Chapter 1 – Introduction

What are Operating Systems?

A program that manages the computer hardware. Therefore, it acts as an intermediary between a user of a computer and the computer hardware.

Why we need an Operating system?

Generally an operating system is needed for the following reasons:

- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.

Computer systems

Computer systems can be divided into four components

 Hardware –provides basic computing resources CPU, memory, I/O devices

 Operating system-Controls and coordinates use of hardware among various applications and users

 Application programs –Define the ways in which the system resources are used to solve the computing problems of the users like, Word processors, compilers, web browsers, database systems, video games

 Users: People, machines, other computers

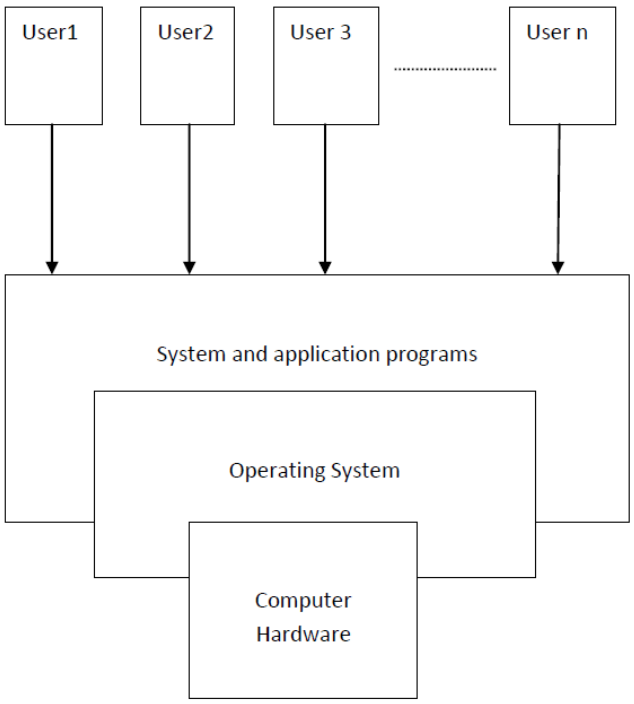
User View

The user view of computer varies by the interface being used. The operating systems are designed mostly for ease of use. Others are designed to maximize resource utilization. Other operating systems are designed to compromise between individual usability and resource utilization.

System view

From the computer's point of view, the OS is a:

- **resource allocator:** Manages all resources and decides between conflicting requests for efficient and fair resource use.
- **control program:** Controls execution of programs to prevent errors and improper use of the computer



Computer Structure

Operating systems goals:

1- Make the computer convenient to the user.
2- Use the computer hardware in an efficient manner.

- OS is like a government, it performs no useful functions by itself, but it provides an environment within which other programs can do useful work.

OS functions:

1- Convenient to the user.
2- Efficient operation of the computer.
3- Resource allocator: OS acts like a manager of resources and allocate them to a specific programs and users.
4- OS is a control program: controls the execution of the programs and prevent errors and improper use of computer and controls the I/O devices

Types of OS: generally there are two types (single user and multiuser) and another classification is as following:

1- Batch systems.
2- Time sharing systems.

3- PC systems.
4- Parallel systems (tightly coupled systems).
5- Real time systems.
6- Distributed systems (loosely coupled systems).

Operating System Historical Review

Operating systems and computer architecture have influenced each other. To facilitate the use of the hardware, researchers developed operating systems. In the following historical review, we will notice the mutual effect between operating systems and computer hardware which led to developments in both sides.

Mainframe systems

Mainframe systems grow on three stages:

1- Batch systems: In this type of computer systems, the operator batch together jobs with similar needs and ran through the computer as group. The operating system was simple and its major task was to transfer control automatically from one job to the next.
2- Multi-programmed systems: The operating system keeps several jobs in memory simultaneously. Operating systems for the Multi-programmed is the first one which make a decision for the users. Making this decision is called job scheduling.
3- Timeshared systems: The CPU executes multiple jobs by switching among them, but the switches occurred so frequently the users can interact with each program while it is running. A Timeshared operating systems allows many user programs (processes) to share the computer simultaneously. The CPU executes multiple jobs by switching among them, but the switches occurred so frequently the users can interact with each program while it is running.

Desktop systems

The operating systems of desktop systems were neither multi-user nor multitasking. Operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems improved to maximize user convenience and responsiveness.

Multiprocessor Systems (Parallel systems or tightly coupled systems)

Such systems have more than one processor in close communication sharing the computer bus, the clock, and sometimes memory and peripheral devices.

Multiprocessor systems have three main advantages

1- Increase throughput.

2- Economy of scale.

3- Increased reliability.

This ability to continue providing service proportional to the level of surviving hardware is called "graceful degradation" is also called "fault tolerant".

There are different architectures for multiprocessor systems.

Distributed Systems

A network is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Using communicates, distributed systems are able to share computational tasks, and provide a rich set of set of feature to users.

- client-server systems
- peer-to-peer systems

Some operating system benefits from ideas of networking and distributed systems in build network operating system.

Clustered Systems

Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work, they composed of two or more individual systems coupled together. The general accepted definition is that clustered computers share storage and is closely linked via LAN networking. Clustering is usually performed to provide high availability.

Real-Time Systems

Special purpose operating system, it is used when there are rigid time requirements on the operation of a processor or the flow of data, thus it is often used as a control device in dedicated application. Real time system need that the processing must be done within the defined time constraints or the system will fail.

There are two flavors of real time system:

- Hard real-time system
- Soft real time system

Handheld Systems

Handheld systems include personal digital assistants (PDAs). Developers of handheld systems and applications face many challenges (due to the limited size of such devices) such as speed of processor, limited size of memory, and small display screen.

Computing Environments

All above systems are used in verity of computing environments settings.

- Traditional Computing.
- Wed-Based computing.
- Embedded Computing.

Chapter 2: Operating System Structures

Computer System Operation:

A modern, general-purpose computer system consists of CPU and a number of device controllers that connected through a common bus that provides access to shared memory system, CPU other devices can execute concurrently competing for memory cycles.

Booting:

It is the operation of bringing operating system kernel from the secondary storage and put it in main storage to execute it in CPU. There is a program bootstrap which is performing this operation when computer is powered up or rebooted.

Bootstrap software: it is an initial program and simple it is stored in read-only memory (ROM) such as firmware or EEPROM within the computer hardware.

Jobs of Bootstrap program:

1- Initialize all the aspect of the system, from CPU registers to device controllers to memory contents.

2- Locate and load the operating system kernel into memory then the operating system starts executing the first process, such as “init” and waits for some event to occur.

The operating system then waits for some event to occur. Types of events are either software events (system call) or hardware events (signals from the hardware devices to the CPU through the system bus and known as an interrupt).

Note: all modern operating system are "interrupt driven".

Trap (exception): it is a software-generated interrupt caused either by an error (ex: division by zero or invalid memory access) or by a specific request from a user program that an operating system service be performed.

Interrupt vector (IV): it is a fixed locations (an array) in the low memory area (first 100 locations of RAM) of operating system when the interrupt occur the CPU stops what it is doing and transfer execution to a fixed location (IV) contain starting address of the interrupt service routine(ISR), on completion the CPU resumes the interrupted computation.

Interrupt Service Routine: is it a routine provided to be responsible for dealing with the interrupt.

Hardware protection:

when we have single user any error occur to the system then we could determine that this error must be caused by the user program ,but when we begin to dealing with spooling ,multiprogramming, and sharing disk to hold many users data this sharing both improved utilization and increase problems .

In multiprogramming system, where one erroneous program might modify the program or data of another program, or even the resident monitor itself. MS-DOS and the Macintosh OS both allow this kind of error.

A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other program to execute incorrectly.

Many programming error are detected by the hardware these error are normally handled by the operating system.

Dual-Mode Operation:

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. The approach taken by many operating systems provides hardware support that allows us to differentiate among various modes of execution.

A bit, called the ***mode bit*** is added to the hardware of the computer to indicates the current mode: monitor (0) or user (1) with mode bit we could distinguish between a task that is executed on behalf of the operating system , and one that is executed on behalf of the user.

I/O Operation Protection:

A use program may disrupt the normal operation of the system by issuing illegal I/O instruction we can use various mechanisms to ensure that such disruption can not take place in the system.

One of them is by defining all I/O instructions to be privileged instructions. Thus users cannot issue I/O instructions directly they must do it through the operating system, by execute a system call to request that the operating system performing I/O in its behalf. The operating system, executing in monitor mode, check that the request is valid, and (if the request is valid) does the I/O requested. The operating system then returns to the user.

Memory Protection:

To insure correct operation, we must protect the interrupt vector and interrupt service routine from modification by a user program. This protection must be provided by the hardware, we need the ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We could provide the protection by using two registers a base register and limit register

- Base register hold the smallest legal physical memory address.
- Limit register: contains the size of the range.

This protection is accomplished by the CPU hardware comparing every address generated in user mode with the registers. Any attempt by a program executing in user mode to access monitor memory or other users' memory results in a trap to the monitor, which treats the attempts as a fatal error.

CPU Protection:

In addition to protecting I/O and memory we must insure that the operating system maintains control. We must prevent the user from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system. To accomplish this goal, we can use ***a timer***.

Timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second) A variable timer is generally implemented by a fixed rate clock and a counter.

We can use the timer to prevent a user program from running too long Simple technique is to initialize a counter with the mount of time that a program is allowed to run.

Amore common use of timer is to implement time sharing. In the most case, the timer could be set to interrupt every N millisecond, where N is the time slice that each user is allowed to execute before the next user get control of the CPU. The operating system is invoked to perform housekeeping tasks.

This procedure is known as a context switching, following a context switch, the next program continues with its execution from the point at which it left off.

Operating System Structure

In the following lectures we will consider the components and services that are provided by different operating systems.

System Components

Many modern computer systems share the goal of supporting the following components:

Process management

A process can be thought of a program in execution. A process needs certain resources to accomplish its task. Also the process various initialization values.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are system processes others are user processes. All processes execute concurrently by multiplexing the CPU among them.

The OS responsible for the following activities in connection with process management:

- Creation and deletion both user and system processes.
- Suspending and resuming processes.
- Providing mechanisms for process synchronization.
- Providing mechanisms for process communication.
- Providing mechanisms for deadlock handling.

Main Memory Management

The main memory is the central to the operation of a modern computer system. For a program to be executed it must mapped to absolute addresses and loaded to the MM.

The OS responsible for the following activities in connection with MM management:

- 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 become available.
- Allocating and deallocating memory space as needed.

File Management

For convenient use of the computer, the OS provides a uniform logical view of information storage. The OS abstracts from the physical properties of its storage device to define the logical storage unit, the file. A file is a collection of related information defined by its creator. These files are organized in directories to ease their use.

The OS responsible for the following activities in connection with file management:

- Creating and deleting files.
- Creating and deleting directories.
- Supporting primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- Backing up files on stable storage media.

I/O System Management

One of the purposes of OS is to hide the peculiarities of specific hardware devices. The OS responsible for the following activities in connection with I/O system management:

- A memory management component that includes buffering, caching and spooling.
- A general device driver interface.
- Derivers for specific hardware devices.

Secondary Storage Management

The computer system must provide secondary storage to back up main memory because that are hold by MM are lost when power is switched of f and the MM is too small to

accommodate all data programs. The OS responsible for the following activities in connection with disk management:

- Free space management
- Storage allocation
- Disk scheduling

Networking

A distributed system collects physically separate heterogeneous system into a single coherent system, providing the user with the access to various resources that the system maintain. Access to a shared resource allows computation speed up, increase functionality, increase data arability, and enhance reliability.

Protection System

Protection is any mechanism for controlling the access programs, processes, or users to the resources defined by the computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

Command Interpreter System

Command Interpreter System is the interface between the user and the OS. Some of these Command Interpreter System are user friendly such as mouse based window and menus. In other shells commands are typed on a keyboard.

Operating System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of these programs. The specific services provided differ from one operating system to another but we can identify common classes. These operating system services are provided for the convenience of the programmer, to make the programming task easier.

1. Program execution. 2. I/O operation. 3. File system manipulation. 4. Communications. 5. Error detection. 6. Resource allocation. 7. Accounting. 8. Protection

System Calls

System calls provide the interface between a process and the operating system. These calls are generally available as assembly language instructions and they are usually listed in the various manuals used by assembly language.

System Programs

System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls others are considerably more complex. They can be divided into these categories:

- File management
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications

System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and to be modified easily. There are three different system structures:

- Simple structure
- Layered Approach
- Microkernal

System programs are the programs that serve as the supplementary environment to the user programs and they are divided into the following categories:

1- File management.
2- Status information.
3- File modification.
4- Programming language support.
5- Program loading and execution.
6- Communication.

Operating system structure:

1- No well defined: started as small, simple, limited and then grow with time (ex. MS-DOS).
2- Limited structuring: such as UNIX, which consist of two parts (kernel and system programs).
3- Partition the task into smaller components (modules): with carefully define I/P and O/P and functions.
4- Layered approach (modularity): OS is divided into a number of layers each built on the top of the lower layers. Layer 0 is the hardware and layer N is the user interface. Layered approach takes the advantage of (Virtual Machine).
5- Micro kernels: as the OS expands, it becomes difficult to manage so, it is divided into smaller kernels called microkernel that provides a communication facility between client programs and various services.
6- Virtual machines: H/W is the lowest level, then the kernel level uses the H/W instructions to create a set of (system calls), next level is the system programs and the top layer is the application programs.

Operating system components:

Process Management, Memory management, File management, Input/Output System Management, Secondary Memory Management, Command Interpreter System, Protection System, Networking (distribution system).

Chapter 3: Processes

Process Concept

An operating system executes a variety of programs such that Batch system executes jobs whereas Time-shared systems executes user programs or tasks. In this course we will use the terms *job* and *process* almost interchangeably.

Process: a program in execution and each process execution must progress in sequential fashion, despite of a single process of multiprocessing operating systems, each process must be executed as a code line by line.

A process is more than a program code (text section) it also includes a program counter represents the count of the current activity, an processor regisers, a stack which contains the

temporary data, data section (which contains global variables) and a heap (which is a memory that is dynamically allocated to a process during run time).

Multiple parts of process includes:

- The program code, also called text section
- Current activity including program counter, processor registers
- Stack containing temporary data and Function parameters, return addresses, local variables
- Data section containing global variables
- Heap containing memory dynamically allocated during run time
- A Program is a passive entity such as a file containing a list of instructions and stored on a disk which is called an executable program. Whereas the process is active entity with a program counter (PC) specifying the next instruction to be executed with a list of associated resources.

Program becomes process when executable file loaded into memory. Execution of program started via GUI mouse clicks, command line entry of its name, etc. One program can be several processes: Consider multiple users executing the same program

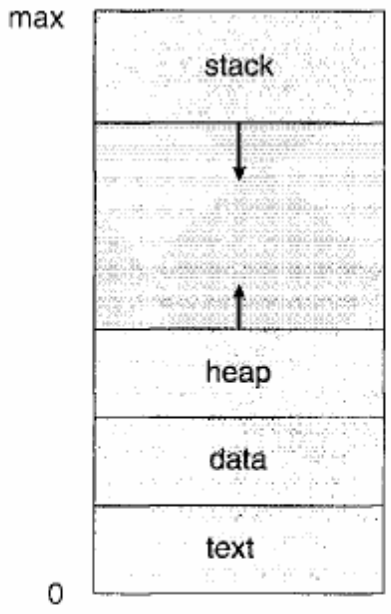
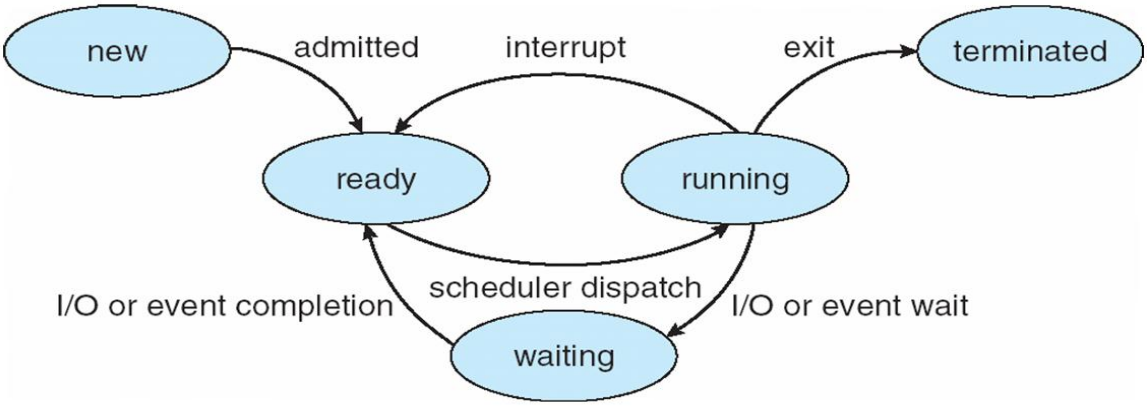


Figure 3.1 Process in memory.

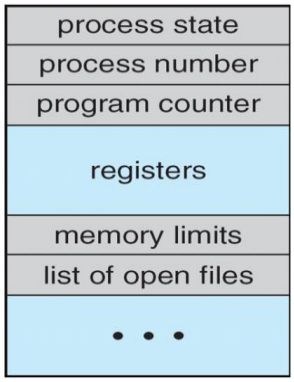
Process State: As a process executes, it changes *state* as the state of the process is defined as part of the current activity of the process execution.

