# ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Course : Operating Systems

Course Code : **24FMCA21T**

Branch : **MCA**

Prepared by : **J.HARIKRISHNA**

Designation : **Assistant Professor**

Department : **MCA**

# ANNAMACHARYA UNIVERSITY, RAJAMPET

**(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016**
**RAJAMPET, Annamayya District, AP, INDIA**

**Title of the Course : Operating Systems**
**Category : PC**
**Course Code : 24FMCA21T**
**Branch : MCA**
**Semester : II Semester**

| Lecture Hours | Tutorial Hours | Practice Hours | Credits |
|---|---|---|---|
| 3 | 0 | 0 | 3 |

## COURSE OBJECTIVES:

- Identify the concepts, principles and services of operating system.
- Understand the operating system functionalities managing with hardware.
- Analyze the structure and design decisions involved in the implementation of an operating system.
- Evaluate different algorithms related to different operating system components.
- Explore various operating system utility commands to manage operating system.

## COURSE OUTCOMES:

**The Student will be able to**

1. Summarize the fundamentals and its components of operating system.
2. Apply scheduling algorithms and techniques.
3. Apply the algorithms to handle the dead lock operations.
4. Describe memory management and file management techniques.
5. Comprehend various protection and security issues.

**UNIT I**

**OPERATING SYSTEM INTRODUCTION:** Operating System Definition, Evolution of Operating Systems- Simple, Batch, Multi Programmed, Time-Shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System Calls, Types of System Calls.

**UNIT II**

**PROCESS AND CPU SCHEDULING:** Process Concepts- The Process, Process State, Process Control Block, Process Scheduling- Scheduling Queues, Schedulers, Context Switch, Preemptive Scheduling, Dispatcher, Scheduling Criteria, Scheduling Algorithms

**PROCESS COORDINATION:** Process Synchronization, the Critical-Section Problem, Semaphores, Classic Problems of Synchronization.

**UNIT III**

**DEADLOCKS:** System model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection and Recovery from Deadlock.

**UNIT IV**

**MEMORYMANAGEMENT:** Swapping, Contiguous Memory Allocation, Paging, Structure of Page Table, Segmentation, Demand Paging, Performance of Demand Paging, Page Replacement Algorithms.
**FILE SYSTEM INTERFACE:** The Concept of a File, Access methods, Directory Structure, Allocation methods, Disk Scheduling algorithms.

**UNIT V**

**PROTECTION:** System Protection- Goals of Protection, Principles of Protection, Access Matrix
**SECURITY: -** The Security Problem, Program Threats, System and Network Threats, User Authentication, Firewalling.

**TEXT BOOKS:**


1. Abraham Silberchatz, Peter B. Galvin. Operating System Principles. Wiley, Global Edition, 2019.


**REFERENCE BOOKS:**


1. Andrew S Tanenbaum. Modern Operating Systems. Pearson/ PHI, 3rd Ed, 2009.
2. R. Elmasri, A.G.Carrick and D.Levine. Operating Systems. MGH 3rd Edition 2011.
3. A.S. Godbole. Operating Systems. TMH, 2ndEd 2009.
4. W. Stallings. Operating Systems-Internal and Design Principles, Pearson Education, 6th Ed.


**CO-PO MAPPING:**

| Course Outcomes | Foundation Knowledge | Problem Analysis | Development of Solutions | Modern Tool Usage | Individual and Teamwork | Project Management and Finance | Ethics | Life-long Learning |
|---|---|---|---|---|---|---|---|---|
| 24FMCA21T.1 | 2 | 2 | 1 | - | - | - | - | - |
| 24FMCA21T.2 | 3 | 2 | 1 | - | - | - | - | - |
| 24FMCA21T.3 | 3 | 2 | 1 | - | - | - | - | - |
| 24FMCA21T.4 | 2 | 2 | 1 | - | - | - | - | - |
| 24FMCA21T.5 | 2 | 2 | 1 | - | - | - | - | - |

## UNIT-I

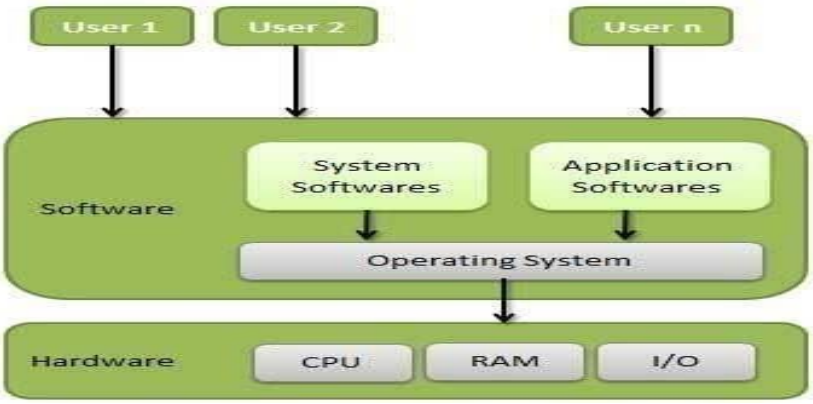
**OPERATING SYSTEM INTRODUCTION:** Operating System Definition, Evolution of Operating Systems- Simple, Batch, Multi Programmed, Time-Shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System Calls, Types of System Calls.

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux, Windows, OS X, VMS, OS/400, AIX, z/OS, etc.

### Definition

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.




### Functions of operating system

Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

**Memory Management**

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management –

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

**Processor Management**

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

**Device Management**

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

**File Management**

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

**Other Important Activities**

Following are some of the important activities that an Operating System performs

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request fora service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobsand users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other software's and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

**Operating system services**

Here is a list of common services offered by an almost all operating systems:

- **User Interface**
- **Program Execution**
- **File system manipulation**
- **Input / output Operations**
- **Resource Allocation**
- **Error Detection**
- **Accounting**
- **Security and protection**

**User Interface of Operating System**

Usually Operating system comes in three forms or types. Depending on the interface their types have been further sub divided. These are:

- Command line interface
- Batch based interface
- Graphical User Interface

The command line interface (CLI) usually deals with using text commands and a technique for entering those commands. The batch interface (BI): commands and directives are used to manage those commands that are entered into files and those files get executed. Another type is the graphical user interface (GUI): which is a window system.

**Program Execution in Operating System**

The operating system must have the capability to load a program into memory and execute that program. Furthermore, the program must be able to end its execution, either normally or abnormally / forcefully.

**File System Manipulation in Operating System**

Programs need has to be read and then write them as files and directories. File handling portion of operating system also allows users to create and delete files by specific name along with extension, search for a given file and / or list file information. Some programs comprise of permissions management for allowing or denying access to files or directories based on file ownership.

**I/O operations in Operating System**

A program which is currently executing may require I/O, which may involve file or other I/O device. For efficiency and protection, users cannot directly govern the I/O devices. So, the OS provide a means to do I/O Input / Output operation which means read or write operation with any file.

**Resource Allocation of Operating System**

When multiple jobs running concurrently, resources must need to be allocated to each of them. Resources can be CPU cycles, main memory storage, file storage and I/O devices.

**Error Detection**

Errors may occur within CPU, memory hardware, I/O devices and in the user program. For each type of error, the OS takes adequate action for ensuring correct and consistent computing.

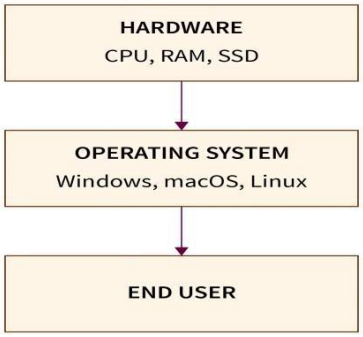
**Accounting**

This service of operating system keeps track of which users are using how much and what kinds of computer resources has been used for accounting or simply to accumulate usage statistics.

**Protection and Security**

Protection includes in ensuring all access to system resources in a controlled manner. For making a system secure, the user needs to authenticate him or her to the system before using (usually via login ID and password).

## Evolution of operating systems



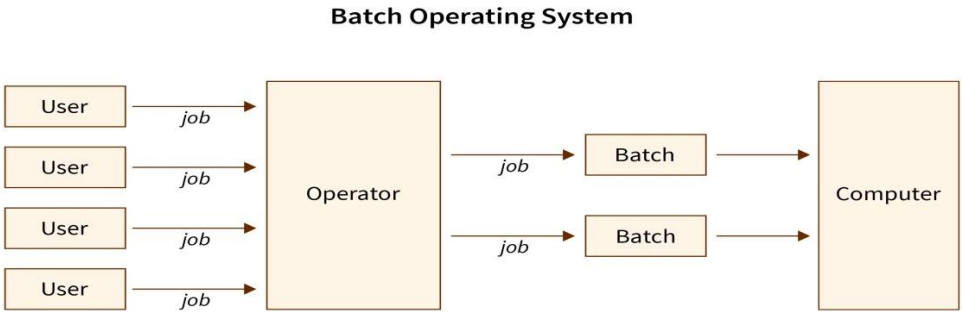

- **Simple Batch system**
- **Multiprogramming Batch Systems**
- **Multiprocessor Systems/parallel systems**
- **Time-sharing operating systems**
- **PC(Personal Computer)**
- **Distributed operating System**
- **Network operating System**
- **Real Time operating system**

### Batch operating system

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

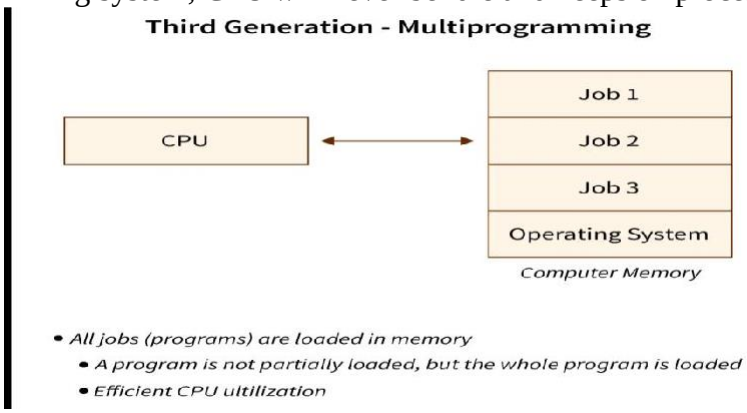
The problems with Batch Systems are as follows −

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.



