's anomaly**:
    - For some algorithms, the page-fault rate may **increase** as the number of allocated frames increases
- **FIFO illustration of Belady's Anomaly**:

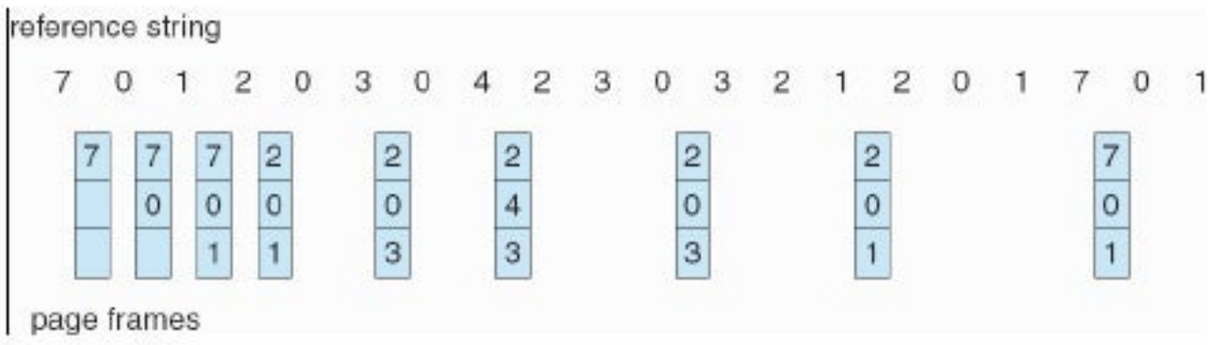


Optimal Page Replacement

- Optimal page replacement was found as a result of Belady's anomaly
- Guarantees the lowest possible page-fault rate for a fixed number of frames
- This algorithm exists and is called either OPT or MIN
- Difficult to implement because we require future knowledge of the reference string
    - Used mainly for comparative studies
- **Algorithm**:
    - Replace the page that **<u>will</u> not be used** for the longest period
- 4 frames **example:**
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



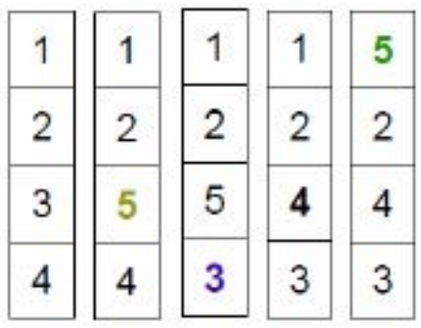
- How do you know this?
- Used for measuring how well your algorithm performs
- Difficult to implement, because you need future knowledge of the reference string
- Optimal Page replacement:



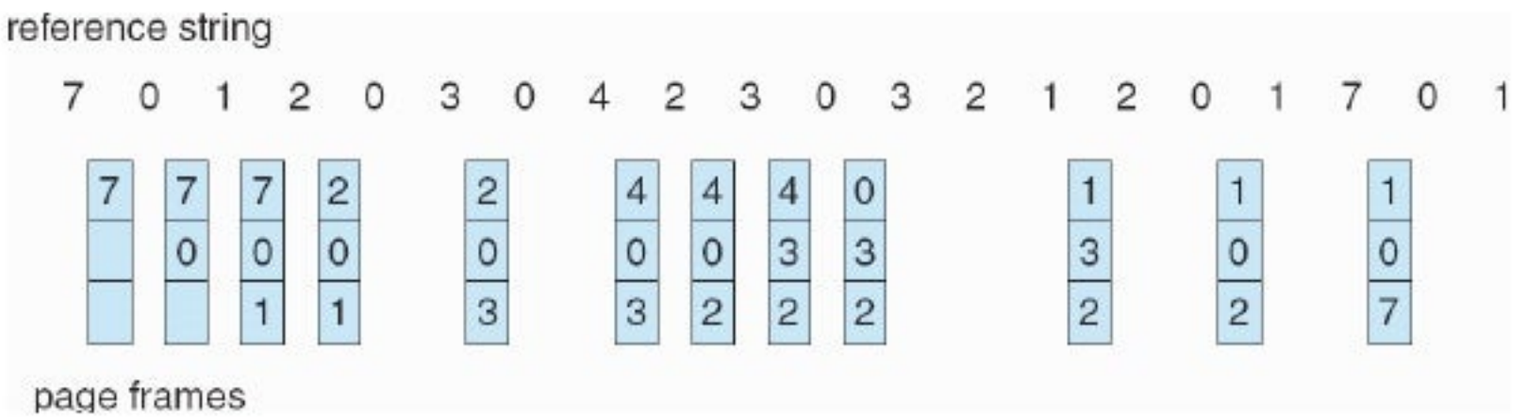
- Yields 9 page faults

LRU (least recently-used) Page Replacement

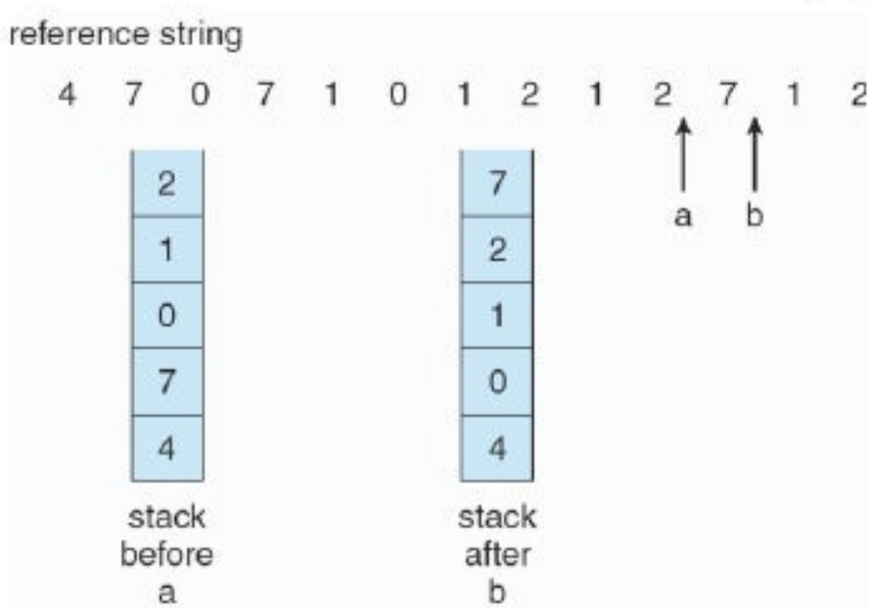
- An approximation of the optimal page replacement algorithm
- We use the recent past as an approximation of the near future
- Replace the page that **has not been used** for the longest period
  - Think of this algorithm as the backward looking optimal page-replacement algorithm
- **Example**:
  - Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



- LRU Page Replacement:



  - Yields 12 page faults
- Two ways to determine an order for the frames defined by the time of last use:
  - Counters:
    - Each page-table entry has a time-of-use field and the CPU gets a logical clock / counter
    - Whenever a page is referenced, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page
  - Stack:
    - Whenever a page is referenced, it is removed from the stack and put on top
    - The bottom of the stack is the LRU page
- Use of a stack to record the most recent page references:



- **Neither optimal replacement nor LRU replacement suffers from Belady's anomaly**

LRU-Approximation Page Replacement

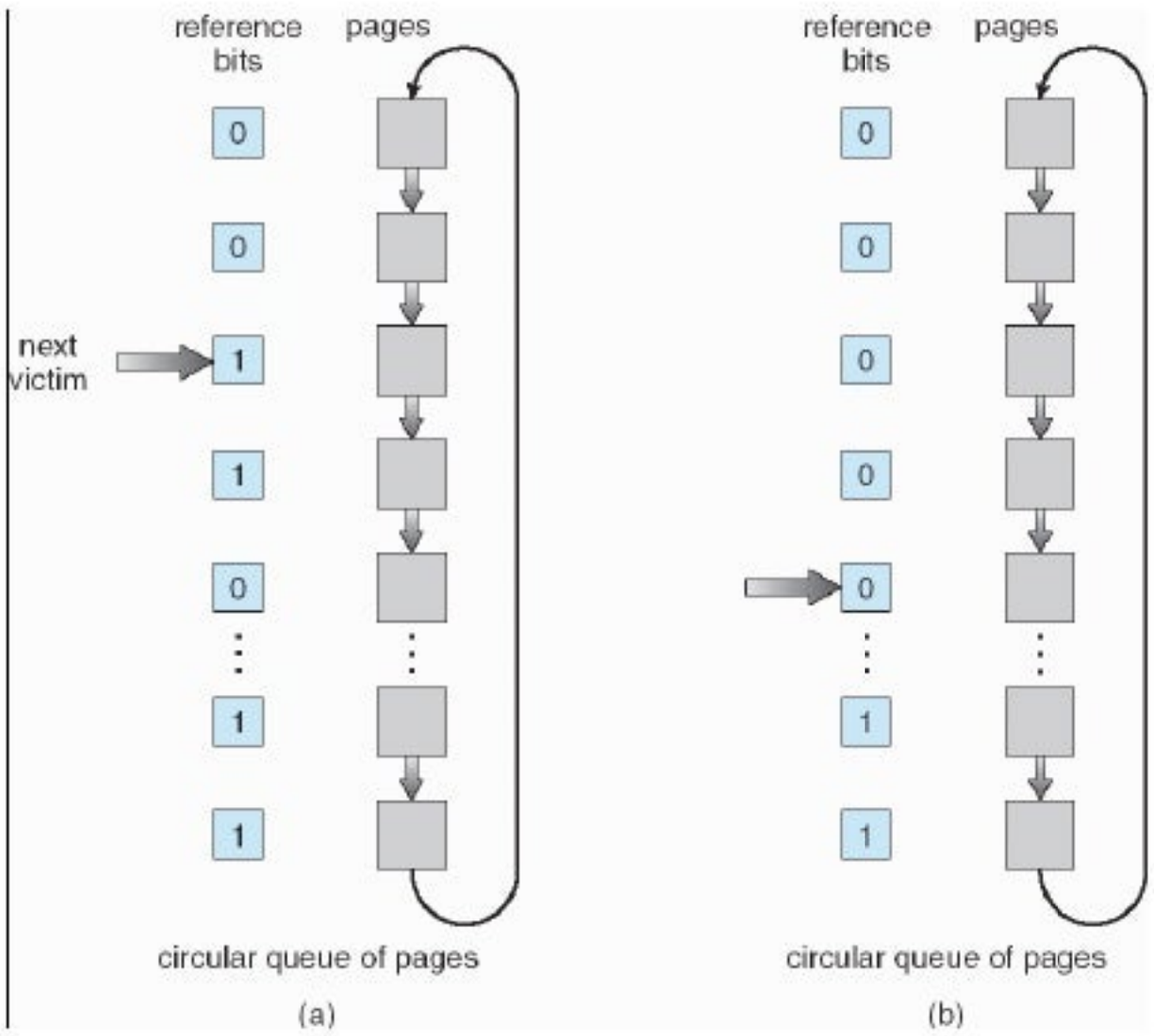
- **Reference bit:**
    - With each page associate a bit, initially = 0
    - When page is referenced bit set to 1 by hardware
    - Replace the one which is 0 (if one exists)
        - We do not know the order, however

Additional-Reference-Bits Algorithm

- You can gain additional ordering info by recording reference bits at regular intervals
- A 8-bit byte is used for each page in a table in memory
- This register is a right-shift register
    - Every 100 ms the pages are referenced and if the page was used then a 1 is moved into the MSB of the byte
- **Examples**:
    - 00000000 - This page has not been used in the last 8 time units (800 ms)
    - 11111111 - Page has been used every time unit in the past 8 time units
    - 11000100 has been used more recently than 01110111
    - These can be treated as unsigned integers and the page with the lowest value is the LRU page
    - If numbers are equal, FCFS is used

Second-Chance Algorithm

- Like the FIFO replacement algorithm, but you inspect the reference bit and replace the page if the value is 0
- If the reference bit is 1, that page gets a second chance, its reference bit is cleared (0), and its arrival time is reset
- A page that has been given a second chance will not be replaced until all other pages are replaced
- If a page is used often enough to keep its reference bit set, it will never be replaced
- **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:



Enhanced Second-Chance Algorithm

- Consider both the reference bit and the modify bit as an ordered pair:
  - (0,0) neither recently used nor modified – best to replace
  - (0,1) not recently used but modified – not quite as good
  - (1,0) recently used but clean – probably used again soon
  - (1,1) recently used and modified – probably used again soon and will need to be written to disk before being replaced

Counting-Based Page Replacement

- Keep a counter of the number of references to each page
- **LFU (least frequently used)** page-replacement algorithm
  - The page with the smallest count is replaced
- **MFU (most frequently used)** page-replacement algorithm
  - The page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- If a page is read into a free frame from the pool **before** a victim frame is written out, the process can restart as soon as possible, without waiting for the victim page to be written out
- Whenever the paging device is idle, a modified page is selected and written to disk, increasing the probability that a page will be clean when selected for replacement
- You can **remember** which page was in each frame from the pool and reuse old pages directly if needed, before the frame is reused

Applications and Page Replacement

- Sometimes applications processing data knows better how to handle their own data then the general-purpose use of page replacement used by the OS
  - **Example**:
    - Databases

- Data warehouses perform massive sequential disk reads, followed by computations and writes
    - MFU would be more efficient than LFU

Allocation of Frames

- p.382 TB
- 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

Minimum Number of Frames

- The instruction-set architecture defines the minimum number of frames that must be allocated
    - The maximum number is defined by the amount of available memory

Allocation Algorithms

- Equal allocation
    - Every process is given an equal share of frames
- Proportional allocation
    - Allocate memory to each process according to its size

Global versus Local Allocation

- Global replacement
    - A process can select a replacement frame from the whole set
    - A process may even select only frames allocated to **other** processes, increasing the number of frames allocated to it
    - Problem: A process can't control its own page-fault rate
    - Global replacement is the more common method since it results in greater system throughput
- Local replacement
    - A process selects only from its own set of allocated frames
    - The number of frames allocated to a process does not change

Non-Uniform Memory Access

- p.385 TB

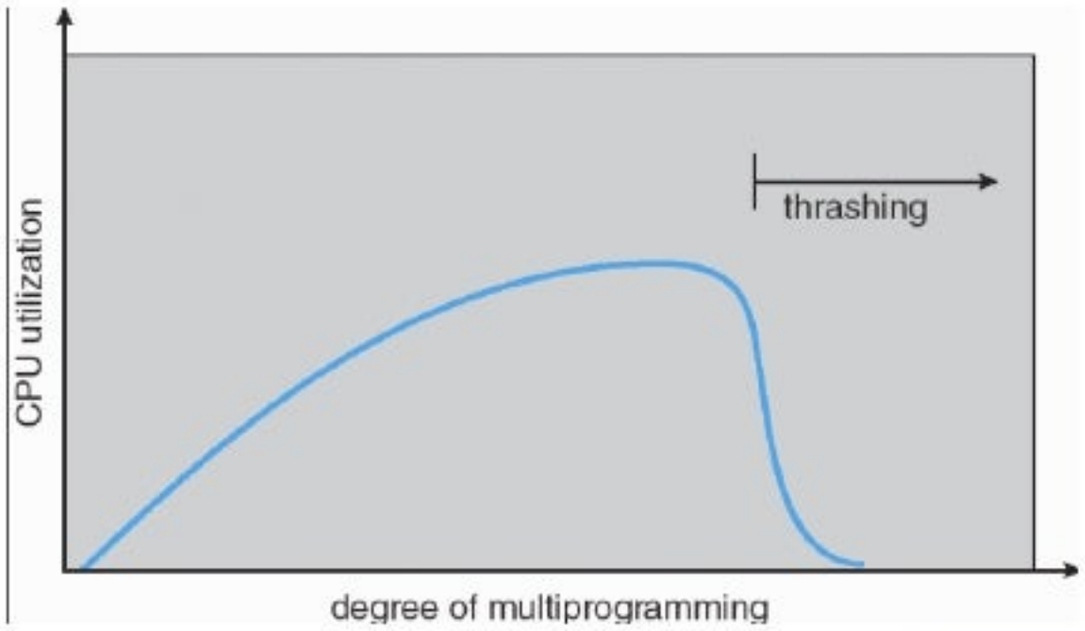
Thrashing

- 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

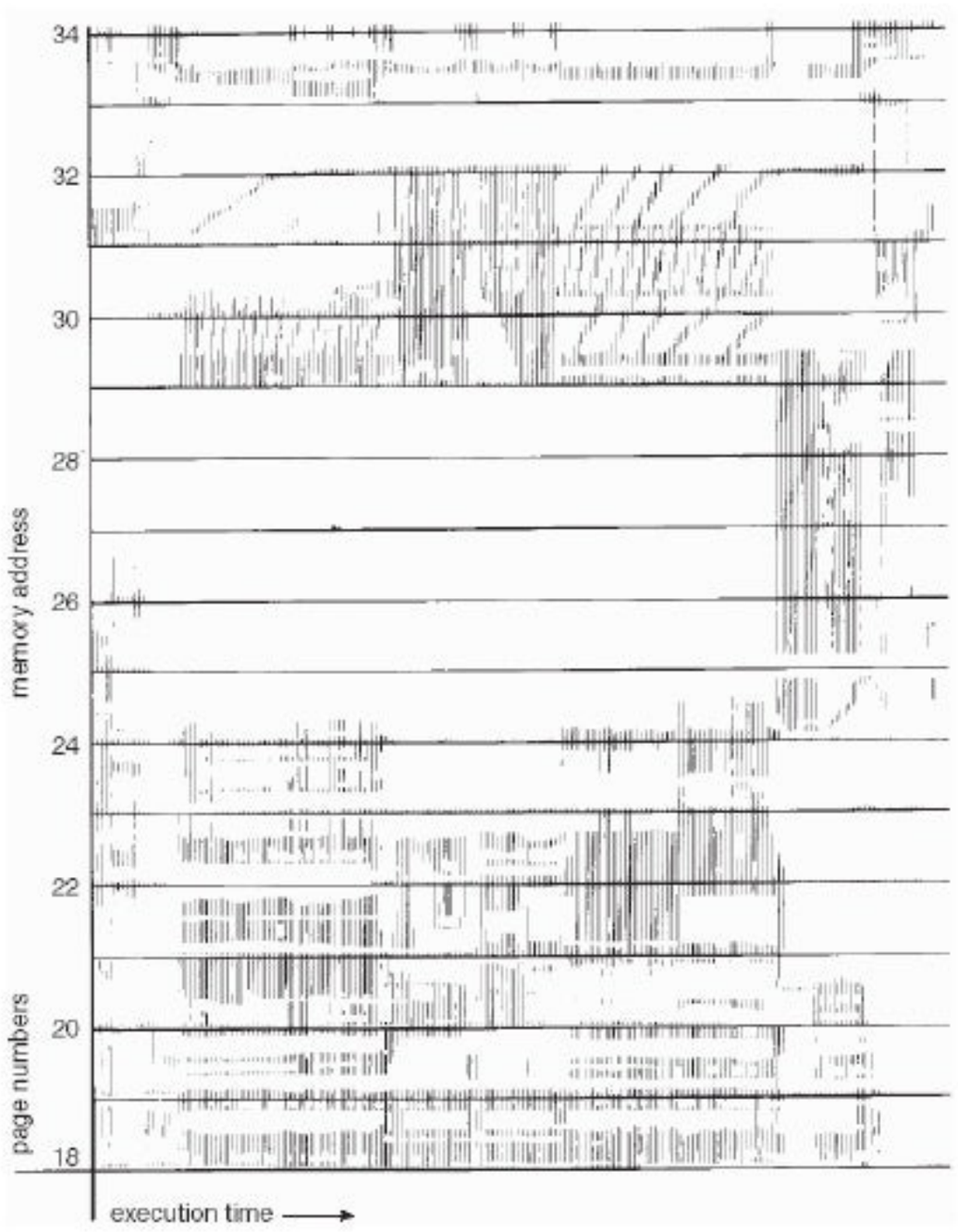
- If the number of frames allocated to a low-priority process falls below the minimum number required, it must be suspended
- A process is **thrashing** if it is spending more time paging than executing (E.g. its pages are **all in use**, and it must replace a page that will be needed again right away)

Cause of Thrashing

- The thrashing phenomenon:
  - As processes keep faulting, they queue up for the paging device, so CPU utilization decreases
  - The CPU scheduler sees the decreasing CPU utilization and **increases** the degree of multiprogramming as a result
  - The new process causes even more page faults and a longer queue!



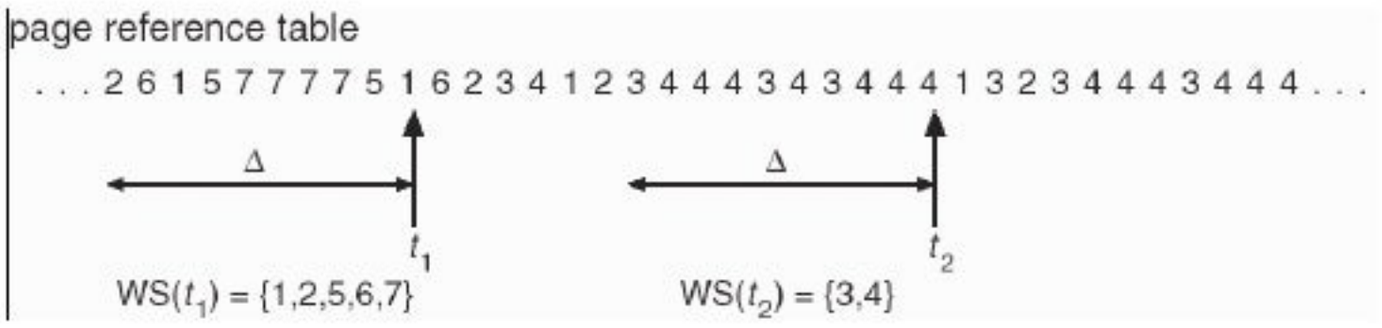
- We can limit the effects of thrashing by using a local replacement algorithm:
  - If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also
  - Pages are replaced with regard to the process of which they are a part
  - However, if processes are thrashing, the effective access time will increase even for a process that is not thrashing
- To prevent thrashing, we must give a process **enough frames**:
  - The **locality model** of process execution:
    - As a process executes, it moves from locality to locality
      - (A locality is a set of pages that are actively used together – a program may have overlapping localities)
    - If we allocate enough frames to a process to **accommodate its current locality**, it will fault for pages in its locality until they are all in memory, and it won't fault again until it changes localities
- Locality in a memory-reference pattern:



Working-Set Model

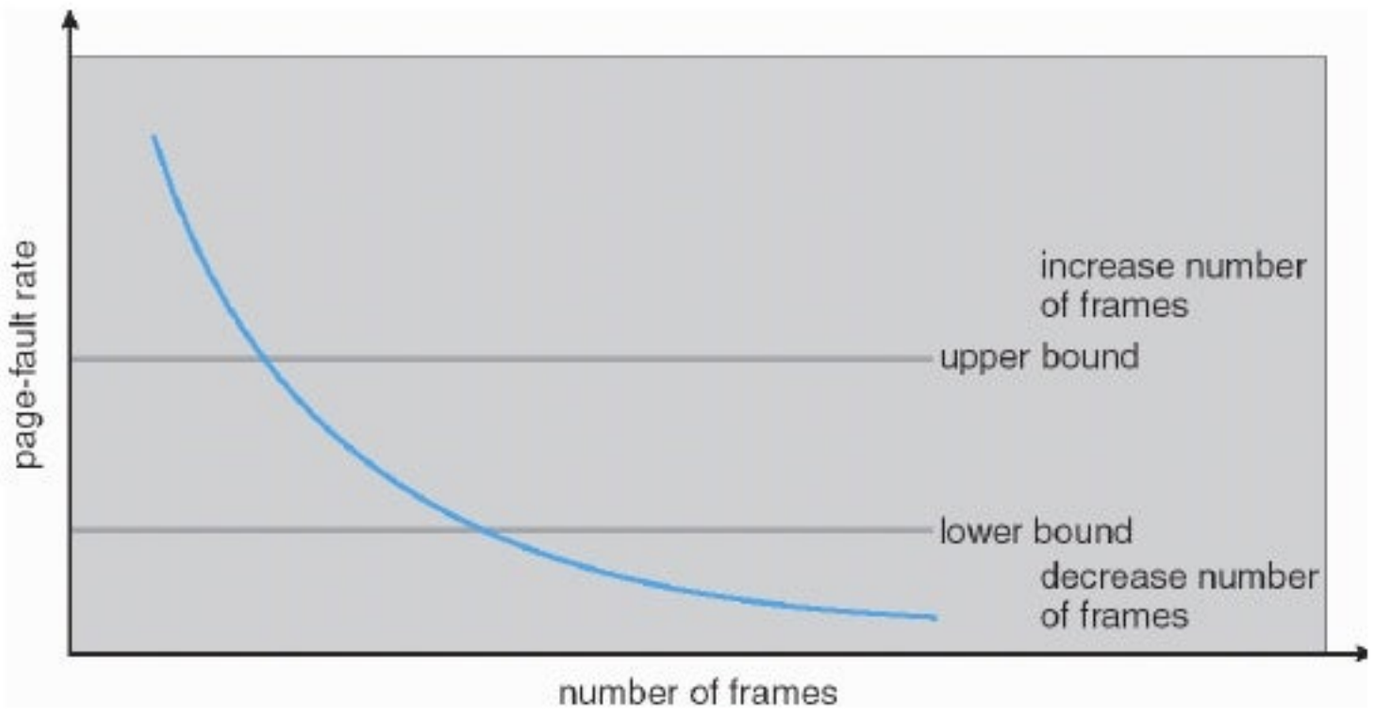
- Based on the assumption of locality
- A parameter, Δ, defines the working-set window
- Working set = set of pages in the most recent Δ page references
- If a page is in use, it will be in the working set, else it will drop from the working set Δ time units after its last reference
- The accuracy of the working set depends on the selection of Δ
    - If Δ is too small, it won't encompass the entire locality
    - If Δ is too large, it may overlap several localities
- The working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
- **Working-Set Model**:
    - Δ ≡working-set window ≡a fixed number of page references

      Example: 10,000 instruction
    - *WSSi* (working set of Process *Pi*) = 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 Δ = ∞⇒ will encompass entire program
    - $D = \Sigma\ WSSi$ ≡total demand frames
    - if $D > m \Rightarrow$ Thrashing
    - Policy if $D >$ m, then suspend one of the processes



Page-Fault Frequency

- This takes a more direct approach than the working-set model
- To prevent thrashing, control the page-fault rate:
    - When it is too high, we know the process needs more frames
    - When it is too low, the process has too many frames
- Establish upper & lower bounds on the desired page-fault rate
    - Allocate / remove frames if needed
- If the page-fault rate increases and no frames are available, select a process to suspend and re-distribute its freed frames



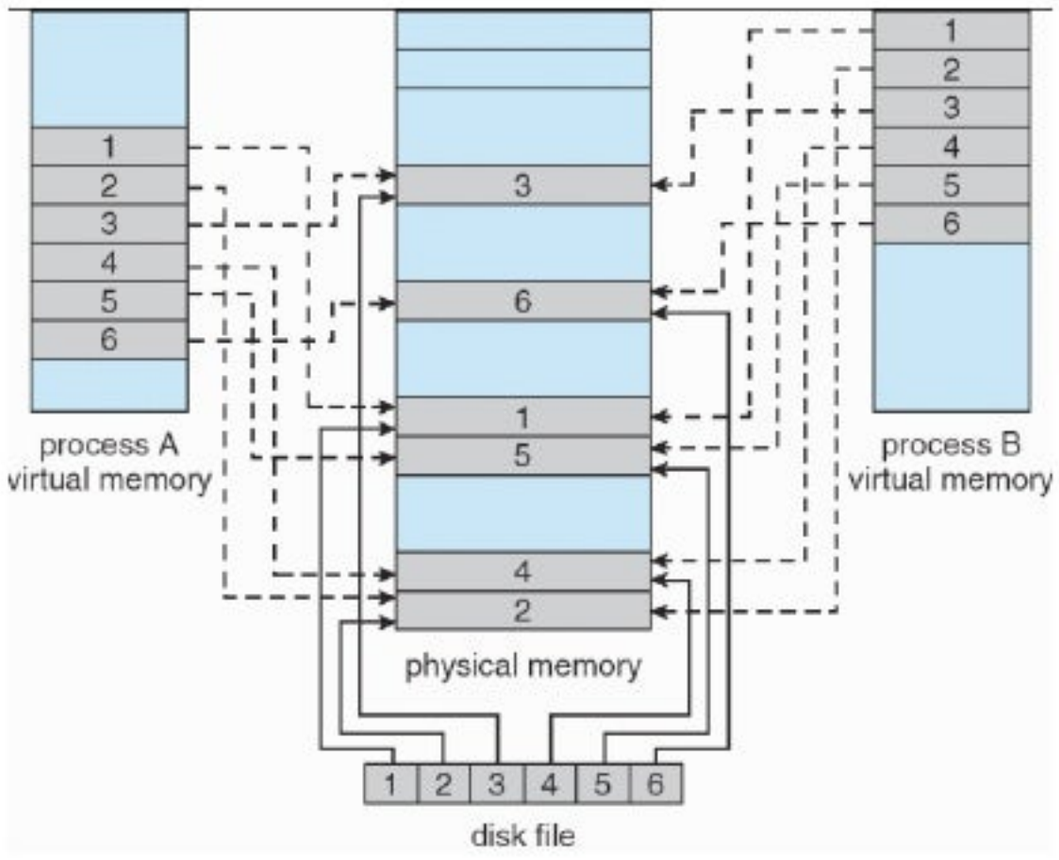
Memory-Mapped Files

- Memory-mapping a file allows a part of the virtual address space to be logically associated with a file
- 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

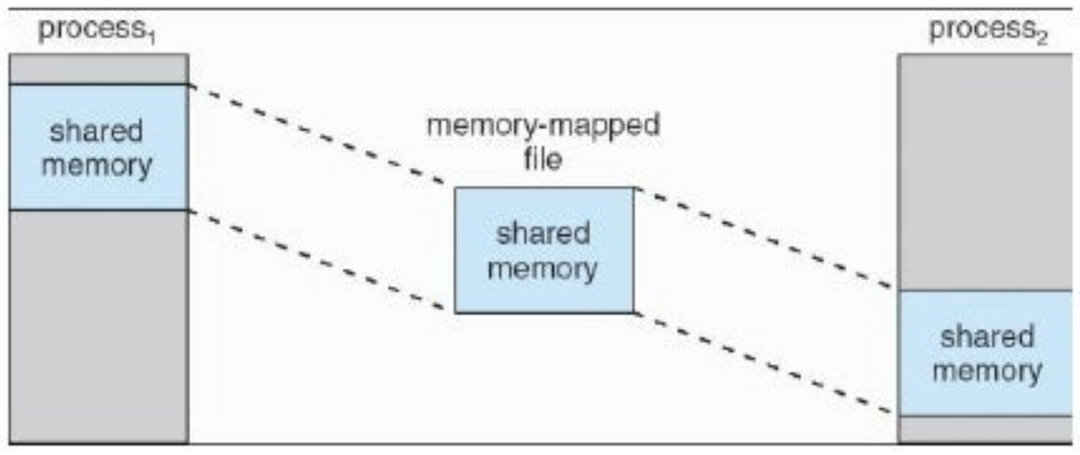
Basic Mechanism

- p.392 TB
- A disk block is mapped to a page in memory
- Initial access to the file proceeds using ordinary demand paging
- Page-sized portions of the file are read from the file system into physical pages
- Subsequent reads & writes to the files are handled as routine memory accesses, simplifying file access and usage

- File manipulation through memory incurs **less overhead** than read() and write() system calls
- Closing the file results in all the memory-mapped data being written back to disk and removed from the virtual memory
- **Memory Mapped files**:



- In many ways, the sharing of memory mapped files is similar to shared memory
- Processes can communicate using shared memory by having the communicating processes memory-map the same file into their virtual address spaces



Shared Memory in the Win32 API

- p.393 - 395 TB

Memory-Mapped I/O

- p.395 - 396 TB

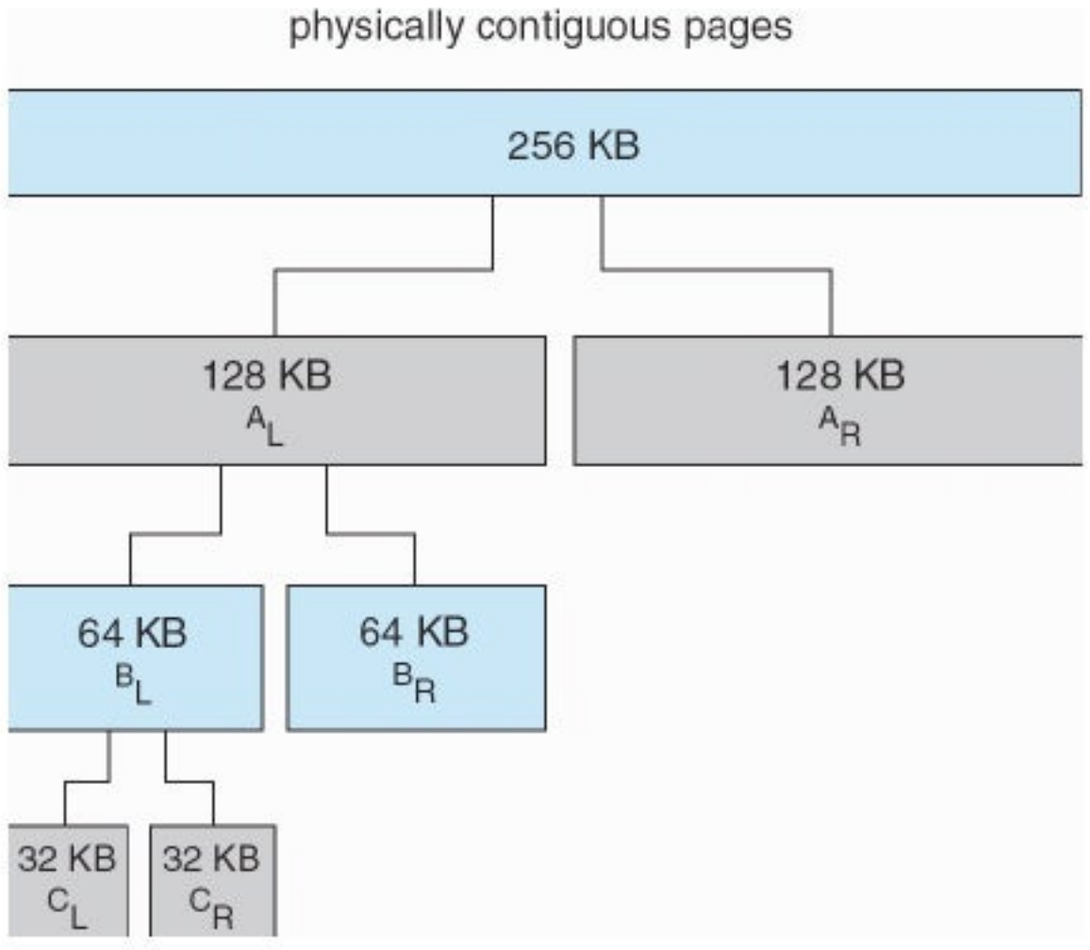
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  1. Kernel requests memory for structures of varying sizes
  2. Some kernel memory needs to be contiguous

Buggy System

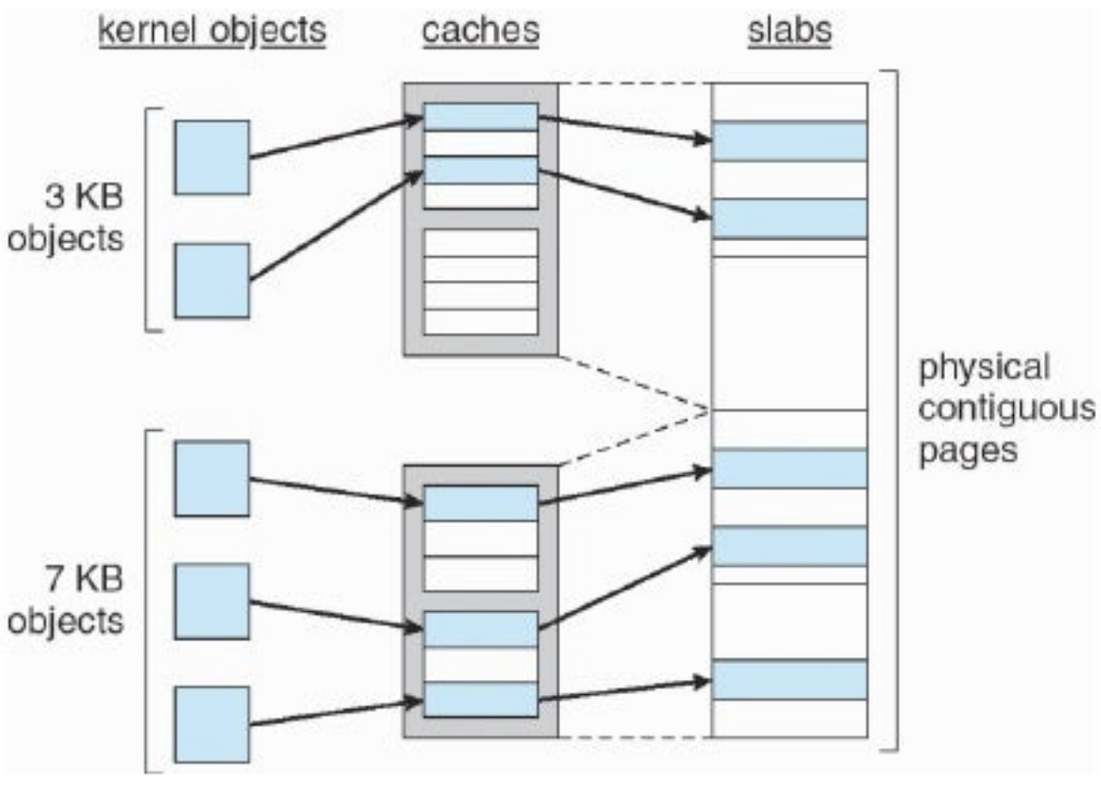
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2

- When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available



Slab Allocation

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



Other Considerations

Prepaging

- p.400 mid TB
- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and α of the pages is used
  - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1-\alpha)$ unnecessary pages?
  - α near zero ⇒ prepaging loses

Page Size

Large page size	**Small page size**
A large page size decreases the number of pages – desirable since each active process must have its own copy of the page table	Memory is better utilized with smaller pages, since it minimizes internal fragmentation
With I/O transfer, latency and seek time dwarf transfer time, so a larger page size minimizes I/O time	Total I/O should be reduced, since locality will be improved
We must allocate and transfer not only what is needed, but also anything else in the page	Better resolution, allowing us to isolate only the memory that is actually needed, resulting in less I/O & less total allocated memory

- The trend is towards **larger** page sizes
- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

TLB Reach

- Hit ratio for the TLB = the % of virtual address translations that are resolved in the TLB rather than the page table
- To increase the hit ratio, increase the number of TLB entries
- However, this is both expensive and power-hungry
- TLB reach = the amount of memory accessible from the TLB (= TLB size x the page size)
- Ideally, the working set for a process is stored in the TLB
- If not, the process will spend a considerable amount of time resolving memory reference in the page table rather than TLB
- To increase the TLB reach, you can
  - increase the size of the page
    - May lead to an increase in fragmentation

- provide multiple page sizes
    - Requires the OS, not hardware, to manage the TLB
    - Managing the TBL in software raises performance costs

Inverted Page Tables

- Create a table that has one entry per physical memory page, indexed by the pair <process-id, page-number>
- Because they keep info about which virtual-memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information
- The inverted page table no longer contains complete info about a process' logical address space, which demand paging requires
- For this information to be available, an external page table (one per process) must be kept
    - These tables are referenced only when a page fault occurs, so they don't need to be available quickly
    - They are paged in and out of memory as necessary

Program Structure

- Demand paging is designed to be transparent to the user program
- Sometimes, system performance can be improved if the user has an awareness of the underlying demand paging
- Careful selection of data structures and programming structures can increase locality and lower the page-fault rate
- E.g. a stack has good locality and a hash table has bad locality
- The choice of programming language can also affect paging: C++ uses pointers which randomize access to memory a bad locality

- **Program Structure**:
    - `int[128,128] data;`
    - Each row is stored in one page
    - **Program 1**

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

        - 128 x 128 = 16,384 page faults
    - **Program 2**

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

        - 128 page faults

I/O Interlock

- A **lock bit** is associated with every frame
- I/O pages are locked, and unlocked when the I/O is complete
- This is because I/O must not be paged out until end of transfer

- Another use for a lock bit involves normal page replacement:
    - To prevent replacing a newly brought-in page until it can be used at least once, it can be locked until used

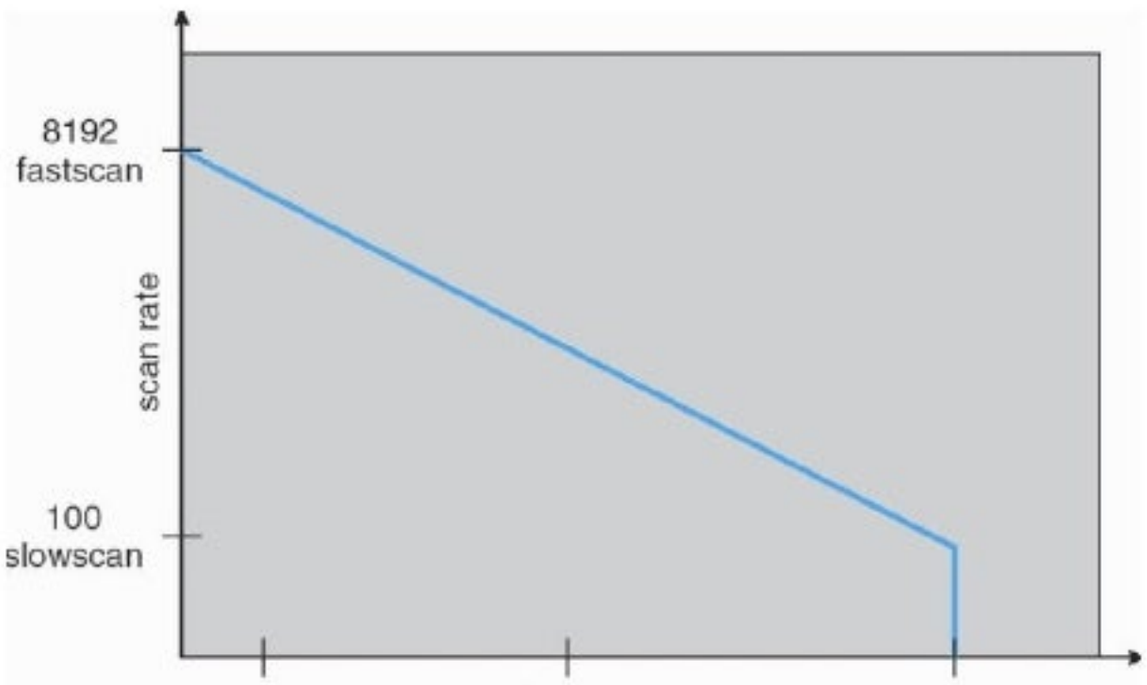
Operating-System Examples

Windows XP

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

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* - threshold parameter (amount of free memory) to begin paging
- *Desfree* - threshold parameter to increasing paging
- *Minfree* - threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned
    - This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available
- **Solaris 2 Page Scanner**:



Summary

Storage Management

- OS provides uniform, logical view of information storage
    - Abstracts physical properties to logical storage unit -**file**

- Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

File-System Management

- Computers can store info on several different types of physical media (e.g. magnetic disk, optical disk…)
- Each medium is controlled by a device (e.g. disk drive)
- The OS maps files onto physical media, and accesses these files via the storage devices
- File = a collection of related information
- The OS implements the abstract concept of a file by managing mass storage media and the devices that control them
- The OS is responsible for these file management activities:
    - Creating and deleting files
    - Creating and deleting directories
    - Supporting primitives for manipulating files & directories
    - Mapping files onto secondary storage
    - Backing up files on stable (non-volatile) storage media

PART FIVE: STORAGE MANAGEMENT

Chapter 10: File System

Objectives:

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

File Concept

- File = a named collection of related info on secondary storage
- Data can't be written to secondary storage unless its in a file
- A file has a certain defined **structure** according to its type
    - **Text file**: sequence of characters organized into lines
    - **Source file**: sequence of subroutines & functions
    - **Object file**: sequence of bytes organized into blocks
    - **Executable file**: series of code sections
- Contiguous logical address space
- **Types**:
    - **Data**
        - numeric
        - character
        - binary

- **Program**

File Attributes

- **Name**: The only info kept in human-readable form
- **Identifier**: Unique tag
- **Type**: Info needed for those systems that support different types
- **Location**: Pointer to a device and to the location of the file
- **Size**: Current size (in bytes, words, or blocks)
- **Protection**: Access-control info
- **Time, date, & user id**: Useful for protection & usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- **Creating a file**:
  - First, space in the file system must be found for the file
  - Then, an entry for the file must be made in the directory
- **Writing a file**:
  - Make a system call specifying both the name of the file and the info to be written to the file
  - The system must keep a write pointer to the location in the file where the next write is to take place
- **Reading a file**:
  - Use a system call that specifies the name of the file and where in memory the next block of the file should be put
  - Once the read has taken place, the read pointer is updated
- **Repositioning within a file**:
  - The directory is searched for the appropriate entry and the current-file-position is set to a given value
- **Deleting a file**:
  - Search the directory for the named file and release all file space and erase the directory entry
- **Truncating a file**:
  - The contents of a file are erased but its attributes stay
- Most of these file operations involve searching the directory for the entry associated with the named file
- To avoid this constant searching, many systems require that an 'open' system call be used before that file is first used
- The OS keeps a small table containing info about all open files
- When a file operation is requested, the file is specified via an index into the open-file table, so no searching is required
- When the file is no longer actively used, it is closed by the process and the OS removes its entry in the open-file table
- Some systems implicitly open a file when the first reference is made to it, and close it automatically when the program ends

- Most systems require that the programmer open a file explicitly with the 'open' system call before that file can be used
- A **per-process** table tracks all files that a process has open and includes access rights to the file & accounting info
- Each entry in the per-process table in turn points to a **system-wide** open-file table, which contains process- independent info, such as the file's disk location, access dates, and file size
- Information associated with an **open file**:
    - **File pointer**:
        - For the system to track the last read-write location
    - **File open count**:
        - A counter tracks the number of opens & closes and reaches zero on the last close
    - **Disk location of the file**:
        - Location info is kept in memory to avoid having to read it from disk for each operation
    - **Access rights**:
        - Each process opens a file in an access mode
- **Open file locking**:
    - Provided by some operating systems and file systems
    - Mediates access to a file
    - **Mandatory or advisory**:
        - **Mandatory**–access is denied depending on locks held and requested
        - **Advisory**–processes can find status of locks and decide what to do

File Types

- If an OS recognizes the type of a file, it can operate on the file in reasonable ways
- A common technique for implementing file types is to include the type as part of the file name
    - Name split into 2 parts:
        - name
        - extension

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

- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file
    - Example:
        - Only a file with a .com, .exe, or .bat extension can be executed
        - .com and .exe are two forms of binary executable files
        - .bat file is a batch file containing, in ASCII format, commands to the operating system
- **MS-DOS** only recognizes a few files but application programs also use extensions to indicate file types they are interested in
    - Because application extensions are not supported by the operating system, they can be considered as "hints" to the applications that operate on them
- The **TOPS-20** operating system will automatically recompile an object program if the source code was modified or edited
    - This way the user always runs with an up-to-date object file
    - For this purpose the operating system must be able to discriminate the source file from the object file, to determine if and when it was modified
    - File extensions are used for this purpose
- In the **Mac OS X** operating system every file has a type, such as TEXT or APPL
    - Each file has a creator attribute that contain the name of the program that created it, set by the operating during the create() call - use is enforced by the system
- **UNIX** uses a crude magic number stored at the beginning of some files to indicate roughly the type of file, executable program; batch file (shell script), PostScript; etc
    - Not all files have magic numbers so system features cannot be based solely on this information
    - Does not record the name of the creating program
    - File extensions are meant mostly to aid users in determining what type of contents a file contains
    - Extensions can be used or ignored by applications
        - Up to the applications programmer