- new: The process is being created
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur
- ready: The process is waiting to be assigned to a processor
- terminated: The process has finished execution

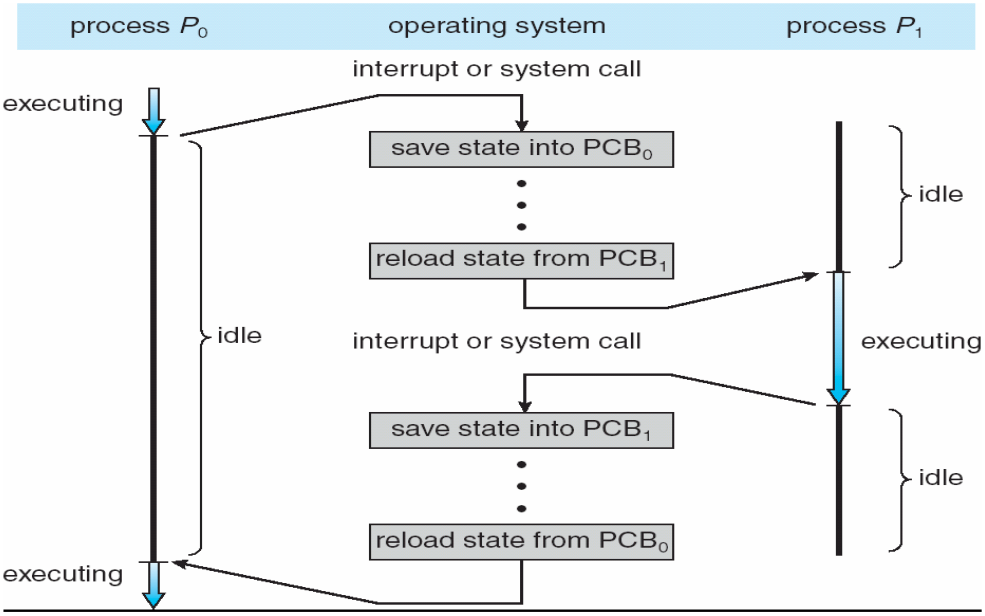
Diagram of Process State



Process Control Block (PCB): each process in the operating system is represented by a process control block (PCB) or task control block (TCB) which is the Information associated with each process and consist of: Process state, Program counter, CPU registers, CPU scheduling information, Memory-management information, Accounting information, I/O status information as in the figure:



CPU Switch From Process to Process

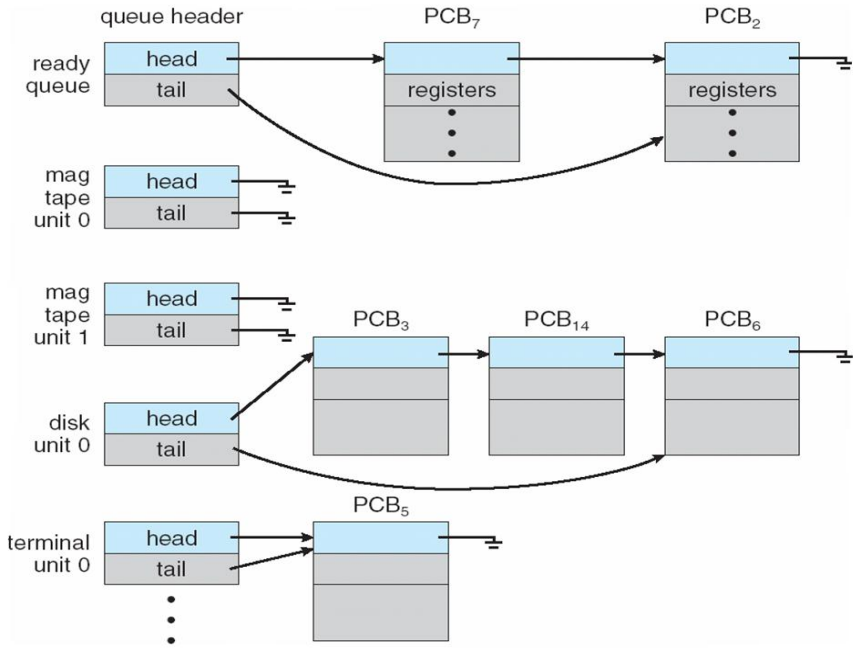


Process Scheduling

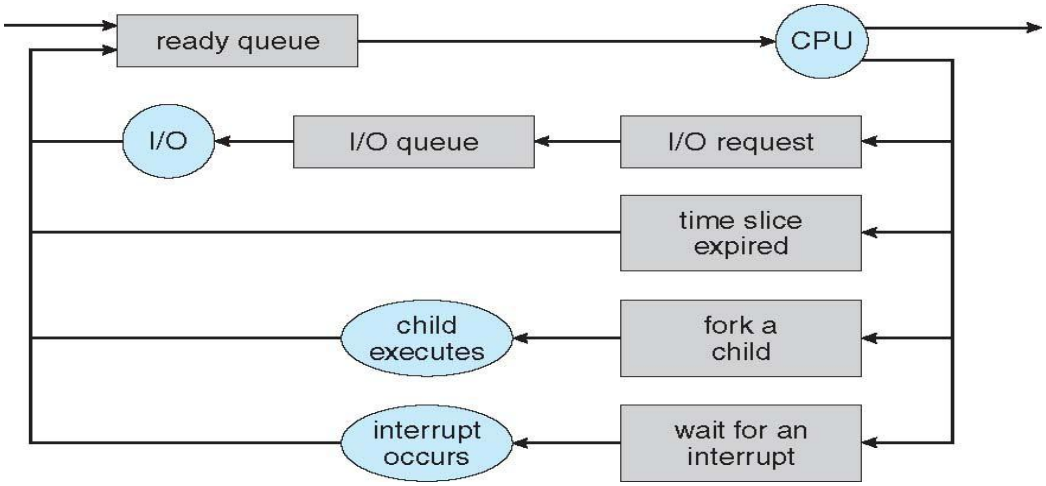
Multiprogramming and time sharing properties of modern operating systems aims to keep many processes in the memory and make the CPU as busy as possible by switching back and front among these processes which Maximize CPU utilization and reduce total execution time by quickly switch processes onto CPU for time sharing. Process scheduler (as a part of the OS) selects among available processes for next execution on CPU and Maintains scheduling queues of processes

- Job queue: set of all processes in the system
- Ready queue: set of all processes residing in main memory, ready and waiting to execute
- Device queues: set of processes waiting for an I/O device to be available.
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Queueing Diagram Representation of Process Scheduling



Process Schedulers

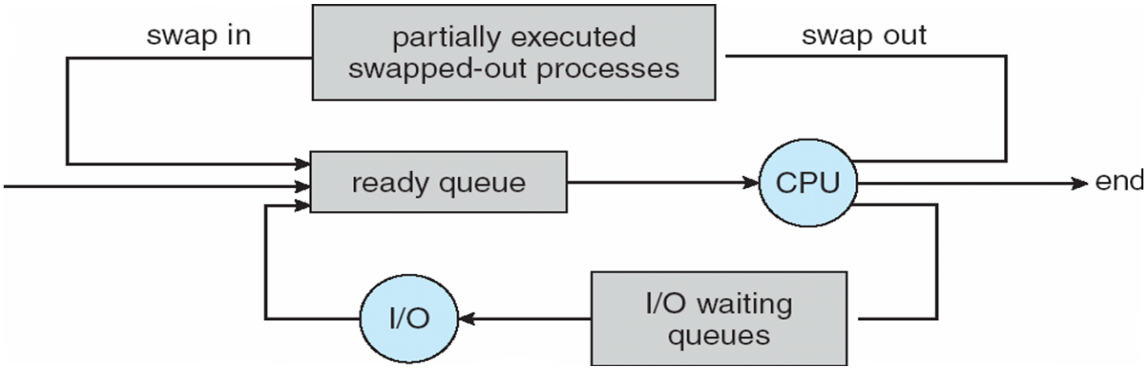
Long-term scheduler (job scheduler): selects which processes should be brought into the ready queue. Long-term scheduler is invoked very infrequently (once every some seconds or minutes so it may be slow). Also the long-term scheduler controls the *degree of multiprogramming*.

Short-term scheduler (CPU scheduler): selects which process should be executed next and allocates CPU to it, Sometimes it is the only scheduler in a system. Short-term scheduler is invoked very frequently every few milliseconds so it must be fast).

Medium term scheduler: some time sharing OS introduce additional intermediary schedulers to (sometimes) remove some processes form memory and from active contention for the CPU and reduce the degree of multiprogramming and continue these processes later.

Processes can be described as either I/O-bound process – spends more time doing I/O than computations, many short CPU bursts or CPU-bound process – spends more time doing computations; few very long CPU bursts.

Addition of Medium Term Scheduling (swapping process)



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

Process Creation

Parent process create children processes, which, in turn create other processes, forming a tree of processes. Generally, process identified and managed via a process identifier (PID)

Resource sharing can be with one of the following schemes (options):

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

Execution options:

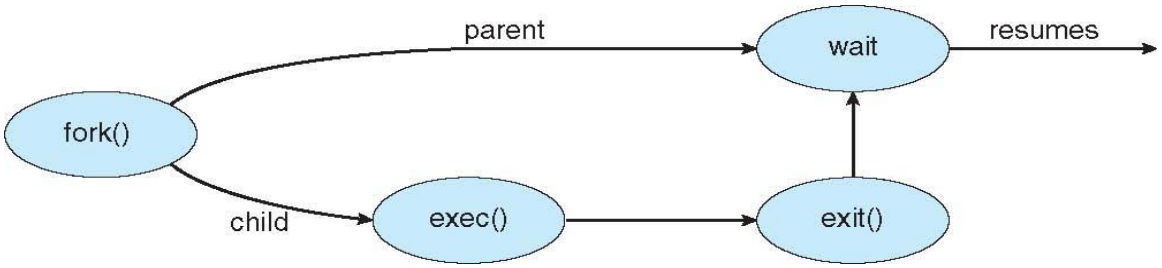
- Parent and children execute concurrently
- Parent waits until children terminate

Address space sharing options:

- Child duplicate of parent
- Child has a program loaded into it

UNIX examples for process creation:

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



Process Termination

Process executes last statement and asks the operating system to delete it (exit). Output data from child to parent (via wait). Process' resources are deallocated by operating system. Parent may terminate execution of children processes (abort). Child has exceeded allocated resources. Task assigned to child is no longer required. If parent is exiting some operating system do not allow child to continue if its parent terminates where all children terminated immediately in a process called cascading termination.

Interprocess Communication

Processes within a system may be independent or cooperating. Cooperating process can affect or be affected by other processes, including sharing data. Reasons for cooperating processes: Information sharing, Computation speedup, Modularity, Convenience

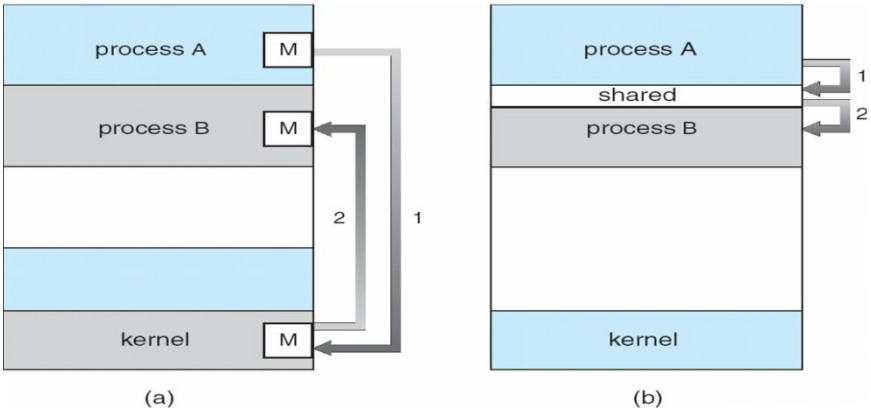
Cooperating processes need interprocess communication (IPC)

There are two models of IPC

1- Shared memory

2- Message passing

Communications Models



Cooperating Processes

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

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

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

Producer-Consumer Problem: Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

unbounded-buffer places no practical limit on the size of the buffer

bounded-buffer assumes that there is a fixed buffer size

Interprocess Communication – Message Passing

Mechanism for processes to communicate and to synchronize their actions. Message system – processes communicate with each other without resorting to shared variables.

IPC facility provides two operations:

1- send(*message*) – message size fixed or variable
2- receive(*message*)

If *P* and *Q* wish to communicate, they need to:

1- establish a *communication link* between them
2- exchange messages via send/receive

Implementation of communication link

1- physical (e.g., shared memory, hardware bus)
2- logical (e.g., logical properties)

Direct Communication

In this type of communication, the Processes must name each other explicitly:

send (*P, message*) – send a message to process P.

receive(*Q, message*) – receive a message from process Q.

Properties of communication link

1- Links are established automatically
2- A link is associated with exactly one pair of communicating processes
3- Between each pair there exists exactly one link

The link may be unidirectional, but is usually bi-directional

Indirect Communication

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

Each mailbox has a unique id. Processes can communicate only if they share a mailbox

Properties of communication link

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

Operations

1- create a new mailbox
2- send and receive messages through mailbox
3- destroy a mailbox
4- Primitives are defined as:

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

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

Mailbox sharing problems:

1- P_1, P_2, and P_3 share mailbox A
2- P_1, sends; P_2 and P_3 receive
3- Who gets the message?

Solutions

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

Synchronization

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

Buffering

Buffering is the process of sending and receiving a messages between two devices or two processes with different speeds were a Queue of messages attached to the link; implemented in one of three ways

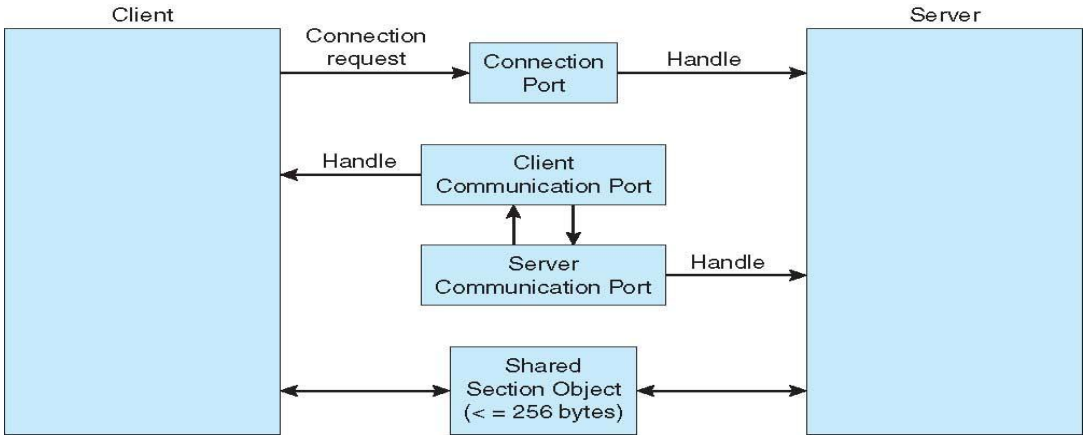
1. Zero capacity – 0 messages: Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages: Sender must wait if link full
3. Unbounded capacity – infinite length: Sender never waits

Examples of IPC Systems – Windows XP

Message-passing centric via local procedure call (LPC) facility. Only works between processes on the same system. Uses ports (like mailboxes) to establish and maintain communication channels. Communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows XP



Communications in Client-Server Systems

1- Sockets

2- Remote Procedure Calls

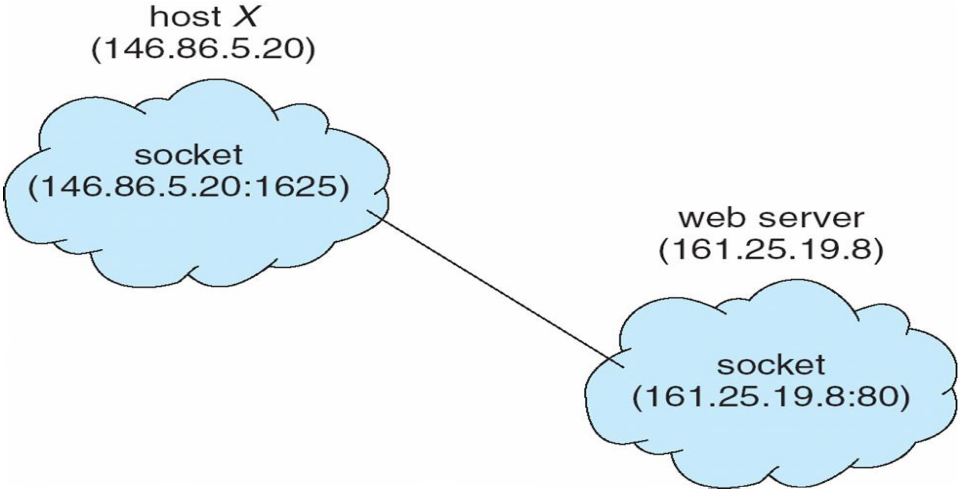
3- Pipes

4- Remote Method Invocation (Java)

1- Sockets

A socket is defined as an *endpoint for communication* a pair of processes communicating over a network employ a pair of sockets. Concatenation of IP address and port number, sockets can be identified. The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8. Communication consists between a pair of sockets.

Socket Communication



Java programming language provides three types of sockets as following:

1- Connection oriented sockets (TCP sockets) are implemented with socket class.
2- Connectionless socket (UDP sockets) use the DatagramSocket class.
3- Multicast socket class which is a subclass of the DatagramSocket class.

2- Remote Procedure Calls

One of the most common forms of remote service is the Remote procedure call (RPC) which abstracts procedure calls mechanism between processes on networked systems. RPC allow users to invoke a procedure stored on a remote host just as it is invoked locally using some kind of code called (stub). Stubs are client-side proxy for the actual procedure on the server. The client-side stub locates the server and *Marshals* the parameters. The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server.

3- Pipes

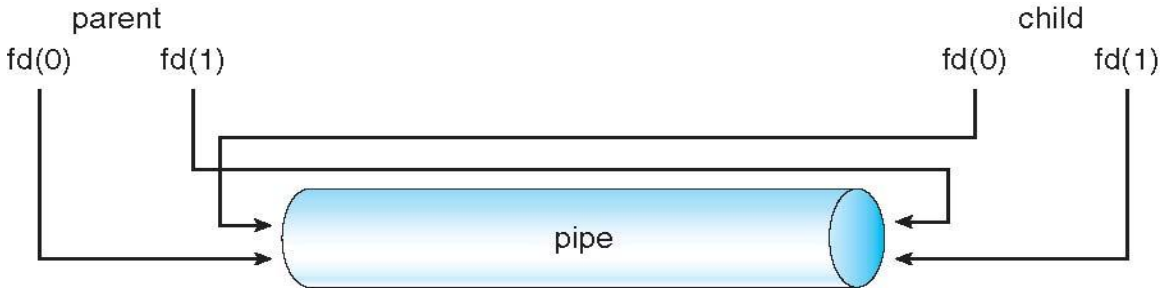
One of the communication tools among processes in a coordinated processes system are the pipes which Acts as a conduit allowing two processes to communicate. The Issues controlling the use of pipes can be classified as:

- Is communication unidirectional or bidirectional?