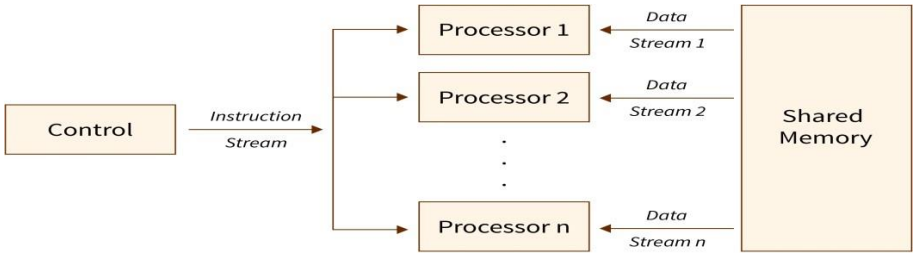

## Multiprogramming Systems

- In this the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job(CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of **CPU Scheduling**.
- In Non-multi programmed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.



## Multiprocessor Systems/parallel systems

A Multiprocessor system consists of several processors that share a common physical memory. Multiprocessor system provides higher computing power and speed. In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.



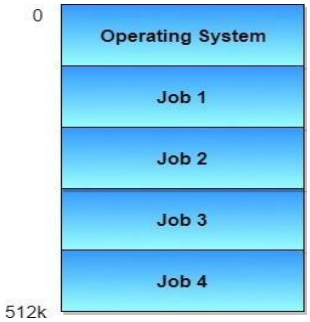

### Advantages of Multi-processor Systems

1. Enhanced performance
2. Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task.
3. If possible, system divides task into many subtasks and then these subtasks can be executed in parallel in different processors. Thereby speeding up the execution of single tasks.

## Time-sharing/multitasking operating systems

**Time Sharing Systems** are very similar to Multiprogramming batch systems. In fact time sharing systems are an extension of multiprogramming systems.

In Time sharing systems the prime focus is on **minimizing the response time**, while in multiprogramming the prime focus is to maximize the CPU usage.




Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

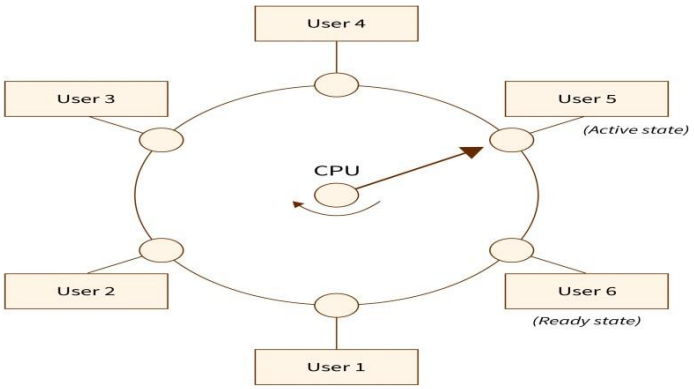
The main difference between Multi programmed Batch Systems and Time- Sharing Systems is that in case of Multi programmed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if **n** users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows −

- Provides the advantage of quick response.
- Reduces CPU idle time.




## PC(Personal Computer)

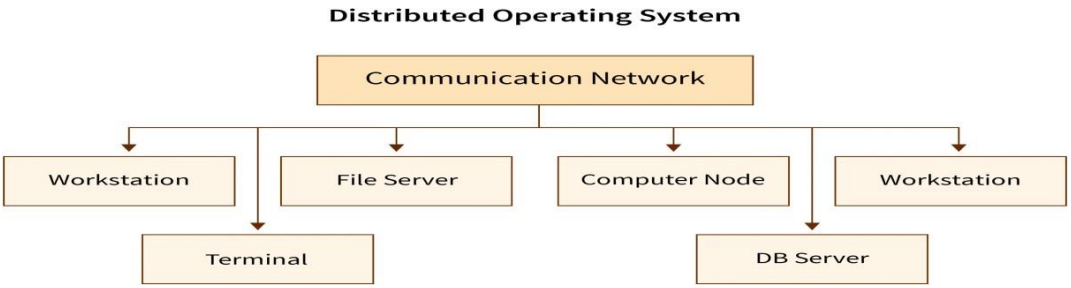
A PC can be defined as a small, relatively inexpensive computer designed for an individual user. PCs are based on the microprocessor technology that enables manufacturers to put an entire CPU on one chip. Businesses use personal computers for word processing, accounting, desktop publishing, and for running spreadsheet and database management applications. At home, the most popular use for personal computers is playing games and surfing the Internet.

Although personal computers are designed as single-user systems, these systems are normally linked together to form a network. In terms of power, now-a-days high-end models of the Macintosh and PC offer the same computing power and graphics capability as low-end workstations by Sun Microsystems, Hewlett- Packard, and Dell.

## Distributed operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers and so on.




The advantages of distributed systems are as follows −

- With resource sharing facility, a user at one site may be able to use there sources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

## Network operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows −

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows −

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

## Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

### Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

### Soft real-time systems

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc.

**System calls**

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks may need to be written using assembly-language instructions.

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used:

**writing a simple program to read data from one file and copy them to another file.**

The first input that the program will need is the names of the two files: the input file and the output file.

These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names of the two files.

In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files.

On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/0 system calls.

**Once the two file names are obtained, the program must open the input file and create the output file.** Each of these operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call).

If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (another system call).

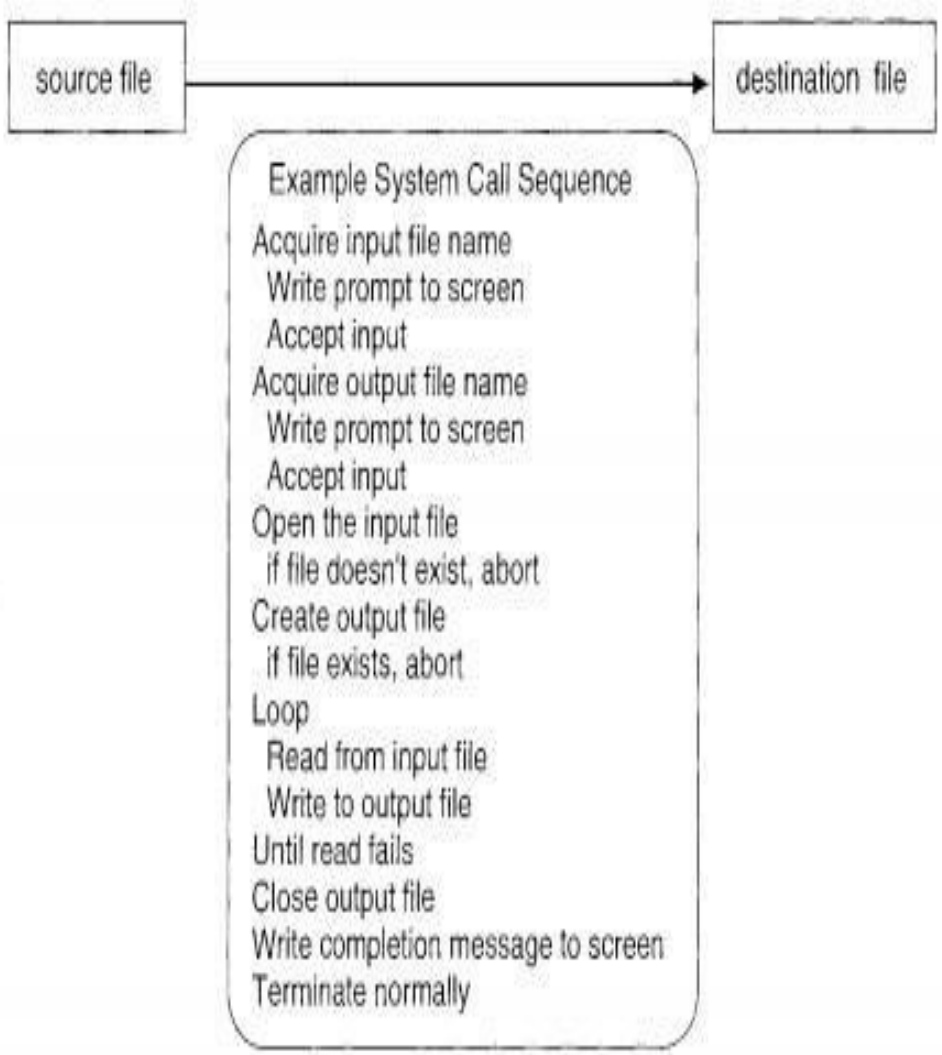
Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the termin.al) whether to replace the existing file or to abort the program.

Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions.

On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (no more disk space, printer out of paper, and so on).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). As we 56 Chapter 2 can see1 even simple programs may make heavy use of the operating system. Frequently/ systems execute thousands of system calls per second.

This system call sequence is shown in below Figure.



- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

**Process control**

A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.

**File Management**

We can, however, identify several common system calls dealing with files. Wefirst need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open it and to use it. We may also read, write, or reposition (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it.

**Device Management**

A process may need several resources to execute-main memory, disk drives, accessto files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first request the device, to ensure exclusive use of it. After we are finished with the device, we release it. Once the device has been requested, we can read, write, and (possibly) reposition the device, just as we can with files.

**Information Maintenance**

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time and date. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on. Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump memory.

**Communication calls**

These allow a program to communicate with other programs and devices, including inter-process communication (IPC) and networking. Examples include socket, bind, and listen.

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess() | fork() |
| | ExitProcess() | exit() |
| | WaitForSingleObject() | wait() |
| File Manipulation | CreateFile() | open() |
| | ReadFile() | read() |
| | WriteFile() | write() |
| | CloseHandle() | close() |
| Device Manipulation | SetConsoleMode() | ioctl() |
| | ReadConsole() | read() |
| | WriteConsole() | write() |
| Information Maintenance | GetCurrentProcessID() | getpid() |
| | SetTimer() | alarm() |
| | Sleep() | sleep() |
| Communication | CreatePipe() | pipe() |
| | CreateFileMapping() | shmget() |
| | MapViewOfFile() | mmap() |
| Protection | SetFileSecurity() | chmod() |
| | InitlializeSecurityDescriptor() | umask() |
| | SetSecurityDescriptorGroup() | chown() |

.

**There are several advantages to using system calls including:**

1. **Improved efficiency:** System calls provide a direct and efficient way for a program to access system resources and services, as they are implemented in the operating system kernel and bypass the overhead of library functions.
2. **Improved security:** System calls provide a way for the operating system to enforce security policies and prevent unauthorized access to system resources.
3. **Improved portability:** Programs that use system calls can be easily ported to other operating systems, as long as the target operating system supports the same system calls.

**However, there are also some potential drawbacks to using system calls including:**

1. **Overhead:** The use of system calls can add additional overhead to the execution of a program, as they require a context switch between user space and kernel space.
2. **Complexity:** The use of system calls can add an additional layer of complexity to the development process, as they require a thorough understanding of the operating system and its interfaces.
3. **Compatibility:** Different operating systems may have different sets of system calls and different interfaces, which can limit the portability of programs that use system calls.