File Structure

- File types can indicate the internal structure of the file
- Disadvantage of supporting multiple file structures: large size
- All OSs must support at least one structure: an executable file
- The Mac OS file structure:
    - Resource fork (contains info of interest to the user)
    - Data fork (contains program code / data)
- Too few structures make programming inconvenient
- Too many structures cause OS bloat and programmer confusion

Internal File Structure

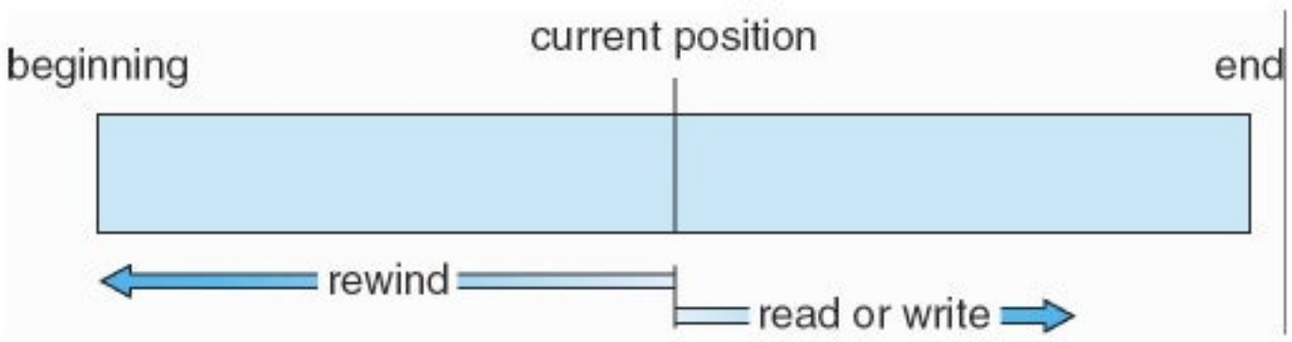
- **p.430 mid bot (make note on this piece)**
- Locating an offset within a file can be complicated for the OS
- Disk systems typically have a well-defined block size
- It is unlikely that the physical record size will exactly match the length of the desired logical record

- Packing a number of logical records into physical blocks is a common solution to this problem
- The logical record size, physical block size, and packing technique determine how many logical records are in each physical block
- The packing can be done either by the user's program or OS

Access Methods

Sequential Access

- Information in the file is processed in order
- The most common access method (e.g. editors & compilers)
- **Read**: reads the next file portion and advances a file pointer
- **Write**: appends to the end of file and advances to the new end



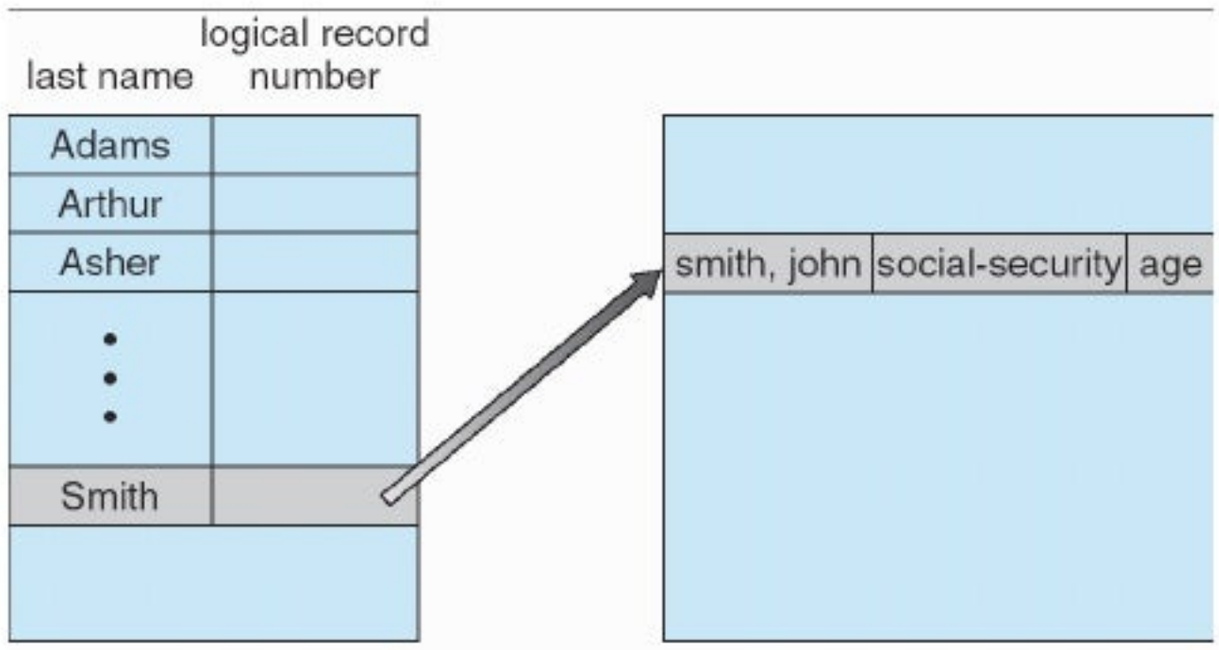
Direct Access

- A file is made up of fixed-length logical records that allow you to read & write records rapidly in no particular order
- File operations include a relative block number as parameter, which is an index relative to the beginning of the file
- The use of relative block numbers
  - allows the OS to decide where the file should be placed and
  - helps to prevent the user from accessing portions of the file system that may not be part of his file

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

Other Access Methods

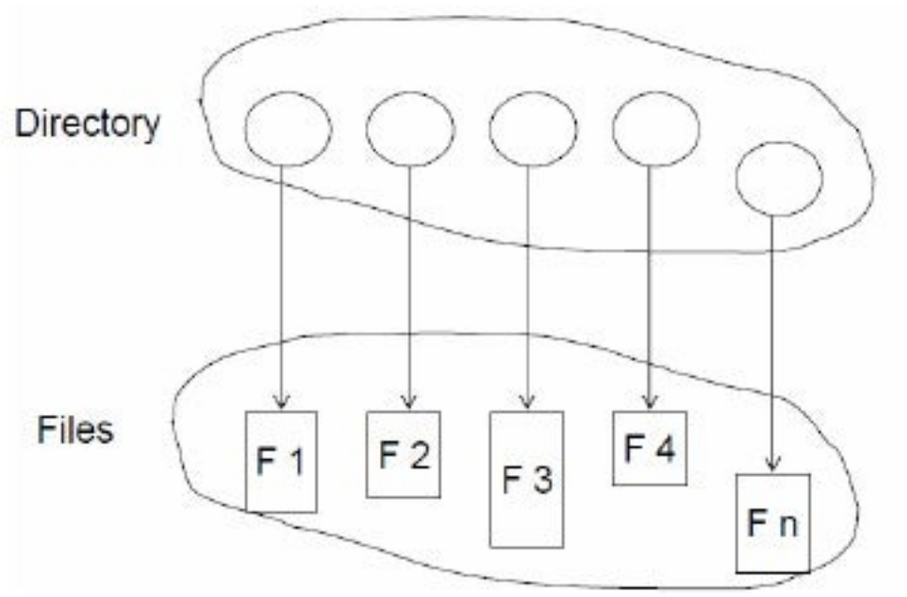
- These methods generally involve constructing a **file index**
- The index contains pointers to the various blocks
- To find a record in the file
  - First search the index
  - Then use the pointer to access the file directly and to find the desired record
- With large files, the index file itself may become too large to be kept in memory
- Solution: create an index for the index file



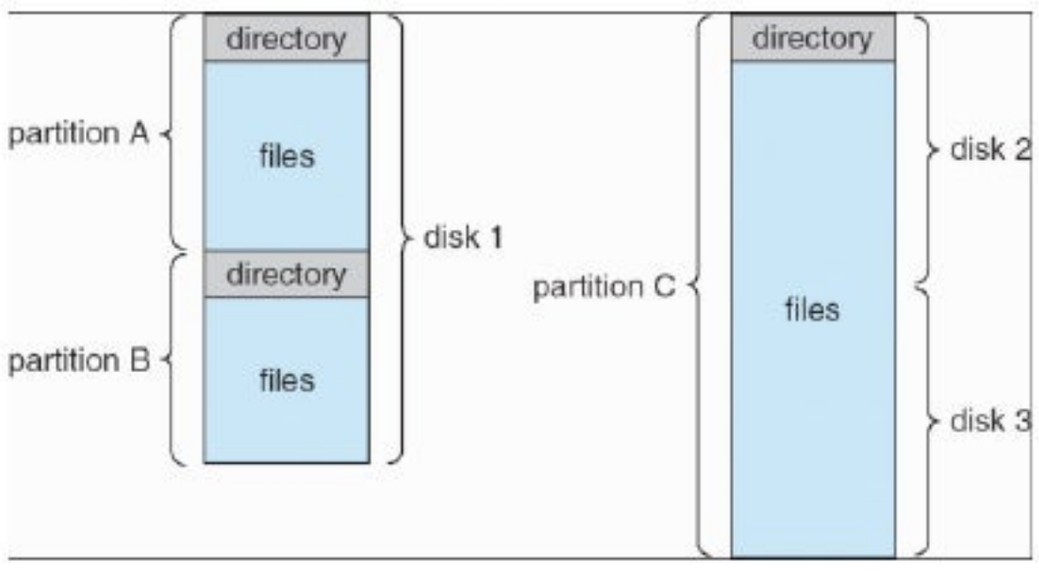
index file relative file

Directory and Disk Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk
- Backups of these two structures are kept on tapes
- **Typical File-System Organization**:



Storage Structure

- **p434 (make sure)**
- Organizing a lot of data can be done in two parts:
  - Disks are split into one or more partitions
  - Each partition contains info about files within it (e.g. name, location, size, type…) in a device directory

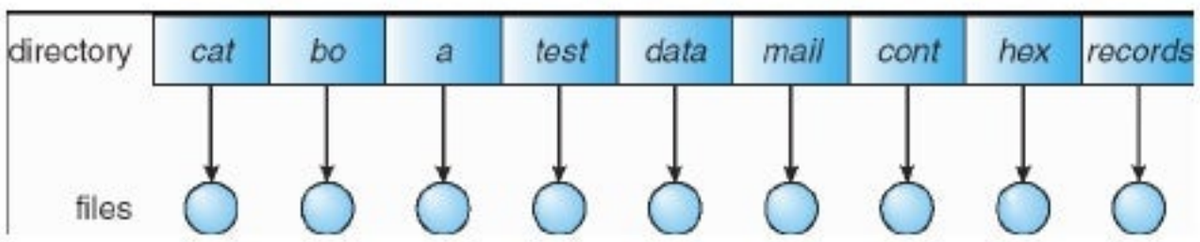
Directory Overview

- **Very Important p.435 (make sure)**
- Operations that can be performed on a directory:

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

Single-Level Directory

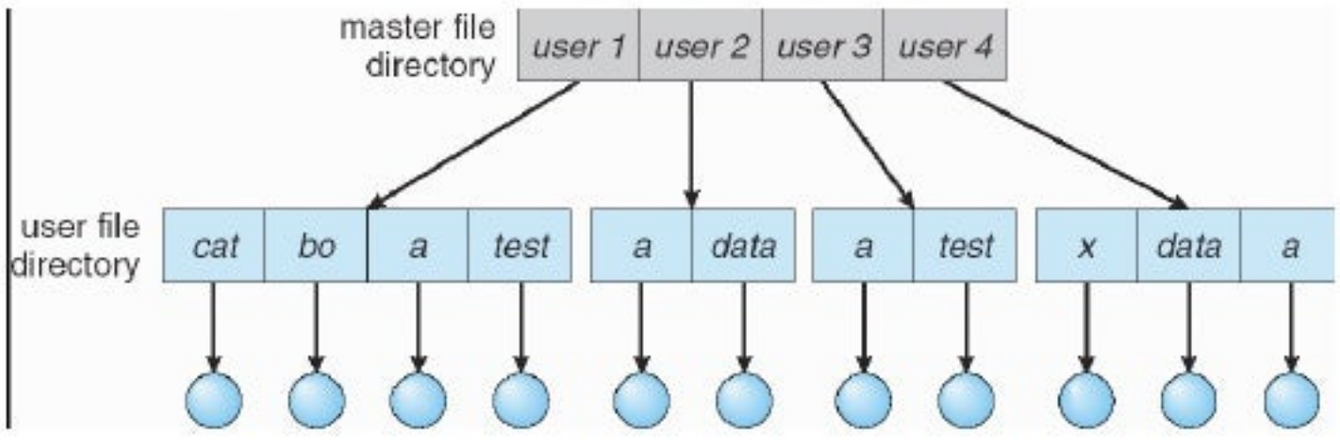
- All files are contained in the same directory
- Limitations because all files must have unique names
- A single directory for all users



- Naming problem
- Grouping problem

Two-Level Directory

- Separate directory for each user (UFD = user file directory)
- Each entry in the MFD (master file directory) points to a UFD
- Advantage: No filename-collision among different users
- Disadvantage: Users are isolated from one another and can't cooperate on the same task
- System files (e.g. compilers, loaders, libraries…) are contained in a special user directory (e.g. user 0) that is searched if the OS doesn't find the file in the local UFD
- Search path = directory sequence searched when a file is named

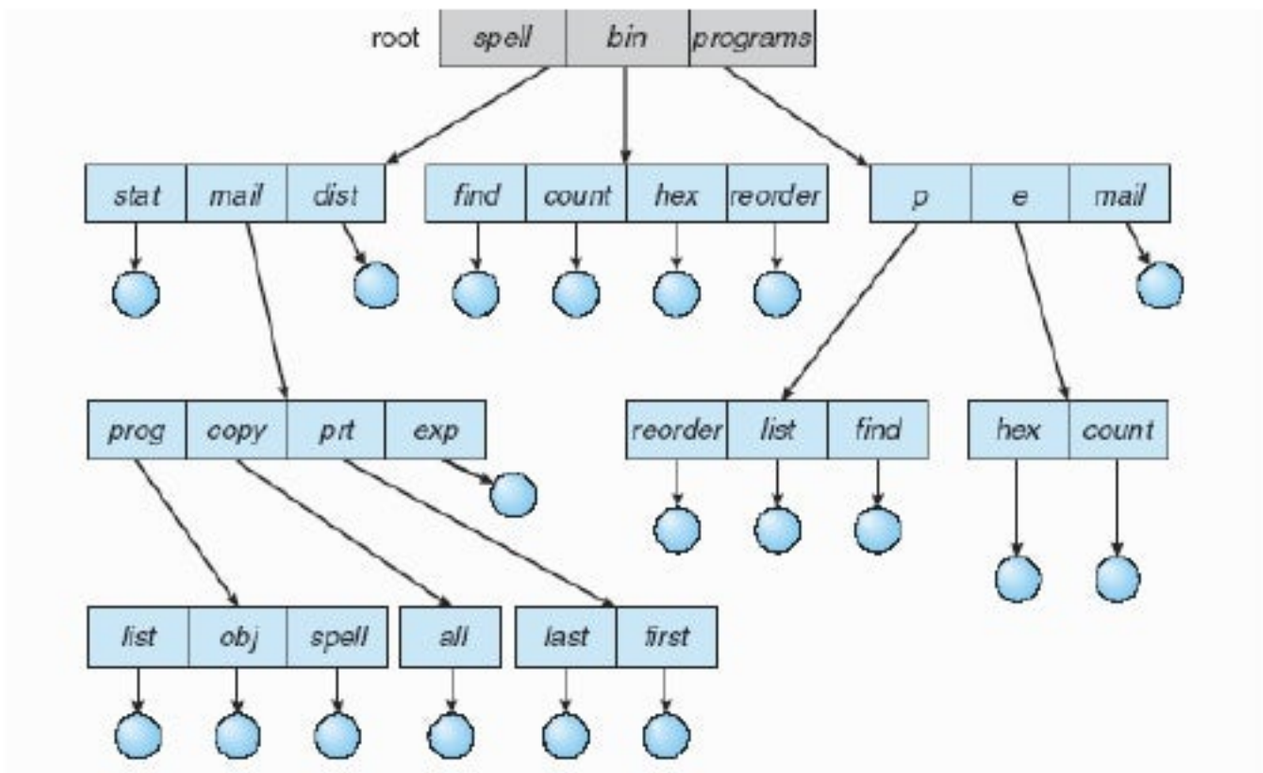


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories

- Users can create their own subdirectories and organize files
- **Absolute path names**: begin at the root
- **Relative path names**: define a path from the current directory
- To delete an empty directory:

    - Just delete it
- To delete a non-empty directory:
    - First delete all files in the directory, or
    - Delete all that directory's files and subdirectories
- Efficient searching
- Grouping Capability
- Current directory (working directory)
    - cd /spell/mail/prog
    - type list



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

  rm <file-name>
- Creating a new subdirectory is done in current directory

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

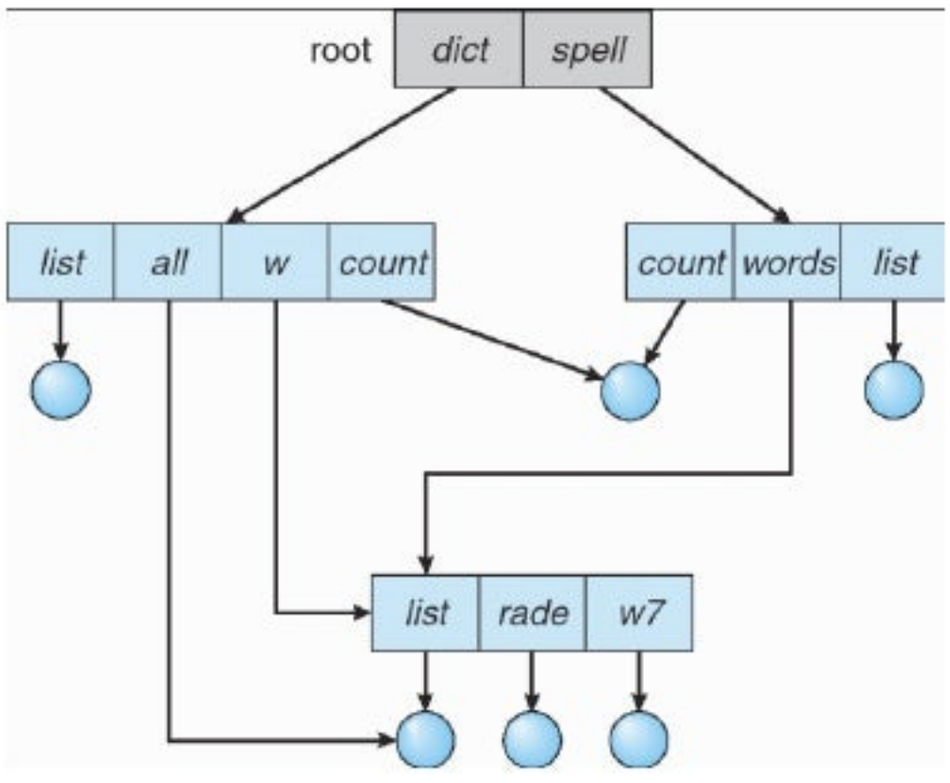
  mkdir count

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

Acyclic-Graph Directories

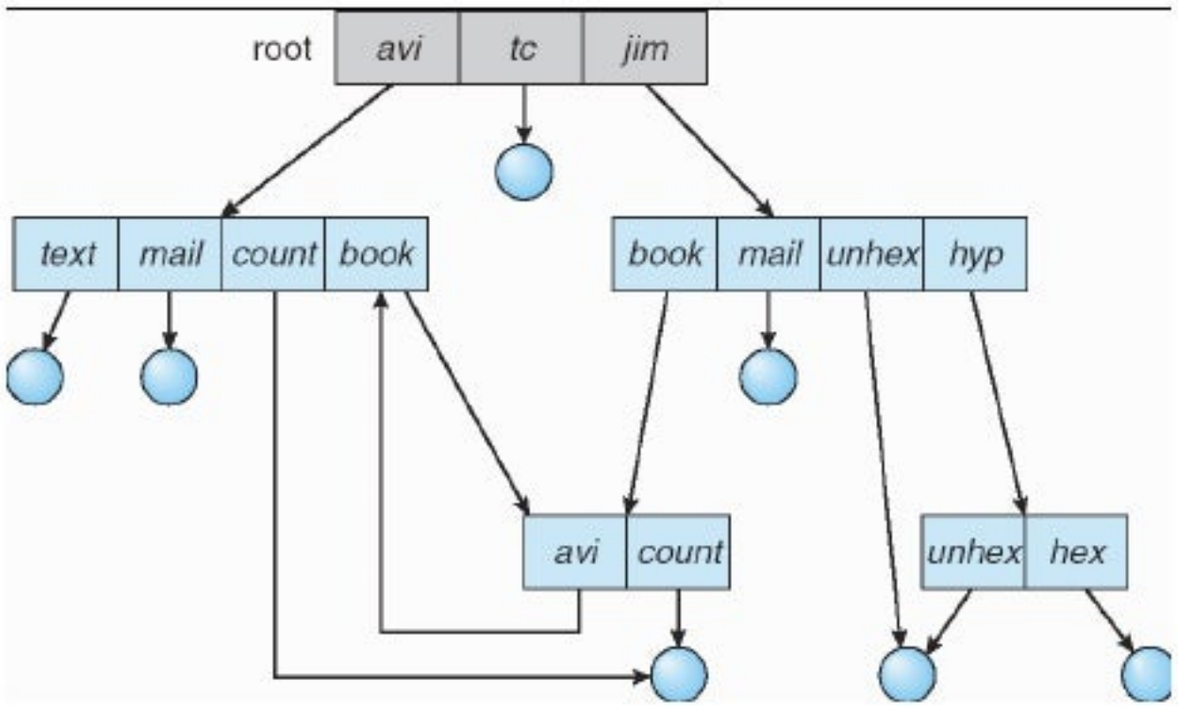
- Directories can have **shared** subdirectories and files
- Advantage: simple algorithms to traverse the graph
- Only one file exists, so changes made by one person are immediately visible to the other
- Ways to implement shared files / subdirectories:
    - Create a new directory entry called a **link**, which is a pointer to another file / subdirectory

    - Duplicate all info about shared files in both sharing directories
- Problems:
    - A file may now have multiple absolute path names
    - Deletion may leave dangling pointers to a non-existent file
- Solutions to deletion problems:
    - Backpointers, so we can delete all pointers
    - Variable size records a problem
    - Backpointers using a daisy chain organization
    - Entry-hold-count solution
- Approaches to deletion:
    - With symbolic links, remove only the link, not the file. If the file itself is deleted, the links are left dangling and can be removed / left until an attempt is made to use them
    - Preserve the file until all references to it are deleted. A mechanism is needed to determine that the last reference to the file has been deleted. Problem: potentially large size of the file- reference list
- New directory entry type
- **Link**-another name (pointer) to an existing file
- **Resolve the link**-follow pointer to locate the file