- In the case of two-way communication, is it half or full-duplex?
- Must there exist a relationship (i.e. parent-child) between the communicating processes?
- Can the pipes be used over a network?

Ordinary Pipes

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



Named Pipes

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

4- **Remote Method Invocation (RMI):** it is a Java feature similar to RPC allows a thread to invoke a method on a remote method.

Chapter 4: Threads

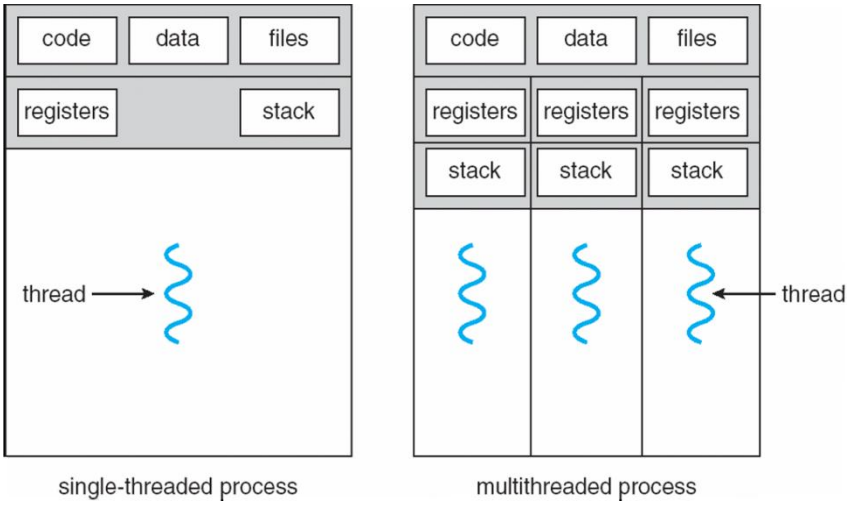
Thread (task) is a basic unit of the CPU utilization consist of tread ID, program counter (PC), register set, and a stack. Thread share with other threads belonging to the same process the code section, the data section, and the OS resources. Threads run within application as a process (single thread process) or a part of process (multithreading process). Multiple tasks with the application can be implemented by separate threads such as Update display, Fetch data, Spell checking, Answer a network request .

Example of multithreads in a single process:

Word processor may have many threads , one for displaying graphics, another for responding for the keystrokes of the used, third for performing spelling and grammar checking and so on.

Process creation is heavy-weight while thread creation is light-weight which means that dealing with threads make the system faster and less need for mare hardware. Dealing with treads instead of processes can simplify code, increase efficiency. OS Kernels are generally multithreaded

Single and multithreaded processes



Benefits of Multithreading Programming:

1- Responsiveness: allow a program to continue running even if a part of a program is blocking or is performing a lengthy operation.

2- Resource Sharing: threads are sharing the memory and resources of the process to which they belong.

3- Economy: resource sharing allow reducing the required memory and resources to create and manage multithreads within a single process.

4- Utilization of Multiprocessor architecture: multithreads can be executed in parallel using multiprocessors in a multiprocessing systems.

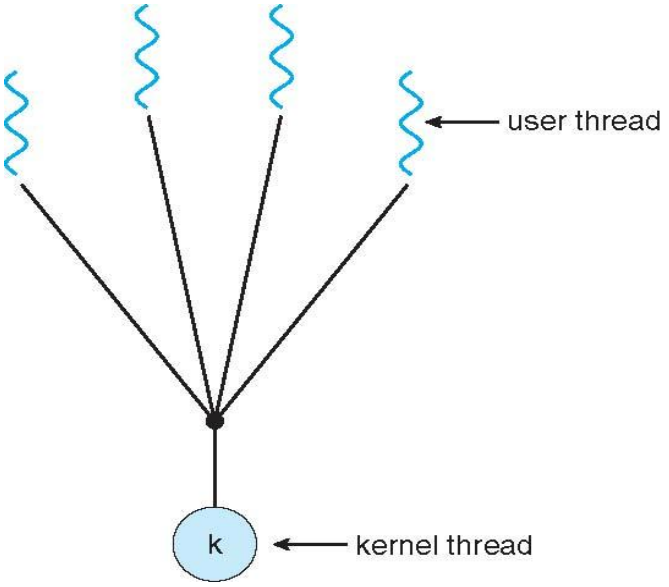
User Threads: where the thread management done by user-level threads library above the kernel and managed without the kernel support. Three primary thread libraries: POSIX Pthreads, Win32 threads, Java threads

Kernel Threads: are the threads that are managed and supported directly by the OS Kernel. Examples (Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X)

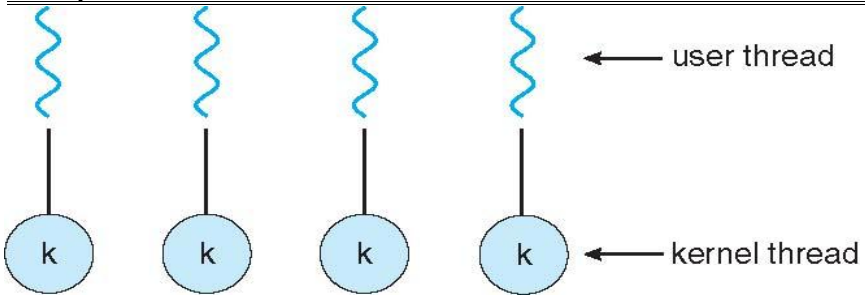
Multithreading Models

1- Many-to-One
2- One-to-One
3- Many-to-Many

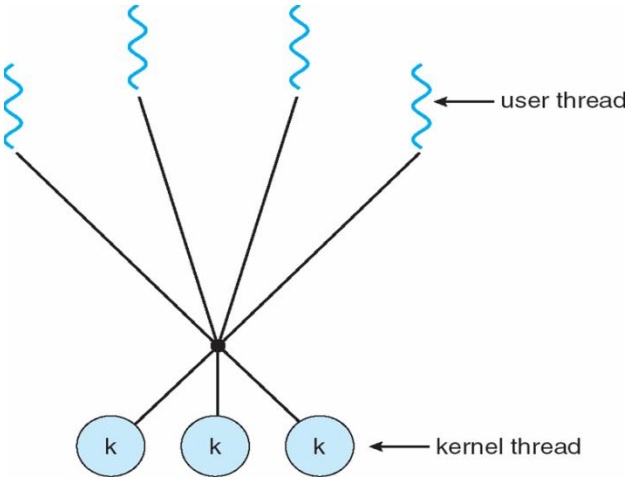
Many-to-One: Many user-level threads mapped to single kernel thread and thread management done by the thread library in the user space. Examples: Solaris Green Threads, GNU Portable Threads



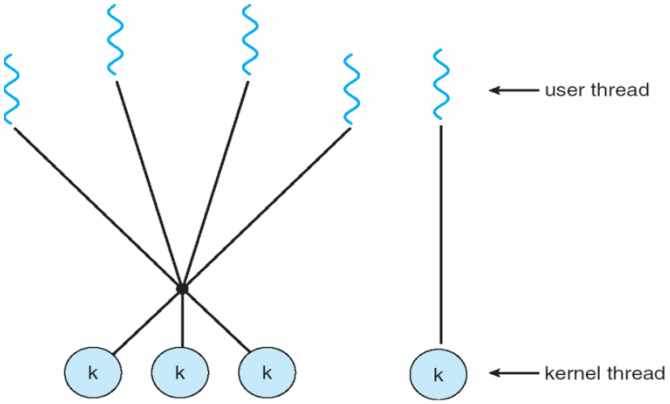
One-to-One: Each user-level thread maps to one kernel thread and it provides more concurrency than the first one by allowing another thread to work if one thread make a blocking system call. Examples: Windows NT/XP/2000, Linux, Solaris 9 and later



Many-to-Many Model: Allows many user level threads to be mapped to many kernel threads. Allows the operating system to create a sufficient number of kernel threads. Examples are Solaris prior to version 9 and Windows NT/2000 with the *Thread Fiber* package



Two-level Model: Similar to M:M model with a little variation where the user and kernel threads are still multiplexed but allow also a user thread to be bound to kernel thread. Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



Thread Libraries: Thread library provides programmer with API for creating and managing threads and there are two primary ways of implementing the thread library either the library is entirely being in user space or the Kernel-level library supported by the OS.

There are three main thread libraries in use today: POSIX PThreads, Win32, and JAVA.

Treading Issues:

1- **Fork() and exec():**

Fork() system call used to create a separate, duplicate processes but here invoking a fork() system call can cause one of two results:

- Duplicating all the threads in the process invoking the fork().
- Duplicating only the thread invoked the fork().

Exec() system call used to save the results of a process and a thread in the same way.

2- **Thread cancellation:**

It is the operation of terminating the thread before it completes its task and always the thread to be canceled called the (target thread). Cancellation done in one of two ways:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.
2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Signal Handling: Signals are used in UNIX systems to notify a process that a particular event has occurred (ex. Illegal memory access, or divide by zero). A signal handler is used to process signals in the following procedure:

1- Signal is generated by particular event

2- Signal is delivered to a process

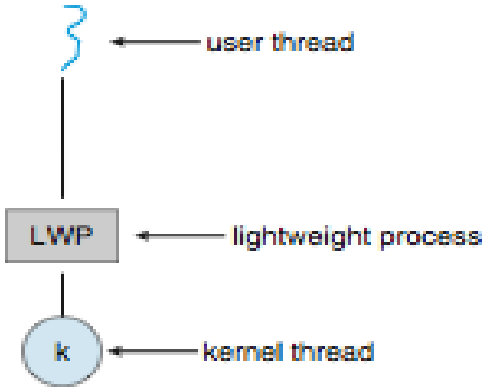
3- Signal is handled

Options:

- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process

Thread Pools means creating a number of threads in a pool where they await working and the advantages of it are that it is usually slightly faster to service a request with an existing thread than create a new thread and allows the number of threads in the application(s) to be bound to the size of the pool.

Lightweight Processes: it is an intermediate data structure between user and kernel threads and used to facilitate communication and coordination between the kernel and thread library.



Windows XP Threads: Implements the one-to-one mapping, kernel-level. Each thread contains: (A thread id, Register set, Separate user and kernel stacks, Private data storage area, The register set, stacks, and private storage area are known as the context of the threads)

The primary data structures of a thread include:

- ETHREAD (executive thread block)
- KTHREAD (kernel thread block)
- TEB (thread environment block)

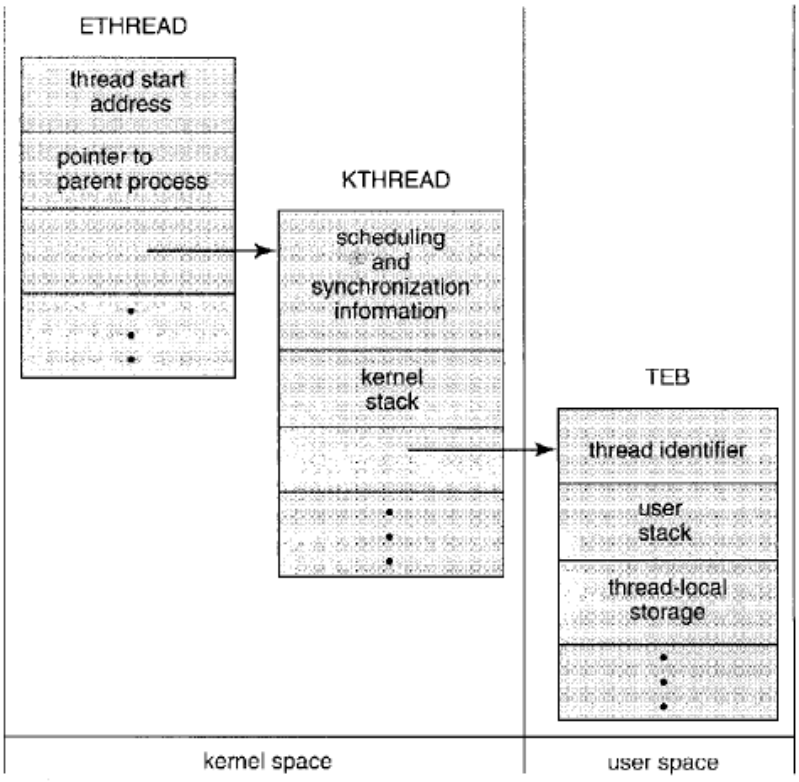


Figure 4.10 Data structures of a Windows XP thread.

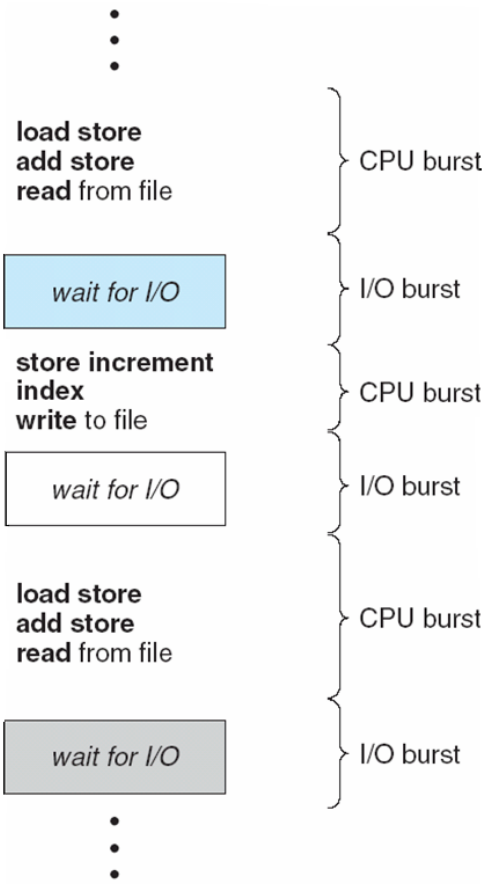
Chapter 5: CPU Scheduling

Basic Concepts

In a single process systems there is only one process for execution in any time, so there is no need for scheduling. In multiprocessing and time sharing systems the scheduling of the SPU is a basic task for maximum CPU utilization and it is simple: each process executed until it must wait (for some I/O device) where another process enters to the CPU for execution.

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

Alternating Sequence of CPU And I/O Bursts



CPU Scheduler: Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them, CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

- Scheduling under 1 and 4 is non-preemptive (the process do not leave the CPU until it finished) , whereas the other scheduling is preemptive (the process can leave the CPU voluntarily for shorter of more priority processes depending on scheduling criteria).

Dispatcher: Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; and this involves switching context, switching to user mode and jumping to the proper location in the user program to restart that program

Dispatch latency– time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

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

Scheduling Algorithm Optimization Criteria (what we want)

1- Max CPU utilization
2- Max throughput
3- Min turnaround time
4- Min waiting time
5- Min response time

First-Come, First-Served (FCFS or FIFO) Scheduling

In this scheduling algorithm, the first process enters the ready queue will enters to the CPU first and the second process enters second and so on.

Process	Burst Time
P_1	24
P_2	3
P_3	3

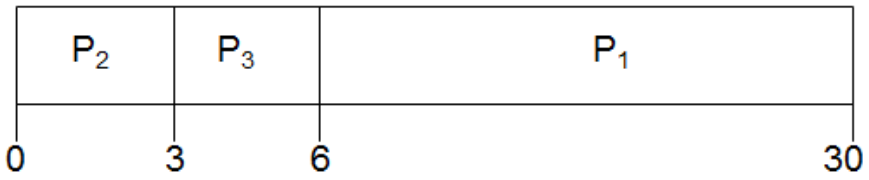
Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order P_2 , P_3 , P_1

The Gantt chart for the schedule is:



Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$

Average waiting time: (6 + 0 + 3)/3 = 3

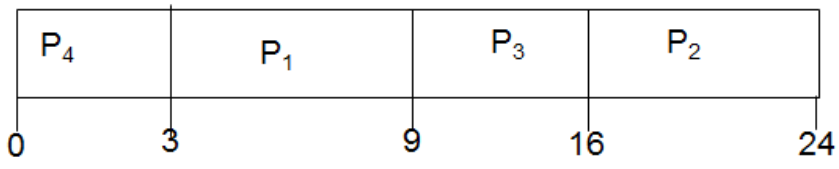
The second case is much better than previous case because of the *Convoy effect* where the FCFS algorithm is doing bad when short process arrives to the ready queue behind long process

Shortest-Job-First (SJF) Scheduling

Sometimes it is called (Shortest Remaining Time First SRTF) and includes associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time. SJF is optimal – gives minimum average waiting time for a given set of processes, The difficulty is knowing the length of the next CPU request. SJF can be preemptive or non-preemptive.

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



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

Priority Scheduling

A priority number (integer) is associated with each process depending on the type of process and the CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority). Equal priority processes are treated in FIFO order. Priority scheduling can be

preemptive or non-preemptive. SJF is a priority scheduling where priority is the predicted next CPU burst time (the shorter the process the higher priority it will get and vice versa). The Problem of this algorithm **is Starvation** which means that low priority processes may never execute as shorter processes keep arriving to the ready queue. The Solution for such problem is the process of **Aging which means that** as time progresses the OS must increase the priority of the process.

Round Robin (RR)

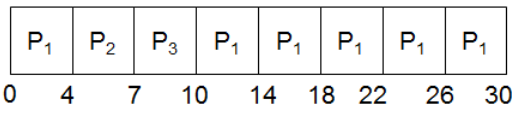
This scheduling algorithm was designed especially for time sharing systems where each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

The Performance of this algorithm can be similar to the FCFS if the time slice (q) is too large and if the (q) is too small then many context switched may occur that make an overhead on the system performance. So the (q) must be with adequate system dependent value.