**In terms of real-time applications, system calls are used in a wide variety of contexts including:**

1. **File I/O:** Programs use system calls to read and write files, including text files, binary files, and special files such as pipes and sockets.
2. **Process management:** Programs use system calls to create and control processes, including forking, execing and waiting for child processes to complete.
3. **Networking:** Programs use system calls to create and manage network connections, including socket creation, binding, listening, and accepting incoming connections.
4. **Memory management:** Programs use system calls to allocate and deallocate memory, as well as to control the mapping of virtual memory to physical memory.

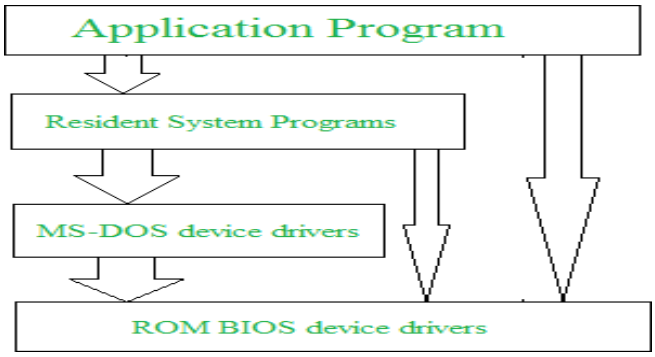
**Layered Structure or approach of OS**

Operating system can be implemented with the help of various structures. The structure of the OS depends mainly on how the various common components of the operating system are interconnected and melded into the kernel.

Depending on this we have following structures of the operating system:

**Simple structure:**

Such operating systems do not have well defined structure and are small, simple and limited systems. The interfaces and levels of functionality are not well separated. MS-DOS is an example of such operating system. In MS-DOS application programs are able to access the basic I/O routines. These types of operating system cause the entire system to crash if one of the user programs fails.
Diagram of the structure of MS-DOS is shown below.




**Advantages of Simple structure:**

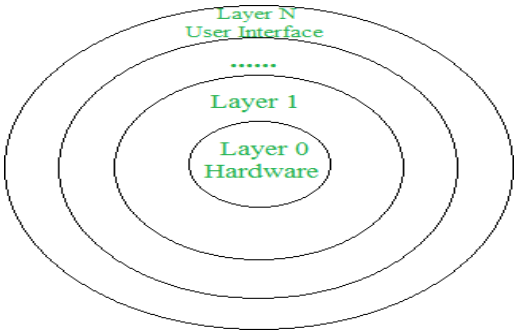
- It delivers better application performance because of the few interfaces between the application program and the hardware.
- Easy for kernel developers to develop such an operating system.

**Disadvantages of Simple structure:**

- The structure is very complicated as no clear boundaries exists between modules.
- It does not enforce data hiding in the operating system.

**Layered structure:**

An OS can be broken into pieces and retain much more control on system. In this structure the OS is broken into number of layers (levels). The bottom layer (layer 0) is the hardware and the topmost layer (layer N) is the user interface. These layers are so designed that each layer uses the functions of the lower level layers only. This simplifies the debugging process as if lower level layers are debugged and an error occurs during debugging then the error must be on that layer only as the lower level layers have already been debugged.
The main disadvantage of this structure is that at each layer, the data needs to be modified and passed on which adds overhead to the system. Moreover careful planning of the layers is necessary as a layer can use only lower level layers. UNIX is an example of this structure.




**Advantages of Layered structure:**
- Layering makes it easier to enhance the operating system as implementation of a layer can be changed easily without affecting the other layers.
- It is very easy to perform debugging and system verification.

**Disadvantages of Layered structure:**
- In this structure the application performance is degraded as compared to simple structure.
- It requires careful planning for designing the layers as higher layers use the functionalities of only the lower layers.

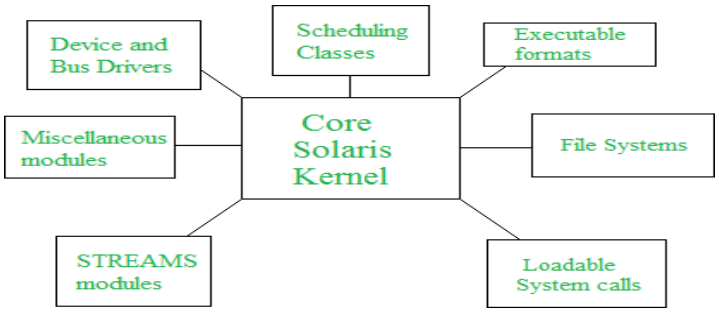
**Micro-kernel**
This structure designs the operating system by removing all non-essential components from the kernel and implementing them as system and user programs. This result in a smaller kernel called the micro-kernel.
Advantages of this structure are that all new services need to be added to user space and does not require the kernel to be modified. Thus it is more secure and reliable as if a service fails then rest of the operating system remains untouched. Mac OS is an example of this type of OS.

**Advantages of Micro-kernel structure:**
- It makes the operating system portable to various platforms.
- As microkernels are small so these can be tested effectively.

**Disadvantages of Micro-kernel structure:**
- Increased level of inter module communication degrades system performance.

**Modular structure or Approach:**

It is considered as the best approach for an OS. It involves designing of a modular kernel. The kernel has only set of core components and other services are added as dynamically loadable modules to the kernel either during run time or boot time. It resembles layered structure due to the fact that each kernel has defined and protected interfaces but it is more flexible than the layered structure as a module can call any other module.



## UNIT II

**PROCESS AND CPU SCHEDULING:** Process Concepts- The Process, Process State, Process Control Block, Process Scheduling- Scheduling Queues, Schedulers, Context Switch, Preemptive Scheduling, Dispatcher, Scheduling Criteria, Scheduling Algorithms
**PROCESS COORDINATION:** Process Synchronization, the Critical-Section Problem, Semaphores, Classic Problems of Synchronization.

**Process and CPU scheduling Process**

**concepts:**

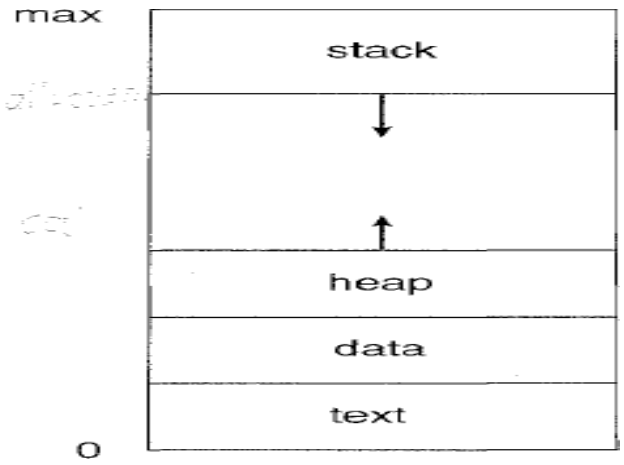
**The process**

A process is a program in execution. A process is more than the program code, which is sometimes known as the text section.

A process generally also includes the

- process stack, which contains temporary data (such as function parameters, return addresses, and local variables).
- A data section, which contains global variables.
- A process may also include a heap, which is memory that is dynamically allocated during process run time.

The structure of a process in memory is shown in below figure.




a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

**The process states**

As the process executes, it changes state. Each process may be in one of the following states:

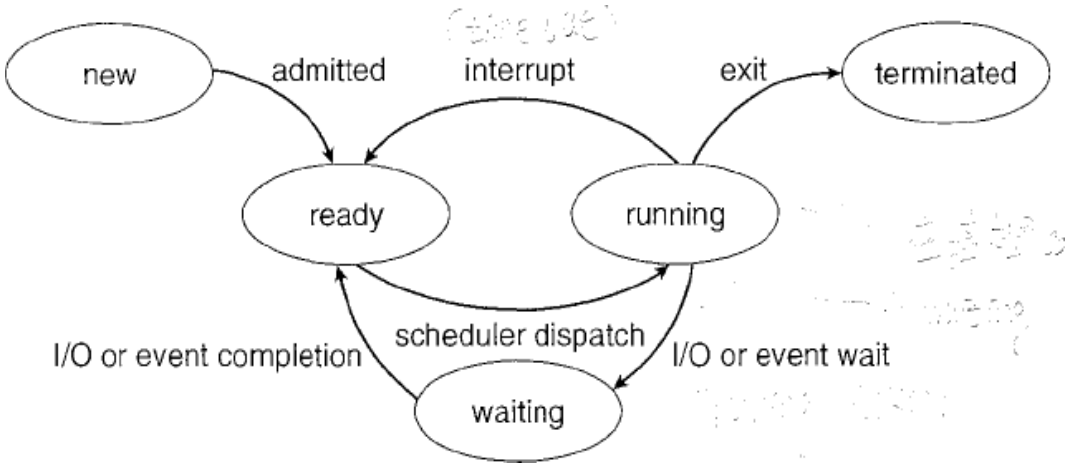
**New:** The process is being created.

**Ready.** The process is waiting to be assigned to a processor.

**Running:** Instructions are being executed.

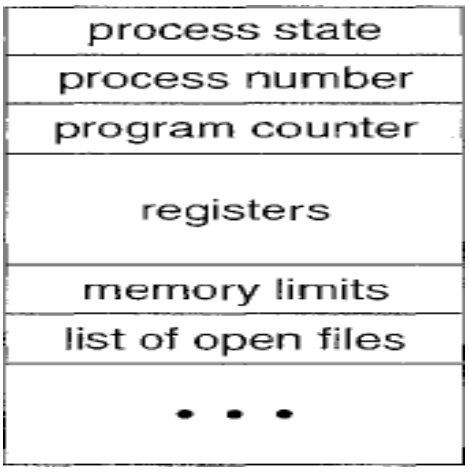
**Waiting:** The process is waiting for some event to occur (such as an I/0 completion or reception of a signal).

**Terminated:** The process has finished execution.



**Process control block**

Each process is represented in the operating system by a process control block(PCB)-also called a **Task Control Block**. A process control block is shown in the following figure.




It contains many pieces of information associated with a specific process, including these:

**Process state:**

The state may be new, ready running, waiting, terminated, and so on.

**Program counter:**

The counter indicates the address of the next instruction to be executed for this process.

