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.

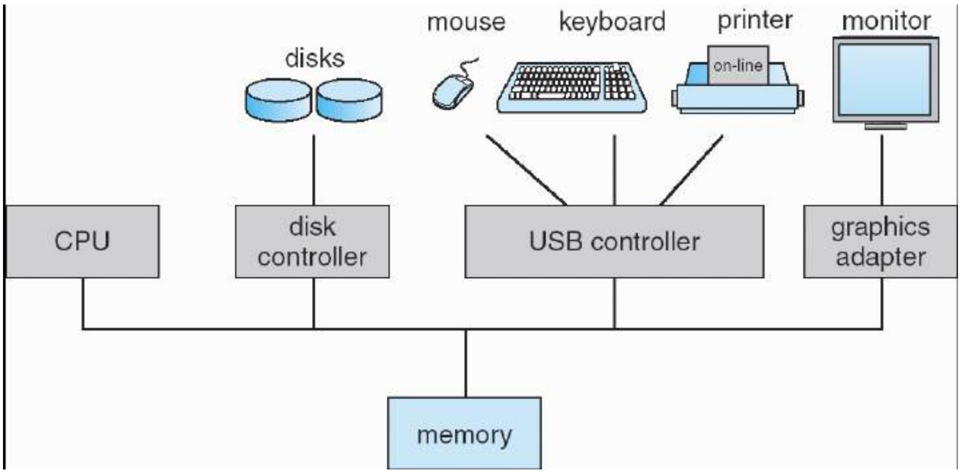
Defining Operating Systems

- 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

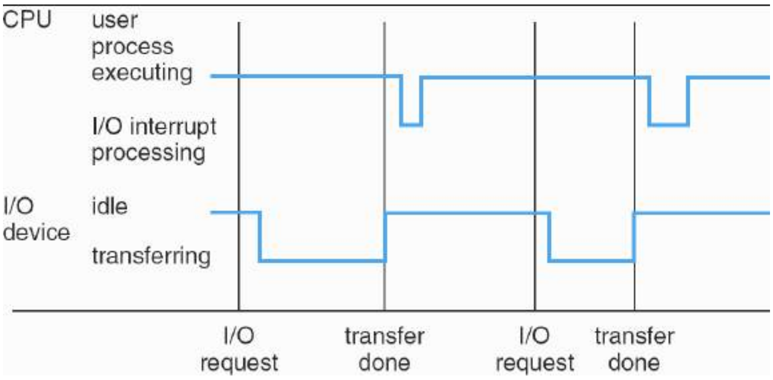
- Computer-system operation

- 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.

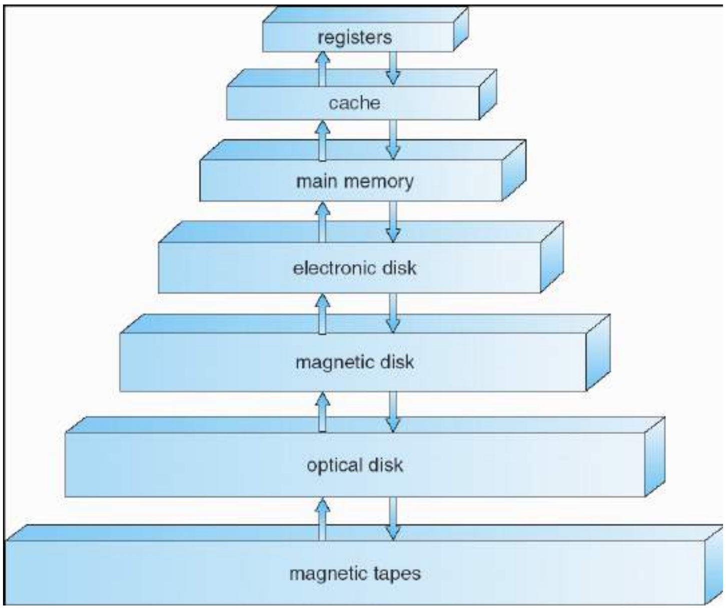


  - Since only a predefined number of interrupts are possible, a table of pointers to interrupt routines is used to increase speed.
  - The table of interrupt pointers is stored in low memory.
  - These locations keep the addresses of the interrupt service routines for the various devices.
  - This array 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)