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



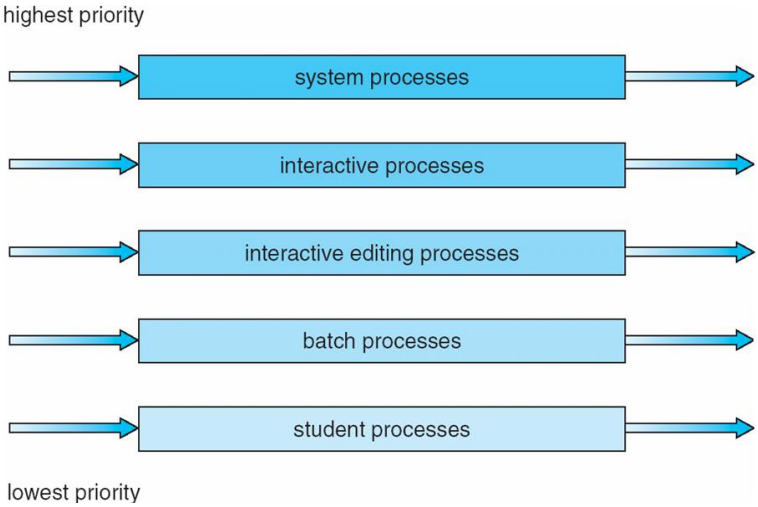
- Typically, higher average turnaround than SJF, but better *response*

Multilevel Queue

Another type of scheduling is designed for situation when processes can be easily classified into different groups. Ready queue is partitioned into separate queues:

foreground (interactive) and background (batch) with Each queue has its own scheduling algorithm. For example foreground – RR and background – FCFS

Scheduling must be done between the queues with either fixed priority scheduling; (i.e., serve all from foreground then from background) and here is possibility of starvation. Or Time slice where each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS



Multilevel Feedback Queue

In spite of the low scheduling overhead of the previous type of scheduling but it is inflexible so we use this type where a process can move between the various queues; aging can be implemented this way. Multilevel-feedback-queue scheduler defined by the following parameters:

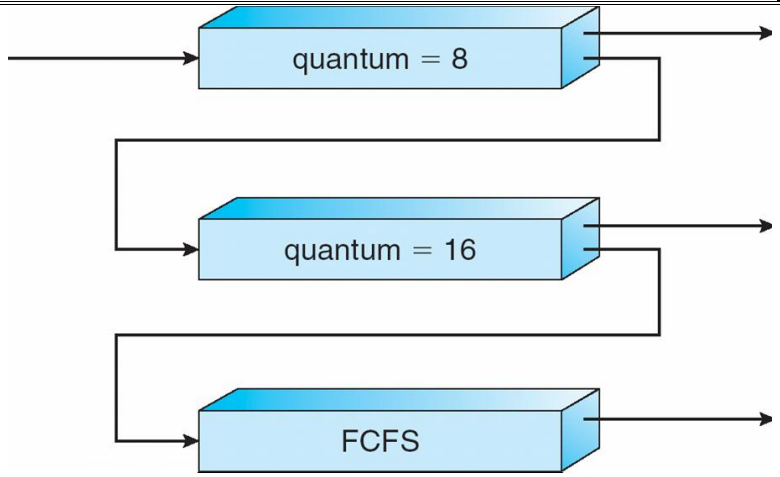
1- number of queues
2- scheduling algorithms for each queue
3- method used to determine when to upgrade a process
4- method used to determine when to demote a process
5- method used to determine which queue a process will enter when it needs service

Example of Multilevel Feedback Queue

Three queues: with Q_0 – RR with time quantum 8 milliseconds and Q_1 – RR time quantum 16 milliseconds and Q_2 – FCFS

Scheduling

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



Multiple-Processor Scheduling

All the above scheduling algorithms were for single processor systems and for multiple processor systems a concept of load sharing must adopted. CPU scheduling more complex when multiple CPUs are available.

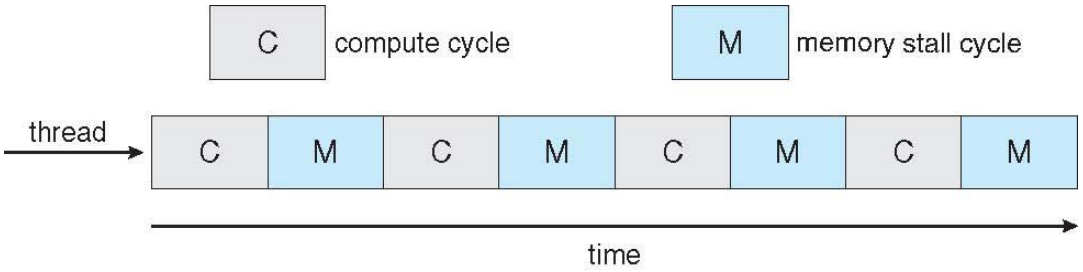
Homogeneous processors within a multiprocessor

1- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
2- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
3- **Processor affinity** – process has affinity for processor on which it is currently running and SMP systems must prevent the process form migrating from one processor to another.

- **soft affinity**: in systems that has a policy attempting to prevent migration but it still could happened.
- **hard affinity**: in systems that has a system calls preventing processes from migrating from a processor to another. Such as in LINUX operating system.

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

Multithreaded Multicore System



Symetric Multithreading Systems (SMT)

Allow several threads to run concurrently by providing multiple physical processors, alternative strategy of SMT is to provide multiple logical processors which is called also hyper threading technology such as in Intel processors.

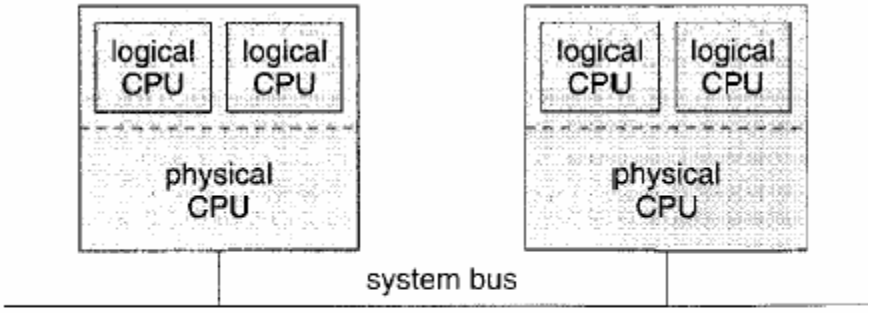


Figure 5.8 A typical SMT architecture

Chapter 6: Process Synchronization

Cooperating process: is a process that affect or being affected by the execution of other processes in the system. These processes share logical address space or data and messages.

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.

Race Condition a situation where several processes access and manipulate the same data concurrently and the results of the execution depends on the particular order in which the access take place. As in the following 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}

Critical section: it is a segment of case in each process in the system where the process may changing common variables, updating a table, writing a file, and so on.

Critical section problem: is to design a protocol that allow processes to cooperate with a condition that when one of them enters execution of its critical section, other are not.

Solution to Critical-Section Problem

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

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

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted, Assume that each process executes at a nonzero speed, No assumption concerning relative speed of the N processes

Critical section problem solutions:

1- **Peterson's Solution**

A classical software based solution that is restricted to only two processes that alternates the execution of the critical section and the remaining section of them. Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted. The two processes share two variables:

int turn;

Boolean flag[2]

The variable turn indicates whose turn it is to enter the critical section. The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

2- **Synchronization Hardware**

Many systems provide hardware support for critical section code by using a uniprocessors environment that prevent interrupts from occurring while a shared variable war being modified. Currently running code would execute without preemption. Generally it is too inefficient on multiprocessor systems. Operating systems using this not broadly scalable. Modern machines provide special atomic (non-interruptible) hardware instructions.

3- **Semaphore:** is a synchronization tool that does not require busy waiting, Semaphore is an integer variable with two standard atomic operations: wait() and signal() Originally called P() and V(). All the modifications to the integer value of the semaphore in the wait() and signal () operations must be executed indivisibly.

Less complicated. Can only be accessed via two indivisible (atomic) operations

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

Semaphore as General Synchronization Tool: there are two types of semaphores:

Counting semaphore – integer value can range over an unrestricted domain

Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement, Also known as mutex locks . Can implement a counting semaphore S as a binary semaphore. Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {    wait (mutex);
        // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

Semaphore Implementation

Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time. Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section. Could now have busy waiting in critical section implementation

But implementation code is short. Little busy waiting if critical section rarely occupied. Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items: value (of type integer), pointer to next record in the list. Two operations:

- Block – place the process invoking the operation on the appropriate waiting queue.
- Wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Deadlock and Starvation

Deadlock (indefinite blocking): happens when two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. Let S and Q be two semaphores initialized to 1

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

Starvation – indefinite blocking means that a process may never be removed from the semaphore queue in which it is suspended

Priority Inversion Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

N buffers, each can hold one item

Semaphore mutex initialized to the value 1

Semaphore full initialized to the value 0

Semaphore empty initialized to the value N.

Readers-Writers Problem

A data set is shared among a number of concurrent processes

Readers – only read the data set; they do not perform any updates

Writers – can both read and write

Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

Shared Data

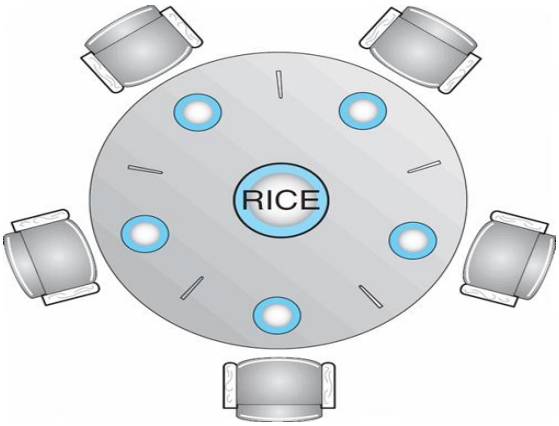
Data set

Semaphore mutex initialized to 1

Semaphore wrt initialized to 1

Integer readcount initialized to 0

Dining-Philosophers Problem



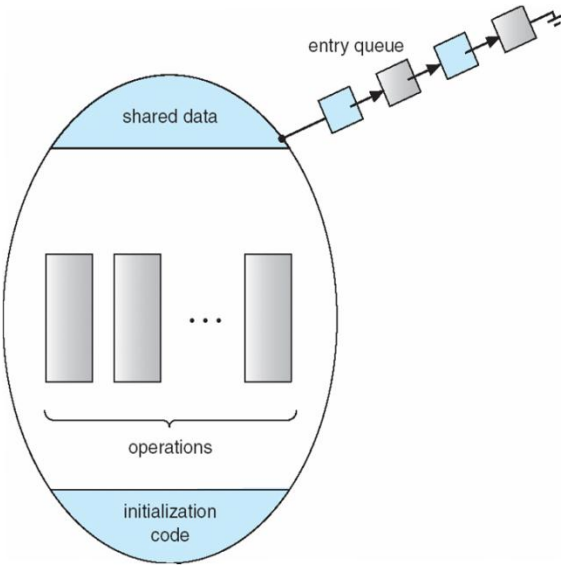
Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.14). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The *dining-philosophers problem* is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore; she releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

Monitors: 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

Schematic view of a Monitor



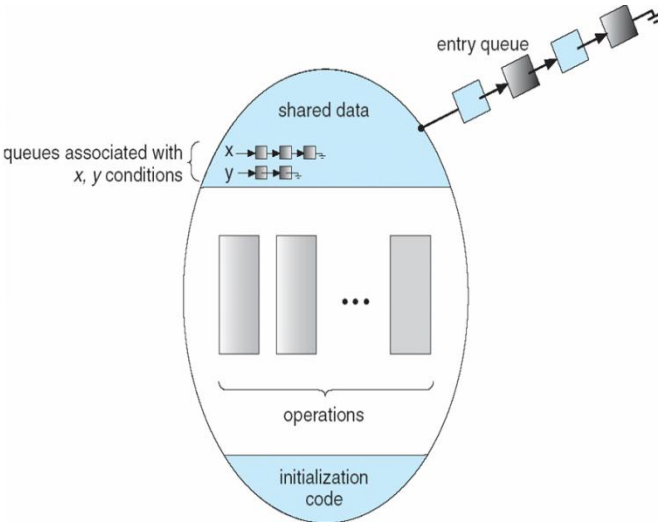
Condition Variables

- condition x, y; and two operations on a condition variable:

  x.wait () – a process that invokes the operation is suspended.

  x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Synchronization Examples

Solaris, Windows XP, Linux, Pthreads

Solaris Synchronization

Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing. Uses adaptive mutexes for efficiency when protecting data from short code segments. Uses condition variables and readers-writers locks when longer

sections of code need access to data. Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

Uses interrupt masks to protect access to global resources on uniprocessor systems. Uses spinlocks on multiprocessor systems. Also provides dispatcher objects which may act as either mutexes and semaphores. Dispatcher objects may also provide events, An event acts much like a condition variable

Linux Synchronization

Linux: Prior to kernel Version 2.6, disables interrupts to implement short critical sections.

Version 2.6 and later, fully preemptive, also Linux provides: semaphores, spin locks

Pthreads Synchronization

Pthreads API is OS-independent, it provides: mutex locks, condition variables, non-portable extensions include: read-write locks, spin locks

Chapter 7: Deadlocks

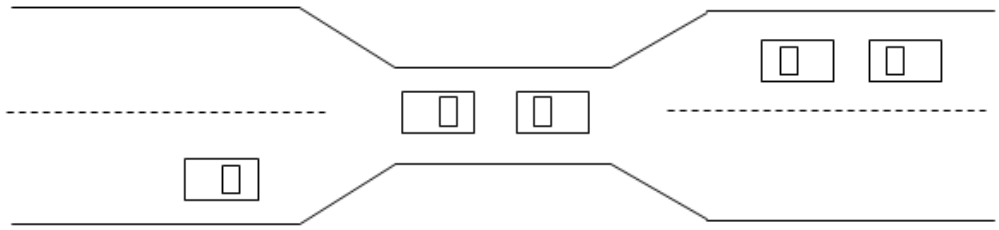
The Deadlock Problem:

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

Example: semaphores *A* and *B*, initialized to 1

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

Bridge Crossing Example



Traffic only in one direction. Each section of a bridge can be viewed as a resource

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

Several cars may have to be backed up if a deadlock occurs. Starvation is possible

Note – Most OSs do not prevent or deal with deadlocks

System Model

Resource types $R_1, R_2, \ldots, R_m$: could be *CPU cycles, memory space, I/O devices*

Each resource type R_i has W_i instances. Each process utilizes a resource as follows:

- **request**
- **use**
- **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

1- **Mutual exclusion:** only one process at a time can use a resource

2- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

3- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

4- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource-Allocation Graph

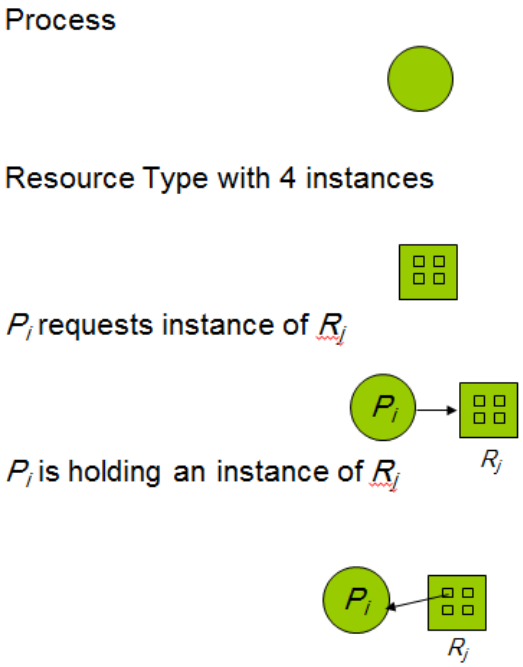
A set of vertices V and a set of edges E.

V is partitioned into two types:

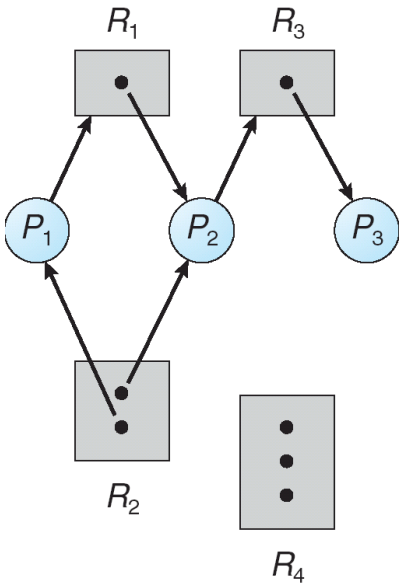
- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
- $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

request edge – directed edge $P_1 \rightarrow R_j$

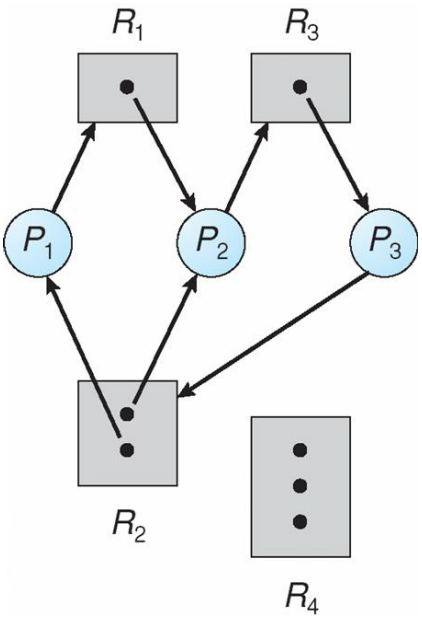
assignment edge – directed edge $R_j \rightarrow P_i$



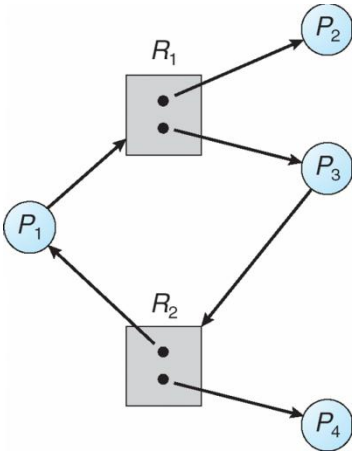
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

1- If graph contains no cycles ⇒ no deadlock

2- 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

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

Deadlock Prevention: Restrain the ways request can be made

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

2- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
- Low resource utilization; starvation possible

3- **No Preemption** – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

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

Deadlock Avoidance

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

Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait

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

Safe State

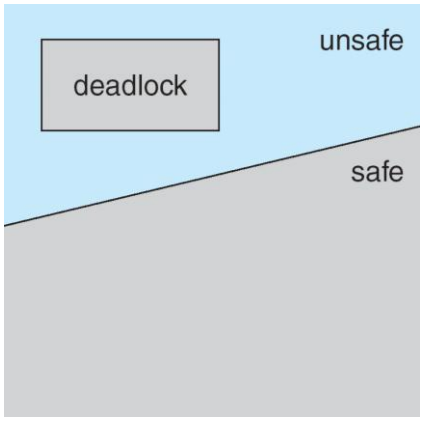
When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$. That is:

- If P_i resource needs are not immediately available, then P_i can wait until all $P_j$have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Safe, Unsafe , Deadlock State



Avoidance algorithms

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

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

Resource-Allocation Graph Scheme

Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line

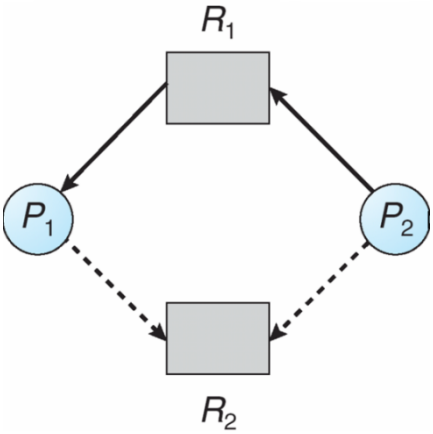
Claim edge converts to request edge when a process requests a resource

Request edge converted to an assignment edge when the resource is allocated to the process

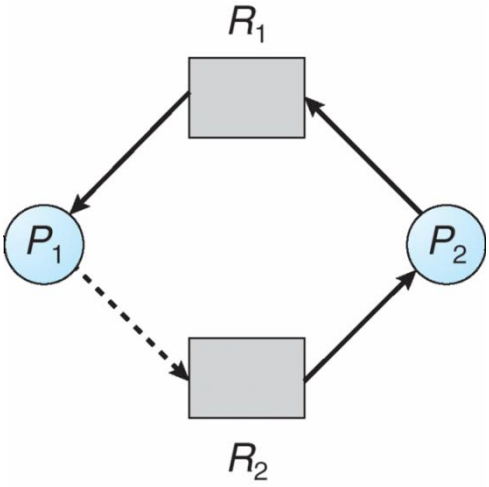
When a resource is released by a process, assignment edge reconverts to a claim edge

Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

Suppose that process P_i requests a resource R_j. The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

It is used for Multiple instances resources. each process must a priori claim maximum use. When a process requests a resource it may have to wait. When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

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

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

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

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

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

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

Safety Algorithm

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

$Work = Available$

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

2- Find and i such that both:

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

(b) $Need_i \leq Work$

If no such i exists, go to step 4

3- $Work = Work + Allocation_i$

$Finish$[i] = $true$

go to step 2

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

Resource-Request Algorithm for Process P_i

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

1.If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2.If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3.Pretend to allocate requested resources to P_i by modifying the state as follows:

Available = Available – Request;

$Allocation_i = Allocation_i + Request_i$;

$Need_i = Need_i - Request_i$;

1- *If safe ⇒ the resources are allocated to Pi*

2- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

Assume that you have a system with 5 processes P_0 through P_4; and 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: if P_1 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

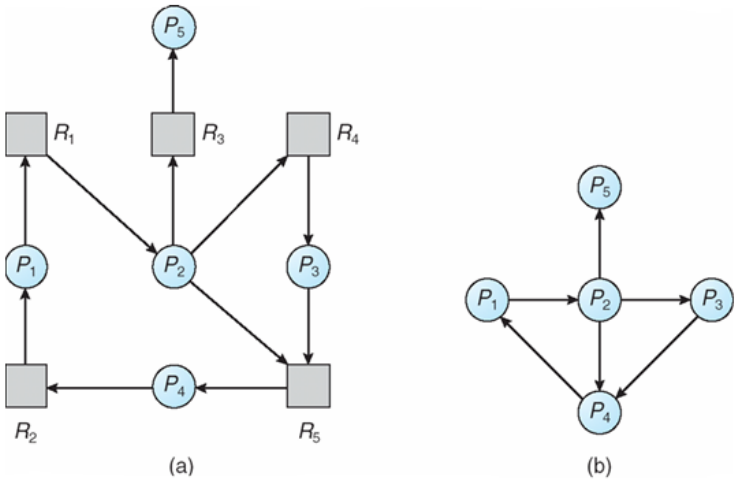
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

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

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

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

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

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

Detection Algorithm

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

(a) *Work* = *Available*

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

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

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

(b) $Request_i \leq Work$

3- If no such *i* exists, go to step 4

$Work = Work + Allocation_i$

Finish[*i*] = *true*

go to step 2

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

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

Example of Detection Algorithm

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

Snapshot at time T_0:

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

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[i] = true for all i

P_2 requests an additional instance of type C

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

State of system?

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

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

Detection-Algorithm Usage

When, and how often, to invoke depends on:

- How often a deadlock is likely to occur?
- How many processes will need to be rolled back? one for each disjoint cycle

If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock

Recovery from Deadlock: Process Termination

1- Abort all deadlocked processes
2- Abort one process at a time until the deadlock cycle is eliminated
3- 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?

Recovery from Deadlock: Resource Preemption

Selecting a victim – minimize cost

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

Starvation – same process may always be picked as victim, include number of rollback in cost factor

Chapter 8: Main Memory Management

Background

Program must be brought from HD 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)

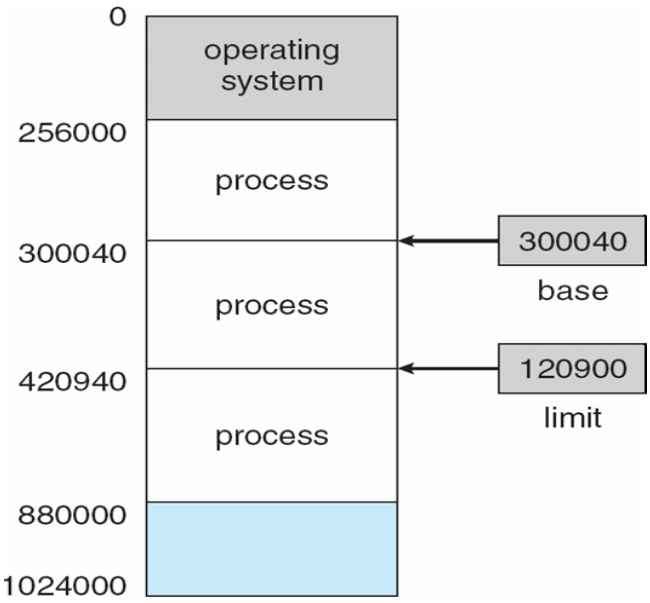
Main memory can take many cycles

Cache sits between main memory and CPU registers

Protection of memory required to ensure correct operation

Base and Limit Registers

A pair of base and limit registers define the logical address space



Binding of Instructions and Data to Memory

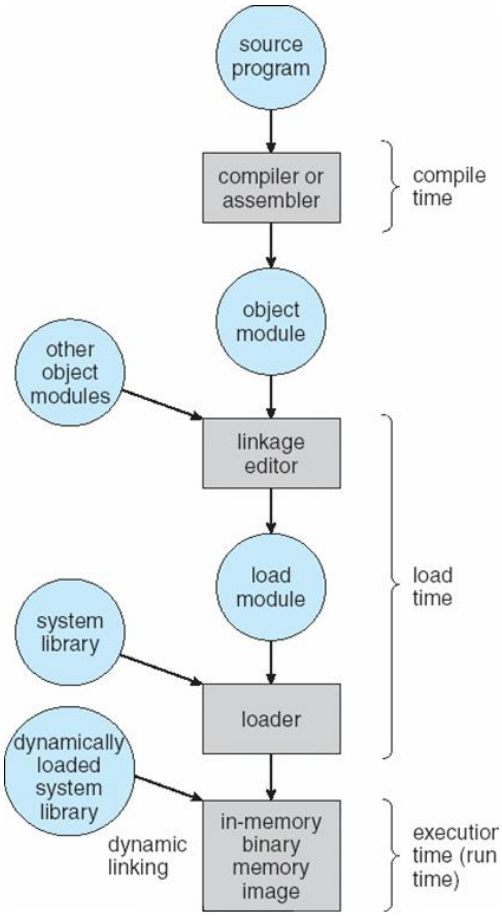
Address binding of instructions and data to memory addresses can happen at three different stages

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

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

Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

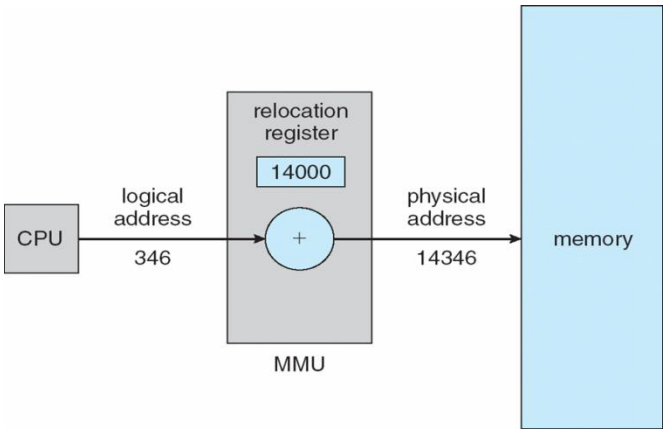
The concept of a logical address space that is bound to a separate physical address space is central to proper memory management. Logical address – generated by the CPU; also referred to as virtual address. Physical address is the address seen by the memory unit.

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

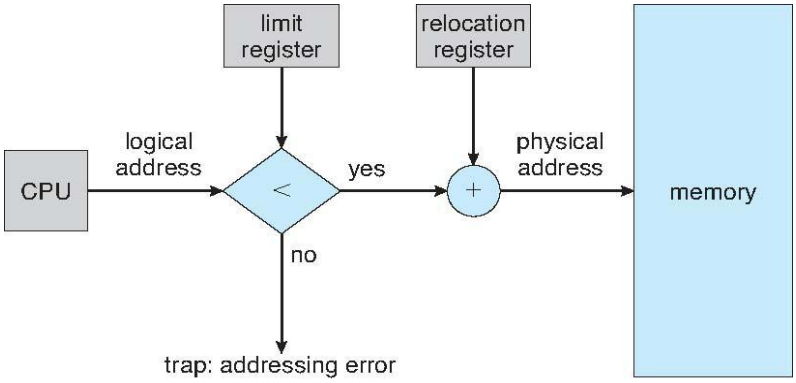
Memory-Management Unit (MMU)

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

Dynamic relocation using relocation register



Hardware Support for Relocation and Limit Registers



Dynamic Storage-Allocation Problem

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

- First-fit: Allocate the *first* hole that is big enough

- Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size, Produces the smallest leftover hole

- Worst-fit: Allocate the *largest* hole; must also search entire list, Produces the largest leftover hole

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

Paging

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

Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes). Divide logical memory into blocks of same size called pages

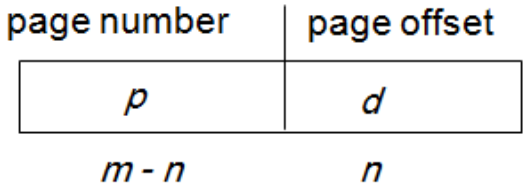
Keep track of all free frames. To run a program of size *n* pages, need to find *n* free frames and load program. Set up a page table to translate logical to physical addresses

Address Translation Scheme

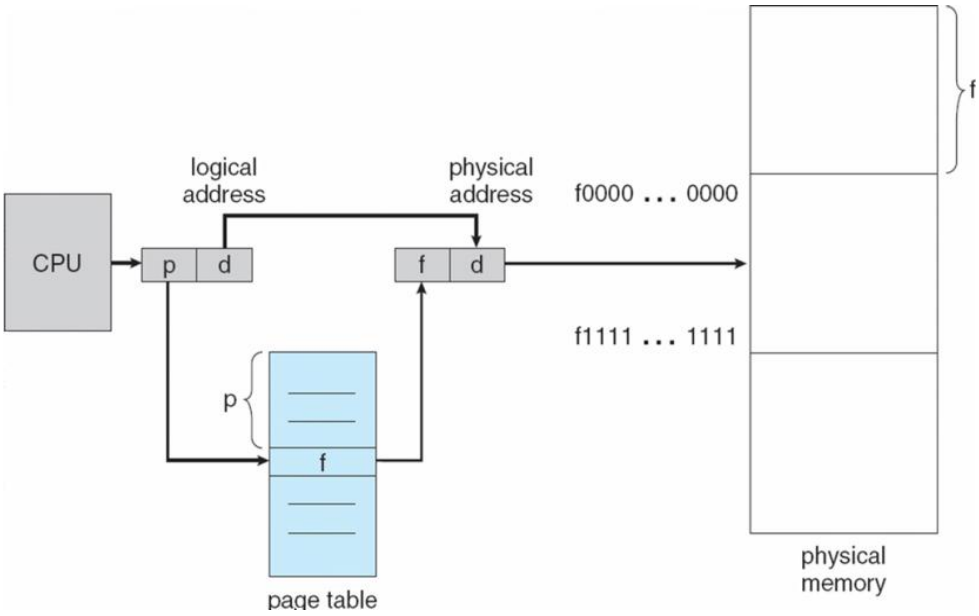
- Address generated by CPU is divided into:

Page number (*p*) – used as an index into a *page table* which contains base address of each page in physical memory

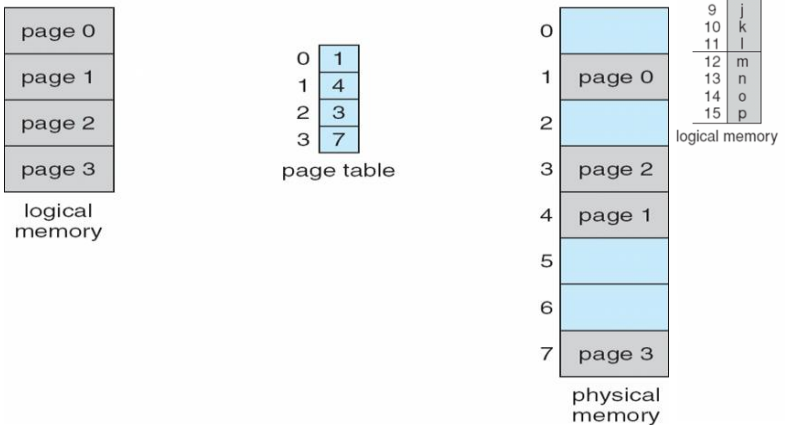
Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit. For given logical address space 2^m and page size 2^n



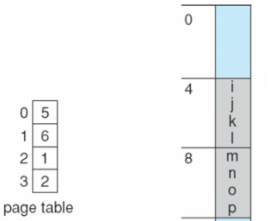
Paging Hardware

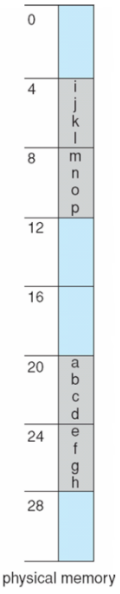


Paging Model of Logical and Physical Memory



Paging Example





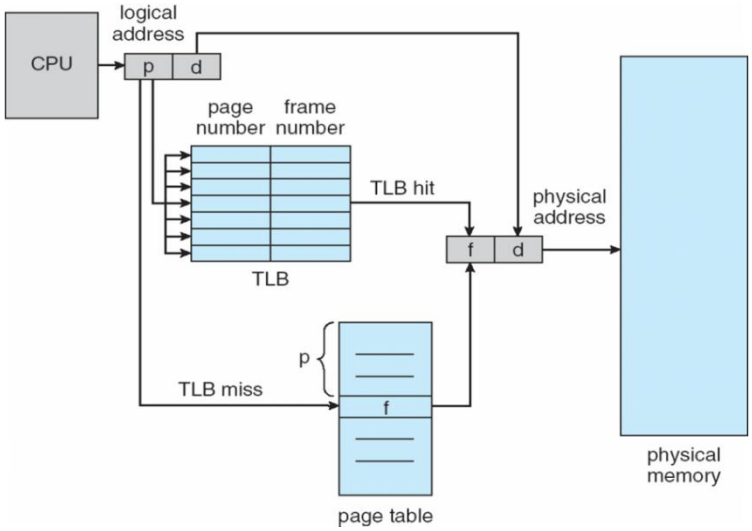
Implementation of Page Table:

Page table is kept in main memory and usually consist of:

- Page-table base register (PTBR) points to the page table
- Page-table length register (PRLR) indicates size of the page table

In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction. The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs). Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Paging Hardware with TLB



Shared Pages

1- Shared code:

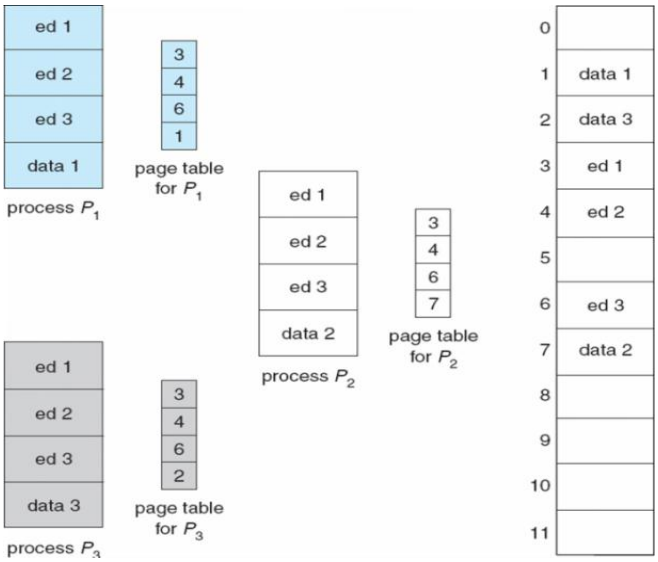
One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems). Shared code must appear in same location in the logical address space of all processes