**CPU registers:**

The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

**CPU-scheduling information**:

This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

**Memory-management information:**

This information may include value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

**Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

**I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

**Process scheduling**

**(scheduling queues, schedulers, context switch)**

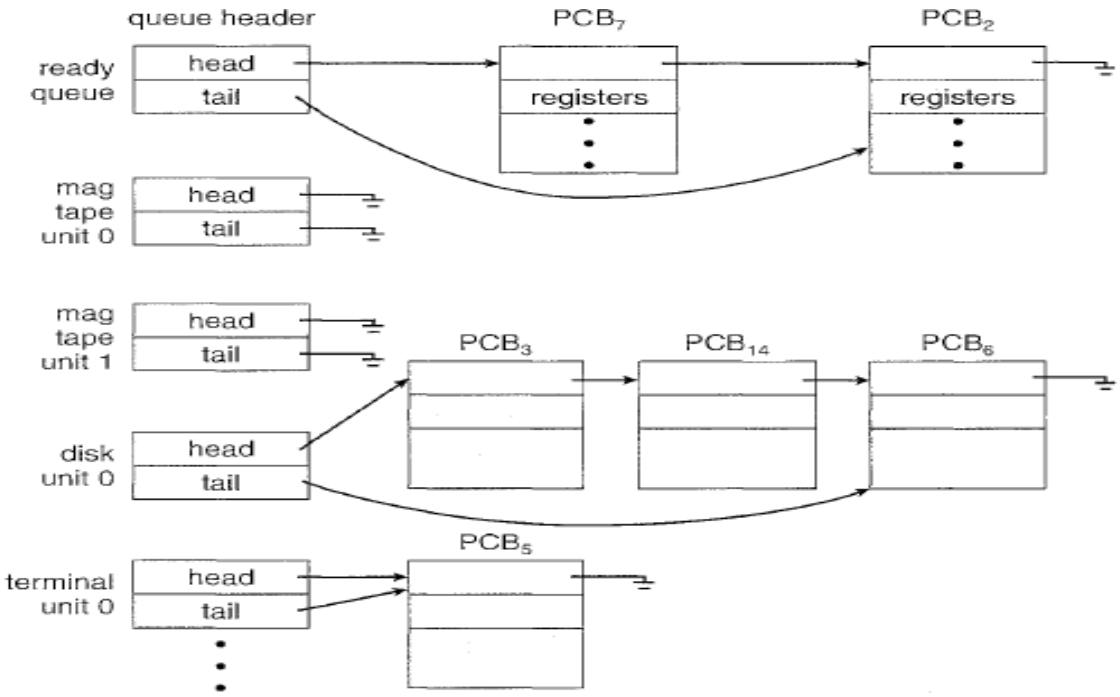
The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

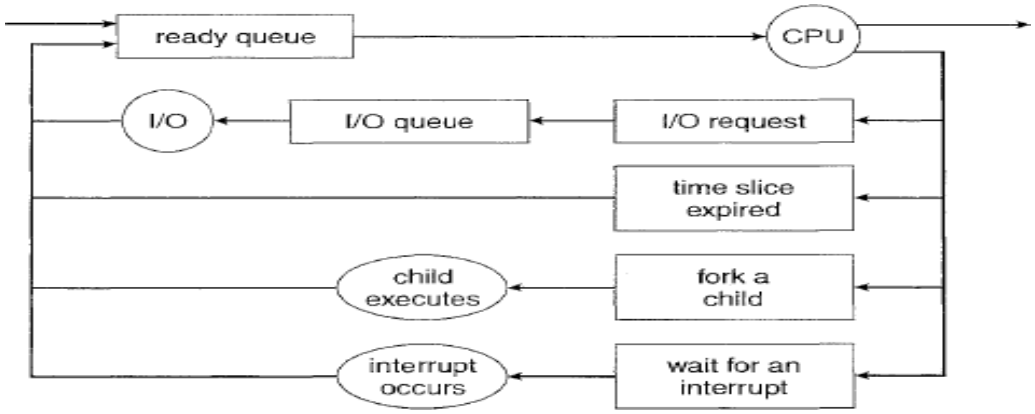
**Scheduling queues**

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.




The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/0 requests. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/0 request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/0 devices is called a device queue. Each device has its own device queue.

A common representation of process scheduling is a queuing diagram, each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.



The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur.

The process could issue an I/0 request and then be placed in an I/0 queue. The process could create a new sub-process and wait for the sub-process's termination.

The process could be removed forcibly from the CPU, as a result of an interrupt and be put back in the ready queue.

**Schedulers**

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

The **long-term scheduler** or **job scheduler** selects processes from this pool and loads them into memory for execution. The **short-term scheduler** or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in **frequency of execution**. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/0 request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.

If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10 I (100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

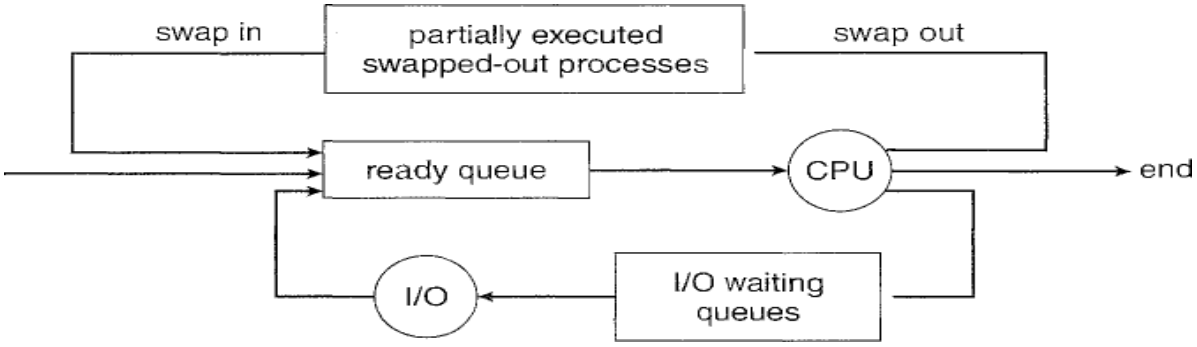
The long-term scheduler executes much less frequently. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).

If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Most processes can be described as either I/ 0 bound or CPU bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.

A CPU-bound process, in contrast, generates I/0 requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/0 bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/0 waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal.

For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.
Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler is shown in the following diagram.



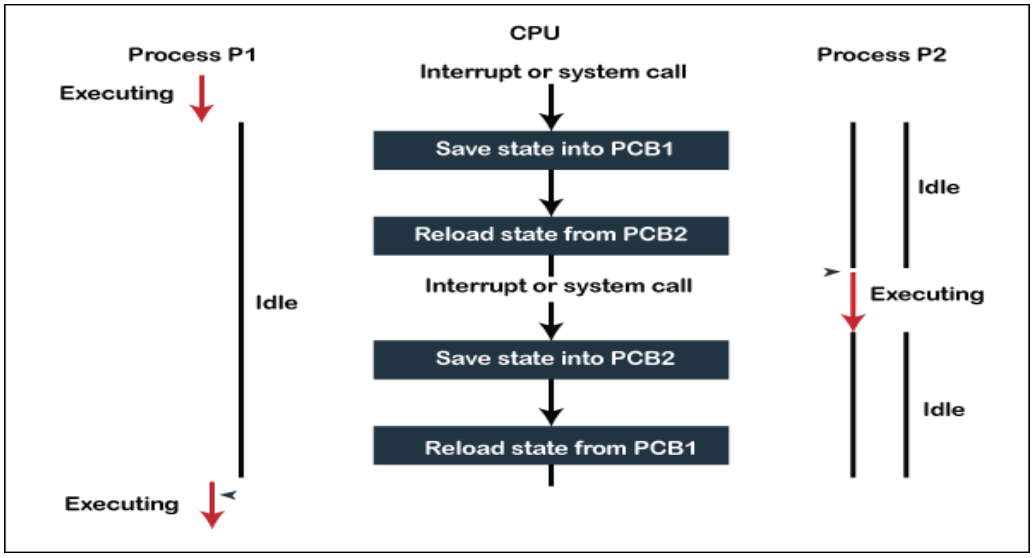

The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multi programming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

**Context switch**

When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are a few milliseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch.




**Process scheduling-basic concepts**

- **CPU-I/O Burst Cycle**
- **CPU Scheduler**
- **Preemptive Scheduling**
- **Dispatcher**

**Basic concepts**

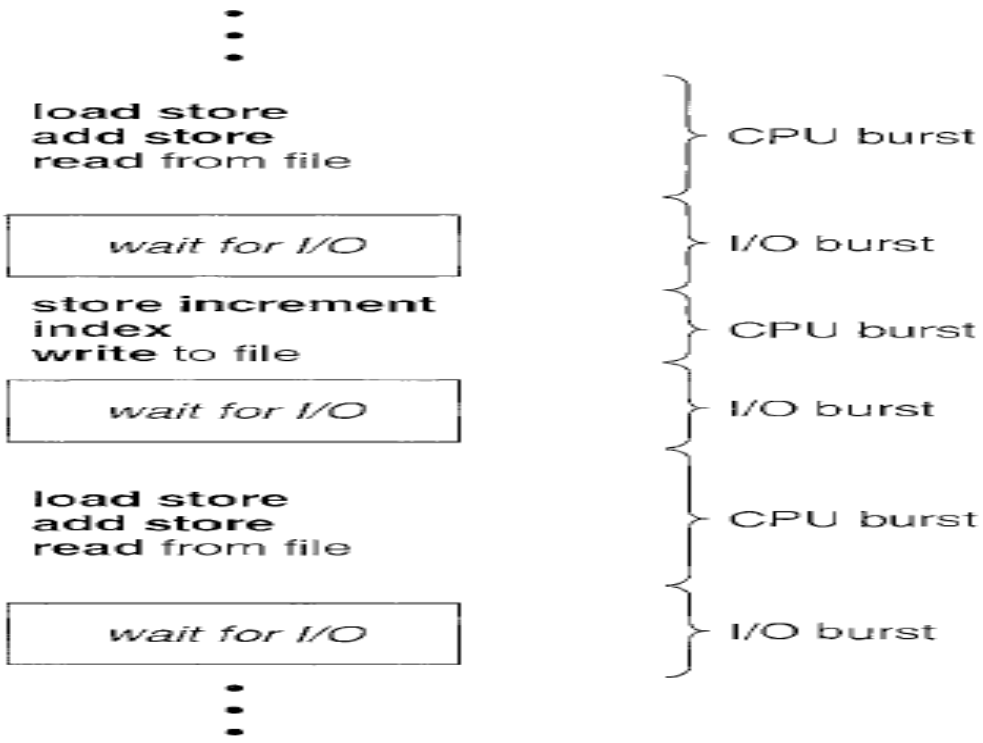
In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multi programming is to have some process rum1ing at all times, to maximize CPU utilization.

The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All its waiting time is wasted; no useful work is accomplished.

With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use.

The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.



**CPU-I/O Burst Cycle**

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/0 wait. Processes alternate between these two states.

Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/0 burst, and so on.

Eventually, the final CPU burst ends with a system request to terminate execution.