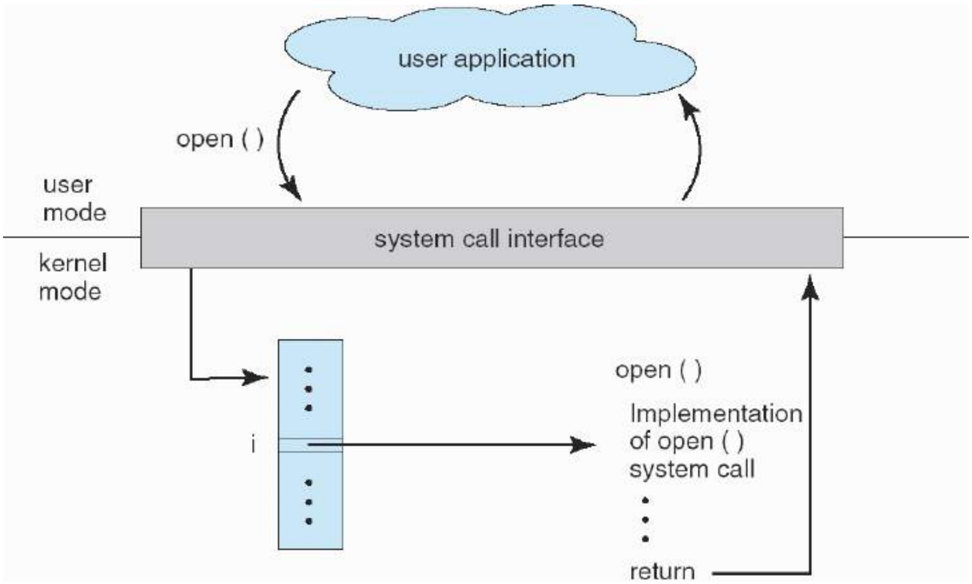
- No entering of commands but the use of a mouse-based window-and-menu system characterized by a **desktop** metaphor.
- The mouse is used to move a pointer to the position of an icon that represents a file, program or folder and by clicking on it the program is invoked.

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.
- <u>**System calls**</u>:
    - 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