2- 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



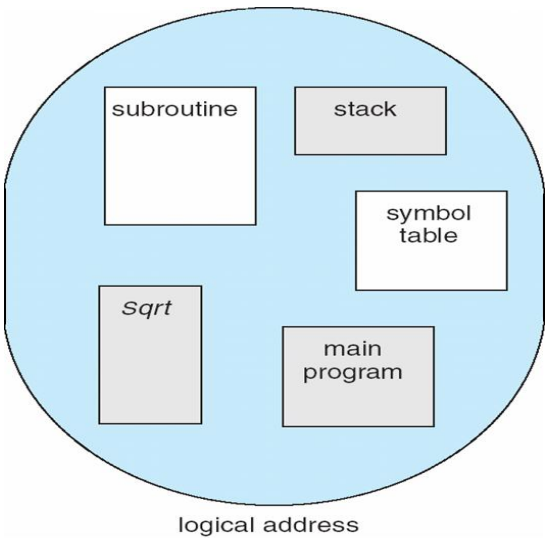
Structure of the Page Table

Hierarchical Paging, Hashed Page Tables, Inverted Page Tables

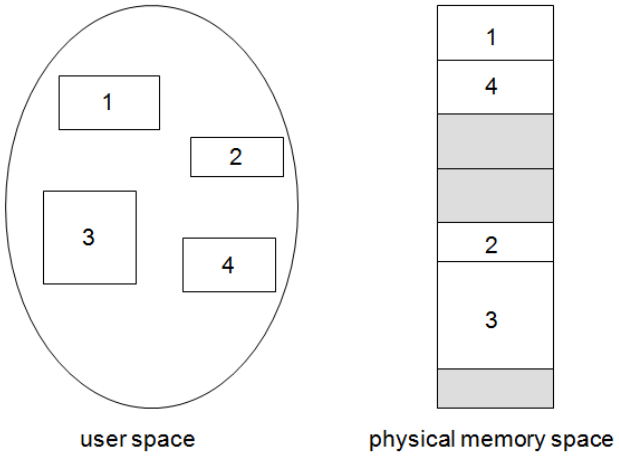
Segmentation

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

User's View of a Program



Logical View of Segmentation



Segmentation Architecture

Logical address consists of a two tuple: <segment-number, offset>,

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

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

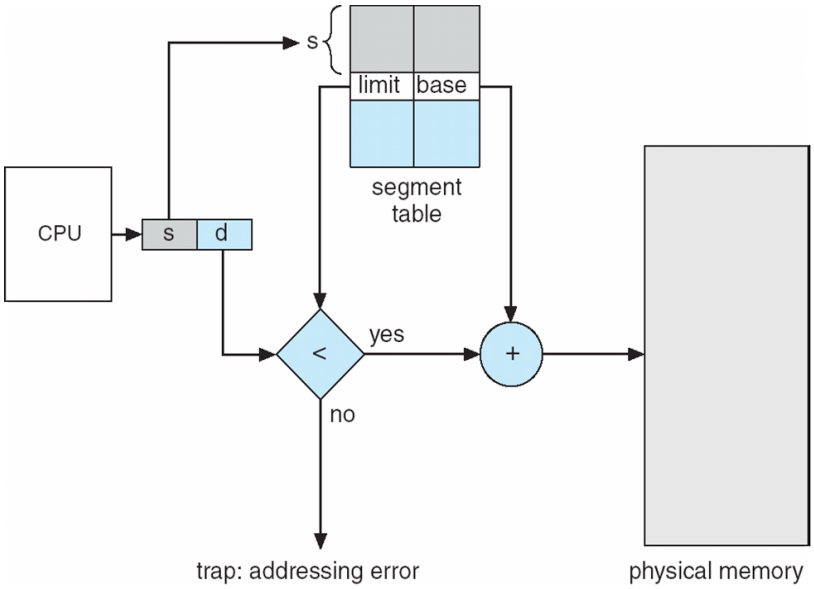
limit – specifies the length of the segment

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

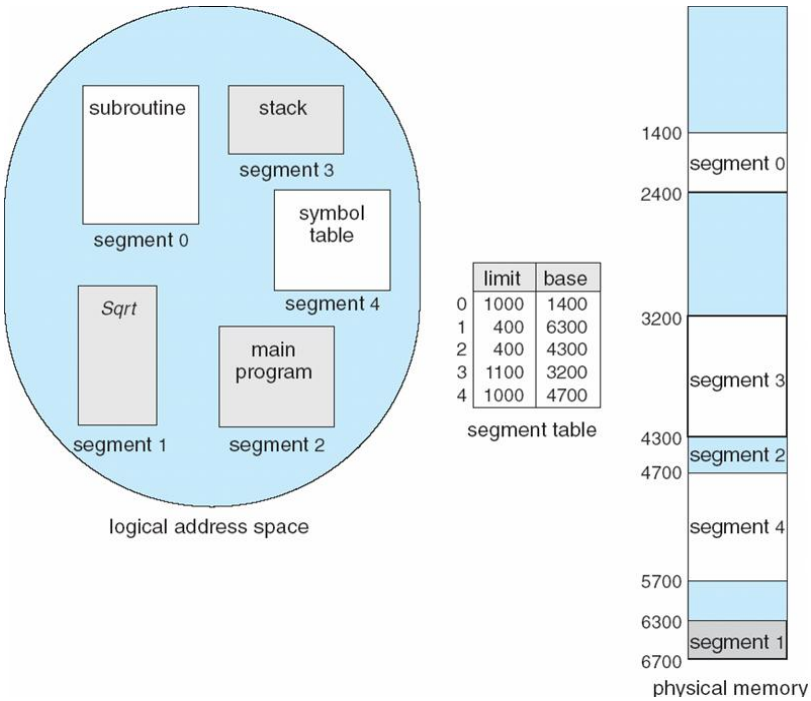
Segment-table length register (STLR) indicates number of segments used by a program;

(segment number *s* is legal if *s*< STLR)

Segmentation Hardware



Example of Segmentation



Example: The Intel Pentium

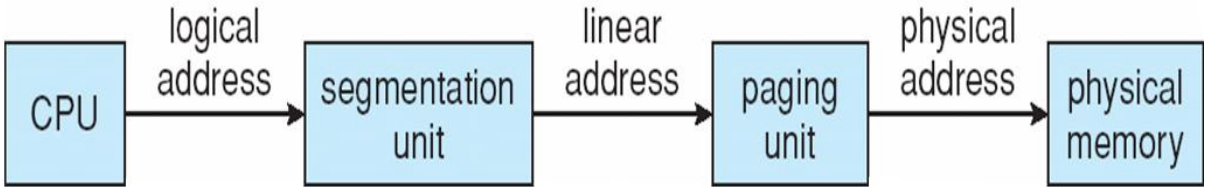
Supports both segmentation and segmentation with paging

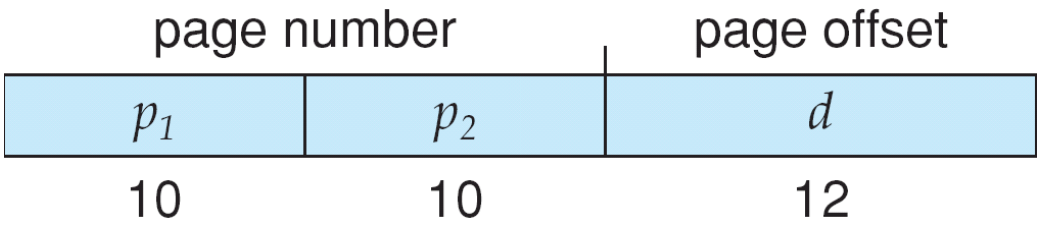
CPU generates logical address Given to segmentation unit, Which produces linear addresses

Linear address given to paging unit, Which generates physical address in main memory

Paging units form equivalent of MMU

Logical to Physical Address Translation in Pentium





Chapter 9: Virtual Memory

Virtual memory: is the process of separation of user logical memory from physical memory. Only part of the program needs to be in memory for execution. Logical address space can therefore be much larger than physical address space. Allows address spaces to be shared by several processes. Allows for more efficient process creation.

Virtual memory can be implemented via:

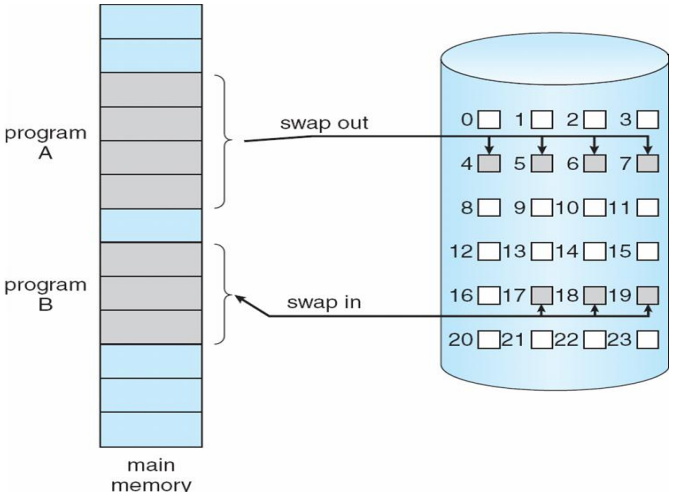
1- Demand paging

2- Demand segmentation

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

Transfer of a Paged Memory to Contiguous Disk Space

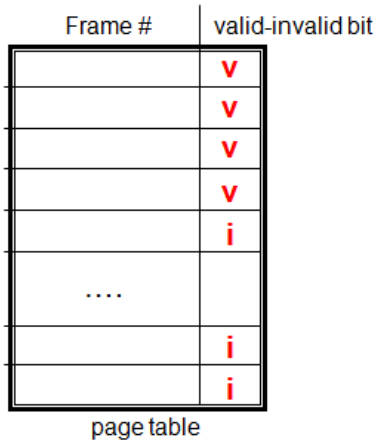


Valid-Invalid Bit

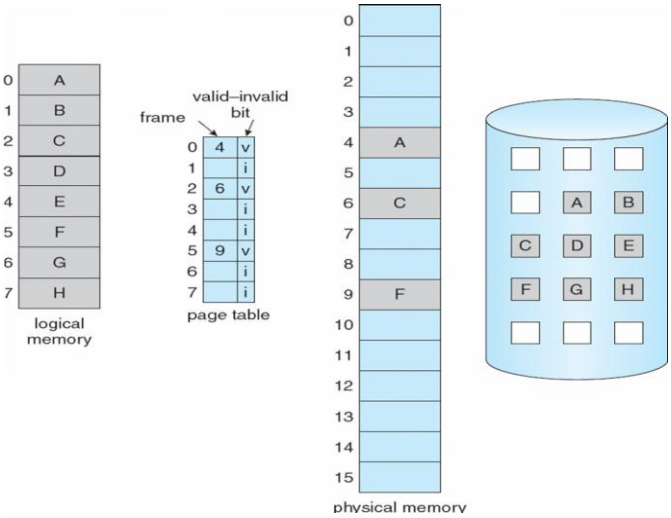
With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, i ⇒ not-in-memory). Initially valid–invalid bit is set to i on all entries

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

Example of a page table snapshot:



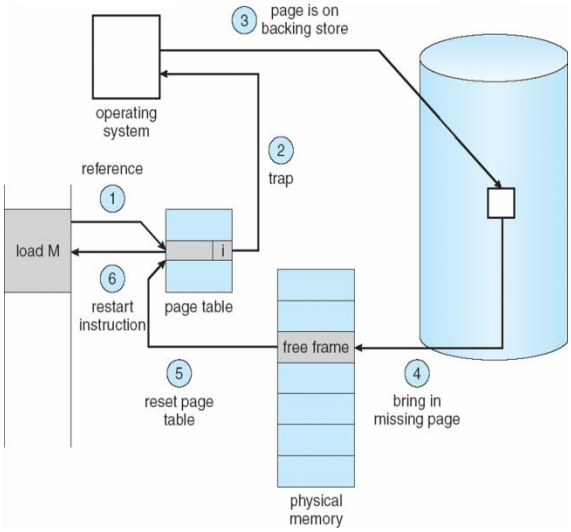
Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: page fault. Operating system looks at another table to decide:
- Invalid reference ⇒ abort
- Just not in memory
- Get empty frame
- Swap page into frame
- Reset tables
- Set validation bit = v
- Restart the instruction that caused the page fault
- Restart instruction→ block move
- auto increment/decrement location

Steps in Handling a Page Fault



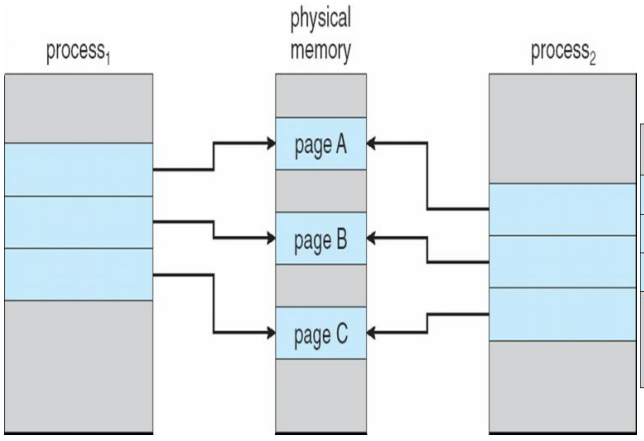
Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files.

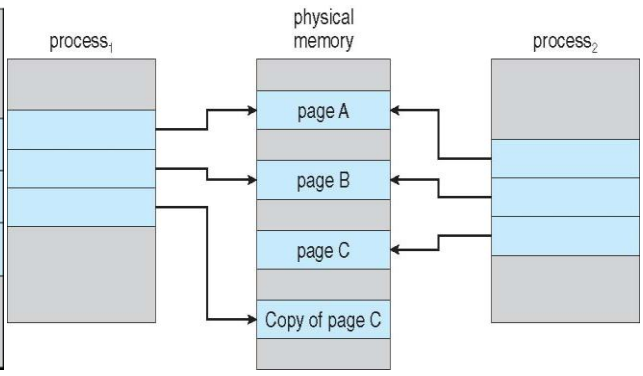
Copy-on-Write

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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



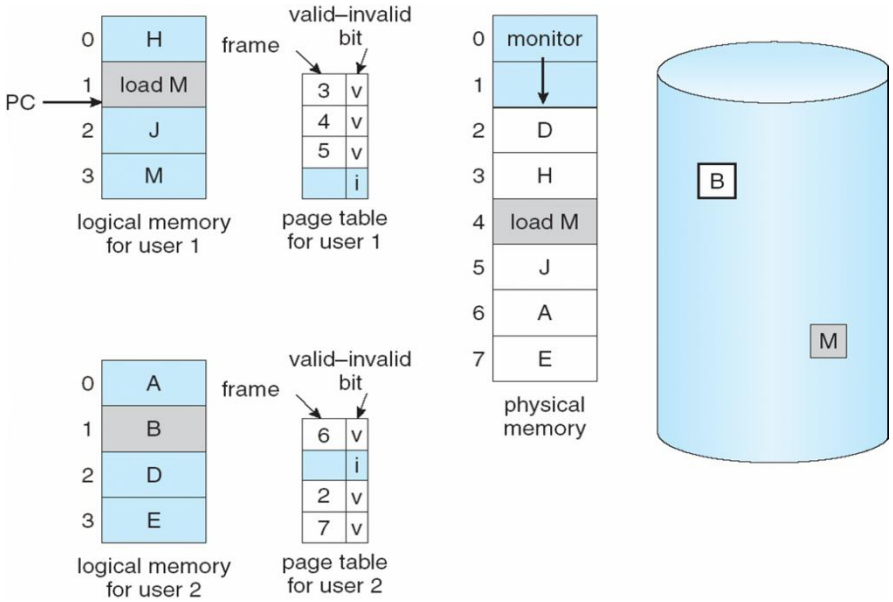
What happens if there is no free frame?

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

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

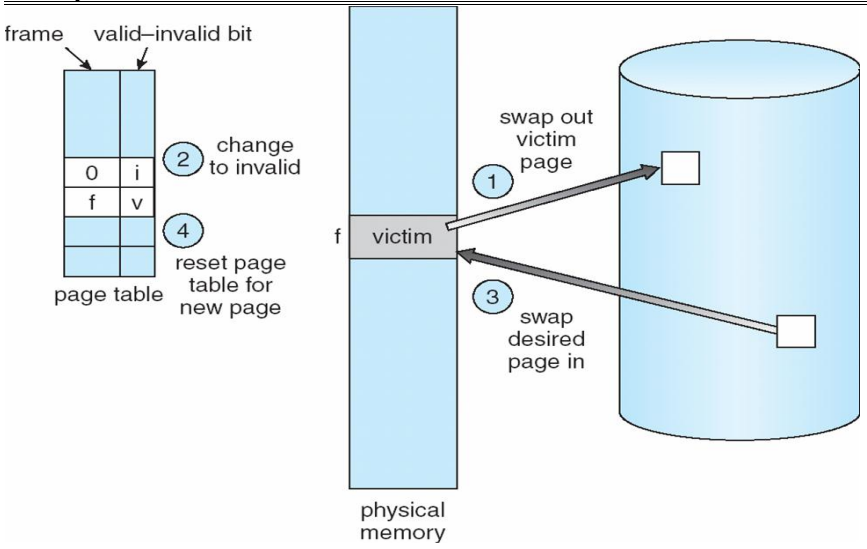
Need For Page Replacement



Basic Page Replacement

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

Page Replacement



Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First-In-First-Out (FIFO) Algorithm

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

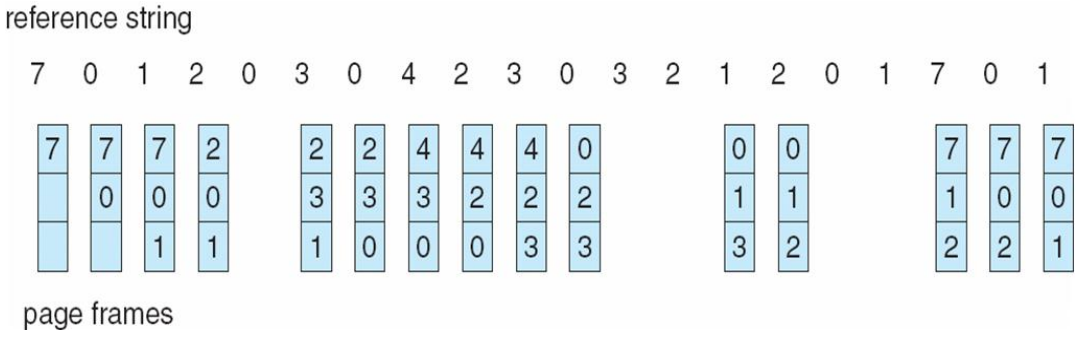
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