**CPU Scheduler**

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler** (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the **ready queue** is not necessarily a **first-in, first-out (FIFO) queue**. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented **as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.** Conceptually, however all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

**Preemptive Scheduling**

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the **running state to the waiting state** (for example, as the result of an I/0 request or an invocation of wait for the termination of one of the child processes)
2. When a process switches from the **running state to the ready state** (for example, when an interrupt occurs)
3. When a process switches from the **waiting state to the ready state** (for example, at completion of I/0)
4. When a **process terminates**

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative; otherwise, it is preemptive.

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

**Dispatcher**

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program The

dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

**Scheduling criteria**

Different CPU-scheduling algorithms have different properties. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

- CPU utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

- **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

- **Turnaround time:** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0.
- **Waiting time:** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0; it affects only the an1.ount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

**Scheduling algorithms**

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

- **First-Come, First-Served Scheduling**
- **Shortest-Job-First Scheduling**
- **Priority Scheduling**
- **Round-Robin Scheduling**
- **Multilevel Queue Scheduling**
- **Multilevel Feedback Queue Scheduling**

- **First-Come, First-Served Scheduling**

The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

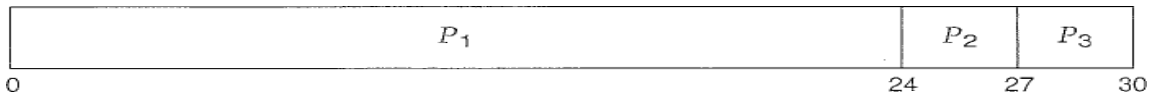
When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If the processes arrive in the order P1, P2, P3, and are served in FCFS order,

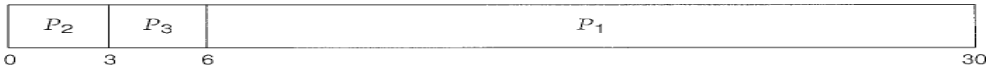
We get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:




The waiting time is 0 milliseconds for process P1, 24 milliseconds for process

P2 , and 27 milliseconds for process P3 . Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in the order P2, P3 , P1, however, the results will be as shown in the following Gantt chart:




The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly. The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0.

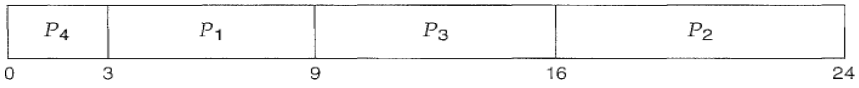
### Shortest-Job-First Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because

scheduling depends on the length of the next CPU burst of a process, rather than its total length. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:




The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4 .

Thus, the average waiting time is (3 + 16 + 9 + 0) I 4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds. The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

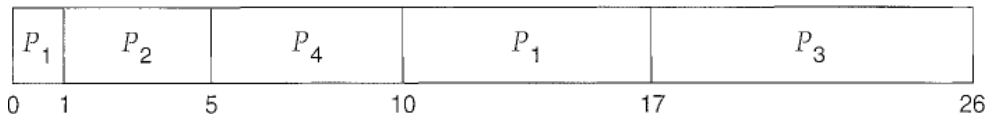
The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

If the processes arrive at the ready queue at the times shown and need the

indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:




### Priority Scheduling

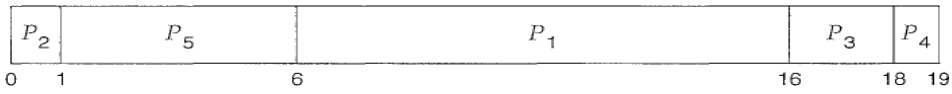
The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

**Note**: that we discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

Using priority scheduling, we would schedule these processes according to the following Gantt chart:




The average waiting time is 8.2 milliseconds.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

### Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length.

The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum.

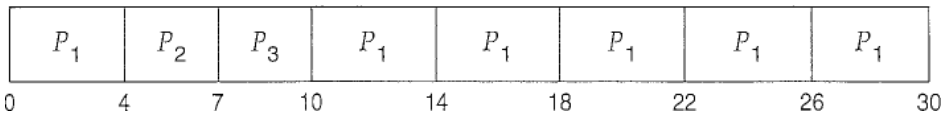
In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.

A context switch will be executed, and the process will be put at the **tail** o£ the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

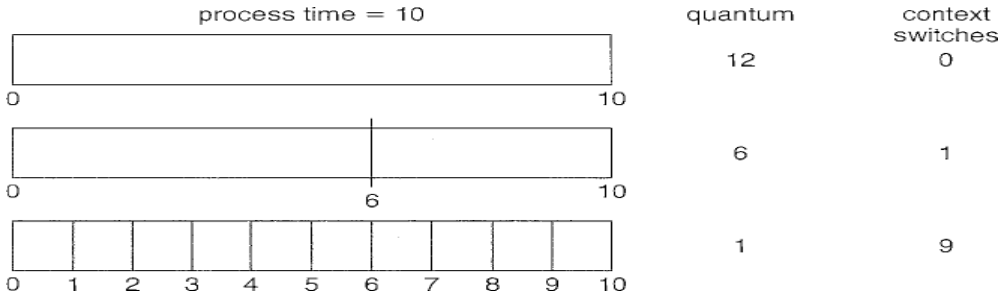
| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2 . Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:




Let's calculate the average waiting time for the above schedule. P1 waits for 6 milliseconds (10-4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

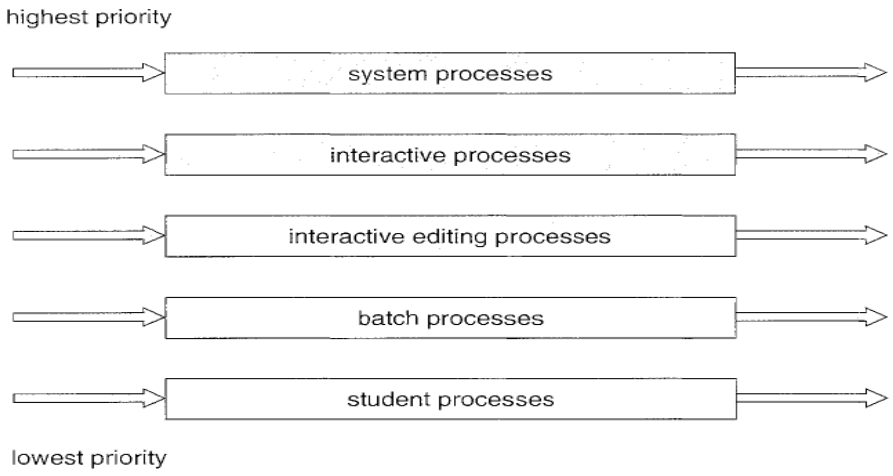



### Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.



**order of priority:**

- System processes
- Interactive processes
- Interactive editing processes
- Batch processes
- Student processes

No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets
a certain portion of the CPU time, which it can then schedule among its various processes.

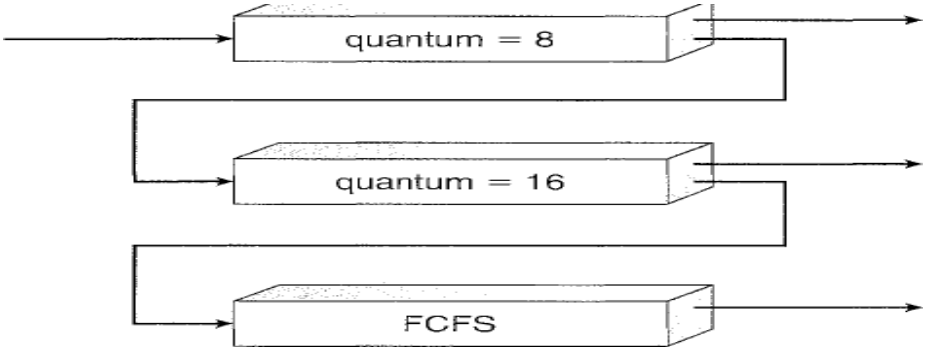
### Multilevel Feedback Queue Scheduling

When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible. The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.

This form of aging prevents starvation. For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

In general, a multilevel feedback queue scheduler is defined by the following




parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

**Process coordination**

1. **Process synchronization**
2. **deadlocks**

**Process synchronization**

- **The critical-section problem**
- **Peterson's solution**
- **Semaphores**
- **Classic problems of synchronization**

**The critical-section problem**

Consider a system consisting of n processes {Po, P1 , ... , P11 _ I}. Each process has a segment of code, called a critical section in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is in the remainder section. The general structure of a typical process Pi is shown in the following figure.

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (TRUE);
```

The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion.** If process Pi 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting.** There exists a bound, or limit, 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.

**Peterson's solution:**

One solution to the critical section problem is peterson's solution.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 - i.
Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section.
For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.

The value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P; enters its critical section only if either flag [j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] ==true. These two observations imply that Po and P1 could not have successfully executed their while statements at about the same time, since the value of

turn can be either 0 or 1 but come both. Hence, one of the processes -say, Pi -must have successfully executed the while statement, whereas P; had to execute at least one additional statement ("turn== j"). However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P; can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] ==true and turn=== j; this loop is the only one possible. If Pi is not ready to enter the critical section, then flag [j] ==false, and P; can enter its critical section. If Pj has set flag [j] to true and is also executing in its while statement, then either turn === i or turn === j. If turn == i, then P; will enter the critical section. If turn== j, then Pi will enter the critical section. However, once Pi exits its critical section, it will reset flag [j] to false, allowing P; to enter its critical section. If Pi resets flag [j] to true, it must also set turn to i.

Thus, since P; does not change the value of the variable turn while executing the while statement, P; will enter the critical section (progress) after at most one entry by P1 (bounded waiting).

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

**Semaphores**

The hardware-based solutions(synchronization hardware) to the critical-section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations:

wait () and signal ()

The wait () operation was originally termed P ; signal() was originally called V. The definition of wait () is as follows:

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

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait (S), the testing of the integer value of S (S :S 0), as well as its possible modification (S--), must be executed without interruption.

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by **P(S) and V(S)** respectively.

In very simple words, **semaphore** is a variable which can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:

```
P(S): if S ≥ 1 then S := S - 1
      else <block and enqueue the process>;

V(S): if <some process is blocked on the queue>
        then <unblock a process>
      else S := S + 1;
```

The classical definitions of **wait** and **signal** are:

- **Wait**: Decrements the value of its argument S, as soon as it would become non- negative (greater than or equal to 1).
- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue.

**Properties of Semaphores**

1. It's simple and always has a non-negative Integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

### Types of Semaphores

Semaphores are mainly of two types:

1. **Binary Semaphore:**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during execution of a program.

2. **Counting Semaphores:**

   These are used to implement bounded concurrency.

### Example of Use

Here is a simple step wise implementation involving declaration and usage of semaphore.