General Graph Directory

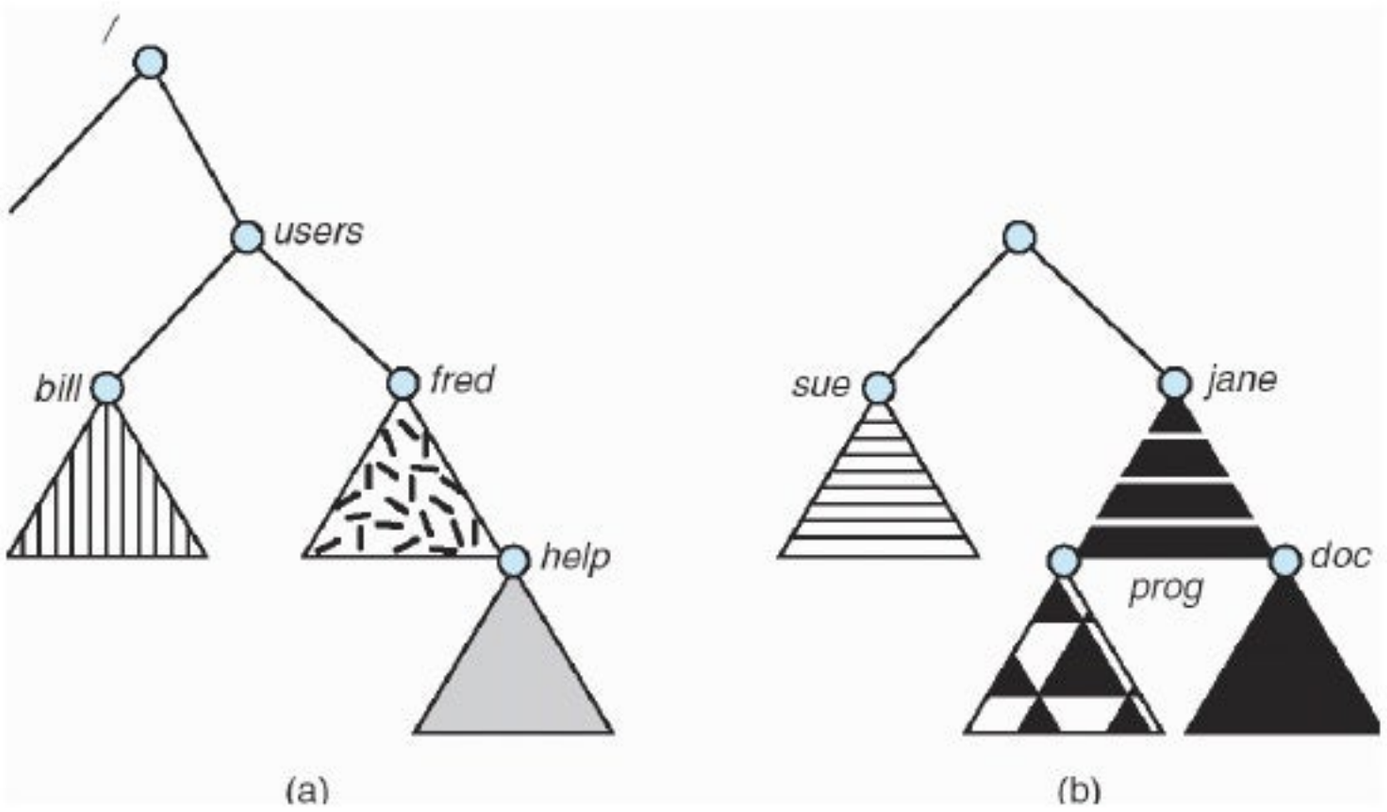
- Can have cycles: links are added to an **existing** directory
- If there are cycles, we want to avoid searching components twice
    - Solution: limit the no of directories accessed in a search
- Similar problem when determining when a file can be deleted:
    - With cycles, the reference count may be nonzero, even when it is no longer possible to refer to a directory / file (This anomaly results from the possibility of self-referencing in the directory structure)
    - **Garbage collection** is needed to determine when the last reference has been deleted, only because of possible cycles



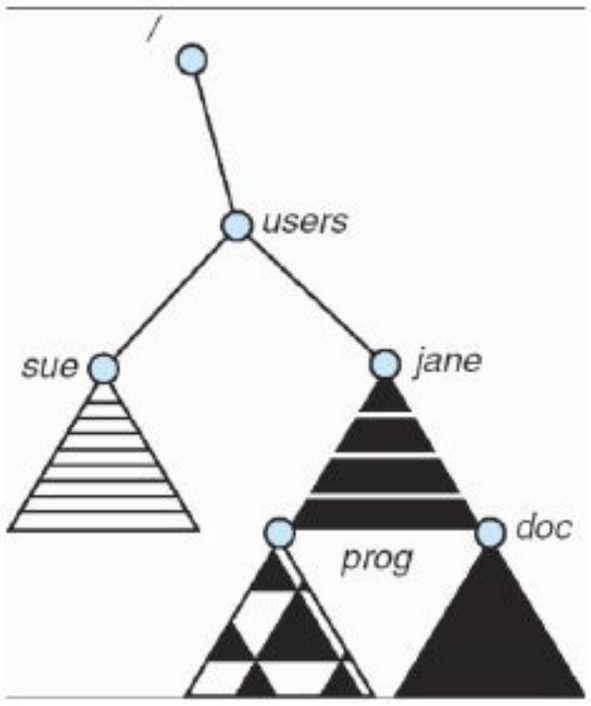
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File-System Mounting

- A file system must be mounted before it can be available to processes on the system
- Mount procedure:
  - The OS is given the name of the device and location within the file structure at which to attach the file system
  - The OS verifies that the device contains a valid file system
  - The OS notes in its directory structure that a file system is mounted at the specified mount point
- **Existing (a) ; Unmounted Partition (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

Multiple Users

- File sharing
  - The system can allow a user to access the files of other users by default, or
  - It may require that a user specifically grant access
- To implement sharing & protection, the system must maintain more file & directory attributes than on a single-user system
- Most systems use concepts of file/directory owner and group
- Owner = the user who may change attributes, grant access, and has the most control over the file / directory
  - Most systems implement owner attributes by managing a list of user names and user IDs
- Group = the attribute of a file that is used to define a subset of users who may share access to the file
  - Group functionality can be implemented as a system-wide list of group names and group IDs
- The owner and group IDs of a file / directory are stored with the other file attributes, and can be used to allow / deny ops

Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**

The Client-Server Model

- The server specifies which resources (files) are available to which clients
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients

- Client and user-on-client identification is insecure or complicated
- **NFS** is standard UNIX client-server file sharing protocol
- **CIFS** is standard Windows protocol
- Standard operating system file calls are translated into remote calls

Distributed Information Systems

- Provide unified access to info needed for remote computing
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing
- DNS provides host-name-to-network-address translations

Failure Modes

- RAID can prevent the loss of a disk
- Remote file systems have more failure modes because the network can be interrupted between two hosts
- Protocols can enforce delaying of file-system operations to remote hosts, for when the host becomes available again
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

Consistency Semantics

- **Consistency semantics** represent an important criterion of evaluating file systems that supports file sharing
    - These semantics specify how multiple users of a system are to access a shared file simultaneously
    - In particular, they specify when modifications of data by one user will be observed by other users
    - These semantics are typically implemented as code with the file system
- Consistency semantics are directly related to the process-synchronization algorithms of chapter 6
    - However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks
    - For example, performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both
    - Systems that attempt such a full set of functionalities tend to perform poorly
    - A successful implementation of complex sharing semantics can be found in the Andrew file system
- For the following discussion, we assume that a series of file accesses (that is reads and writes) attempted by a user to the same file is always enclosed between the open() and close() operations
    - The series of accesses between the open() and close() operations make up a file session
    - To illustrate the concept, we sketch several prominent examples of consistency semantics

UNIX Semantics

- Unix file system (UFS) uses the following consistency semantics:
    - Writes to an open file by a user are visible immediately to other users who have this file open
    - One mode of sharing allows users to share the pointer of current location into a file
        - Thus, the advancing of the pointer by one user affects all sharing users

        - Here, a file has a single image that interleaves all accesses, regardless of their origin
- In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource
- Contention for this single image causes delays in user processes

Session Semantics

- Andrew File System (AFS) uses the following consistency semantics
    - Writes to an open file by a user are not visible immediately to other users that have the same file open
    - Once a file is closed, the changes made to it are visible only in sessions starting later
        - Already open instances of the file do not reflect these changes
- According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time
    - Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay
    - Almost no constraints are enforced on scheduling accesses

Immutable-Shared-Files Semantics

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

Protection

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

Types of Access

- Systems that don't permit access to other users' files don't need protection so protection can provided by prohibiting access
- This is too extreme, so controlled access is needed:
    - Limit the types of file access that can be made
    - You can control operations like Read, Write, Delete, List…
- Types of access:
    - **Read**
    - **Write**
    - **Execute**
    - **Append**

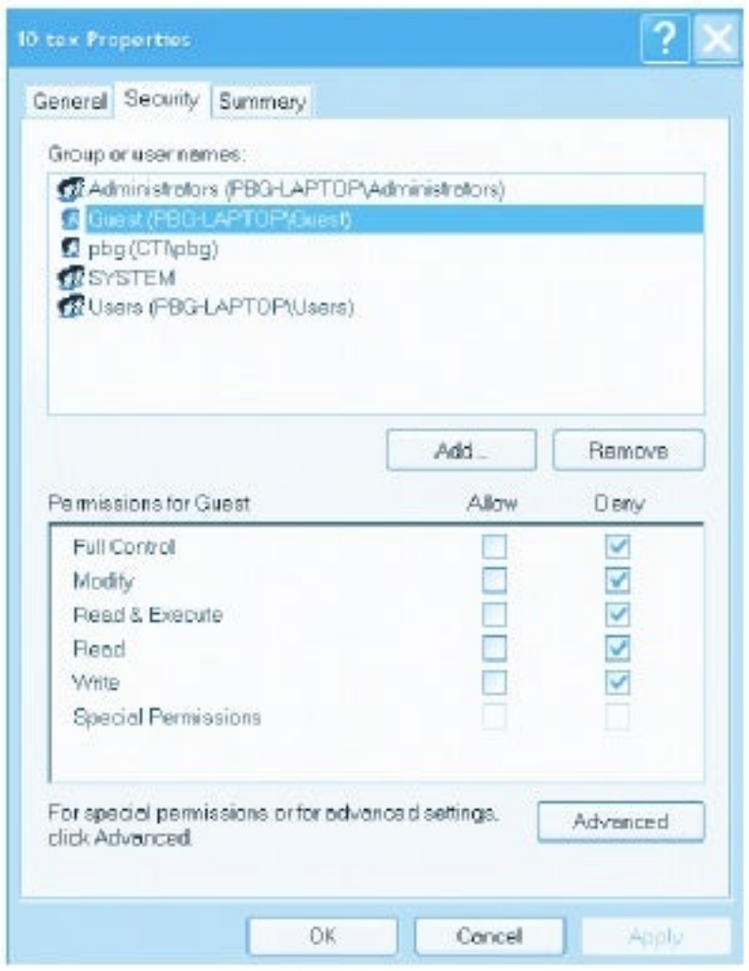
- **Delete**
- **List**

Access Control

- The most common approach to the protection problem is to make access dependent on the identify of the user
- Files can be associated with an access-control list (ACL) specifying the user name and the types of access allowed for each user
- Problems:
  - Constructing a list can be tedious
  - The directory entry now needs to be of variable size, resulting in more complicated space management
- These problems can be resolved by combining access control lists with an 'owner, group, universe' access-control scheme
  - To condense the length of the access-control list, many systems recognize three classifications of users in connection with each:
    - Owner
      - The user who created the file is the owner
    - Group
      - A set of users who are sharing the file and need similar access is a group, or work group
    - Universe
      - All other users in the system constitute the universe
  - **Samples**:

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

  - E.g. rwx bits indicate which users have permission to read/write/execute
- **Windows XP Access-control list management**:



- **A Sample UNIX directory listing**:

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

Other Protection Approaches

- A password can be associated with each file
- Disadvantages:
  - The no of passwords you need to remember may become large
  - If only one password is used for all the files, then all files are accessible if it is discovered
  - Commonly, only one password is associated with all of the user's files, so protection is all-or-nothing
- In a multilevel directory structure, we need to provide a mechanism for **directory protection**
- The directory operations that must be protected are different from the file operations:
  - Control creation & deletion of files in a directory
  - Control whether a user can determine the existence of a file in a directory (i.e. the 'dir' command in DOS)

Summary

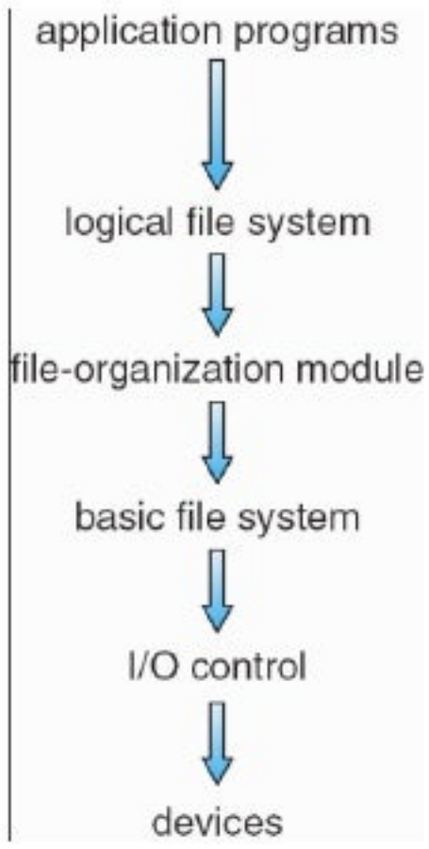
Chapter 11: Implementing File System

Objectives:

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

File-System Structure

- Disks provide the bulk of secondary storage on which a file system is maintained
    - They have two characteristics that make them a convenient medium for storing multiple files:
        - They can be rewritten in place
            - Can read a block from disk, modify the block, and write it back into the same place
        - A disk can access directly any block of information it contains
            - Thus, it is simple to access any file either sequentially or randomly , and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of **blocks**
    - Each block has one or more sectors
    - Depending on the disk drive, sector size varies from 32 bytes to 4096 bytes
    - The usual size is 512 bytes
- File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
- Design problems of file systems:
    - Defining how the file system should look to the user
    - Creating algorithms & data structures to map the logical file system onto the physical secondary-storage devices
- The file system itself is generally composed of many different levels
    - Every level in the design uses the features of lower levels to create new features for use by higher levels
- **A layered File System**:



- Levels of the file system:
    - **I/O Control** (lowest level)
        - Consists of **device drivers** & interrupt handlers to transfer info between main memory & disk
    - **Basic file system**

        - Need only issue generic commands to the appropriate device driver to read & write blocks on the disk
- **File-organization module**
    - Knows about files and their logical & physical blocks
    - Translates logical block address to physical ones
- **Logical file system**
    - Manages metadata information
    - Manages the directory structure
        - Maintains file structure via **file control blocks (FCB)**
    - **Application programs**

File-System Implementation

- Here we delve into the structures and operations used to implement file-system operations, like the open() and close() operations

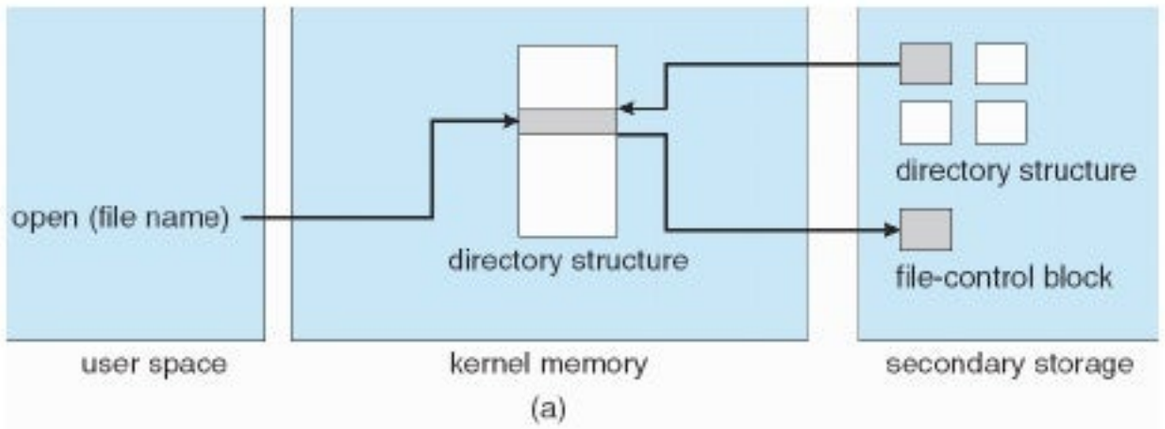
Overview

- On-disk & in-memory structures used to implement a file system:
- On-disk structures include:
    - A **boot control block**
        - Contains info needed to boot an OS from that partition
    - A **partition control block**
        - Contains partition details, like the no of blocks in the partition, size of the blocks, free-block count…
    - A **directory structure**
        - Used to organize the files
    - An **FCB (file control block)**
        - Contains file details, e.g. ownership, size…
- **A Typical File Control Block**:

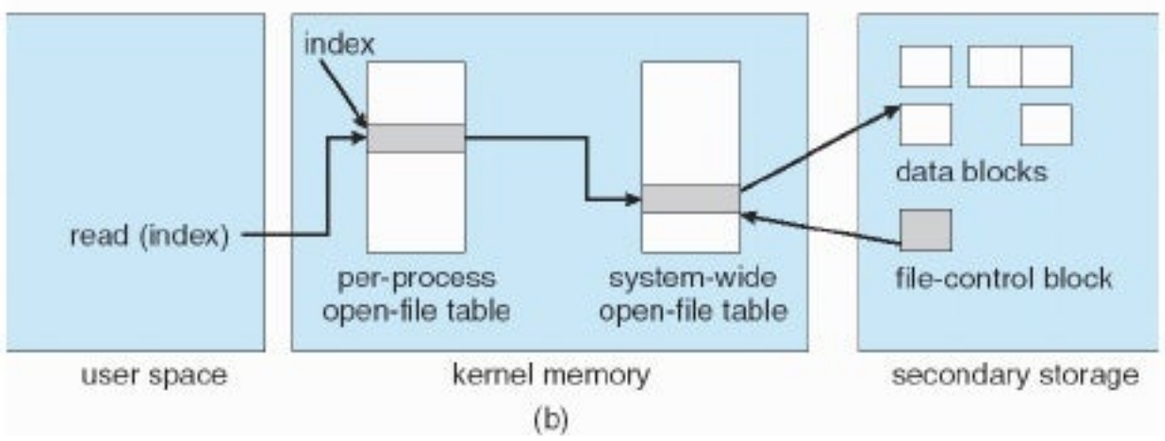
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

- In-memory structures is used for both file-system management and performance improvement via caching:
    - An **in-memory mount table**
        - Contains information about each mounted partition
    - An **in-memory directory structure**

        - Holds directory info of recently accessed directories
    - The **system-wide open-file table**
        - Contains a copy of the FCB of each open file
    - The **per-process open-file table**
        - Contains a pointer to the appropriate entry in the system-wide open-file table
    - **Buffers** hold file-system blocks when they are being read from disk or written to disk
- **In-Memory File System Structure: (a) File open, (b) File read**



(a)



(b)

- To create a new file, a program calls the logical file system (LFS)
- The 'LFS' knows the format of the directory structures
- To **create** a new file, it
    - Allocates a new FCB
    - Reads the appropriate directory into memory
    - Updates it with the new file name and FCB
    - Writes it back to the disk
- After a file has been created, it can be used for I/O
    - First the file must be opened
    - FCB: copied to a system-wide open-file table in memory
    - An entry is made in the **per-process** open-file table, with a pointer to the entry in the **system-wide** open- file table
    - The open call returns a pointer to the appropriate entry in the per-process file-system table
    - All file operations are then performed via this pointer
    - When a process closes the file
        - The per-process table entry is removed
        - The system-wide entry's open count is decremented

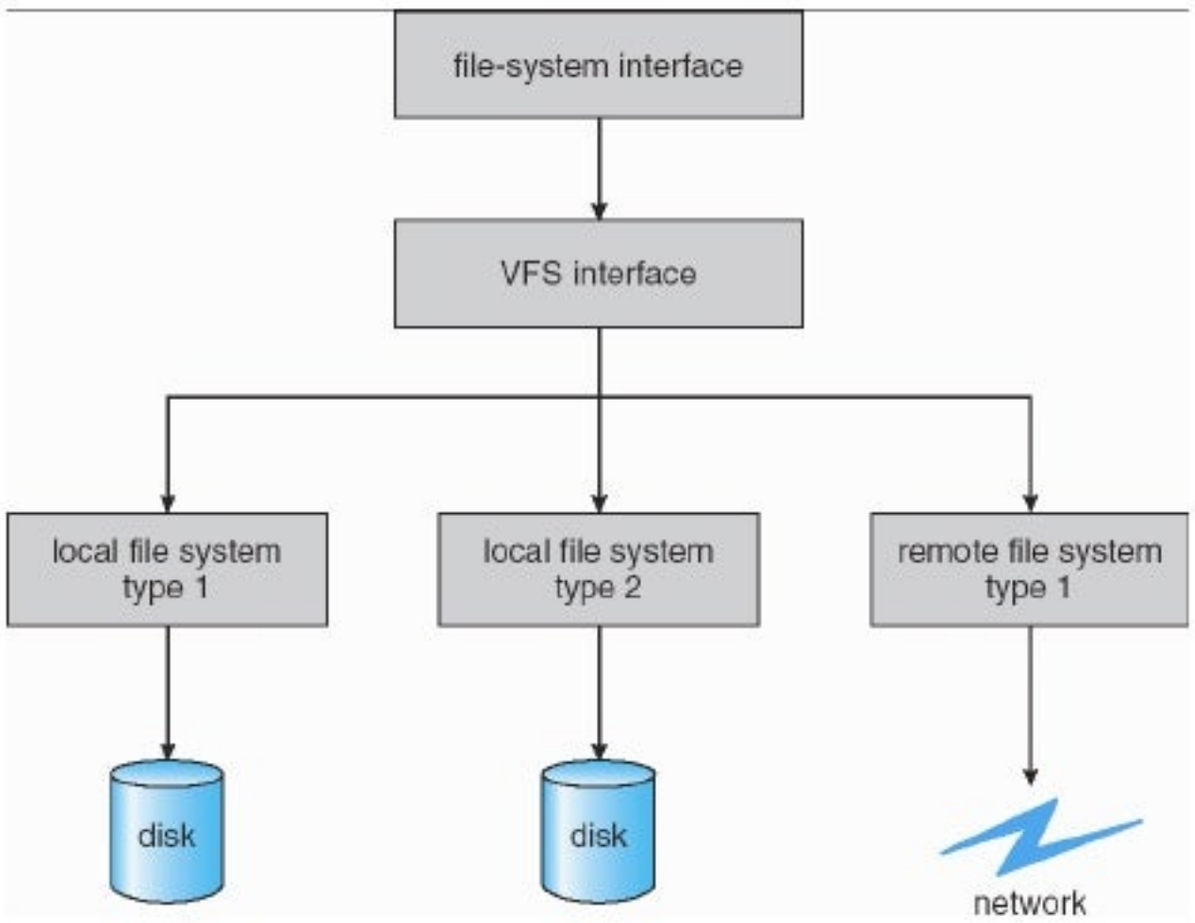
Partitions and Mounting

- Disk layouts:

    - A disk can be sliced into multiple partitions, or
    - A partition can span multiple disks (RAID)
- Each partition can either be:
    - Raw (containing no file system), or
    - Cooked (containing a file system)
- Boot info can be stored in a **separate partition** with its own format, since at boot time the system doesn't have file- system device drivers loaded and can't interpret the file-system format
    - Boot info format:
        - A sequential **series of blocks**, loaded as an **image** into memory, and execution of the image starts at a predefined location, such as the first byte
- The root partition containing the kernel is mounted at boot time
- Other partitions can be mounted at boot time, or later, manually
- As part of a successful mount operation, the OS verifies that the device contains a valid file system
- The OS notes in its in-memory mount table structure that a file system is mounted, and the type of the file system

Virtual File Systems

- The OS allows **multiple types** of file systems to be integrated into a directory structure (and be distinguished by the VFS!)
- The file-system implementation consists of 3 major layers:
    - File-system interface
    - Virtual File System (VFS) interface (serves 2 functions):
        - Separates file-system-generic operations from their implementation by defining a clean VFS interface
        - The VFS is based on a file-representation structure (vnode) that contains a numerical designator for a network-wide unique file, to support NFS
    - Local file system
- Schematic view of a virtual file system:



Directory Implementation

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system
- Here we look at the trade-offs involved in choosing one of these algorithms

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks
    - A linear list of file names has pointers to the data blocks
    - Requires a linear search to find a particular entry
    - Simple to program, but time-consuming to execute (linear search)
- A cache can store the most recently used directory information
    - A sorted list allows a binary search and decreases search times

Hash Table

- A linear list stores the directory entries, but a hash data structure is also used
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list
    - Some provision must be made for collisions
        - Situation in which two file names hash to the same location
- **Disadvantages**:
    - Fixed size of hash table and the dependence of the hash function on that size

Allocation Methods

- The direct-access nature of disks allows us flexibility in the implementation of files
    - In almost every case, many files are stored on the same disk
    - The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly
    - Three major methods of allocating disk space are in wide use:
        - **Contiguous**
        - **Linked**
        - **Indexed**
    - Each method has advantages and disadvantages
    - Some systems support all three (Data General's RDOS for its Nova line of computers)
    - More commonly, a system uses one method for all files within a file-system type

Contiguous Allocation

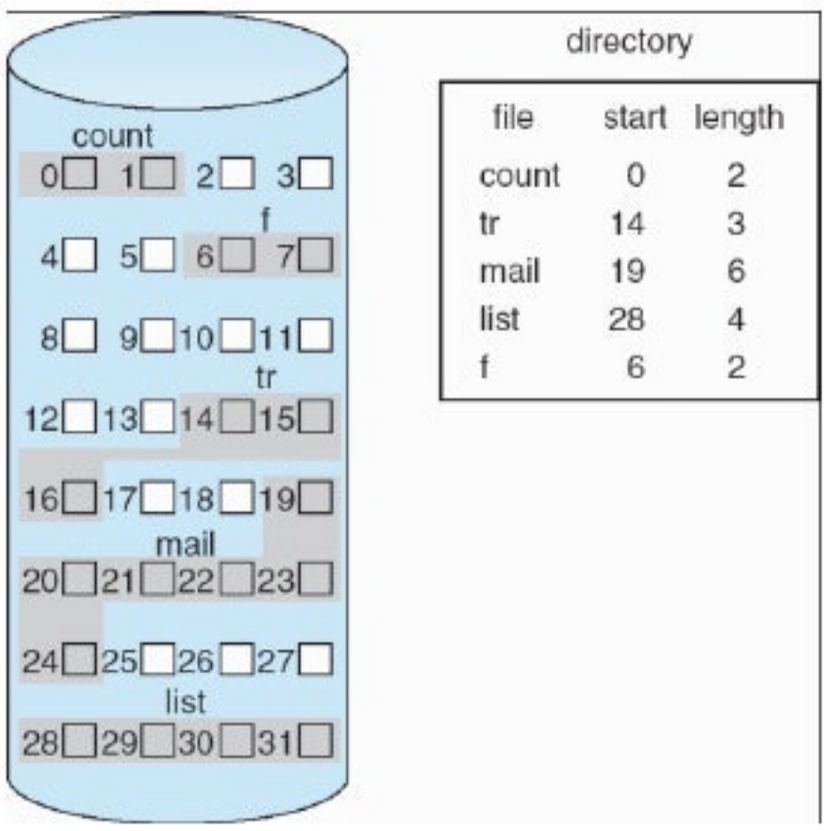
See also: Memory Allocation

Read p.472 mid NB!!!!!