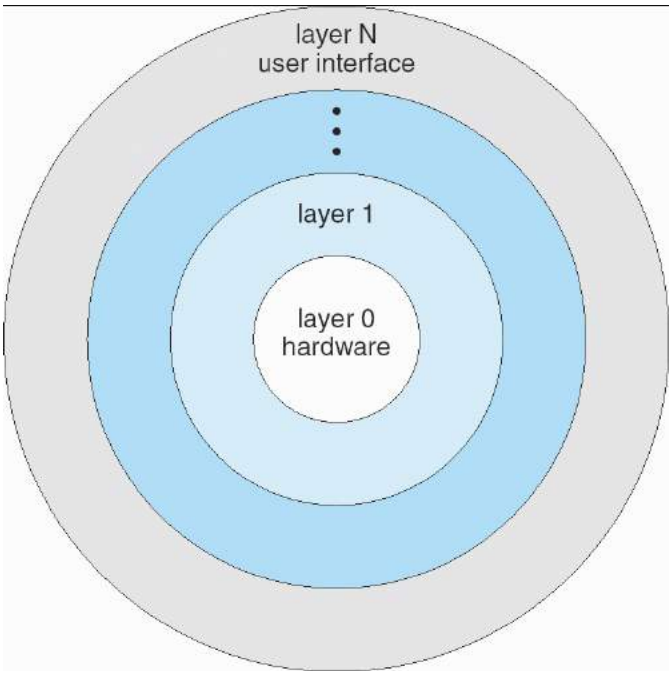
Operating-System Structure

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

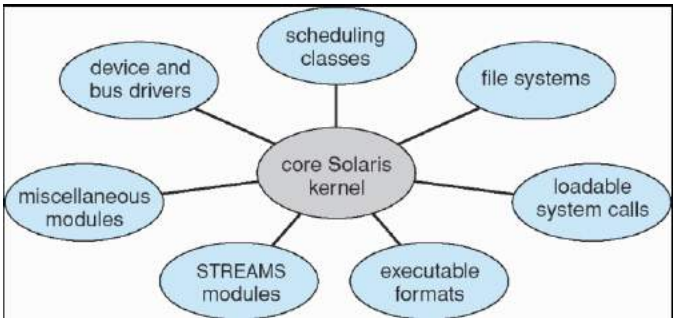


Microkernels

- 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

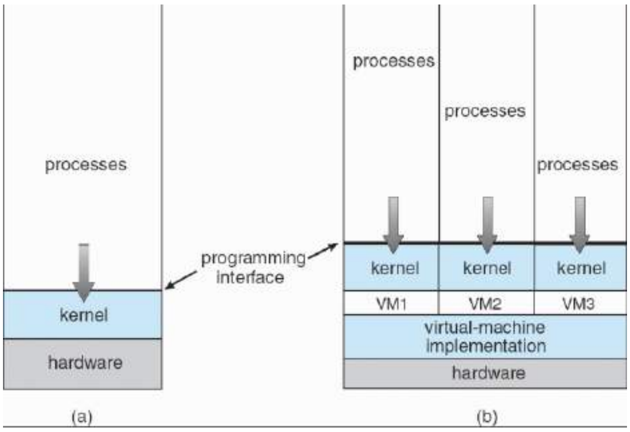
Modules

- 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

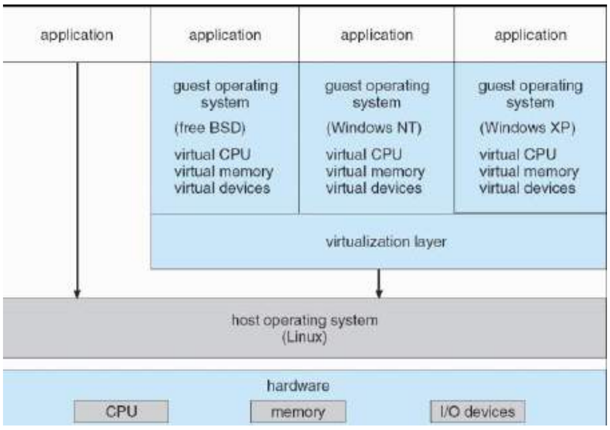
Para-virtualization

Implementation

Examples

VMware

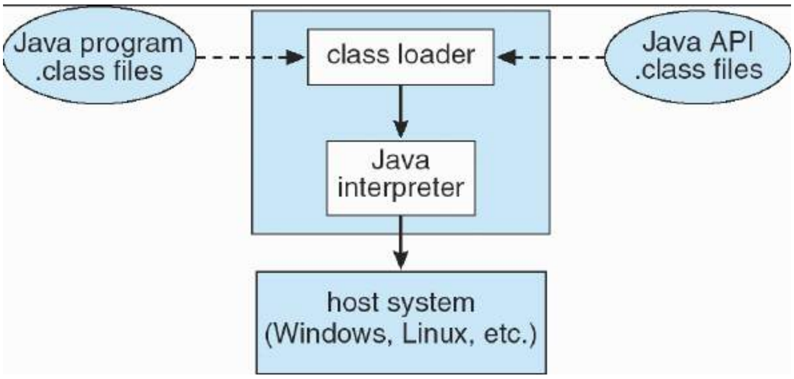
- VMware Architecture



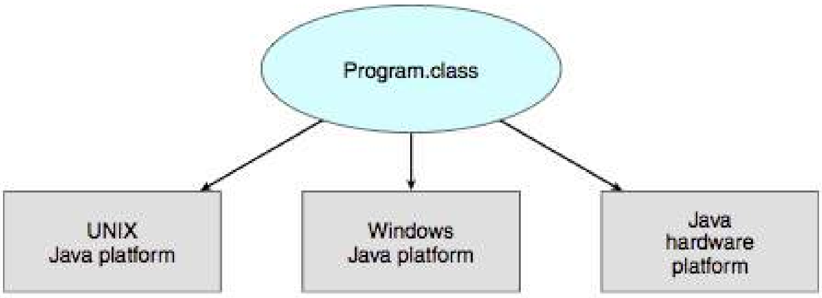
The Java Virtual Machine

Java consists of:

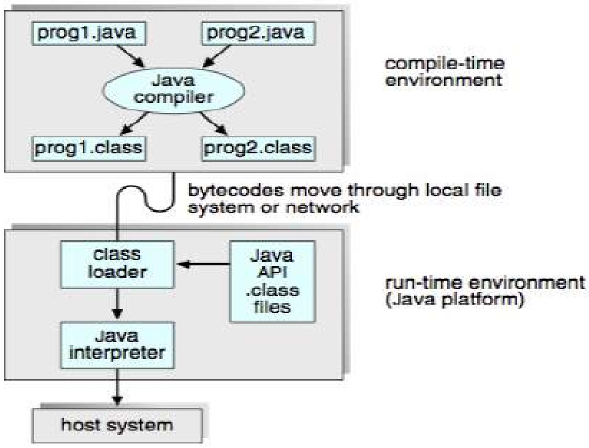
- Programming language specification
- Application programming interface (API)
- Virtual machine specification



- 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

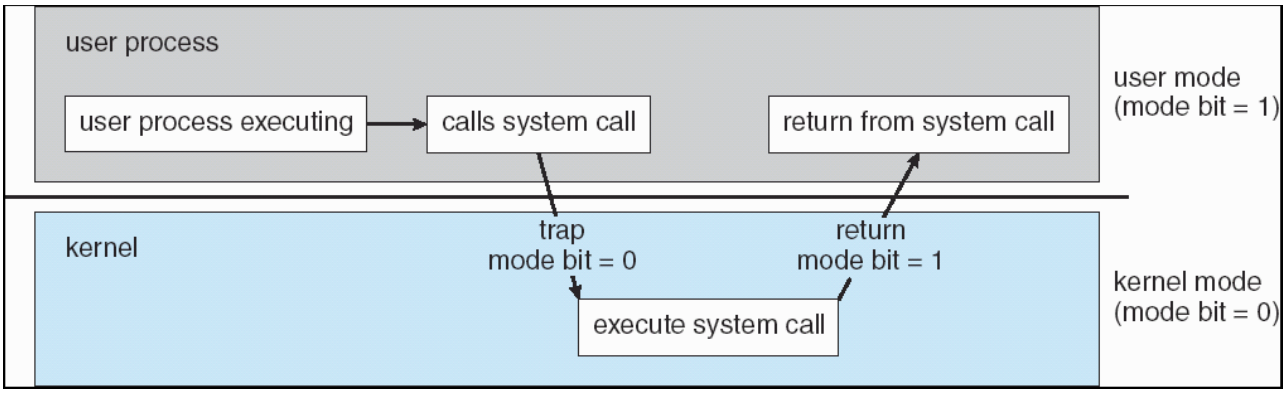
Summary

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

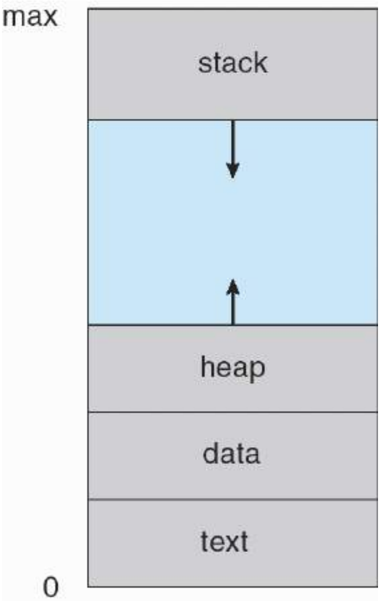
- **Objectives**:
    - To introduce the notion of a process - a program in execution, which forms the basis of all computation.
    - To describe the various features of processes, including scheduling, creation and termination, and communication.
    - To describe communication in client-server systems.