```
Shared var mutex: semaphore = 1;
Process i
   begin
   .
   .
   P(mutex);
   execute CS;
   V(mutex);
   .
   .
   End;
```

### Limitations of Semaphores

1. **Priority Inversion** is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in details in coming lessons.

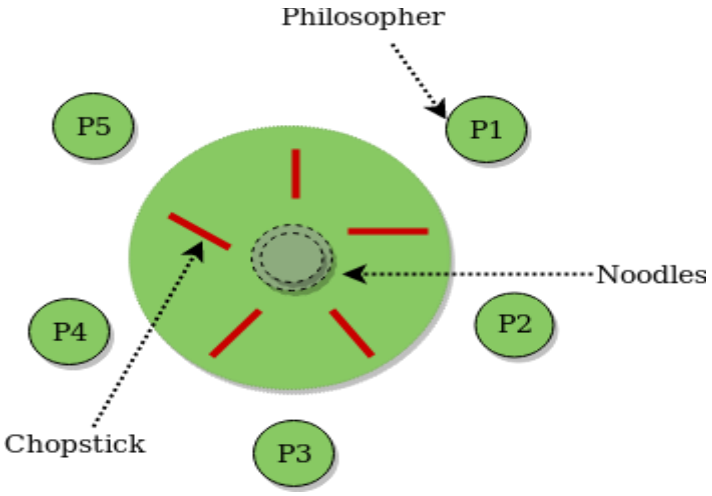
**Classical Problem Synchronization:** These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

1. Bounded-buffer (or Producer-Consumer) Problem,
2. Dining-Philosophers Problem,
3. Readers and Writers Problem,
4. Sleeping Barber Problem

These are summarized, for detailed explanation, you can view the linked articles for each.

- **Bounded-buffer (or Producer-Consumer) Problem:**
  Bounded Buffer problem is also called producer consumer problem. This problem is generalized in terms of the Producer-Consumer problem. Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
  Producers produce a product and consumers consume the product, but both use of one of the containers each time.

- **Dining-Philosophers Problem:**
  The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



- **Readers and Writers Problem:**
  Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers- writers' problem. Problem parameters:
    - One set of data is shared among a number of processes.
    - Once a writer is ready, it performs its write. Only one writer may write at a time.
    - If a process is writing, no other process can read it.
    - If at least one reader is reading, no other process can write.
    - Readers may not write and only read.

**Sleeping Barber Problem:**

The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system.

There is a barber shop with one barber and a number of chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.

The problem is to write a program that coordinates the actions of the customers and the barber in a way that avoids synchronization problems, such as deadlock or starvation. One solution to the Sleeping Barber problem is to use semaphores to coordinate access to the waiting chairs and the barber chair.

## UNIT III

**DEADLOCKS:** System model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection and Recovery from Deadlock.

### System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances. If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer. A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

***Request:*** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

***Use:*** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

***Release:*** The process releases the resource.

A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated. If a process

Requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource. A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors). However, other types of events may result in deadlocks. To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type. Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P; is holding the DVD and process Pi is holding the printer. If P; requests the printer and P1 requests the DVD drive, a deadlock occurs. A programmer who is developing multithreaded applications must pay particular attention to this problem. Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

## DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

**Necessary Conditions**

**Resource-Allocation Graph**

### Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

**Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

**Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

**No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

**Circular wait:** A set { *P0* , Pl, ... , P11 } of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by *P, v* and P11 is waiting for a resource held by *Po.*

**Resource-Allocation Graph**

Deadlocks can be described more precisely in terms of a directed graph called A **Resource-Allocation Graph** This graph consists of a set of vertices *V* and a set of edges *E.* The set of vertices Vis partitioned into two different types of nodes: *P* == { P1, P2, ... , Pn}, the set consisting of all the active processes in the system, and *R* == {R1, R2, ... , *Rm}* the set consisting of all resource types in the system.

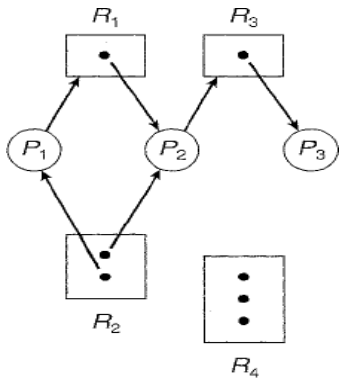
A directed edge from process $p_i$ to resource type *Rj* is denoted by $P_i$ - > *Rj;*

it signifies that process *P;* has requested an instance of resource type *Rj* and is currently waiting for that resource. A directed edge from resource type *Rj* to process *P;* is denoted by *R1- >P;;* it signifies that an instance of resource type *R1* has been allocated to process *P;*. A directed edge $P_i$- > *Rj;* is called a **request edge**. a directed edge *R1-> P;* is called an **assignment edge**

Pictorially we represent each process *P;* as a circle and each resource type *Rj* as a rectangle. Since resource type *Ri* may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *R1,* whereas an assignment edge must also designate one of the dots in the rectangle.

When process *P;* requests an instance of resource type *Ri,* a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown below




- The sets $P$, $R$, and $E$:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - One instance of resource type $R_1$
  - Two instances of resource type $R_2$
  - One instance of resource type $R_3$
  - Three instances of resource type $R_4$
- Process states:
  - Process $P_1$ is holding an instance of resource type $R_2$ and is waiting for an instance of resource type $R_1$.
  - Process $P_2$ is holding an instance of $R_1$ and an instance of $R_2$ and is waiting for an instance of $R_3$.
  - Process $P_3$ is holding an instance of $R_3$.

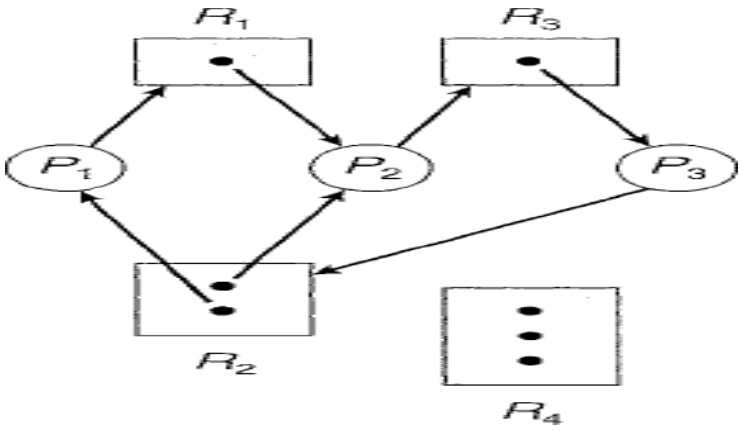
Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that

a deadlock has occurred. In this case, a cycle in. the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in the above figure. Suppose that process *P3* requests an instance of resource type R2. Since no resource instance is currently available, a request edge *P3* - > R2 is added to the below graph. At this point, two minimal cycles exist in the system:

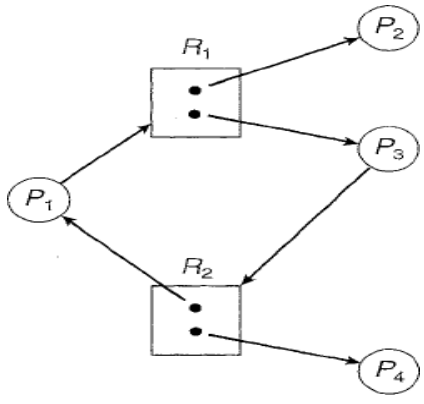
P1 -> + *R* 1 -> P2 -> R3 -> *P3* -> R2 -> P1

P2 -> R3 -> *P3* -> R2 -> P2




Processes *P1, P2,* and *P3* are deadlocked. Process *P2* is waiting for the resource *R3,* which is held by process *P3.* Process *P3* is waiting for either process *P1* or process *P2* to release resource R2. In addition, process *P1* is waiting for process *P2* to release resource R1.

Now consider other resource-allocation graph given below



In this example, we also have a cycle:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process *P4* may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

**METHODS FOR HANDLING DEADLOCKS**

We can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually. In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods, which must be used constantly. Also, in some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process running on a non- preemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

## Deadlock prevention

for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

### Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

**Hold and Wait**

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file and then again request the disk file and printe1~ only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.

Second starvation is possible.

A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

## No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

## Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let $R = \{ R1, R2, \ldots , Rm\}$ be the set of resource types.