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

- Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



Optimal Algorithm

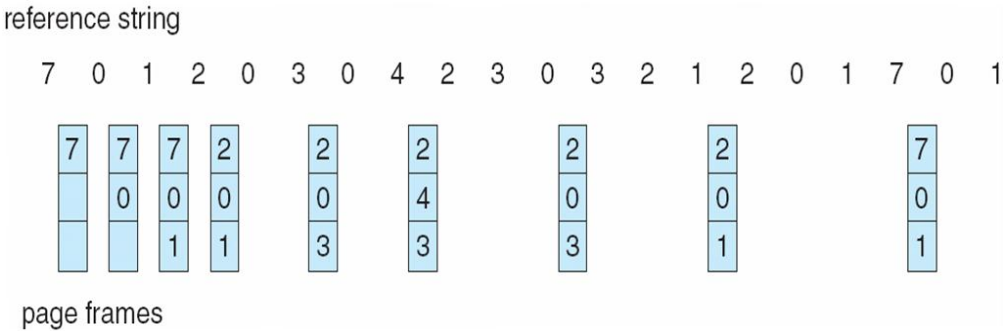
- Replace page that will not be used for longest period of time
- 4 frames example

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

1	4
2	6 page faults
3	
4	5

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



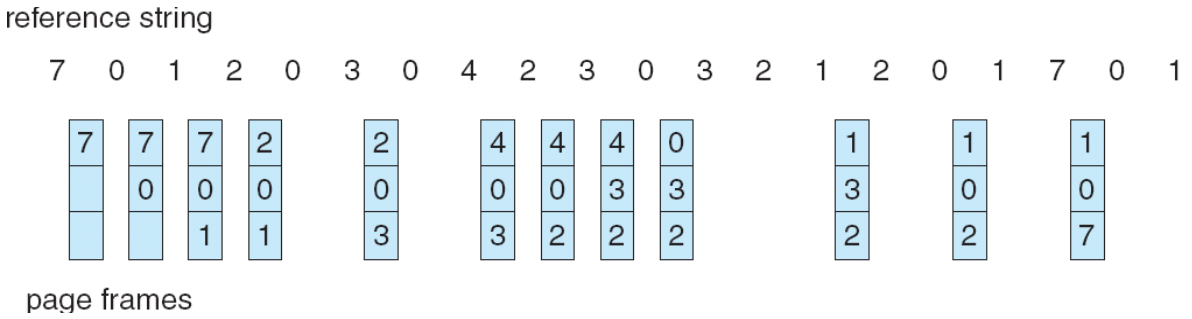
Least Recently Used (LRU) Algorithm

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

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

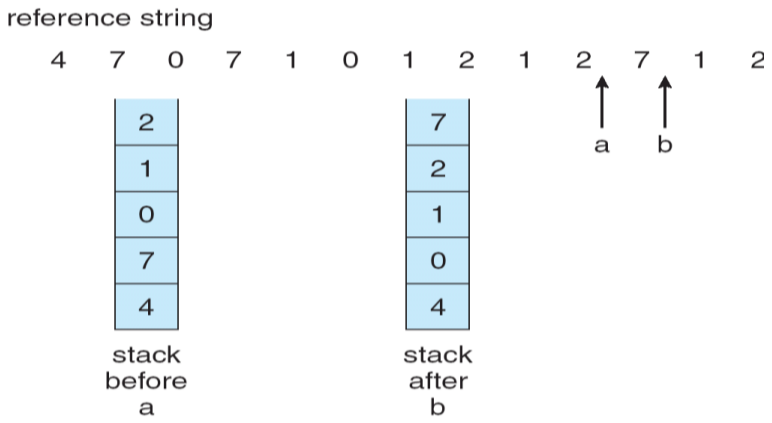
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement



- Stack implementation – keep a stack of page numbers in a double link form:
- Page referenced:
- move it to the top
- requires 6 pointers to be changed, No search for replacement

Use of A Stack to Record The Most Recent Page References



Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
- instruction is 6 bytes, might span 2 pages
- 2 pages to handle *from*
- 2 pages to handle *to*
- Two major allocation schemes: (fixed allocation, priority allocation)

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

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

Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another
- Local replacement – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
- low CPU utilization
- operating system thinks that it needs to increase the degree of multiprogramming
- another process added to the system
- Thrashing ≡ a process is busy swapping pages in and out

Other Issues – Page Size

- Page size selection must take into consideration:

1- fragmentation
2- table size
3- I/O overhead
4- locality

Other Issues – TLB Reach

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

Chapter 10: File-System Interface

File Concept: a file is a contiguous logical address space of many types as Data, numeric, character, binary, Program. It provides the mechanism for on line storage of data and programs of the OS and the users.

File: is a collection of related information recorded on a secondary storage and it is the smallest allotment logical secondary storage from the user point of view. File Structure can be None -sequence of words, bytes, Simple record structure, Lines, Fixed length, Variable length, Complex Structures, Formatted document, or Relocatable load file.

File Attributes

- **Name**–only information kept in human-readable form.
- **Identifier**–unique tag (number) identifies file within file system
- **Type**–needed for systems that support different types
- **Location**–pointer to file location on device
- **Size**–current file size
- **Protection**–controls who can do reading, writing, executing
- **Time, date, and user identification**–data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **Delete**
- **Truncate**: reset file length to zero with fixing all the other attributes.
- *Open(Fi)*–search the directory structure on disk for entry *Fi*, and move the content of entry to memory
- *Close (Fi)*–move the content of entry *Fi* in memory to directory structure on disk

Open Files: Several pieces of data are needed to manage open files:

- File pointer: pointer to last read/write location, per process that has the file open

- File-open count: counter of number of times a file is open –to allow removal of data from open-file table when last processes closes it
- Disk location of the file: cache of data access information
- Access rights: per-process access mode information

Open File Locking: Provided by some operating systems and file systems

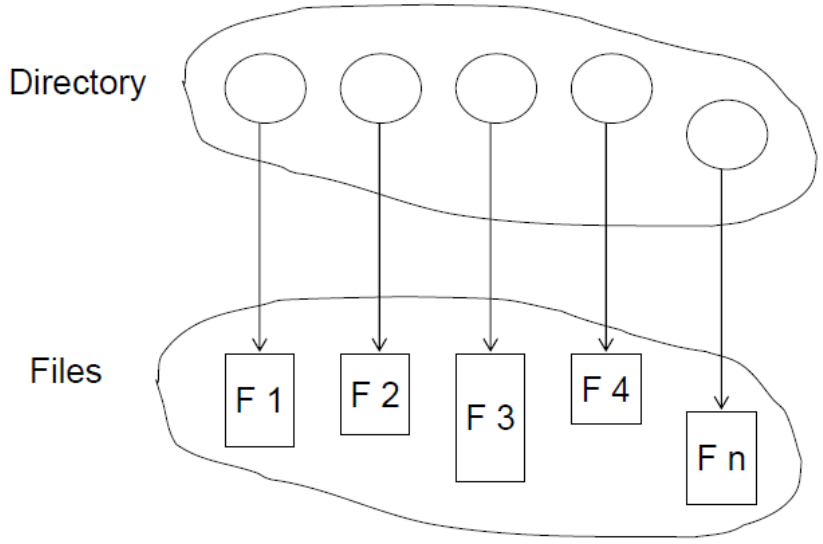
- Mediates access to a file
- Mandatory or advisory:
- **Mandatory**–access is denied depending on locks held and requested
- **Advisory**–processes can find status of locks and decide what to do

File Types –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

Directory Structure

- A collection of nodes containing information about all files

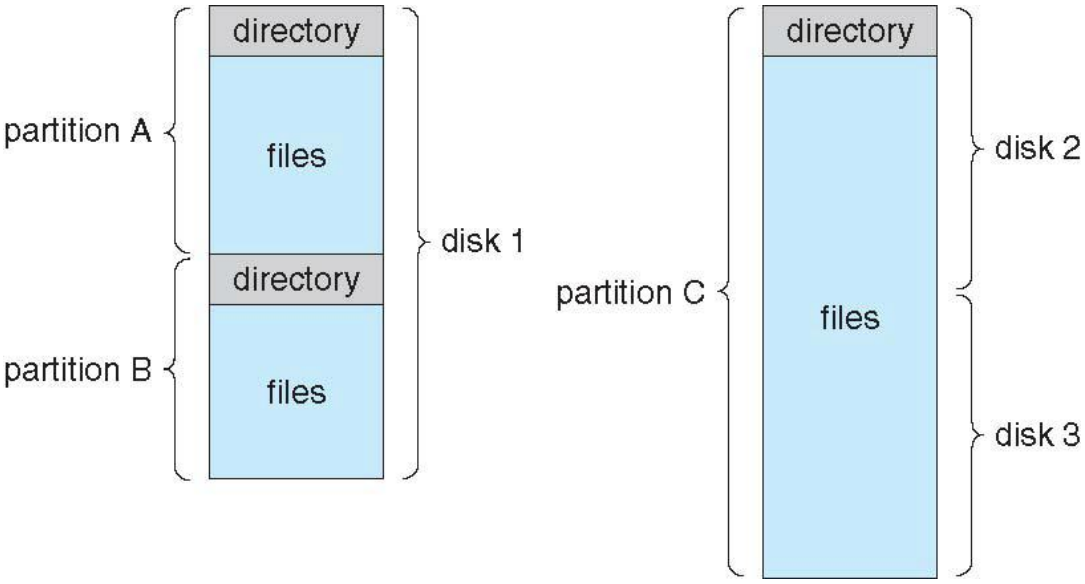


Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

Disk Structure: Disk can be subdivided into **partitions**

- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw**–without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**. As well as **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

A Typical File-system Organization



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

File Sharing –Multiple Users

User IDs identify users, allowing permissions and protections to be per-user

Group IDs allow users to be in groups, permitting group access rights

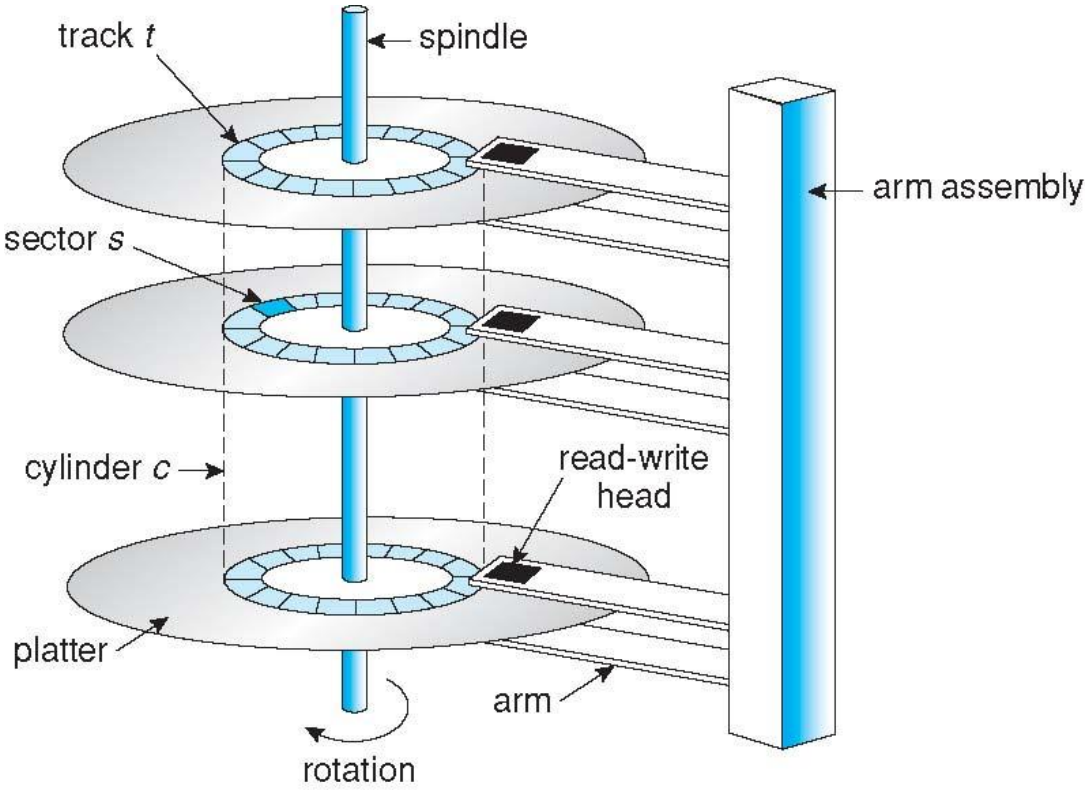
File Sharing –Remote File Systems

Uses networking to allow file system access between systems can be Manually via programs like FTP or Automatically, seamlessly using **distributed file systems** or Semi automatically via the **world wide web** (WWW).

- **Client-server** model allows clients to mount remote file systems from servers. Server can serve multiple clients. Client and user-on-client identification is insecure or complicated
- **NFS** is standard UNIX client-server file sharing protocol
- **CIFS** is standard Windows protocol
- Standard operating system file calls are translated into remote calls
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

Chapter 12: Mass Storage Systems:

- **Magnetic disks** provide bulk of secondary storage of modern computers
- Drives rotate at 60 to 200 times per second
- **Transfer rate** is rate at which data flow between drive and computer
- **Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
- **Head crash** results from disk head making contact with the disk surface, that's bad
- Disks can be removable
- Drive attached to computer via **I/O bus**
- Busses vary, including **EIDE**, **ATA**, **SATA**, **USB**, **Fiber Channel**, **SCSI**
- **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array



Disk Scheduling

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

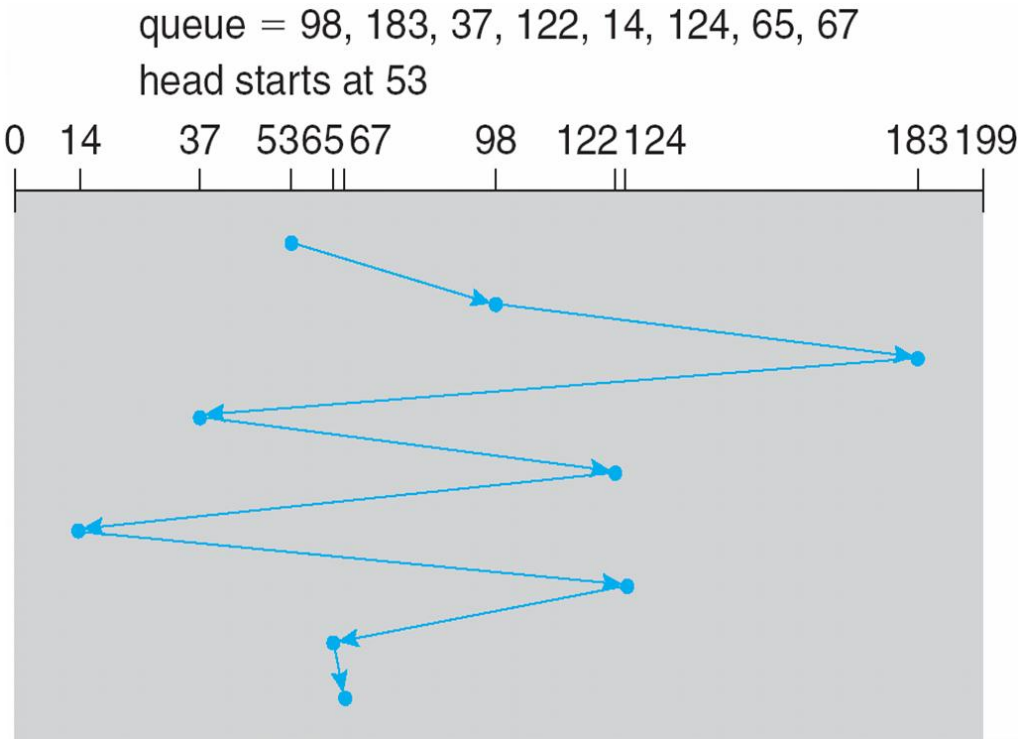
1- **Seek time** is the time for the disk are to move the heads to the cylinder containing the desired sector

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

The aim of the disk scheduling algorithms is to Minimize seek time. Seek time ≈ seek distance

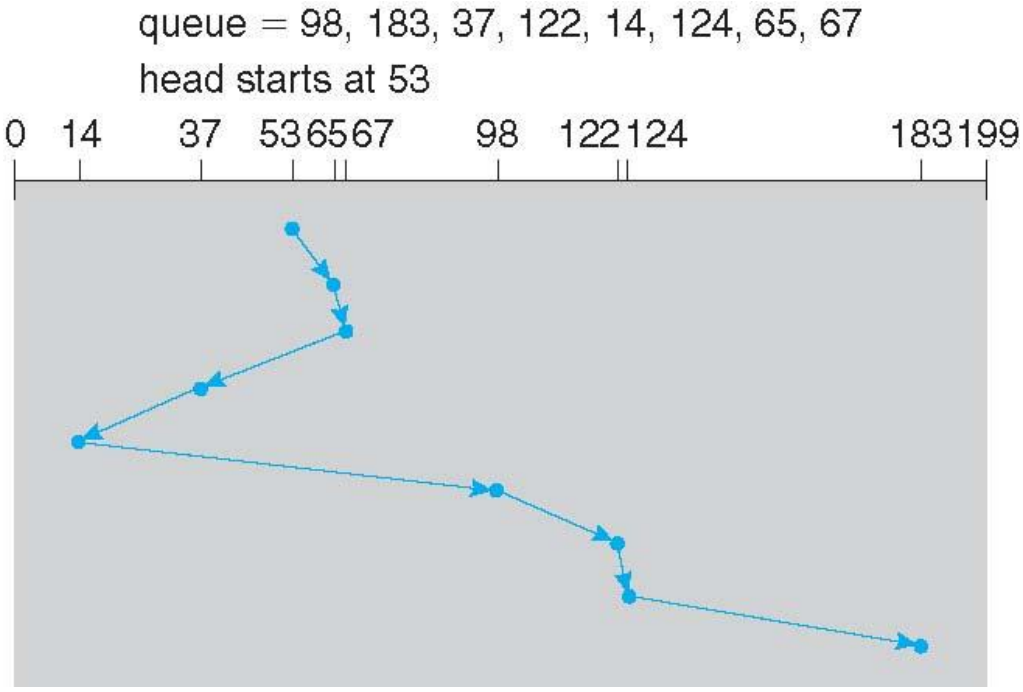
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
- Several algorithms exist to schedule the servicing of disk I/O requests, we illustrate them with a request queue (0-199): 98, 183, 37, 122, 14, 124, 65, 67
- Head pointer currently on : 53

1- <u>First come first served (FCFS)</u>: first request served first and so on:

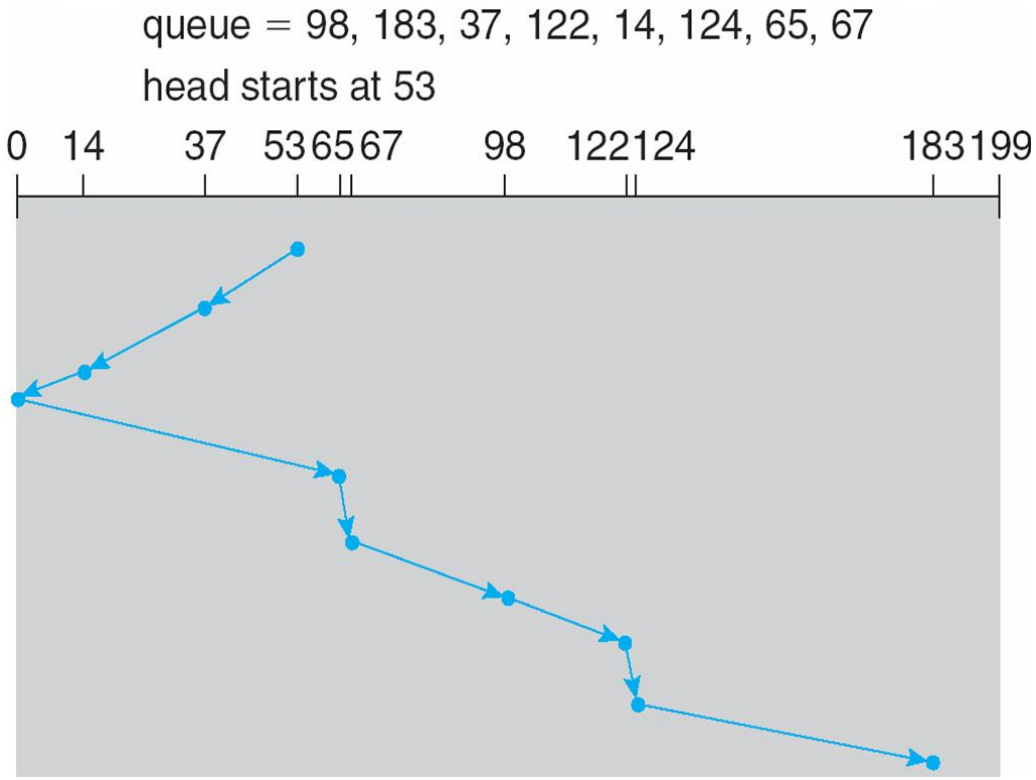


2- <u>Shortest Seek Time First (SSTF):</u>

- Selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders

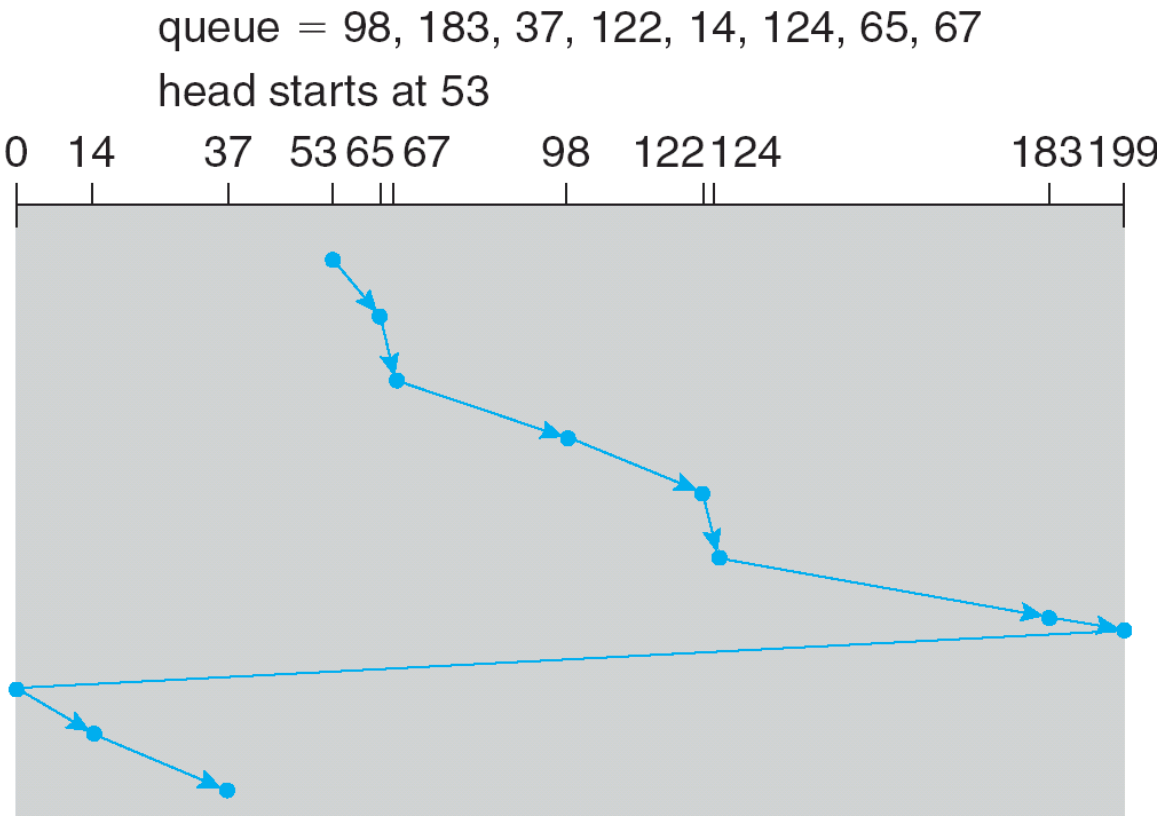


3- **SCAN algorithm:** The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues. **SCAN algorithm** Sometimes called the **elevator algorithm**. Illustration shows total head movement of 208 cylinders

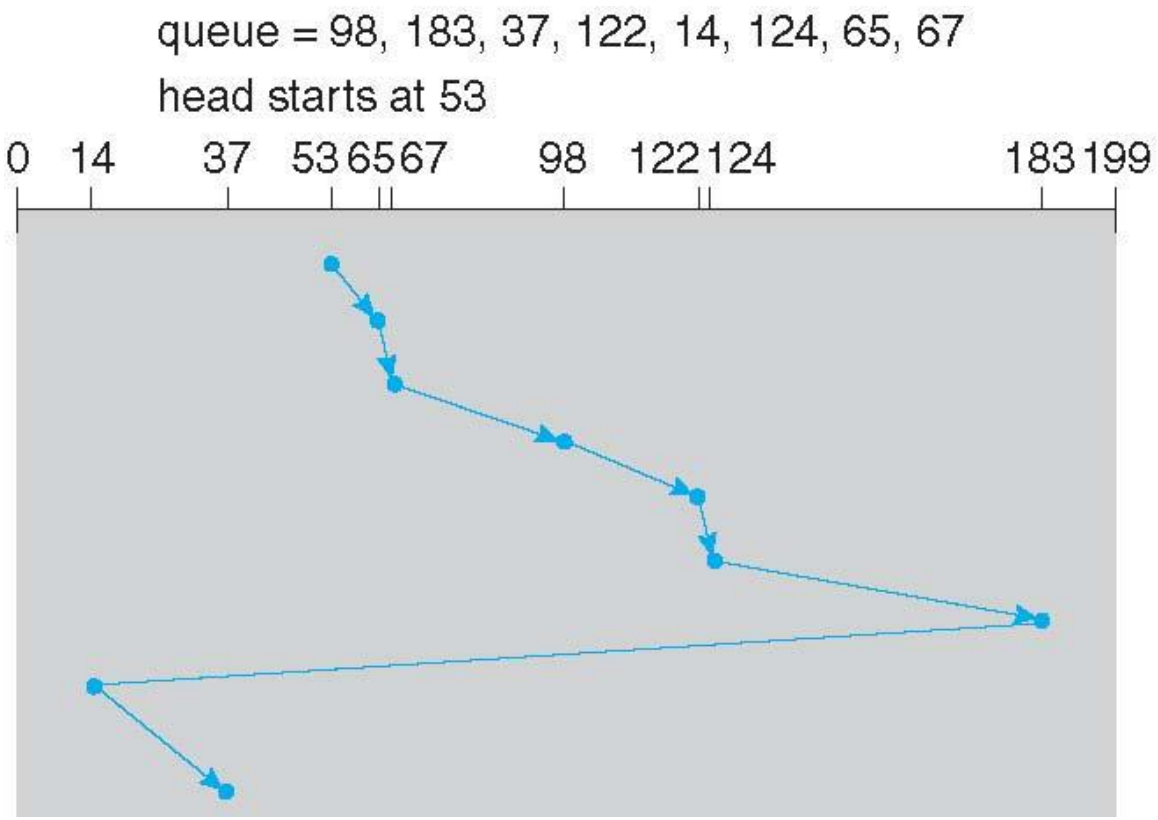


4- **Circular SCAN Algorithm**: Provides a more uniform wait time than SCAN. The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk,

without servicing any requests on the return trip. Treats the cylinders as a circular list that wraps around from the last cylinder to the first one



5- **Circular LOOK Algorithm:** Version of C-SCAN. Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk



Chapter 14 Protection

Protection is the mechanism of controlling the access of programs, processes, or users to the resources defined by the computer system. This mechanism must provide a means for specifying the controls to be imposed and the means for enforcement.

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

Domain Structure

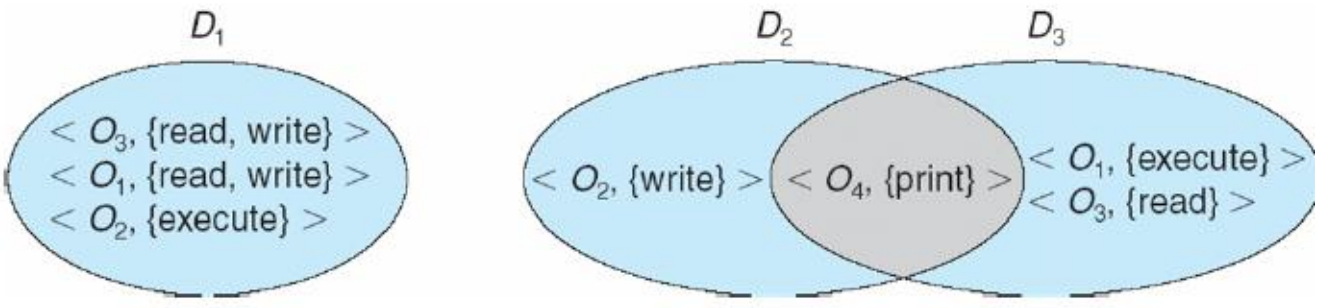
Each process work within a protection domain that specify the resources that process can access and each domain defines a set of objects and the types of operations that can be invoked on these objects.

The ability to execute an operation to an object is defined as:

Access-right = <object-name, rights-set>

where rights-set is a subset of all valid operations that can be performed on the object.

And Domain = set of access-rights.



Access matrix:

Our model of protection can be viewed abstractly as a matrix called access matrix where the rows of the matrix are the domains and the columns representing the objects. Each entry in the matrix represent a set of access rights.

*Access(i, j)*is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

The use of the access matrix:

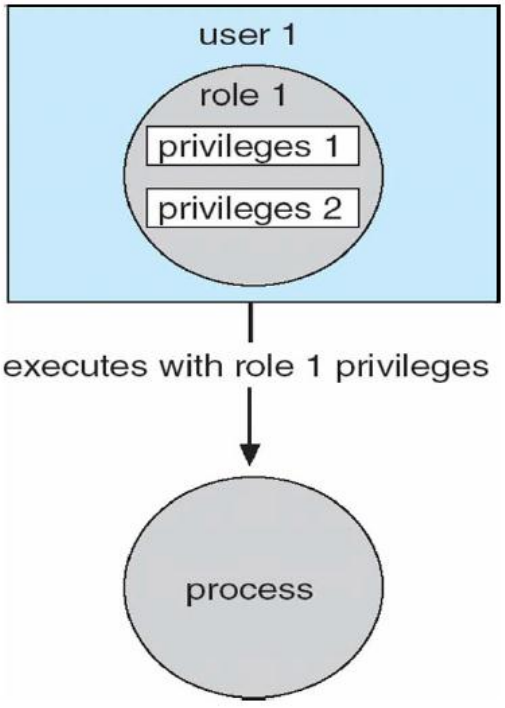
If a process in Domain Di tries to do "op" on object Oj, then "op" must be in the access matrix

Also this can be expanded to dynamic protection where:

- there are Operations to add, delete access rights
- Special access rights:
- owner of Oi
- copy op from Oi to Oj
- control –Di can modify Dj access rights
- transfer –switch from domain Di to Dj

Access Control

As we see the access control of a file within a file system can give the ability to use of deny using a specific file. The same thing can be applied on a non-file resources such as in the Solaris 10 operating system 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.

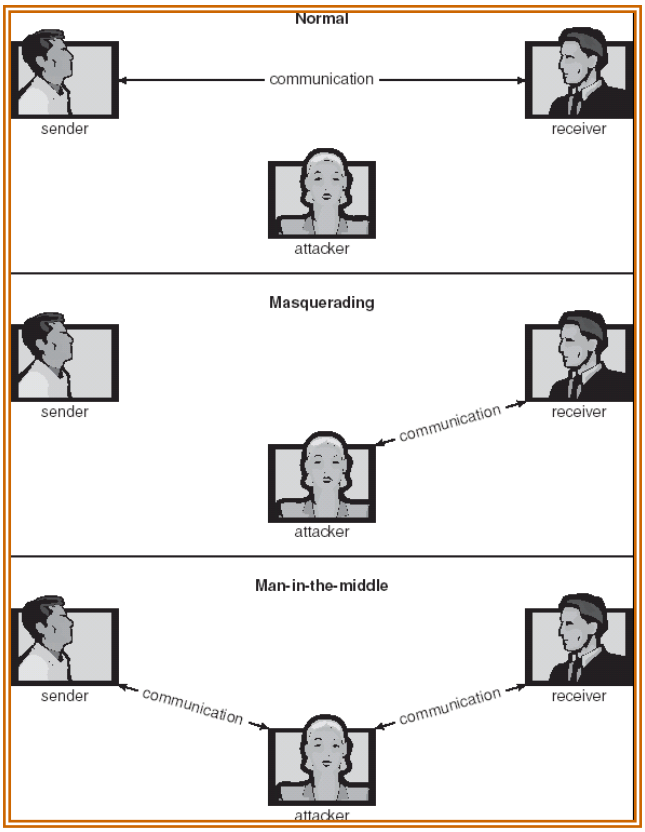


Chapter 15: Security

Protection is strictly an internal problem whereas the security on the other hand requires not only the protection of each computer system parts from one another but also must take the external environment within which the system operates.

Intruders (crackers) attempt to breach security. **Threat** is potential security violation. **Attack** is attempt to breach security. Attack can be accidental or malicious. Easier to protect against accidental than malicious misuse.

Standard Security attacks:



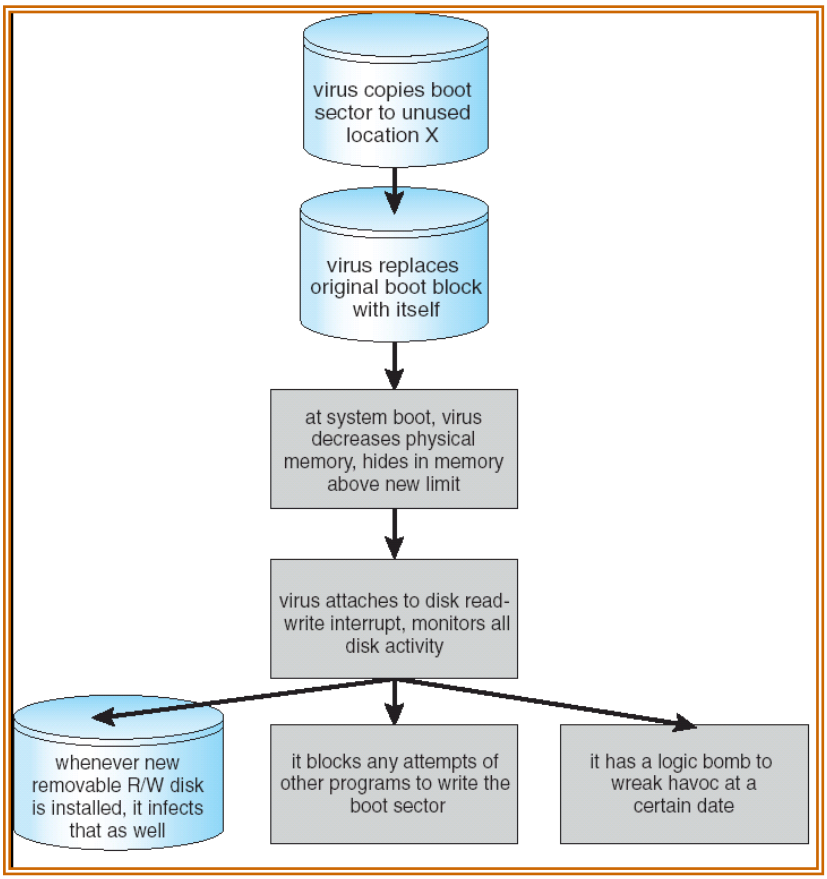
Security measure levels: Security must occur at four levels to be effective Physical, Human, Operating System, and Network. Security is as weak as the weakest chain.

Program Threats:

- Trojan Horse: is a Code segment that misuses its environment. Exploits mechanisms for allowing programs written by users to be executed by other users. **Spyware, pop-up browser windows, covert channels**
- Trap Door: Specific user identifier or password that circumvents normal security procedures. Could be included in a compiler

- Logic Bomb: is a Program that initiates a security incident under certain circumstances
- Stack and Buffer Overflow: Exploits a bug in a program (overflow either the stack or memory buffers).

Boot sector Computer Virus



System and Network Threats:

- Worms – use **spawn** mechanism; standalone program
- Internet worm: Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs. **Grappling hook** program uploaded main worm program
- Port scanning: Automated attempt to connect to a range of ports on one or a range of IP addresses.
- Denial of Service: Overload the targeted computer preventing it from doing any useful work. Distributed denial-of-service (**DDOS**) come from multiple sites at once.