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

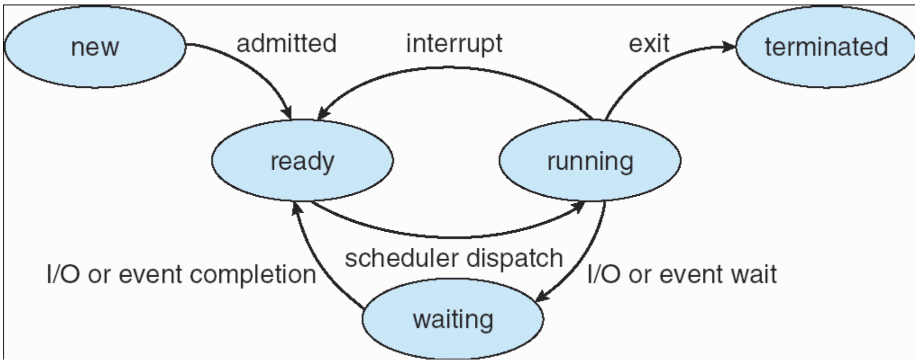
- Process = an active entity, with a program counter (to indicate the current activity), process stack (with temporary data), and a data section (with global variables)
- Text section = the program code
- If you run many copies of a program, each is a separate process (The text sections are equivalent, 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



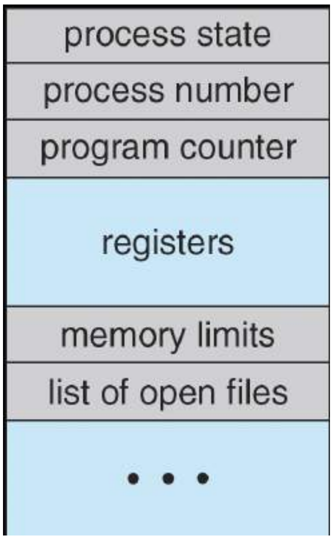
- p.102 give description of stack, heap, data and text areas

Process State

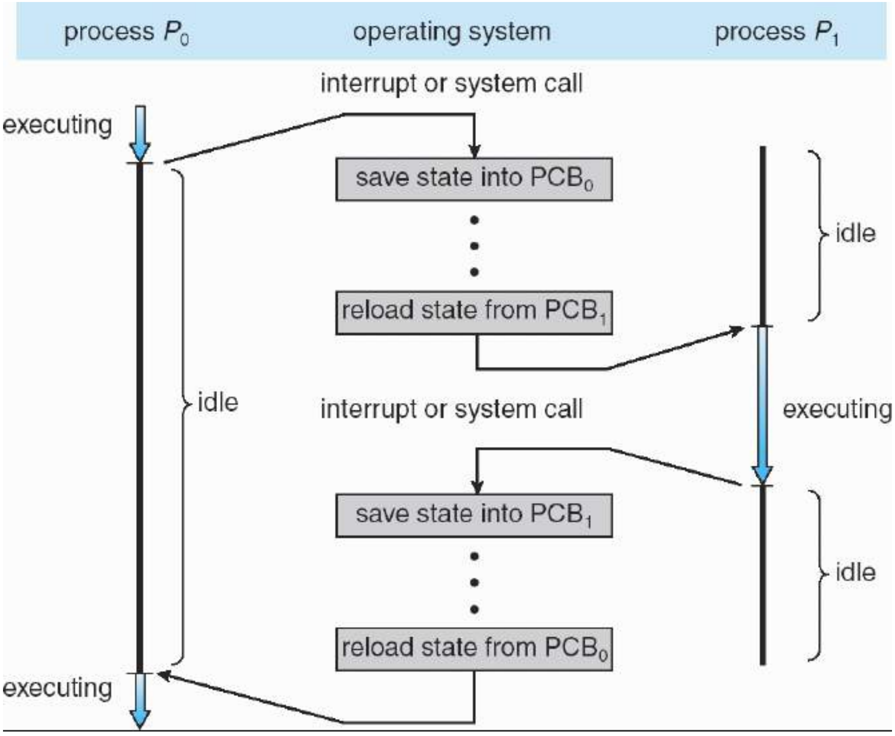
- Each process may be in one of the following states:
    - New (Process is being created)
    - Running (Instructions are being executed)
    - Waiting (Process is waiting for an event, e.g. I/O)
    - Ready (Process is waiting to be assigned to a processor)
    - Terminated (Process has finished execution)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