$F$ (tape drive) = 1

$F$ (disk drive) = 5

$F$ (printer) = 12

**We can now consider the following protocol to prevent deadlocks:**

Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -say, *R;*. After that the process can request instances of resource type *Rj* if and only if F(Rj) > F(R;).

For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type *Rj* must have released any resources *R;* such that F(Ri) ≥ F(Rj). It must also be noted that if several instances of the same resource type are needed, a *single* request for all of them must be issued.

## Deadlock avoidance

For example, in a system with one tape drive and one printer, the system might need to know that process *P* will request first the tape drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources For example, in a system with one tape drive and one printer, the system might need to know that process *P* will request first the tape drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources.

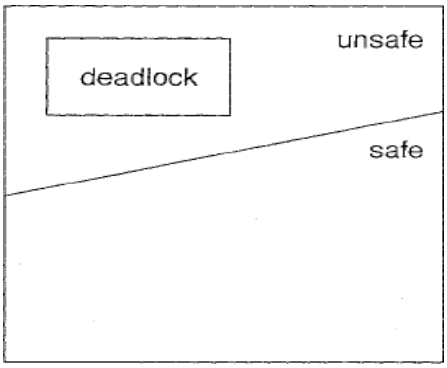
**Safe State**

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, ... , *Pn*> is a safe sequence for the current allocation state if, for each *Pi,* the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj,* with *j* < *i.* In this situation, if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished.

When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources and terminate. When *Pi* terminates, *Pi+l* can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe.*

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however an unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.




To illustrate, we consider a system with twelve magnetic tape drives and three processes: *Po,* P1, and P2. Process *Po* requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time *to,* process *Po* is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

| | Maximum Needs | Current Needs |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | 2 |

At time *t0,* the system is in a safe state. The sequence <P1, *P0,* P2> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *Po* can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P2* can get all its tape drives and return them (the system will then have all twelve tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process *Po* is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock.

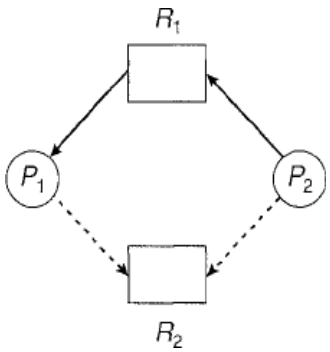
Our mistake was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state. In this scheme, if a process requests a resource that is currently available, it may still have to wait.

**Resource-Allocation-Graph Algorithm**

In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge.

A claim edge *Pi* --> *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process *Pi* requests resource



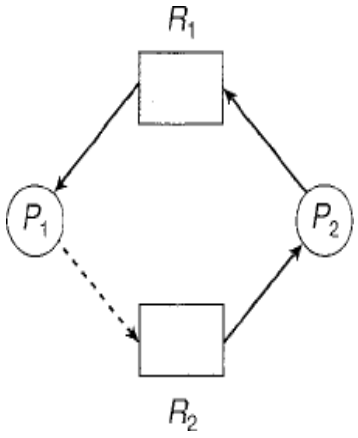
To illustrate this algorithm, we consider the above resource-allocation graph. Suppose that *P2* requests R2. Although R2 is currently free, we cannot allocate it to *P2*, since this action will create a cycle in the graph A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

$R_j$, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource $R_j$ is released by $P_i$, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

We note that the resources must be claimed a priori in the system. That is, before process $P_i$ starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process $P_i$ are claim edges.

Now suppose that process $P_i$ requests resource $R_j$. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of $n^2$ operations, where $n$ is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process $P_i$ will have to wait for its requests to be satisfied.



## Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the *banker's algorithm.* The name was chosen because the algorithm. Could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This nun1.ber may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:

**Available.** A vector of length *m* indicates the **number of available resources** of each type. If *Available[j]* equals *k,* then *k* instances of resource type *Ri* are available.

**Max.** An *n* x *m* matrix defines the **maximum demand of each process**.

If *Max[i] [j]* equals *k,* then process *P;* may request at most *k* instances of resource type Ri.

**Allocation.** An *11* x *m* matrix defines the **number of resources of each type currently allocated to each process**. If *Allocation[i][j]* equals lc, then process *P;* is currently allocated lc instances of resource type *Rj*.

**Need.** An *n* x *m* matrix indicates **the remaining resource need of each process**. If *Need[i][j]* equals *k,* then process *P;* may need *k* more instances of resource type *Ri* to complete its task. Note that *Need[i][j]* equals *Max[i][j] - Allocation [i][j]*.

## Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length $m$ and $n$, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, \ldots, n - 1$.
2. Find an index $i$ 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.

This algorithm may require an order of *m* x *n2* operations to determine whether a state is safe.

## Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

Let *Request;* be the request vector for process *P;*. If *Request; [j] == k*, then process *P;* wants *k* instances of resource type *Rj*. When a request for resources is made by process *P;*, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, $P_i$ must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process $P_i$ by modifying the state as follows:

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

**Example**

| | *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* and is as follows:

| | *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 |

We claim that the system is currently in a safe state. Indeed, the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria. Suppose now that process $P_1$ requests one additional instance of resource type *A* and two instances of resource type *C*, so $Request_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$—that is, that $(1,0,2) \leq (3,3,2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

| | *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 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 1 | |

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P1, P3, P4, Po, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by *Po* cannot be granted, even though the resources are available, since the resulting state is unsafe.

**Deadlock detection**

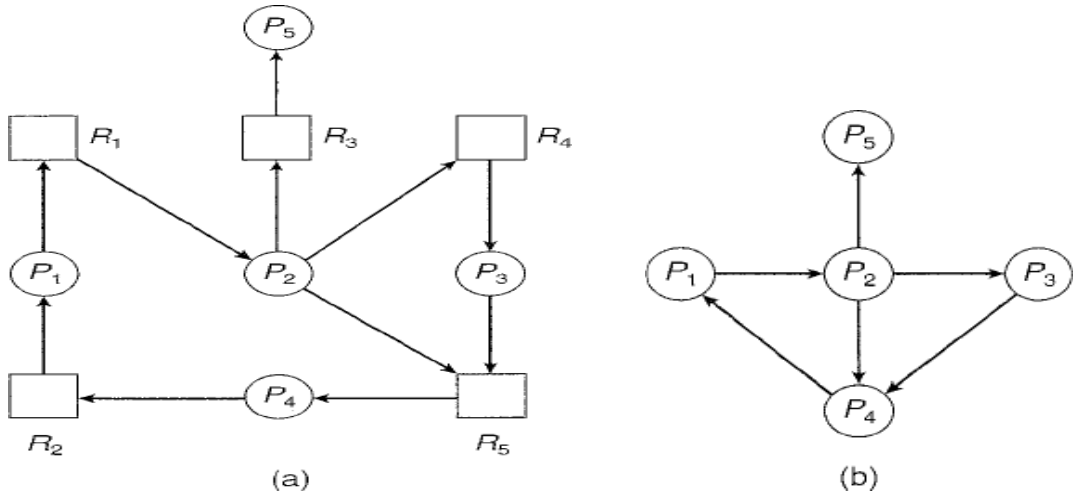
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

**Single Instance of Each Resource Type**

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.




(a) Resource-allocation graph. (b) Corresponding wait-for graph.

A deadlock exists in the system if and only if the wait-for graph

contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of *n2* operations, where *n* is the number of vertices in the graph.

## Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm

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

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

**Request.** An *n* x *m* matrix indicates the **current request of each process**.

If *Request[i][j]* equals *k*, then process *Pi* is requesting *k* more instances of resource type *Rj*.

1. Let *Work* and *Finish* be vectors of length $m$ and $n$, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.