- Each file occupies a set of contiguous blocks on the disk
- Disk addresses define a linear ordering on the disk
- The number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time

- Both sequential and direct access can be supported
- Problems with contiguous allocation:
    - Finding space for a new file
        - External fragmentation can occur
    - Determining how much space is needed for a file
        - If you allocate too little space, it can't be extended
        - If you allocate too much space, it may go unused
    - To minimize these drawbacks:
        - A contiguous chunk of space can be allocated initially, and then when that amount is not large enough, another chunk of contiguous space (an '**extent**') is added

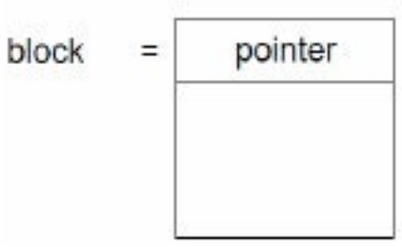
- **Contiguous Allocation of Disk Space**:



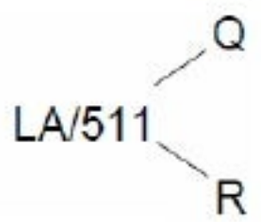
- An **extent** is a contiguous block of disks
    - Extents are allocated for file allocation
    - A file consists of one or more extents

Linked Allocation

- Each file is a linked list of disk blocks (scattered anywhere)

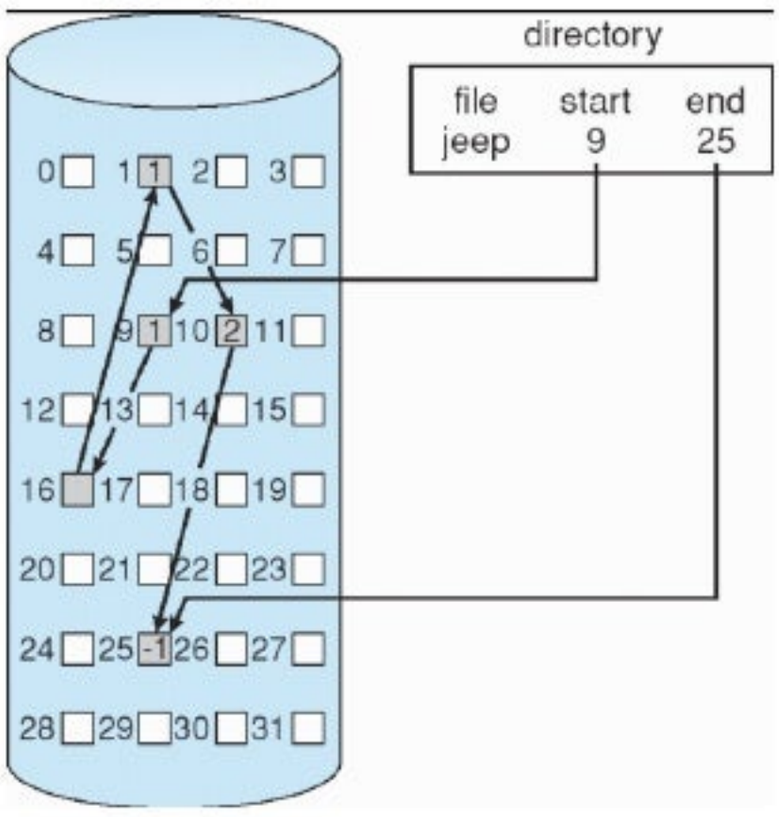


- The **directory contains a pointer to the first & last blocks**
- Each block contains a pointer to the next block
    - **Mapping**:

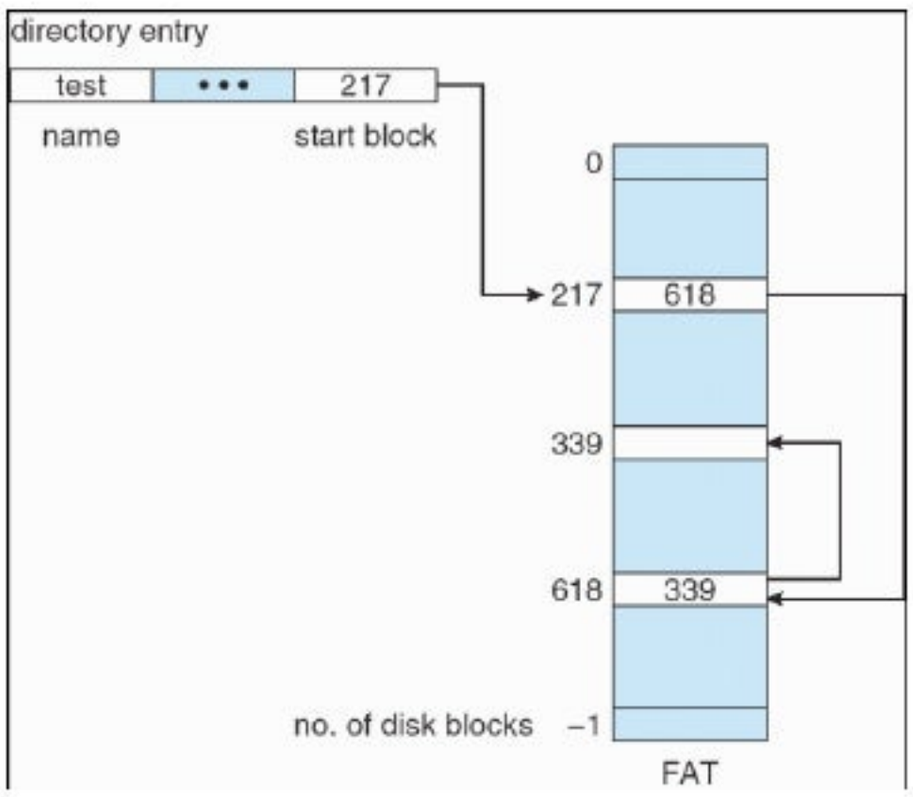


    - Block to be accessed is the Qth block in the linked chain of blocks representing the file
    - Displacement into block = R + 1

- To create a new file, just create a new entry in the directory
    - A write to the file causes a free block to be found
    - This new block is then written to and linked to the end of file
- **No external fragmentation, and any free block on the free-space list can be used to satisfy a request**
- The size of the file doesn't need to be declared on creation
- **Disadvantages**:
    - Can be used effectively only for **sequential-access** files
    - Space required for the **pointers**
        - Solution: collect blocks into multiples (‘**clusters**’) and allocate the clusters rather than blocks
    - **Reliability** (Problem if a pointer is lost / damaged)
- **Linked Allocation**:

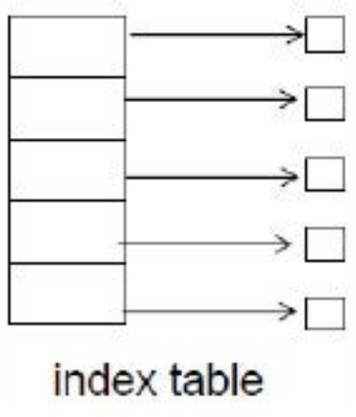


- **File Allocation Table (FAT)** = a variation on linked allocation:
    - A section of disk at the beginning of each partition is set aside to contain the table
    - The table has one entry for each disk block, and is indexed by block number
    - The **directory** entry contains the block number of the first block in the file
    - The table entry indexed by that block number then contains the block number of the next block in the file
    - This chain continues until the last block, which has a special end-of-file value as the table entry
    - The FAT can be cached to reduce the no. of disk head seeks
    - **Benefit**: improved access time, since the disk head can find the location of any block by reading the info in the FAT
- **File-Allocation Table**:

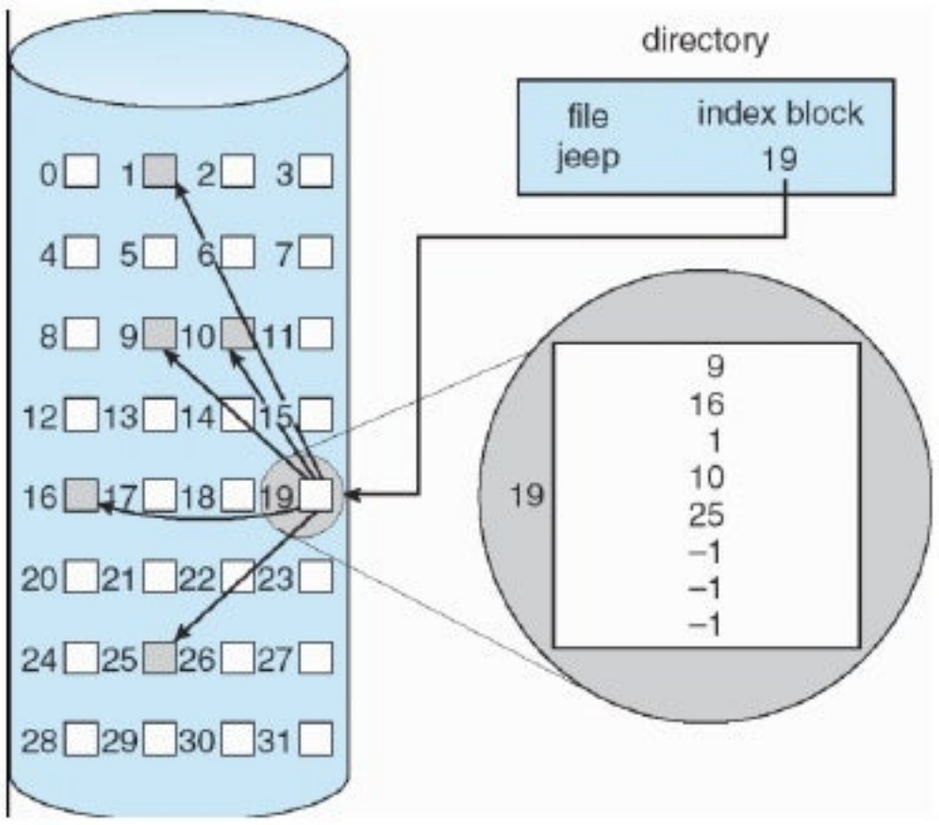


Indexed Allocation

- Solves the problems of linked allocation (without a FAT) by bringing all the pointers together into an index block
- Each file has an index block (an array of disk-block addresses)
- Logical view of the Index Table:

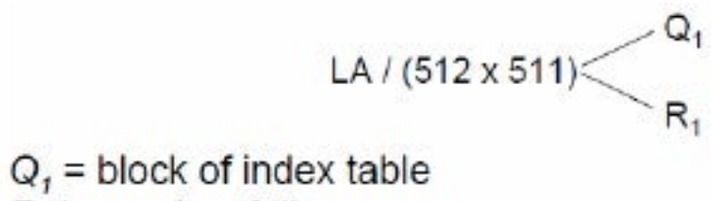


- The ith entry in the index block points to the ith file block
- The directory contains the address of the index block
- **Example of Index Allocation**:

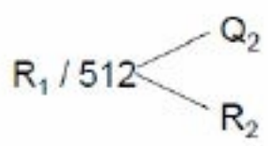


- When writing the ith block, a block is obtained from the free- space manager, and its address put in the ith index- block entry
- Supports **direct access**, without external fragmentation

- **Disadvantage**: **wasted space**: pointer overhead of the index block is greater than the pointer overhead of linked allocation
- We want the index block to be as small as possible, but what if file size is large?
    - Mechanisms to deal with the problem of index block size:
        - **Linked scheme**
            - To allow for large files, link several index blocks

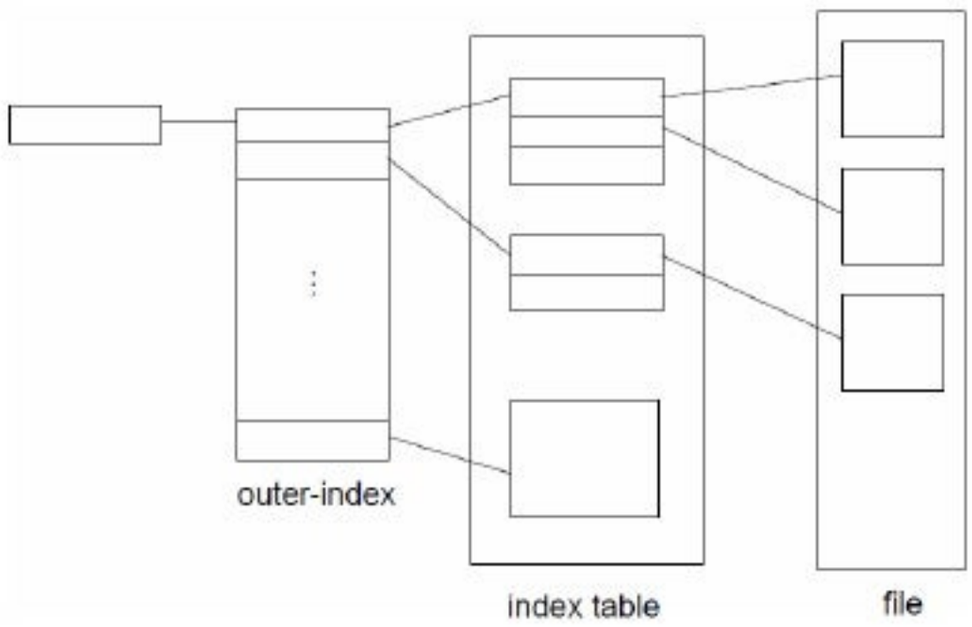


Q_1 = block of index table
R_1 is used as follows:

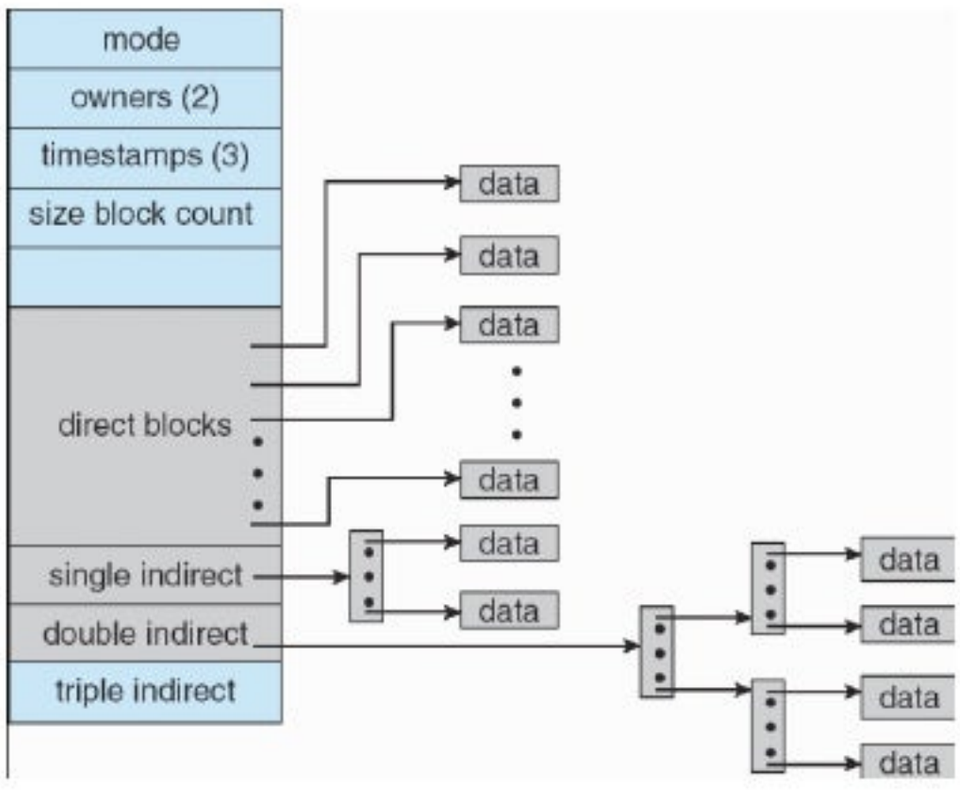


Q_2 = displacement into block of index table
R_2 displacement into block of file:

- **Multilevel index**
    - A first-level index block points to second-level ones, which in turn point to the file blocks



- **Combined scheme**
    - The first few pointers point to direct blocks
    - The next few point to indirect blocks
    - (Indirect blocks contain addresses of blocks)



- Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation

Performance

- The above allocation methods vary in their storage efficiency and data-block access times
- Contiguous allocation
    - Requires only one access to get a disk block
    - Since we can keep the file's initial address in memory, we can calculate immediately the disk address of the ith block
    - Good for **direct access**
- Linked allocation
    - For direct access, you need i disk reads to access block i
    - Good for **sequential access**
- Indexed allocation
    - Performance depends on the index structure, the size of the file, and the position of the block desired

Free-Space Management

- A **free-space list** keeps track of free disk space
- To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file
    - When a file is deleted, its disk space is added to the list

Bit Vector

- The free-space list is implemented as a bit map / bit vector
- Each block is represented by a bit: 1 = free; 0 = allocated
- Advantage: relative simplicity & efficiency in finding the first free block, or n consecutive free blocks
- Disadvantage: Inefficient unless the entire vector is kept in main memory (and is written to disc occasionally for recovery)
- Also look in PDF and PTT notes

Linked List

- Link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk
- The first block contains a pointer to the next free one, etc.
- Not efficient, because to traverse the list, each block is read
- Usually the OS simply needs a free block, and uses the first one

Grouping

- The addresses of n free blocks are stored in the 1st free block
- The first n-1 of these blocks are actually free
- The last block contains addresses of another n free blocks, etc

- Advantage: Addresses of a large no of free blocks can be found quickly, unlike in the standard linked-list approach

Counting

- Takes advantage of the fact that, generally, several contiguous blocks may be allocated / freed simultaneously
- Keep the address of the first free block and the number n of free contiguous blocks that follow the first block
- Each entry in the free-space list then consists of a disk address and a count

Space Maps

- p481 No description in notes...

Efficiency and Performance

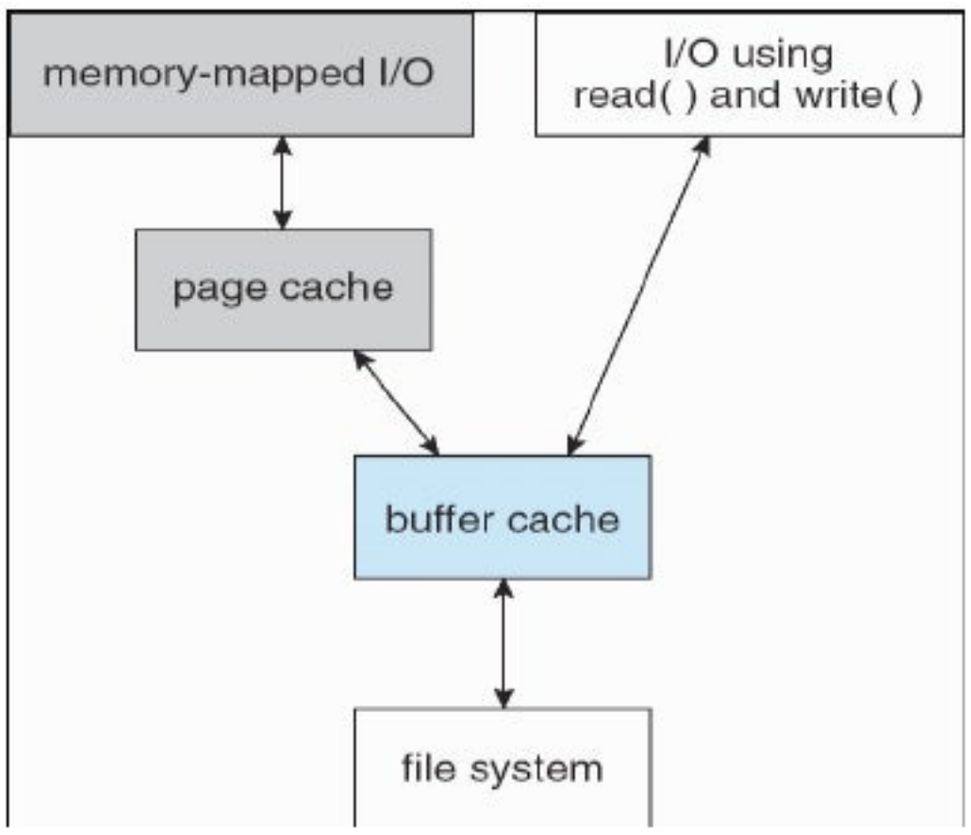
Efficiency

- The efficient use of disk space is heavily dependent on the disk allocation and directory algorithms in use
- The type of data kept in a directory also affect efficiency
    - Information like 'last write / access date' affect efficiency
- Small pointer sizes limit file length, but large ones take more space to store and make allocation methods use more disk space

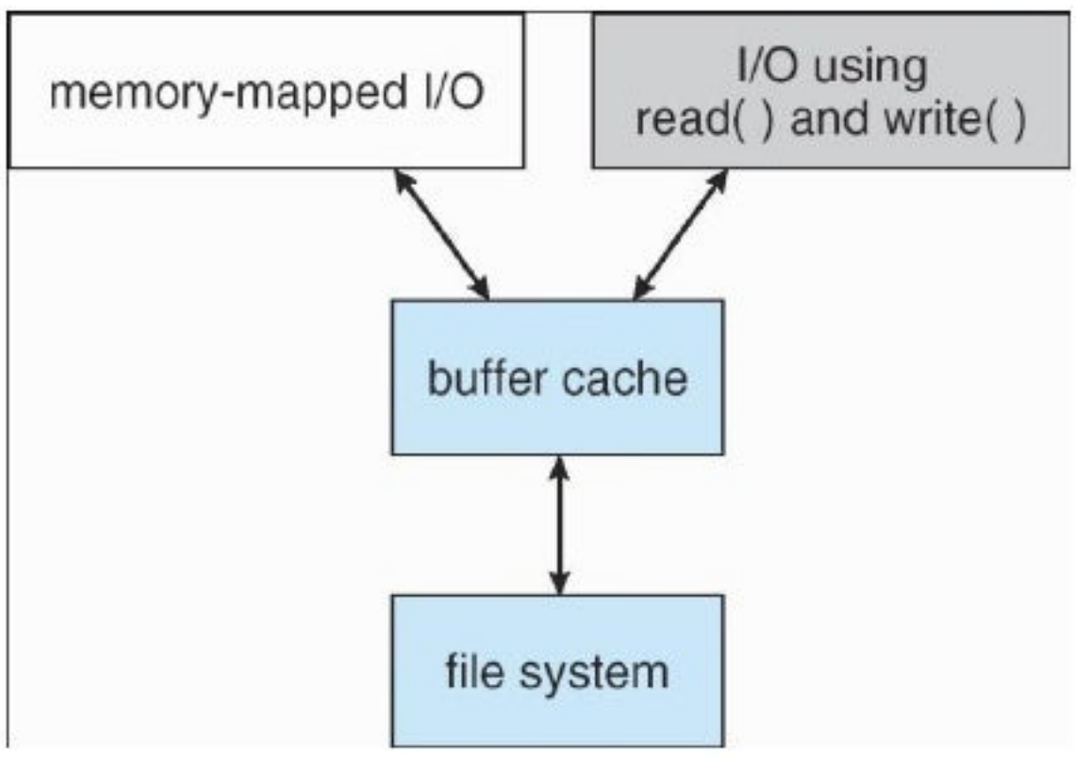
Performance

p483 - 486 for better description

- Most disk controllers include local memory to form an on-board **cache** that is large enough to store entire tracks at a time
- **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
- 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:



- A unified buffer cache uses the same page cache to cache both memory- mapped pages and ordinary file system I/O



Recovery

- **System crashes** can cause inconsistencies among on-disk file-system data structures, such as directory structures, free-block pointers, and free FCB pointers.
    - Changes to these file-system structures can be performed in place and if interrupted by system crashes.
    - This causes the actual data on disk to be inconsistent to the state of the file-system structures.
- In addition to crashes, bugs in the file-system implementation, disk controllers, and even user applications can corrupt a file system.
- File systems have varying methods to deal with corruption, depending on the file-system data structures and algorithms.
- We deal with those issues next.