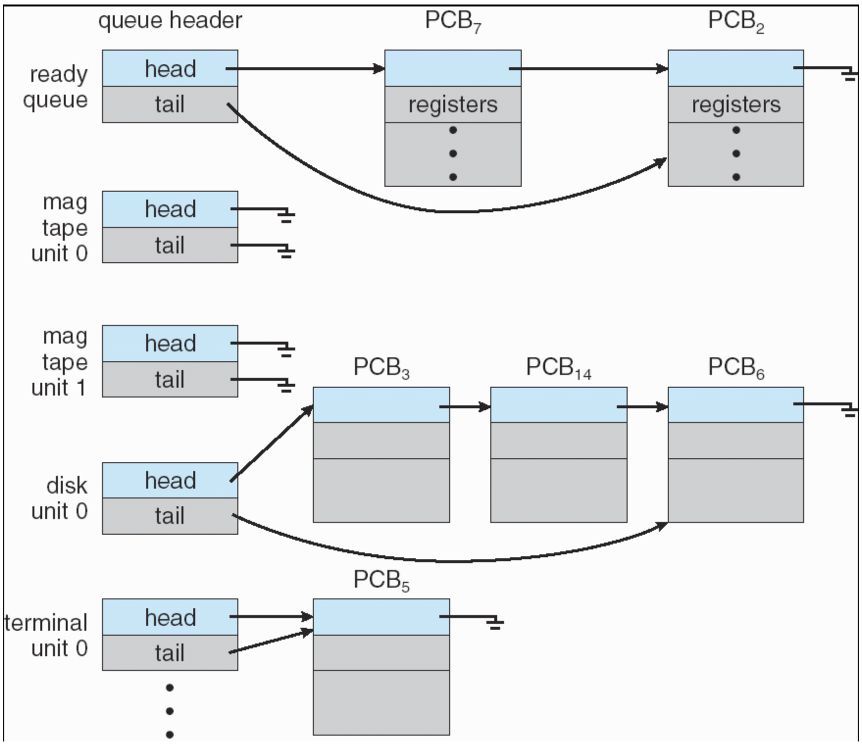
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization
    - The objective of time sharing is to switch the CPU among processes so frequently that the user can interact with each program while it is running
    - To meet this objectives, the process 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

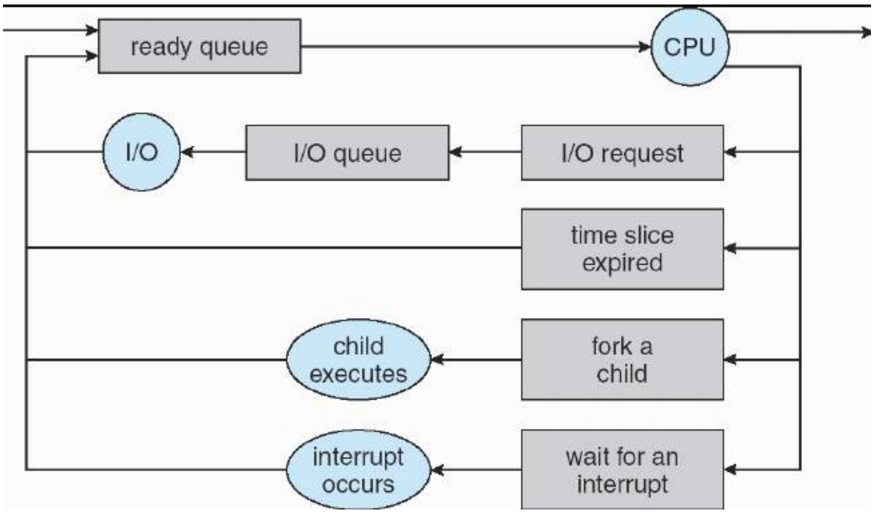
- As processes enter the system, they are put into a **job queue**
- Processes in memory, waiting to execute, are in the **ready queue**
- A ready queue header contains pointers to the fist & last PCBs in the list, each of which has a pointer to the next PCB
- **Device queue** = the list of processes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

- it could create a <u>sub-process</u> and wait for its termination
- it could be <u>interrupt</u>ed 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

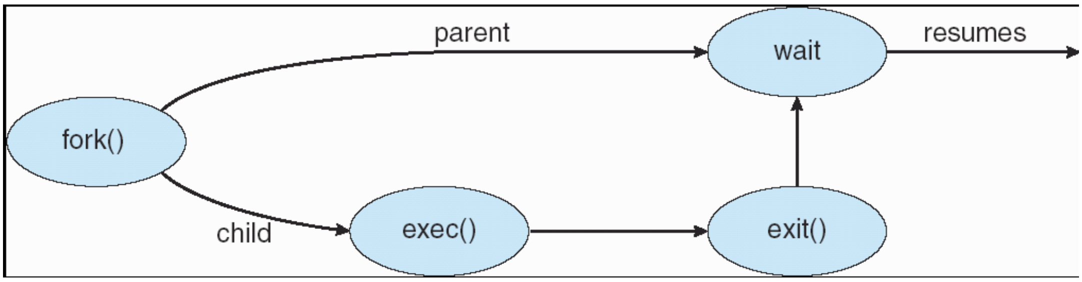
- Context switch = saving the state of the old process and switching the CPU to another process
- The context of a process is represented in the PCB of a process
- (It includes the value of the CPU registers, process state, and memory-management information)
- Context-switch time is pure overhead, because 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();
 }
}
```

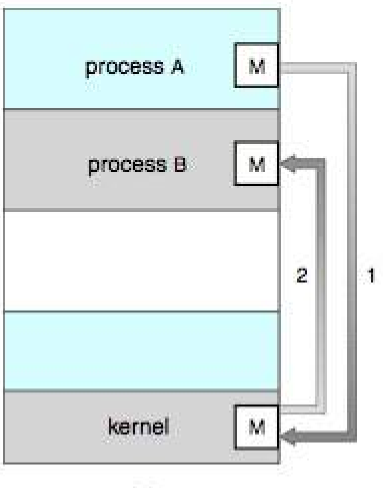
Process Termination

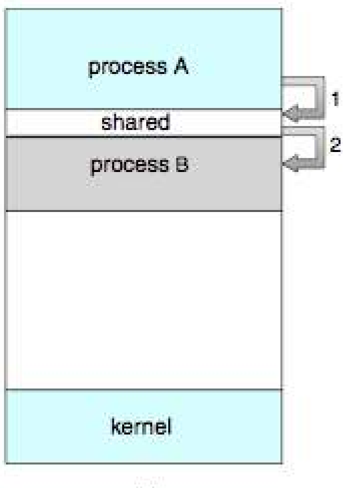
- A process terminates after it executes its final statement
- At that point the process may return data to its parent process
- All the process' resources (memory, open files, I/O buffers) are de-allocated by the OS
- A parent process can cause its child processes to terminate
- Parents therefore need 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)

<table>
<tr><td>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</td><td>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</td></tr>
<tr><td>Symmetric addressing:
Both sender and receiver processes must name the other to communicate</td><td>Mailbox owned by a process:
The owner can only receive, and the user can only send
The mailbox disappears when its owner process terminates</td></tr>
<tr><td>Asymmetric addressing:
Only the sender names the recipient; the recipient needn't name the sender</td><td>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</td></tr>