Consistency Checking

- Part of the directory info is kept in main memory (cache) to speed up access
- The directory info in main memory is more up to date than is the corresponding info on the disk
- When the computer crashes, the table of opened files is lost, as well as changes in the directories of opened files

- This event can leave the file system in an inconsistent state
- The consistency checker compares the data in the directory structure with the data blocks on disk and fixes inconsistencies (fsck in UNIX or chkdsk in Windows)
    - p487 top

Log-Structured File Systems

See also: Log-Based Recovery

- Similar idea to p260 section 6.9.2
- All metadata changes are written sequentially to a log
- Transaction = a set of operations that perform a specific task
- Once the changes are written to this log, they are committed
- When an entire committed transaction is completed, it is removed from the log file which is actually a circular buffer
- Side benefit of using logging on disk metadata updates: those updates proceed much faster than when they are applied directly to the on-disk data structures

Other Solutions

p488

Backup and Restore

- System programs can back up data from disk to permanent storage
- Use system programs to **back up** data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup
- To minimize copying, use info from each file's directory entry
- E.g. backup programs see when the last backup of a file was done
- Look at typical backup schedule on p.489 mid

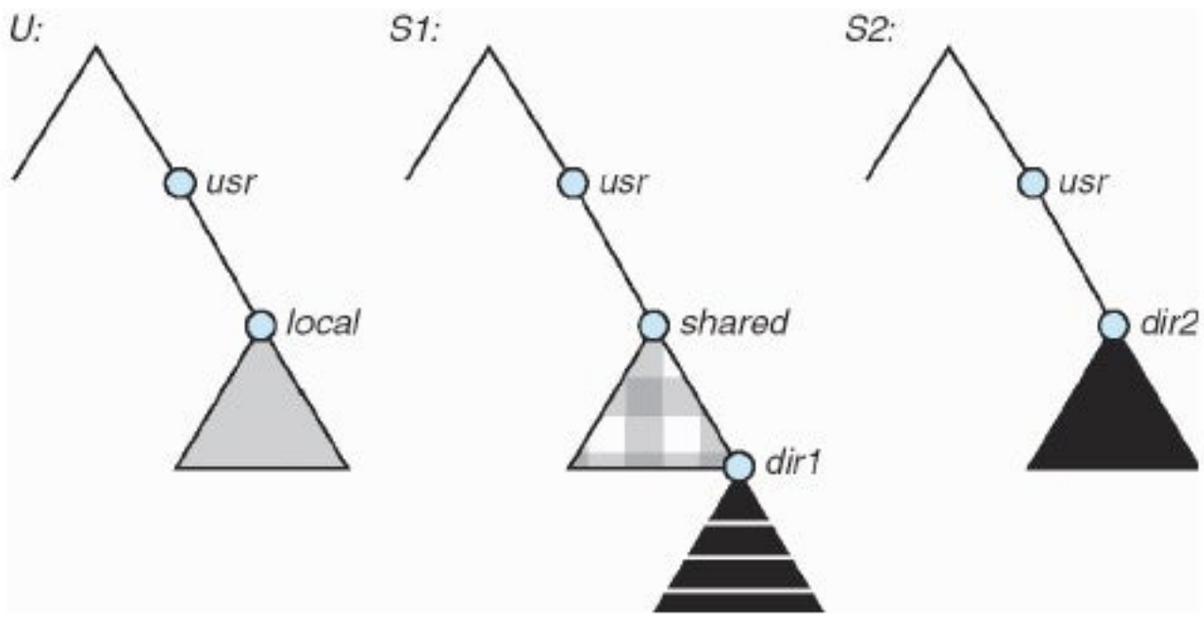
NFS

- 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

Overview

- Sharing is based on a client-server relationship
- Sharing is allowed between any pair of machines
- To ensure machine independence, sharing of a remote file system affects only the client machine and no other machine
- For a remote directory to be accessible from a machine, a client of that machine has to carry out a **mount** operation first
- A remote directory is **mounted over** a directory of a local file system and looks like a sub-tree of the local file system
- The local directory becomes the name of the root of the newly mounted directory

- No transitivity: the client doesn't gain access to other file systems that were mounted over the former file system
- 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 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



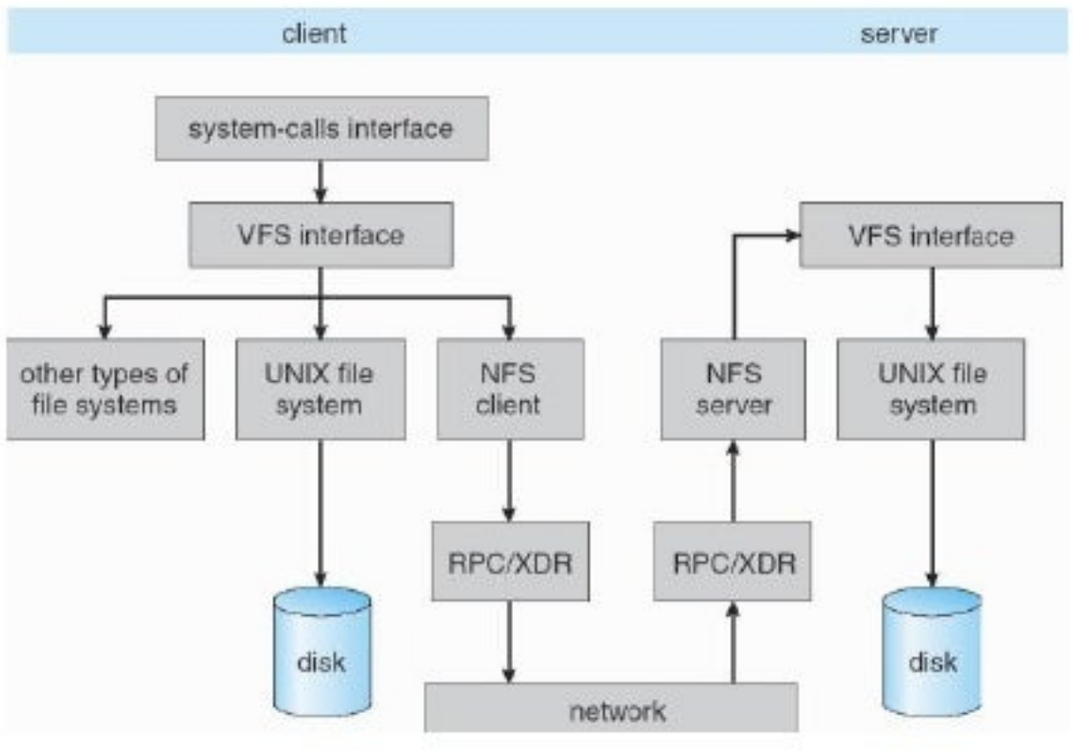
The Mount Protocol

- 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
- The server also maintains a list of clients and currently mounted directories, which it uses for admin purposes

The NFS Protocol

- Provides a set of RPCs for remote 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 & writing files
- These procedures can be invoked only after a file handle for the remotely mounted directory has been established
- NFS servers are **stateless** and don't maintain info about clients
- A server crash and recovery will be invisible to a client
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol doesn't 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



Path-Name Translation

- Path-name translation is done by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- Once a mount point is crossed, every component lookup causes a separate RPC to the server

Remote Operations

- A remote file operation can be translated directly to a RPC

Example: The WAFL File System

- Used on Network Appliance "Filers"–distributed file system appliances

- "Write-anywhere file layout"
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
- NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications

Summary

Mass-Storage Management

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

Caching

- Cache management is important because of cache's limited size
- Data transfer from cache to CPU and registers is usually a hardware function, with no OS intervention
- Transfer of data from disk to memory is controlled by the OS
- Coherency and consistency
    - In the storage hierarchy, data appears in different levels
    - When it is modified, its value differs in the various storage
    - Its value only becomes the same throughout after being written from the internal register back to the e.g. magnetic disk
    - In a computing environment where one process executes at a time, access to the data will be to the copy at the highest level
    - In a multitasking environment, care must be taken to ensure that the several processes access the most recently updated value
    - In a multiprocessor environment, where the data exists in several caches simultaneously, an update to the data in one cache must be immediately reflected in all other caches where it resides (Cache coherency)
- In a distributed environment, when a replica is updated in one place, all other replicas must be brought up-to-date

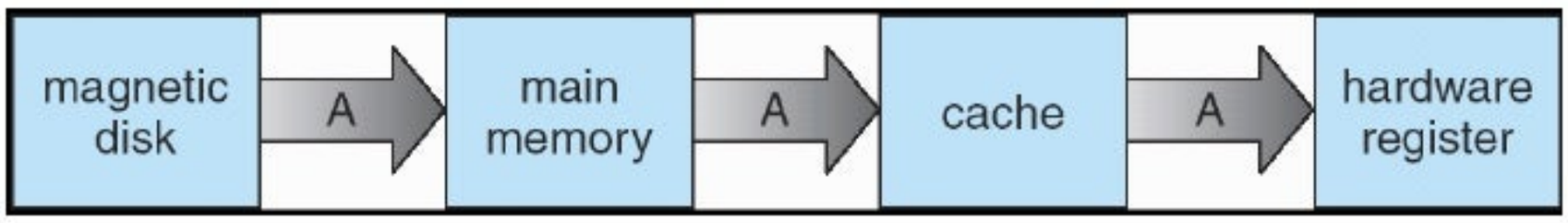
****

- Caching is an important principle of computer systems
    - Information is normally kept in some storage system (such as main memory)
    - As it is used, it is copied into a faster storage system - the cache - on a temporary basis
    - When we need a particular piece of information, we first check whether it is in the cache
        - If it is, we use the information directly from the cache
        - If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon
- In addition, internal programmable registers, such as index registers, provide high-speed cache for main memory
    - The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory
    - There are also caches that are implemented totally in hardware
        - For instance, most systems have an instruction cache to hold the instructions expected to be executed next
    - Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory
    - For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy
    - We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system
- Because caches have limited size, **cache management** is an important design problem
    - Careful selection of the cache size and of a replacement policy can result in greatly increased performance
    - The figure below compares storage performance in large workstations and small servers

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

    - Various replacement algorithms for software-controlled caches are discussed in Chapter 9
- Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping
    - The file-system data, which resides on secondary storage, may appear on several levels in the storage hierarchy
        - At the highest level, the operating system may maintain a cache of file-system data in main memory

        - In addition, electronic RAM disks (also known as solid-state disks) may be used for high-speed storage that is accessed through the file-system interface
    - The bulk of secondary storage is on magnetic disks
        - The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of hard-disk failure
        - Some systems automatically archive old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost
- The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software
    - For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating system intervention
    - In contrast, transfer of data from disk to memory is usually controlled by the operating system
- In a hierarchical storage structure, the same data may appear in different levels of the storage system
    - For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk
        - The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory
        - This operation is followed by copying A to cache and to an internal register
        - Thus, the copy of A appears in several places:
            - on the magnetic disk
            - in main memory
            - in the cache
            - and in an internal register
            - (see the next figure)



        - Once the increment takes place in the internal register, the value of A differs in the various storage systems
        - The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk
- In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy
    - However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A
- The situation becomes even more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache
    - In such an environment, a copy of A may exist simultaneously in several caches
    - Since the various CPUs can all execute concurrently, we must make sure that an update to the value A in one cache is immediately reflected in all other caches where A resides
    - This situation is called **cache coherency**, and it is usually a hardware problem (handled below the operating-system level)

- In a distributed environment, the situation becomes even more complex
  - In this environment, several copies (or replicas) of the same file can be kept on different computers that are distributed in space
  - Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible
  - There are various ways to do this - discussed in Chapter 17

I/O Systems

- The OS must hide peculiarities of hardware devices from users
- In UNIX, the peculiarities of I/O devices are hidden from the bulk of the OS itself by the I/O subsystem
- The I/O subsystem consists of
  - A memory-management component that includes buffering, caching, and spooling
  - A general device-driver interface
  - Drivers for specific hardware devices
- Only the device driver knows the peculiarities of the specific device to which it is assigned

Protection and Security

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

PART SIX: PROTECTION AND SECURITY

Chapter 14: System Protection

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems

Goals of 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.
- Reasons for providing protection:
  - To prevent mischievous violation of an access restriction

    - To ensure that each program component active in a system uses system resources only in ways consistent with policies
- Mechanisms are distinct from policies
    - Mechanisms determine how something will be done
    - Policies decide what will be done
- This principle provides flexibility

Principles of Protection

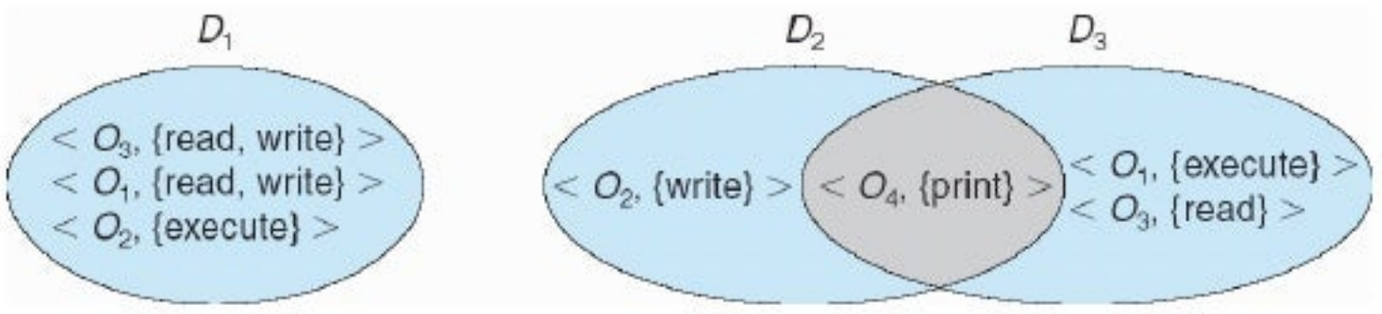
- Guiding principle –principle of least privilege
    - Programs, users and systems should be given just enough privileges to perform their tasks
- p.592

Domain of Protection

- A process should be allowed to access only authorized resources
- Need-to-know principle: a process should be able to access only those resources that it currently requires to complete its task

Domain Structure

- A protection domain specifies the resources a process may access
- A domain is a collection of access rights, each of which is an ordered pair <object-name, rights-set>
    - Access right = the ability to execute an operation on an object
    - Access-right = <*object-name*, *rights-set*> where *rights-set* is a subset of all valid operations that can be performed on the object
    - Domains also define the types of operations that can be invoked



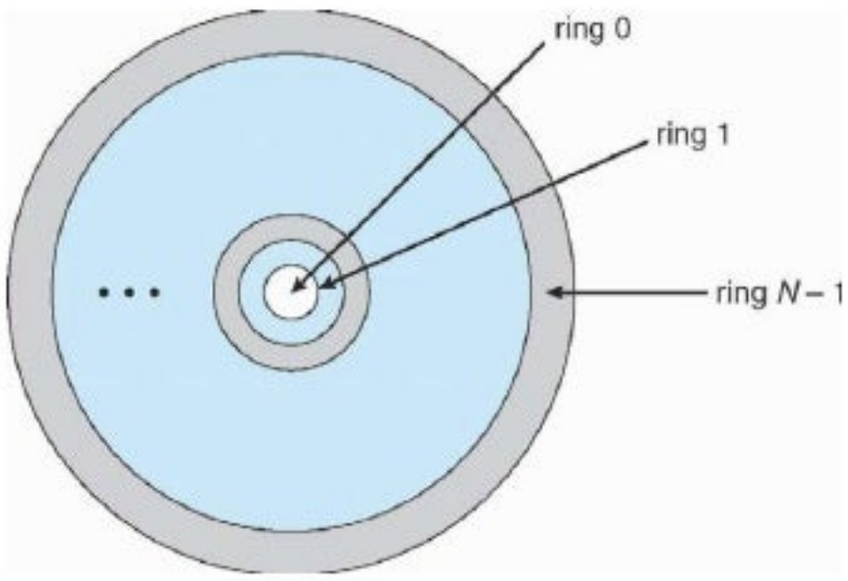
- The association between a process and a domain may be
    - Static (if the process' life-time resources are fixed)
        - Violates the need-to-know principle
    - Dynamic
        - A process can switch from one domain to another
- A domain can be realized in several ways:
    - Each user may be a domain
        - Domain switching occurs when a user logs out
    - Each process may be a domain
        - Domain switching occurs when a process sends a message to another process and waits for a response
    - Each procedure may be a domain
        - Domain switching occurs when a procedure call is made

An Example: 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 (set uid 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.

An Example: MULTICS

- Let *Di* and *Dj* be any two domain rings.
- If $j < i \Rightarrow Di \subseteq Dj$



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 Domaini can invoke on Objectj
- Example:

	F1	**F2**	**F3**	**Printer**
D1	Read		Read	
D2				Print
D3		Read	Execute	
D4	Read, Write		Read, Write	

- When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain)
- We can include domains in the matrix to control domain switching
- In the following table, D1 can only switch to D2:

	F1	**F2**	**F3**	**Printer**	**D1**	**D2**	**D3**	**D4**

D1	Read		Read			Switch		
D2				Print			Switch	Switch
D3		Read	Execute					
D4	Read, Write		Read, Write		Switch			

- Allowing controlled change to the contents of the access-matrix entries requires three additional operations:
- **Copy**
    - * denotes the ability for one domain to copy the access right to another domain (row)
    - If the right is then removed from the original domain, it is called a **transfer**, not a copy
    - The * isn't copied as well
- **Owner**
    - Allows the addition and removal of rights
    - **'Owner' in a column means that the process executing in that domain can add / delete rights in that column**
- **Control**
    - The control right is applicable only to domain objects
    - 'Control' in access(D2,D4) means that a process executing in domain D2 can modify the row D4
- The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environments is called the **confinement problem**

Implementation of Access Matrix

- Here we look at ways to implement the access matrix.

Global Table

- Contains a set of ordered triples: <domain, object, rights-set>
- Drawbacks:
    - The table is usually large and can't be kept in main memory
    - It is difficult to take advantage of groupings, e.g. if all may read an object, there must be an entry in each domain

Access Lists for Objects

- Each column in access list can be implemented as an access list for one object
- Each column = Access-control list for one object
    - Defines who can perform what operation.

        Domain 1 = Read, Write

        Domain 2 = Read

        Domain 3 = Read

        ⋮
- Resulting list: <domain, rights-set> for all non-empty columns

- Corresponds to users' needs: When you create an object, you can specify which domains may access it
- Determining the set of access rights for a domain is difficult

Capability Lists for Domains

- A **capability list** for a domain is a list of objects together with the operations allowed on those objects
  - An object is represented by its physical name or address, called its capability
  - Simple possession of the capability means that access is allowed
- Resulting list: Objects, with operations allowed on them
- Capabilities are useful for localizing info for a given process
- Capabilities are distinguished from other data in one of 2 ways:
  - Each object has a **tag** to denote its type as either a capability or as accessible data
  - The program's address space can be split into two parts:
    - One part contains data, accessible to the program
    - The other part contains the capability list, accessible only to the OS
- 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

A Lock-Key Mechanism

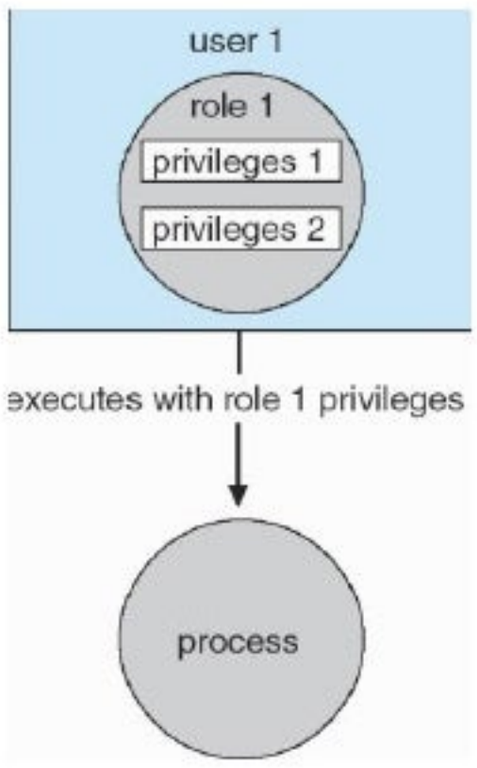
- Compromise between access lists and capability lists
- Each object has locks and each domain has keys
  - Unique bit patterns
- A process executing in a domain can access an object only if that domain has a key that matches one of the object's locks
- Not accessible by users directly

Comparison

- Check p.604

Access Control

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



Revocation of Access Rights

- Questions about revocation that may arise:
    - Does it occur immediately or is it delayed?
    - Does it affect all users or only a group?
    - Can a subset of rights be revoked, or must they all be?
    - Can access be revoked permanently or temporarily?
- *Access List* - Delete access rights from access list.
    - Simple
        - Search the access list for the right(s) to be revoked, then delete it once found
    - Immediate
- Revocation is more difficult with capabilities:
    - Capabilities are distributed throughout the system, so we must find them first
    - *Capability List* - Scheme required to locate capability in the system before capability can be revoked:
        - Reacquisition
            - Periodically, capabilities are deleted from each domain, and the process may try to reacquire the capability
        - Back-pointers
            - Pointers point to all capabilities associated with an object, and can be followed when revocation is required
        - Indirection
            - Each capability points to an entry in a global table, which in turn points to the object
        - Keys
            - A key is associated with each capability and can't be modified / inspected by the process owning the capability
            - Master key is associated with each object; can be defined or replaced with the *set-key* operation

Capability-Based Systems

- These systems are not widely used.