</table>

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

<table>
<tr><th>Synchronous message passing (blocking)</th><th>Asynchronous passing (non-blocking)</th></tr>
<tr><td>Blocking send:
The sending process is blocked until the message is received by the receiving process or by the mailbox.</td><td>Non-blocking send:
The sending process sends the message and resumes operation.</td></tr>
<tr><td>Blocking receive:</td><td>Non-blocking receive:</td></tr>
</table>

• 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

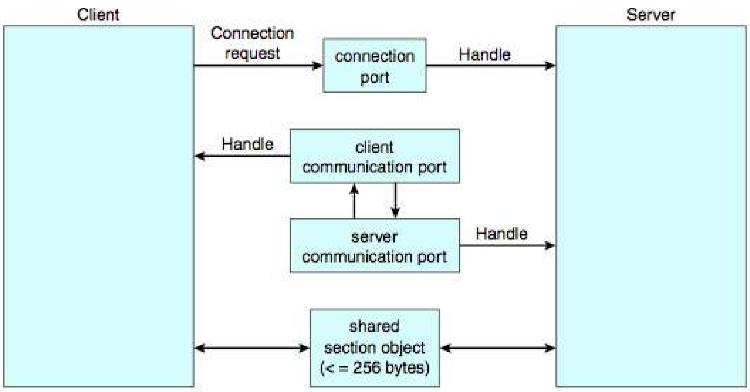
- p.123 - 124

An Example: Mach

- p.124 - 126

An Example: Windows XP

- p.127 - 128

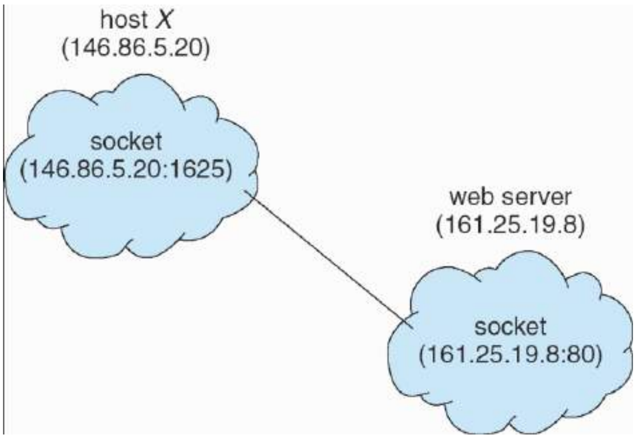


Communication in Client-Server Systems

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

Sockets

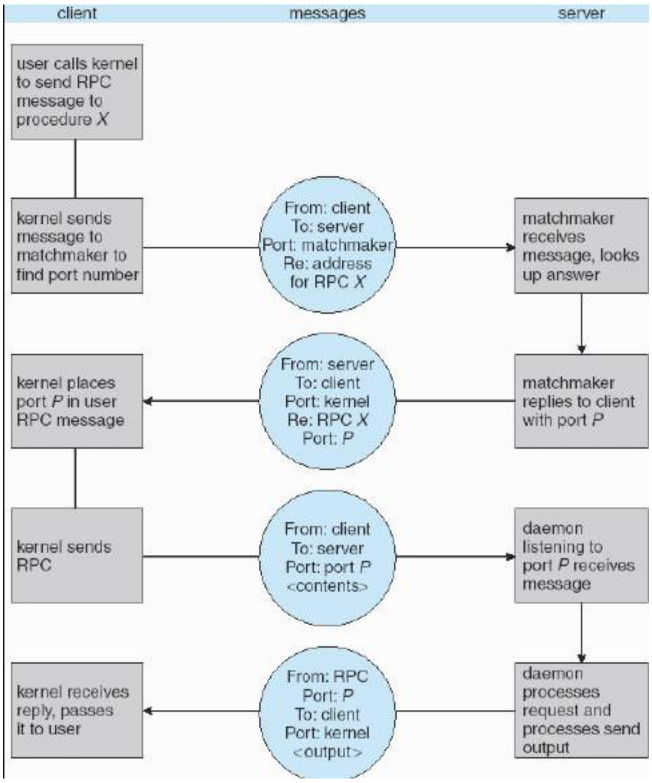
- **Socket** = an endpoint for communication
- A pair of processes communicating over a network employs a pair of sockets - one for each process
- A socket is identified by an IP address together with a port no
    - The socket **161.25.19.8:1625**refers to port **1625**on host **161.25.19.8**
- In general, 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

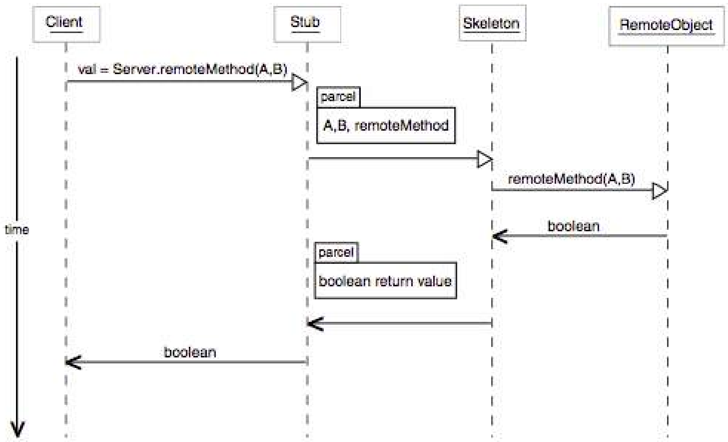


Remote Procedure Calls

- 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.