An Example: Hydra

- Capability based system that provides flexibility
- Fixed set of access rights known to and interpreted by the system
    - Access rights:
        - read
        - write
        - execute a memory segment
- User can declare other rights.
- Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights.
- Operations on objects are defined procedurally.
    - These procedures are objects themselves and are accessed indirectly by capabilities.
    - When the definition of an object is made known by Hydra, the names of operations on the type become **auxiliary rights.**
- Auxiliary rights are described in a capability for an instance of the type.
- Provides **rights amplification.**
    - Procedures are certified as trustworthy to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure.
- Amplification allows implementation procedures access to the representation variables of an abstract data type.
- When a process invokes the operation *P* on an object *A*, however, the capability for access to *A* may be amplified as control passes to the code body of *P*.
- When a user passes an object as an argument to a procedure, we may need to ensure that the procedure cannot modify the object.
- The procedure-call mechanism of Hydra was designed as a direct solution to the problem of mutually suspicious subsystems.
- A Hydra subsystem is built on top of its protection kernel and may require protection of its own components.

An Example: Cambridge CAP System

- CAP's capability system is simpler and less powerful than that of Hydra.
- It also provide secure protection of user-defined objects.
- Two kinds of capabilities:
    - 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.
- The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains.

Language-Based Protection

- To the degree that protection is provided in existing computer systems, it is usually achieved through an operating-system kernel, which acts as a security agent to inspect and validate each attempt to access a protected resource.
- Protection systems are now concerned not only with the identity of a resource to which access is attempted but also with the functional nature of that access. In the newest protection systems, concern for the function to be invoked extends beyond a set of system-defined functions, such as standard file access methods, to include functions that may be user-defined as well.
- Protection can no longer be considered a matter of concern only to the designer of an operating system. It should also be available as a tool for use by the application designer, as that resources of an applications subsystem can be guarded against tampering or the influence of an error.

Compiler-Based Enforcement

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.
- When protection is declared along with data typing, the designer of each subsystem can specify its requirements for protection, as well as its need for use of other resources in a system.
    - Such specification should be given directly as a program is composed, and in the language in which the program itself is stated.
- This approach has several significant advantages:
    - Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.
    - Protection requirements can be stated independently of the facilities provided by a particular operating system.
    - The means for enforcement need not be provided by the designer of a subsystem.
    - A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.
- A variety of techniques can be provided by programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system.
- If a system does not provide a protection kernel as powerful as those of Hydra or CAP, protection can still be implemented using specifications given in a programming language.
    - This kind of security will not be as good as protection implemented using a protection kernel. This is because the mechanism relay more on assumptions about the operational state of the system.
- The merits of security enforced by kernel opposed to enforcement by compiler:
    - Security
        - Enforcement by a kernel provides a greater degree of security of the protection system itself than does the generation of protection-checking code by a compiler.
    - Flexibility
        - There are limits to the flexibility of a protection kernel in implementing a user defined policy, although it may supply adequate facilities for the system to provide enforcement of its own policies
        - With a programming language, protection policy can be declared and enforcement provided as needed by an implementation.

- Efficiency
    - The greatest efficiency is obtained when enforcement of protection is supported directly by hardware (microcode).
    - Language based enforcement has the advantage that static access enforcement can be verified off-line at compile time.
    - Intelligent compilers can tailor the enforcement mechanism so that the fixed overhead of kernel calls can often be avoided.

- Read the summary on p.612 (mid)
- What is needed is a safe, dynamic access-control mechanism for distribution capabilities to system resources among user processes.
- To be useful in practice it soul be reasonably efficient.
    - This has led to the development of a number of language constructs that allow the programmer to declare various restrictions on the use of specific managed resources.
    - These resources provide mechanisms for three functions:
        - Distributing capabilities safely and efficiently among customer processes.
        - Specifying the type of operations that a particular process may invoke on an allocated resource.
        - Specifying the order in which a particular process may invoke the various operations of a resource.
- 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

- 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.
    - Look at example p.614 (Top)
- Stack inspection:
    - This philosophy require the class to explicitly permit a network connection.
    - By doing this the method takes responsibility for the request.
- The following figure shows 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);
			...

- A Java program cannot directly access memory; it can manipulate only an object for which it has a reference.
- Java's load-time and run-time checks enforce **type safety** of Java classes
    - Type safety ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways.
- This is the foundation of Java protection, since it enables a class to effectively encapsulate and protect its data and methods from other classes loaded in the same JVM.

Summary

Chapter 15: System Security

- **Protection** (Ch 14) is strictly an internal problem: How controlled access to programs and data stored in a computer is provided.
- **Security** on the other hand, requires not only an adequate protection system but also consideration of the external environment within which the system operates.
- A protection system is effective if user authentication is compromised or a program is rum by an unauthorized user.
- Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.
- These resources include information stored in the system (code and data), as well as the CPU, memory, disks, tapes, and networking.
- Here we start by examining ways in which resources may be accidentally or purposely misused.
- We also look at key security enablers like cryptography.
- We then look at mechanisms to guard against or detect attacks.

Chapter Objectives:

- To discuss security threads and attacks
- To explain the fundamentals of encryption, authentication, and hashing
- To examine the uses of cryptography in computing
- To describe various countermeasures to security attacks

The Security Problem

- A system is **secure** if its resources are used and accessed as intended under all circumstances.
- Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental.
    - Easier to protect against accidental misuse than against malicious misuse.
- Note the following:
    - An **intruder** or **attacker** are people who are attempting to break security.
    - A **thread** is the potential for a security violation, whereas an **attack** is the attempt to break security.
- A list of several forms of accidental and malicious security violations (p.622):
    - Breach of **confidentiality**:
        - The unauthorized reading of data
    - Breach of **integrity**:
        - The unauthorized modification of data

- Breach of **availability**:
  - The unauthorized destruction of data (ex. defacement)
- **Theft of service**:
  - The unauthorized use of resources
- **Denial of service** (DOS):
  - The prevention of the legitimate use of a system

- Standard methods used to attempt to breach security:
  - **Masquerading**:
    - One participant in a communication pretends to be someone else
  - **Replay attack** (Captured exchange):
    - The malicious or fraudulent repeat of a valid data transmission
    - Used with message modification to escalate privileges
  - **Man-in-the-middle attack**:
    - The attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa
    - Used after a session hijacking took place where the active communication session was intercepted

- To protect a system, we must take security measures at **4 levels**:

  1) Physical (Armed entry)

  2) Human (Users being bribed / tricked)
    - Social Engineering
    - Phishing
    - Dumpster diving

  3) Operating System (Security breaches to the system)

  4) Network (Intercepting data over private lines)

- Security at the first two levels must be maintained if operating-system security is to be secured.
- The system must provide protection (ch 14) to allow the implementation of security features.

Program Threats

- Writing a program to create a breach of security or causing a normal process to change its behavior and create a breach is a common goal of crackers
- While it is useful to log into a system without authorization it is even more useful to leave behind a back-door daemon that provides information or allows easy access even if the original exploit is blocked
- Multi-user computers are generally not prone to viruses because the executable programs are protected from writing by the OS

Trojan Horse

- A code segment that misuses its environment
- Examples:
  - login program emulation
  - Spyware (pop-ups, covert channels - where surreptitious communication occurs

Trap Door

- A programmer leaves a hole in the software that only he can use

Logic Bomb

- A program that initiates a security incident only under certain circumstances (e.g. date and time or looking for a specific parameter to change)

Stack and Buffer Overflow

- The most common way for an attacker to gain unauthorized access
- An authorized user may also use this exploit for privilege escalation, to gain privileges beyond those allowed for him
- Essentially, the attack exploits a bug in a program
- The attacker finds the vulnerability and writes a program to:
    - Overflow an input field, command-line argument, or input buffer… until it writes into the stack
    - Overwrite the current return address on the stack with the address of the exploit code loaded in step 3
    - Write a simple set of code for the next space in the stack that includes commands that the attacker wants to execute
- This attack can be countered by doing bounds checking on inputs
- **Read p.628-629 NB!!!**

Viruses

- A virus is a fragment of code embedded in a legitimate program
- Whereas a worm is structured as a complete, standalone program, a virus is a fragment of code embedded in a legitimate program
- Viruses are very specific to architectures
- Viruses are normally hidden in Trojan horse programs acting as virus droppers
- Viruses normally belong to more than one category:
    - **File**:
        - A virus appends itself to a file then after the program is executed it removes itself from the file and return control to the program
    - **Boot:**
        - It infects the boot sector and executes every time the computer boots. It sits in memory and infects all other boot sector disks which is inserted to the computer
    - **Macro:**
        - Written in a high-level language and triggered when a program capable of executing the macro is run
    - **Source code:**
        - This virus looks for source code and include the virus to help distribute the virus
    - **Polymorphic:**
        - Virus changes each time installed to avoid detection. Done to change virus's signature. Signature is a pattern that is used to detect the virus
    - **Encrypted:**
        - Encrypted to avoid detection. Decryption included to decrypt and to infect a target
    - **Stealth:**
        - It attempts to avoid detection by modifying parts of the system that normally detects it

- **Tunneling:**
    - Bypass detection by anti-virus programs by installing itself in interrupt-handler chain and into device drivers
- **Multipartite:**
    - Infect multiple parts of the system (Boot sector; memory; files)
- **Armored:**
    - Compressed to avoid detection and disinfection by anti-virus programs. Difficult to understand by anti- virus researchers

- **Keystroke logger**:
    - Record all thing entered on keyboard
- **Monoculture**:
    - An environment where many systems run the same hardware, operating system, and/or application software

System and Network Threats

- System and network threads involve the abuse of services and network connections
- System and network threats create a situation in which operating-system resources and user files are misused
- System and network attacks are used to launch a program attack, and vice versa
- The more **open** an operating system is the more services are running and the more likely a bug is available to be exploit
- The **attack surface** is the set of ways an attacker can try to break into a system

Worms

- **(p.634)**
- A worm spawns copies of itself, using up system resources
- The worm spawns copies of itself using up all system resources and locking out all other processes

Port Scanning

- Port scanning is a means for a cracker to detect a system's vulnerabilities to attack
- It normally attempts to create a TCP/IP connection to a specific port or a range of ports
- Port scans are normally launched from **zombie systems**
- Such systems are previously compromised systems that are serving their owners while being used for nefarious purposes, including denial-of-service attacks and spam relay

Denial of Service

- Involves disabling legitimate use of a system / facility
- E.g. an intruder could delete all the files on a system
- Generally network based attacks, and fall into two categories:
    - An attack that uses so many facility resources that no useful work can be done
    - Disrupting the network of the facility
- Distributed Denial-Of-Service attacks (DDOS):
    - Attacks launched from multiple sites at once

Cryptography as a Security Tool

- We look at details of cryptography and its use in computer security
- All computers on a network sent and receive bits onto and from the wire without knowing from where they come or where they go to
- Constrains the potential senders and receivers of a message
- Keys are distributed to computers to process messages
- Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key - the key is the source of the of the message

Encryption

- A means for constraining the possible **receivers** of a message
- An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message
- An encryption algorithm consists of the following components:
    - A set K of keys
    - A set M of messages
    - A set C of ciphertexts
    - A function E:K->(M->C)
    - A function D:K->(C->M)
- An encryption algorithm must provide this essential property:
    - Given a chipertext cEC, a computer can compute m such that E(k)(m)=c only if it possesses D(k)
- Two main types of encryption algorithms:
    - Symmetric encryption algorithm
    - Asymmetric encryption algorithm

Symmetric Encryption

- In a symmetric encryption algorithm, the same key is used to encrypt and decrypt
- E(k) is derived from D(k), hence E(k) must be protected to the same extend as D(k)
- Data-Encryption Standard (DES)
    - Breaks messages up into 64-bit blocks
    - Keys are 56-bit key
    - Called a block cipher
    - Cipher-block chaining:
        - The chunks are XORed with the previous ciphertext
        - This is done to prevent that the ciphertext can be used to determine the encryption and decryption keys
    - Triple-DES was developed since DES was deemed insecure
- Advanced Encryption Standard (AES)
    - Breaks messages up into 128-bit blocks
    - Key lengths are 128, 192, 256-bits
- Stream ciphers are designed to encrypt and decrypt a stream of bytes or bits rather than a block

- RC4
- Use pseudo-random-bit generator to produce random bits which is fed a key and delivers a keystream
- A keystream is an infinite set of keys that can be used for the input plaintext stream
- RC4 is used to encrypt streams of data, such as WEP
- Used to encrypt Web communications between Web browsers and Web servers

Asymmetric Encryption

- In an asymmetric encryption algorithm, different keys are used to encrypt and decrypt
- RSA
    - A block-cipher public-key algorithm
    - The encryption key is called the public key and is distributed freely
    - The decryption key needs to be kept secret and is thus called the private key
- It is not feasible to calculate the decryption key from the encrypted text
- More safe and are used authentication, confidentiality and key distribution
- Needs more computer power to execute

Authentication

- Authentication is used to constrain the set of potential **senders** of a message
- Authentication is complementary to encryption
- Used to prove that a message has not been modified
- Components of an authentication algorithm:
    - A set K of keys
    - A set M of messages
    - A set A of authenticators
    - A function S:K->(M->A)
    - A function V:K->(MxA->{true, false}) (Verification function)
- A **hash function** H(m) creates a small fixed-size block of data, known as a **message digest** or **hash value**, from a message m
    - A hash function work by taking a message in n-bit blocks and processing the blocks to produce an n-bit hash
    - H must be collision resistant on m - that is, it must be infeasible to find an m' not equal to m such that H(m)=H(m')
    - If H(m)=H(m') we know that m=m' - that is, we know that the message has not been modified
    - Two common message-digest functions:
        - MD5 outputs 128-hash
        - SHA-1 outputs 160-bit hash
    - Message digests are useful for detecting changed messages but are not useful as authenticators
    - An authentication algorithm takes the message digest and encrypts it
- Two varieties of authentication algorithms:
    - **Message authentication code (MAC)**

        - A cryptographic checksum is generated from the message using a secret key
        - Knowledge of V(k) and knowledge of S(k) are equivalent: one can be derived from the other, so k must be kept secret
        - Because of the collision resistance in the hash function, we are reasonably assured that no other message could create the same MAC
    - **Digital signature algorithm (with public & private keys)**
        - The authenticators produced are called digital signatures
        - In a digital-signature algorithm, it is computationally infeasible to derive S(ks) from V(kv); in particular, V is a one-way function
            - kv is the public key and ks is the private key
            - RSA digital algorithm is one example
- The primary three reasons why we need encryption and authentication algorithms:
    - Authentication algorithms generally require fewer computations
    - The authenticator of the message is almost always shorter than the message and its cipher
    - Sometimes, we want authentication but not confidentiality
- Authentication is the core of nonrepudiation, which supplies proof that an entity performed an action
    - Ex: The filling out of an electronic form

Key Distribution

- Delivery of a symmetric key is a huge challenge
    - Done out-of-band: on paper or conversation
- Asymmetric keys can be exchanged in public and each the user needs only one private key
- To make sure that the public key is legit authentication takes place on the public key using a digital certificate
    - A **digital certificate** is a public key digitally signed by a trusted party
    - The trusted party receives proof of identification from some entity and certifies that the public key belongs to that party
    - The **certificate authorities** have their public keys included within the Web browsers before they are distributed, hence we know it's legit
- The digital certificates are distributed in the **standard X.509 digital certificate format**

Implementation of Cryptography

- Implementation of cryptography can happen at almost any one of the 7 seven OSI Model layers
- In general more protocols benefit from protections placed lower in the protocol stack (not definitive)
    - This might lead to lower protection in higher-layers of the OSI model
- IPSec (IP security) is used as the basis for VPNs
- p.646 - p.647

An Example: SSL

- Protocol that enables two computers to communicate securely
    - To limit the sender and receiver of messages to the other
- With SSL a client and server establishes a secure session key that can be used for symmetric encryption of the session between the two to avoid man-in-the-middle attacks

- Once the session is complete the session key is thrown away
- The SSL protocol is initiated by a **client c** to communicate securely with a **server s**
- The server obtain a certificate (cert) from certification authority **CA**
- The certificate contain the following:
    - Attributes of the server
    - Identity of a public encryption algorithm
    - Public key of this server
    - Validity interval of the certificate
    - Digital signature on the above information made by CA
- The client obtain the public verification algorithm prior to the protocol's use

User Authentication

- Protection depends on the ability to identify the programs and processes currently executing, which then must also be able to identify each user of the system
- How does the system determine if a user is authentic?
    - The user's possession of something (e.g. key or card)
    - The user's knowledge of something (e.g. identifier or password)
    - An attribute of the user (e.g. fingerprint or signature)

Passwords

- Used to protect access to a system together with a username
- Used to protect objects in the system (files etc.)
- Used to determine access rights

Password Vulnerabilities

- **Guessing** – through knowing the user / using brute force
    - Use good (longer) passwords to prevent guessing
- **Shoulder surfing** = looking over the shoulder of a user
    - Make sure people are not watching while entering password
- **Sniffing** = watching all data being transferred on the network
    - Encryption solves sniffing problem
- **Illegal transfer** – when users break account-sharing rules
- Passwords can be system-generated
- Some systems age passwords, forcing them to change at intervals

Encrypted Passwords

- Only encoded passwords are stored
- When a user presents a password, it is encoded and compared
- Thus, the password file doesn't need to be kept secret

One-Time Passwords

- **Different** password every time, so interceptors can't re-use it

- Paired passwords: the system presents one part and the user must supply the other part – the user is challenged
- System & user share a secret, which must not be exposed
- A seed is a random number / numeric sequence
    - The seed is the authentication challenge from the computer
- The secret and seed are input to a function: f(secret,seed)
- The result of the function is transmitted as the password
- Two-factor authentication: using a one-time password + PIN (personal identification number)
- Code book / one-time pad = a list of single-use passwords, which are crossed out after being used

Biometrics

- Biometrics used to secure physical access
    - ex. access to a data center
- Palm / finger-print readers can scan you
- **Multifactor authentication** is the use of various authentication methods to authenticate a user (password, fingerprint, etc...)

Implementing Security Defenses

- Includes improved user education, technology and writing bug-free software (not all)
- **Defense in depth** is a theory where people believe it is better to have more layers of defense than fewer layers

Security Policy

- First step to security is to have a security policy
- Statement of what is secured
- Without policy, users and admins wouldn't know what is permissible, required and what is not allowed
- Security policy document is a living document that is reviewed and updated periodically

Vulnerability Assessment

- To determine if security policy is correctly implemented a **vulnerability assessment** is needed
- This can consist out of broad range of tests, some of which are:
    - Social engineering
    - Risk assessment
    - Port scans
- Risk Assessments:
    - Look at the odds a security incident will affect the entity and decrease its value
- **Core activity on vulnerability assessments is penetration test**:
    - Scan the entity for known vulnerabilities
    - Done when computer use is relatively low
    - Do on test systems
    - Scans can check variety of aspects:
        - Short or easy-to-guess passwords

- Unauthorized privileged programs (setuid)
- Unauthorized programs in system directories
- Unexpectedly long-running processes
- Improper directory protections on user and system directories
- Improper protections on system data files (password file, device drivers, OS kernel)
- Dangerous entries in program search path (section 15.2.1)
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

- A system is only as secure as its most far-reaching connection
    - If a system has a connection outside a building the system is not secure
- Vulnerability scans are used on networks to find problems with network security
    - Scans search the network for ports that responds to a request
    - Access to unsafe open ports can be blocked
    - Scans determine details of applications on ports by listening and try to find vulnerabilities for the application
    - Maybe system needs patches or is misconfigured
- Tools that are used to test security can be dangerous in the hands of a bad person
- **Security through obscurity**:
    - People advocate that no tools should be developed to test security, since it is used to find security holes

Intrusion Detection

- Encompasses many techniques that vary on a number of axes:
    - The time that detection occurs
    - The types of inputs examined to detect intrusive activity
    - The range of response capabilities
- Intrusion-Detection Systems (**IDSs**)
    - Raises an alarm when intrusion is detected
- Intrusion-Prevention Systems(**IDPs**)
    - Acts as routers, passing traffic unless an intrusion is detected
- What constitutes an intrusion?
    - **Signature-based detection**
        - System input is examined for specific behavior patterns
        - E.g. monitoring for multiple failed attempts to log on
        - 'Characterizes dangerous behavior and detects it'
    - **Anomaly detection**
        - E.g. monitoring system calls of a daemon process to detect if its system-call behavior deviates from normal patterns
        - 'Characterizes non-dangerous behavior and detects the opposite'
        - **Tripwire (Example of Anomaly-Detection Tool)**

- Operates on the premise that a large class of intrusions result in anomalous **modification** of system directories & files
- Tripwire is a tool to monitor file systems for added, deleted, or changed files, and to alert admin to these modifications
- **Limitations**:
  - The need to protect the Tripwire program
  - Some security-relevant files are *supposed to* change in time

Virus Protection

Auditing, Accounting, and Logging

- Audit-trail processing: security-relevant events are logged to an audit trail and then matched against attack signatures or analyzed for anomalous behavior

Firewalling to Protect Systems and Networks

- A firewall limits network access between two security domains
- It monitors and logs connections and can also limit connections
- A firewall can separate a network into multiple domains
- A Firewall don't prevent attacks that tunnel, or travel within protocols / connections that the firewall allows
- Another firewall vulnerability spoofing, where an unauthorized host pretends to be an authorized one by meeting some criteria

Computer-Security Classification

- Trusted Computer Base (TCB) = the sum total of all protection systems within a computer system that enforce a security policy

An Example: Windows XP

Summary

Distributed Systems

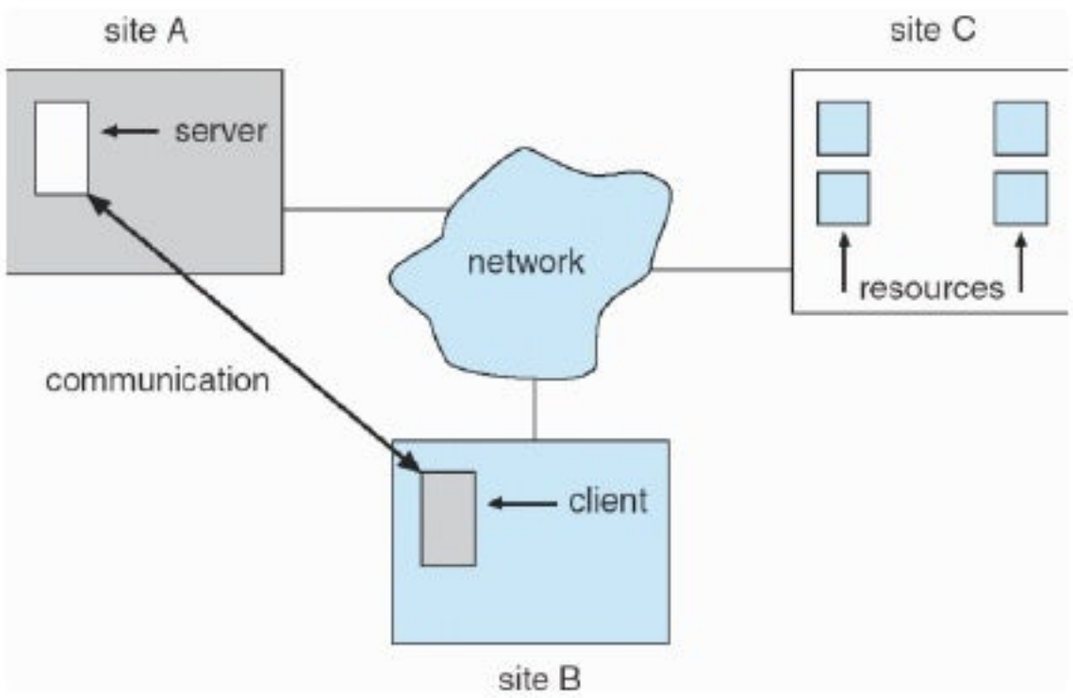
PART SEVEN: DISTRIBUTED SYSTEMS

Chapter 16: Distributed Operating Systems

- A distributed system is a collection of processors that do not share memory or a clock
- Instead, each processor has its own local memory
- The processors communicate with one another through various communication networks, such as high-speed buses or telephone lines
- Here we discuss the general structure of distributed systems and the networks that interconnect them
- We contrast the main differences in operating-system design between these systems and centralized systems
- **Chapter Objectives**:
  - To provide a high-level overview of distributed systems and the networks that interconnect them
  - To discuss the general structure of distributed operating systems

Motivation

- Distributed system is a collection of loosely coupled processors interconnected by a communication network
- Each processor has its own local resources
- The processors communicate through networks
- General structure of distributed system



- Four reasons for building Distributed Systems:
  - resource sharing
  - computation speedup
  - reliability
  - communication

Resource Sharing

- A user at one site can use the resources at another site
- E.g. sharing files, processing info, printing files…

Computation Speedup

- If a computation can be partitioned into sub-computations that can run concurrently, then a distributed system allows us to distribute the sub-computations among various sites
- If a particular site is overloaded with jobs, some may be moved to other lightly loaded sites (= load sharing)

Reliability

- If one site fails, the remaining ones can still operate
- If each machine is responsible for some crucial system function, then a single failure may halt the whole system
- With enough redundancy, the system can continue operation
- The failure of a site must be detected by the system
- The system must no longer use the services of that site
- If the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly
- When the failed site recovers / is repaired, mechanisms must be available to integrate it back into the system

Communication

- Messages passed between systems in the same way a single-computer message system does (section 3.4)
- Functions include file transfer, login, mail, and remote procedure calls (RPCs)
- These functions can be carried out over distances
- Users at different sites can exchange information
- Advantages:
    - Collaboration over distances
    - Downsizing

Types of Network-based Operating Systems

- Two general categories of network-oriented operating systems:
    - Network Operating Systems
    - Distributed Operating Systems

Network Operating Systems

- Simpler to implement, but more difficult for users to access and utilize than distributed OSs
- Provides an environment where users who are aware of multiplicity of machines can access remote resources on each others machines

Remote Login

- The Internet provides the telnet facility for this purpose
    - ex: telnet cs.yale.edu
    - Creates socket connection between the local machine and the cs.yale.edu computer
    - Open bidirectional connection
    - User must enter username and password
    - User can execute any command on remote computer as any local user can

Remote File Transfer

- Each computer maintains its own local file system
- User need access, else use anonymous and arbitrary password
    - Anonymous users can only access files in the directory with public access
    - Care must be taken that anonymous users cannot access files outside this directory
- The Internet provides the FTP program
    - ex. connect with: ftp cs.yal.edu
    - copy the file with: get Server.java
- This does not provide real file sharing
- User must know where the files are in the subdirectories
- Various copies of the same file can exist and they can be inconsistent
- Only predefined set of file-related commands can be used:
    - get - from remote to local machine
    - put - from local to remote machine
    - ls or dir - list files in current directory of remote machine

        - cd - change current directory of remote machine
- A windows user logging into a Unix machine using telnet should change paradigms
    - User must use Unix commands
- Distributed operating systems address this issue

Distributed Operating Systems

- Provide more features than network OSs
- Users access remote resources in the same manner as local ones
- Data and process migration is under control of distributed operating system

Data Migration

- If a user need to work on a remote file:
    - Transfer the entire file to the other site and back after modifying the file
        - Very inefficient if only small part of large file is being processed
        - Efficient if large portions of file needs to be processed
    - Transfer only the necessary portions of the file needed for immediate task
        - If other portion needed, can be transferred
        - After modification only the modified portion is transferred back to remote site

Computation Migration

- Invoke a procedure at another site and get the result
- Access to file carried out remotely and is initiated by RPC
    - RPC uses datagram protocol (UDP on Internet) to execute routine on remote site (section 3.6.2)
    - Can also send message to remote site
        - Remote site opens up new process, executes task, sent result back and close
        - Can be executed concurrently/bidirectional between sites

Process Migration

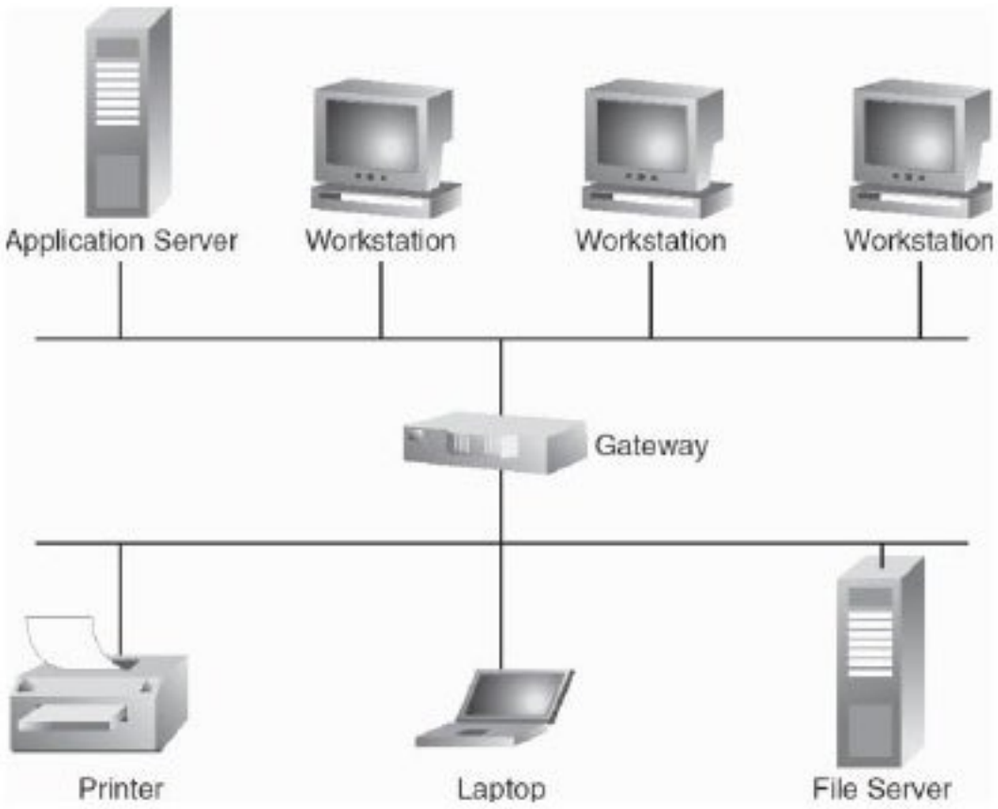
- Logical extension of computation migration
- Why (parts of) a process may be executed at different sites
    - Load balancing - even workload
    - Computation speedup - reduce total process turnaround time
    - Hardware preference - specialized processor/hardware needs
    - Software preference - software only available at specific site and not cost effective to move
    - Data access - if to much data must be moved, remote processes can be more efficient
- Two techniques to move processes in a network
    - The system can hide the fact that the process has migrated from the client
        - No user programming needed to accomplish process migration
    - The user must specify explicitly how the process should migrate
        - Done to satisfy a hardware or software preference

Network Structure

- Two type of networks:
    - Local Area Networks (LAN)
    - Wide Area Networks (WAN)
- Main difference is in the way they are geographically distributed
    - Local Area Networks (LAN)
        - Distributed over small areas (inside single or adjacent buildings)
    - Wide Area Networks (WAN)
        - Distributed over large areas (in a country)
- Differences in speed and reliability of communications networks

Local-Area Networks

- LANs emerged as a substitute for large mainframe computers
    - A number of small computers are used to replace the mainframe computers
    - Small computers have self-contained applications
    - Can be used for data sharing in the enterprise
- Designed to cover small geographical area
- Multiaccess bus, ring, or star network
- Because computers closer together, higher speed and lower error rate than computers in WAN
- Make use of twisted-pair and fiber-optic cabling
- Speeds of 10Mbps to 100Mbps can be obtained with the Ethernet protocol used for LANs
- Normally consists out of computers, shared peripheral devices and one or more gateways

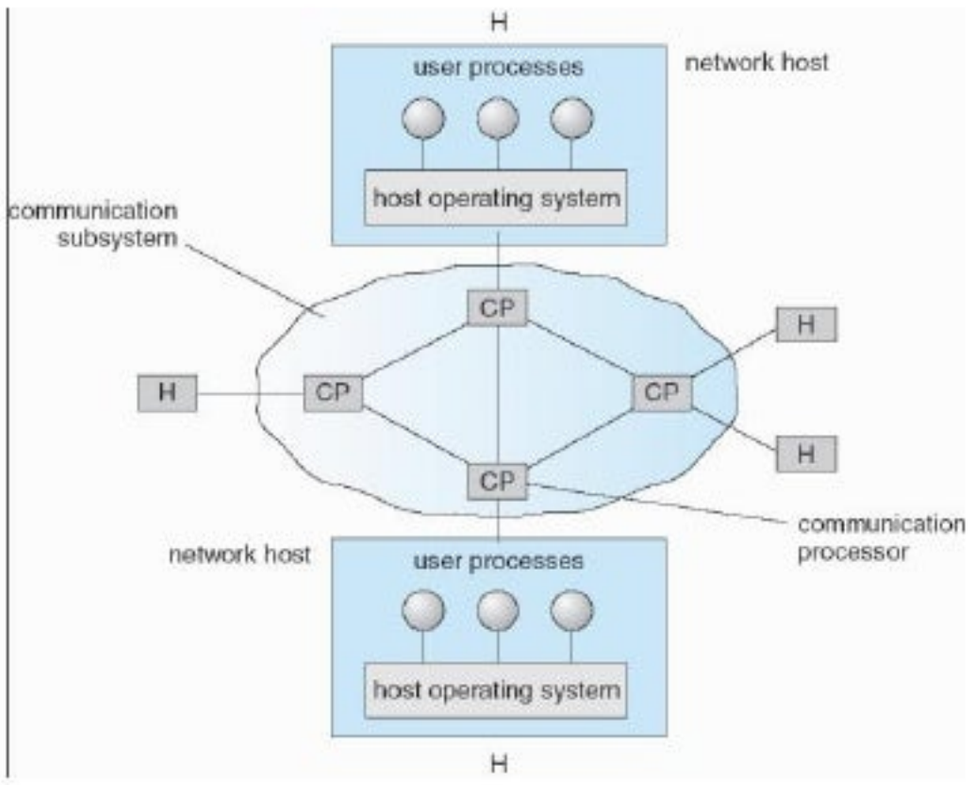


- Wifi is the wireless alternative to Ethernet, but is slower than Ethernet
    - No cables needed to connect hosts to network
- The distance between the wireless router and the host influences the speed of the network

Wide-Area Networks

- WANs emerged mainly as an academic research project
- **Communication processors** control communication links

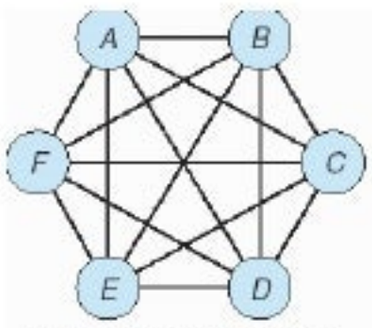
- Responsible for defining the interface through which the sites communicate over the network, as well as for transferring information among the various sites



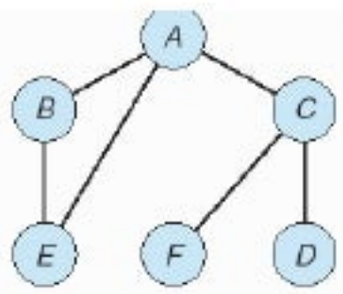
- Links geographically separated sites
- Point-to-point connections over long-haul lines (often leased from a phone company)
- Speed ≈1.544 to 45Mbps
- Broadcast usually requires multiple messages
- Nodes:
    - usually a high percentage of mainframes

Network Topology

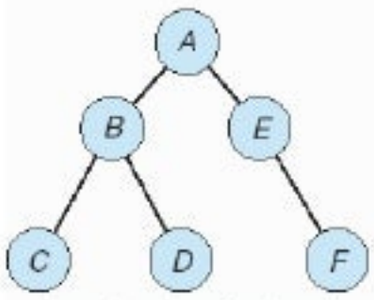
- Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:
    - **Installation cost**
        - How expensive is it to link the various sites in the system?
    - **Communication cost**
        - How long does it take to send a message from site *A* to site *B*?
    - **Availability/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:



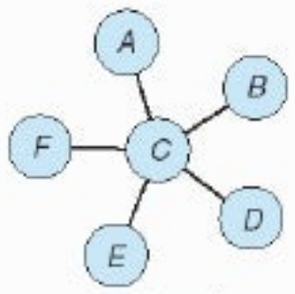
fully connected network



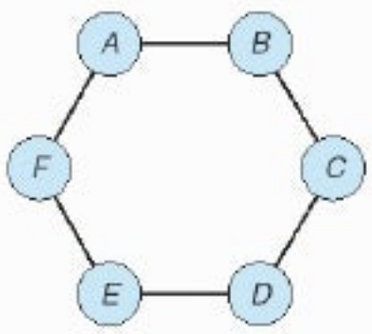
partially connected network



tree-structured network



star network



ring network

- The number of links grows as the square of the number of sites, resulting in a huge installation cost
- Criteria for comparing the different configurations:
    - **Installation cost**
        - Low for tree structured networks
        - High for fully connected networks
    - **Communication cost**
        - Low for tree-structured networks
        - Low for star networks
        - High for ring networks
    - **Availability**
        - High for ring networks
        - Lower for tree-structured networks

Communication Structure

- The design of a *communication* network must address five basic *issues:*
    - **Naming and name resolution**
        - How do two processes locate each other to communicate?
    - **Routing strategies**
        - How are messages sent through the network?
    - **Packet strategies**
        - Are packets sent individually or as a sequence?
    - **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

- Each process has an identifier
- Identify processes on remote systems by <host-name, identifier> pair
- The computer's host-name must be **resolved** into a **host-id**
- *Domain name service* (DNS) –specifies the naming structure of the hosts, as well as name-to-address resolution (Internet)
- The OS is responsible for accepting from its process a message destined for **<host-name, identifier>** and for transferring that message to the appropriate host
- The Kernel on the destination host is then responsible for transferring the message to the process named by the identifier
- Check p.686 for DNS protocol

Routing Strategies

- Each site has a routing table, indicating alternative paths
- Fixed routing
  - A path from A to B is specified in advance
  - Disadvantage: Can't adapt to link failures & load changes
- Virtual routing
  - A path from A to B is fixed for the duration of one session
- Dynamic routing
  - The path is chosen only when a message is sent
  - Messages may arrive out of order, so add sequence numbers
- Routers examine the destination Internet address and examine tables to determine the location of the destination host
- With static routing, this table is changed by manual update
- With dynamic routing, a routing protocol is used between routers so that they can update their routing tables automatically

Packet Strategies

- Communication is commonly implemented with fixed-length messages called packets, frames, or datagrams

Connection Strategies

- Once messages are able to reach their destinations, processes can institute **communications sessions** to exchange information
  - Pairs of processes that want to communicate over the network can be connected in a number of ways
  - The three most common schemes are **circuit switching**, **message switching**, and **packet switching**
- **Circuit switching**
  - If two processes want to communicate, a permanent physical link is established between them

        - This link is allocated for the duration of the communication session, and no other process can use that link during this period (even if the two processes are not actively communicating for a while)
        - This scheme is similar to that used in the telephone system
        - Once a communication line has been opened between two parties (that is, party A calls party B), no one else can use this circuit until the communication is terminated explicitly (for example, when the parties hang up)
- **Message switching**
    - If two processes want to communicate, a temporary link is established for the duration of one message transfer
        - Physical links are allocated dynamically among correspondents as needed and are allocated for only short periods
        - Each message is a block of data with system information - such as the source, the destination, and error-correction codes (ECC) - that allows the communication network to deliver the message to the destination correctly
        - This scheme is similar to the post-office mailing system
            - Each letter is a message that contains both the destination address and source (return) address
        - Many messages (from different users) can be shipped over the same link
- **Packet switching**
    - One logical message may have to be divided into a number of packets
        - Each packet may be sent to its destination separately, and each therefore must include a source and a destination address with its data
        - Furthermore, the various packets may take different paths through the network
        - The packets must be reassembled into messages as they arrive
        - Note that it is not harmful for data to be broken into packets, possibly routed separately, and reassembled at the destination
        - Breaking up an audio signal (say, a telephone communication), in contrast, could cause great confusion if it was not done carefully
- There are obvious **tradeoffs** among these schemes:
    - Circuit switching requires substantial setup time and may waste network bandwidth, but it incurs less overhead for shipping each message
    - Conversely, message and packet switching require less set-up time but incur more overhead per message
    - Also, in packet switching, each message must be divided into packets and later reassembled
    - Packet switching is the method most commonly used on data networks because it makes best use of network bandwidth

Contention

- Depending on the network topology, a link may connect more than two sites in the computer network, and several of these sites may want to transmit information over a link simultaneously
    - This situation occurs mainly in a ring or multi-access bus network
    - In this case, the transmitted information may become scrambled
    - If it does, it must be discarded

        - The sites must be notified about there problem so that they can retransmit the information
        - If no special provisions are made, this situation may be repeated, resulting in degraded performance
- Several techniques have been developed to avoid repeated collisions, including **collision detection** and **token passing**
    - **CSMA/CD (carrier sense with multiple access)**:
        - Before transmitting a message over a link, a site must listen to determine whether another message is currently being transmitted over that link
            - If the link is free, the site can start transmitting
            - Otherwise, it must wait (and continue to listen) until the link is free
        - If two or more sites begin transmitting exactly the same time (each thinking that no other site is using the link), then they will register a **collision detection (CD)** and will stop transmitting
            - Each site will try again after some random time interval
        - The main problem with this approach is that, when the system is very busy, many collisions may occur, and thus performance may be degraded
        - Nevertheless, CSMA/CD has been used successfully in the Ethernet system, the most common local area network system
        - One strategy for limiting the number of collisions is to limit the number of hosts per Ethernet network
            - Adding more hosts to a congested network could result in poor network throughput
        - As systems get faster, they are able to send more packets per time segment
            - As a result, the number of systems per Ethernet network generally is decreasing so that networking performance is kept reasonable
    - **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
            - It then removes the token from the ring and begins to transmit its message
            - When the site completes its round of message passing, it retransmits the token
            - This action, in turn, allows another site to receive and remove the token and to start its message transmission
            - If the token gets lost, the system must detect the loss and generate a new token
                - It usually does that by declaring an **election** to choose a unique site where a new token will be generated
        - A token-passing scheme has been adopted by the IBM and HP / Apollo systems
        - The **benefit** of a token-passing network is that performance is constant
        - Adding new sites to a network may lengthen the waiting time for a token, but it will not cause a large performance decrease, as may happen on Ethernet
        - On lightly loaded networks, however, Ethernet is more efficient, because systems can send messages at any time

Communication Protocols

OSI model

- Physical layer
    - Handles mechanical & electrical details of transmission
- Data-link layer
    - Responsible for handling the frames
- Network layer
    - Responsible for providing connections & routing packets
- Transport layer
    - Responsible for low-level access to the network
- Session layer
    - Responsible for implementing sessions
- Presentation layer
    - Responsible for resolving the differences in formats
- Application layer
    - Responsible for interacting directly with the users

Robustness

Failure Detection

- To detect **link** and **site** failure, use a handshaking procedure:
- At fixed intervals both sites exchange 'I-am-up' messages
- If site A doesn't receive this message, it can assume
    - That site B has failed, or
    - That the link between A & B has failed, or
    - That the message from B has been lost
- At this point, site A can
    - Wait for another 'I-am-up' message from B, or
    - Send an 'Are-you-up?' message to B
- Site A can differentiate between link and site failure by sending an 'Are-you-up?' message by **another** route
- If B then replies, you know B is up and that the failure is in the direct link between A and B

Reconfiguration

- If a direct **link** from A to B has failed,
    - This info must be broadcast to every site in the system so that the routing tables can be updated accordingly
- If the system believes that a **site** has failed,
    - Every site in the system must be notified, so they will no longer attempt to use the services of the failed site

Recovery from Failure

- If a **link** between A and B has failed,
    - When it is repaired, both A and B must be notified
- If **site** B has failed,
    - When it recovers, it must notify all other sites

Fault Tolerance

Design Issues

An Example: Networking

Summary

Special-Purpose Systems

Real-Time Embedded Systems

- The most prevalent form of computers
- They run embedded real-time OS's that provide limited features

PART EIGHT: SPECIAL PURPOSE SYSTEMS

Chapter 19: Real-Time Systems

Overview

- Rigid time requirements on the operation of a processor
- Sensors bring data to the computer, which must analyze the data and possibly adjust controls to modify the sensor inputs
- E.g. home appliance controllers, weapon & fuel-injection systems
- Processing *must* be done within the time constraints, or it fails
- Hard real-time systems
    - Guarantee that critical tasks are completed on time
    - Data is stored in memory / ROM instead of secondary storage
    - No virtual memory, so no time-sharing
    - No general-purpose OS supports hard real-time functionality
- Soft real-time systems
    - A critical real-time task gets priority over other tasks
    - More limited utility than hard real-time systems
    - Risky to use for industrial control and robotics
    - Useful for multimedia, virtual reality, scientific projects
    - Finding its way into most current OS's, including UNIX

Multimedia Systems

Handheld Systems

- Limited memory, so the OS must manage memory efficiently

- Slow processors, so the OS must not tax the processor
- Small display screens, so web clipping is used for displaying

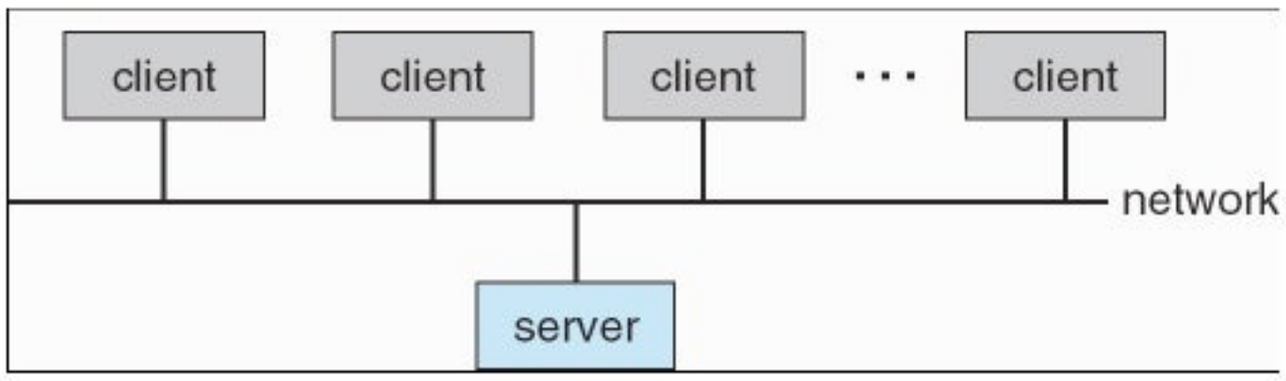
Computing Environments

Traditional Computing

- Companies have portals to provide web access to internal servers
- Network computers are terminals that understand web computing
- Handheld PCs can connect to wireless networks
- Some homes have firewall to protect them from security breaches
- Traditional computer
    - Blurring over time
    - Office environment
        - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
        - Now portals allowing networked and remote systems access to same resources
    - Home networks
        - Used to be single system, then modems
        - Now firewalled, networked

Client-Server Computing

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



Peer-to-Peer Computing

- OS's include system software that enables computers to access the Internet, and several include the web browser itself
- The processors communicate through communication lines
- Network OS = one that provides file sharing across the network
- Another model of distributed system
- P2P does not distinguish clients and servers
    - Instead all nodes are considered peers
    - May each act as client, server or both
    - Node must join P2P network
        - Registers its service with central lookup service on network, or

        - Broadcast request for service and respond to requests for service via *discovery protocol*
    - Examples include *Napster* and *Gnutella*

Web-Based Computing

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

Chapter 23: Influential Operating Systems

Feature Migration

- Features from mainframes have been adopted by microcomputers

Early Systems

- Early systems were
    - first interactive systems (punch cards!)
    - then became batch systems
    - then interactive again
- Early systems were
    - single-user systems (just one user could work at a time)
    - then became multi-user systems
    - and then single-user systems (with the advent of PCs)
    - and are now aimed at a multi-user environment
    - Dedicated Computer Systems

Shared Computer Systems

Overlapped I/O

Open-Source Operating Systems

History

Linux

BSD UNIX

Solaris

Utility

Summary