2. Find an index $i$ such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such $i$ exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some $i$, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process $P_i$ is deadlocked.

This algorithm requires an order of $m$ x $n^2$ operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes *Po* through *P4* and three resource types *A*, *B*, and C. Resource type *A* has seven instances, resource type *B* has two instances, and resource type C has six instances.

Suppose that, at time *T0*, we have the following resource-allocation state:

| | *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 | |

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*Po, P2, P3, Plt P4*> results in *Finish[i] == true* for all *i.*

Suppose now that process *P2* makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

| | *Request* |
|---|---|
| | *A B C* |
| $P_0$ | 0 0 0 |
| $P_1$ | 2 0 2 |
| $P_2$ | 0 0 1 |
| $P_3$ | 1 0 0 |
| $P_4$ | 0 0 2 |

We claim that the system is now deadlocked. Although we can reclaim the resources held by process *Po,* the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, *P2, P3,* and *P4.*

## Recovery from a deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.

Another possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

- Process termination
- Resource preemption

### Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

**Abort all deadlocked processes**.

This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

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

This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then we must determine which deadlocked process should be terminated.

We should abort those processes whose termination will incur
the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many
factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated Whether the process is interactive or batch

**Resource Preemption**

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes 1-m til the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

**Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

**Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

**Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process

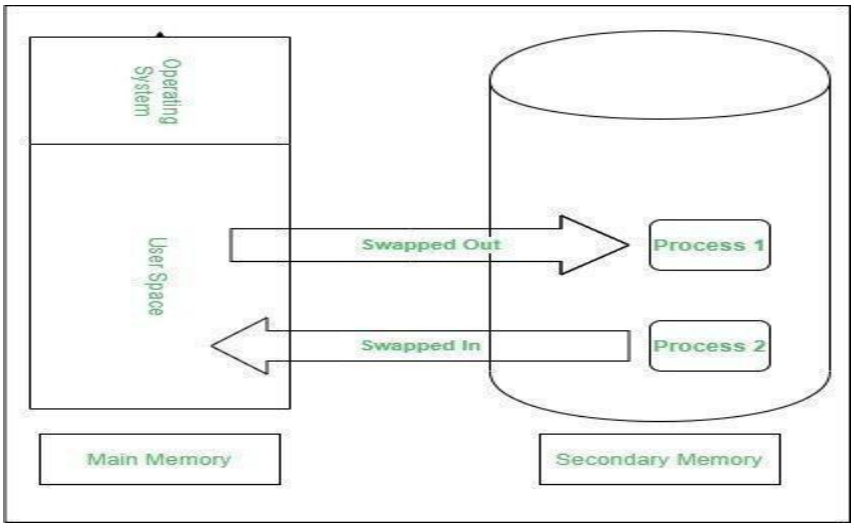
# UNIT IV

**MEMORYMANAGEMENT:** Swapping, Contiguous Memory Allocation, Paging, Structure of Page Table, Segmentation, Demand Paging, Performance of Demand Paging, Page Replacement Algorithms.

**FILE SYSTEM INTERFACE:** The Concept of a File, Access methods, Directory Structure, Allocation methods, Disk Scheduling algorithms.

## Swapping:

Swapping is **a process of swapping a process temporarily to a secondary memory from the main memory which is fast than compared to secondary memory**. But as RAM is of less size so the process that is inactive is transferred to secondary memory.




### Swap In and Swap Out in OS

The procedure by which any process gets removed from the **hard disk** and placed in the **main memory or RAM** commonly known as **Swap In.**

On the other hand, **Swap Out** is the method of removing a process from the **main memory or RAM** and then adding it to the **Hard Disk.**

### Advantages of Swapping

1. The swapping technique mainly helps the CPU to manage multiple processes within a single main memory.
2. This technique helps to create and use virtual memory.
3. With the help of this technique, the CPU can perform several tasks simultaneously. Thus, processes need not wait too long before their execution.
4. This technique can be easily applied to **priority-based** scheduling in order to improve its performance.

**Disadvantages of Swapping**

1. There may occur inefficiency in the case if a resource or a variable is commonly used by those processes that are participating in the swapping process.
2. If the algorithm used for swapping is not good then the overall method can increase the number of page faults and thus decline the overall performance of processing.
3. If the computer system loses power at the time of high swapping activity then the user might lose all the information related to the program.

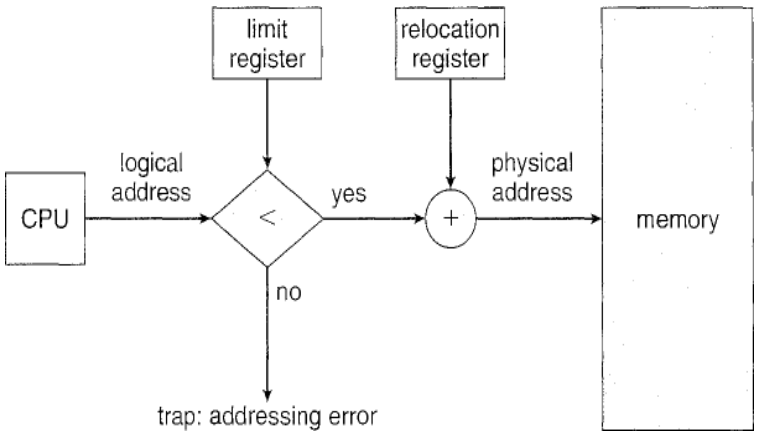
## 2. Contiguous memory allocation:

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one common method, contiguous memory allocation.

**The memory is usually divided into two partitions**: one for the resident **operating system** and one for the **user processes**. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory. We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single contiguous section of memory.

**Memory Mapping and Protection**

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical Addresses. With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory. It is shown in the following figure




Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

## Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized. Each partition may contain exactly one process. Thus, the degree of multi programming is bound by the number of partitions. In this multiple partition method when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (**called MFT**); it is no longer in use. The method described next is a generalization of the fixed-partition scheme (**called MVT**); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management.

In the variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory a hole. Eventually as you will see, memory contains a set of holes of various sizes.

In general as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.
There are many solutions to dynamic storage allocation problem. They are
**first fit, best fit** and **worst fit.**

**First fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

**Best fit**: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

**Worst fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

## Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

The memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**-unused memory that is internal to a partition.

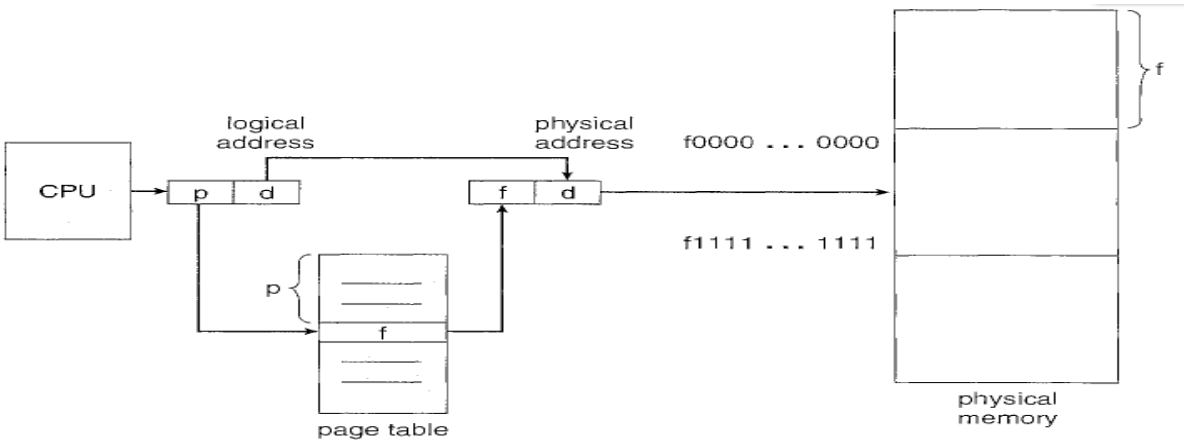
One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done.

**Compaction** is possible only if relocation is dynamic and is done at execution time.
Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.

Two complementary techniques achieve this solution: paging and segmentation. These techniques can also be combined.

## 3. paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction.

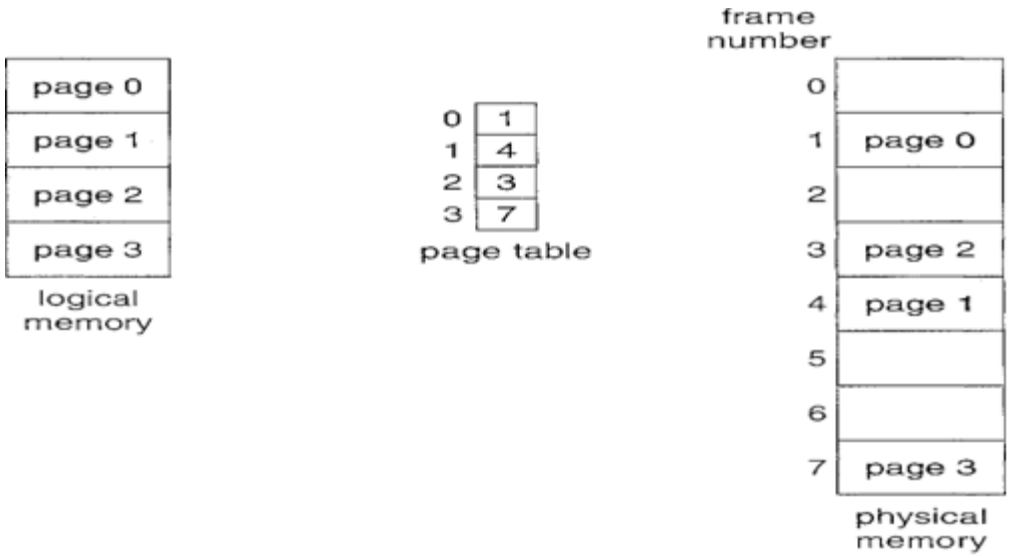



**Paging Hardware**

**Basic Method**

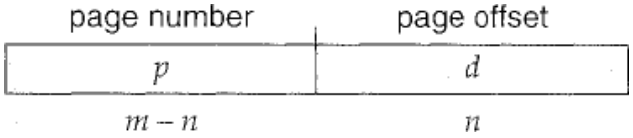
The basic method for implementing paging involves breaking physical memory into fixed- sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The paging model of memory is shown in the below Figure



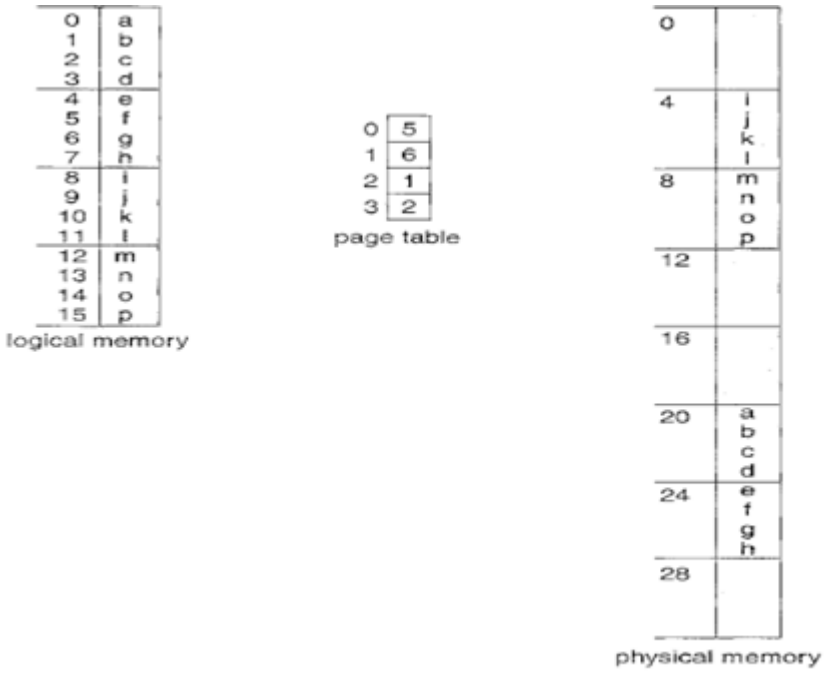

Paging model of logical and physical memory

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.



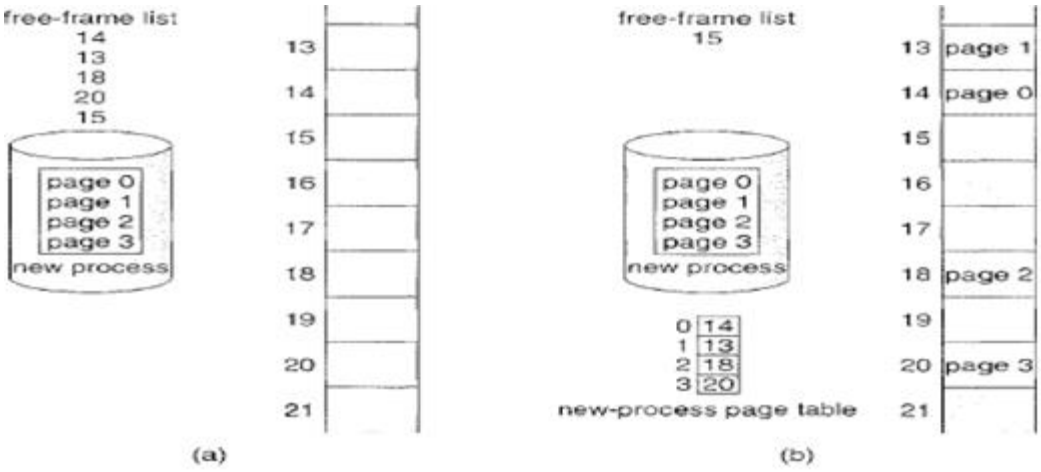

Where p is an index into the page table and d is the displacement within the page.

**Example:**




Paging example for a 32-byte memory with 4-byte pages.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.




Free frames (a) before allocation and (b) after allocation.

**Hardware Support**

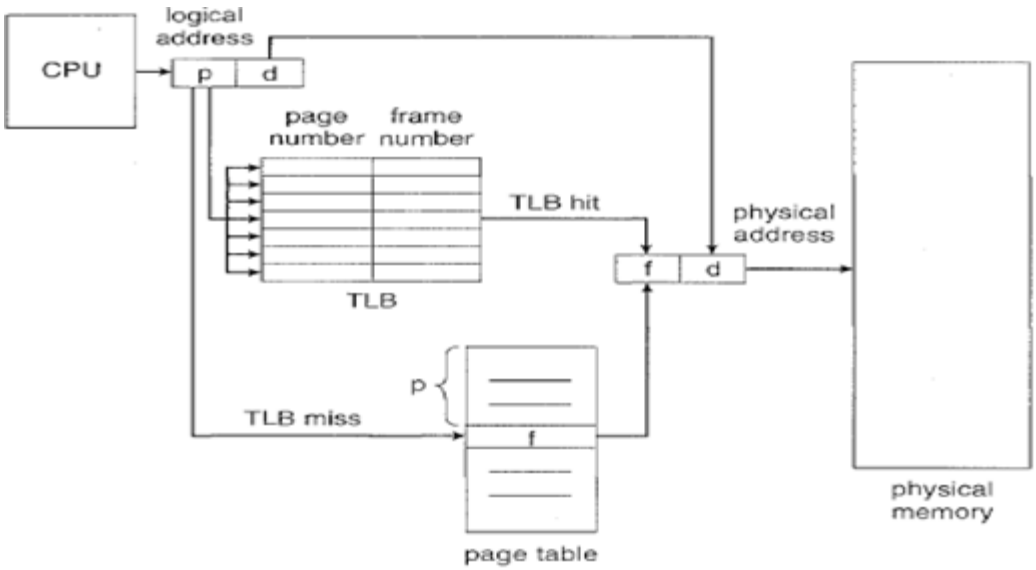
Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page- table values from the stored user page table.

The hardware implementation of the page table can be done in several ways

In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map.

The **TLB** is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.




Paging hardware with TLB.

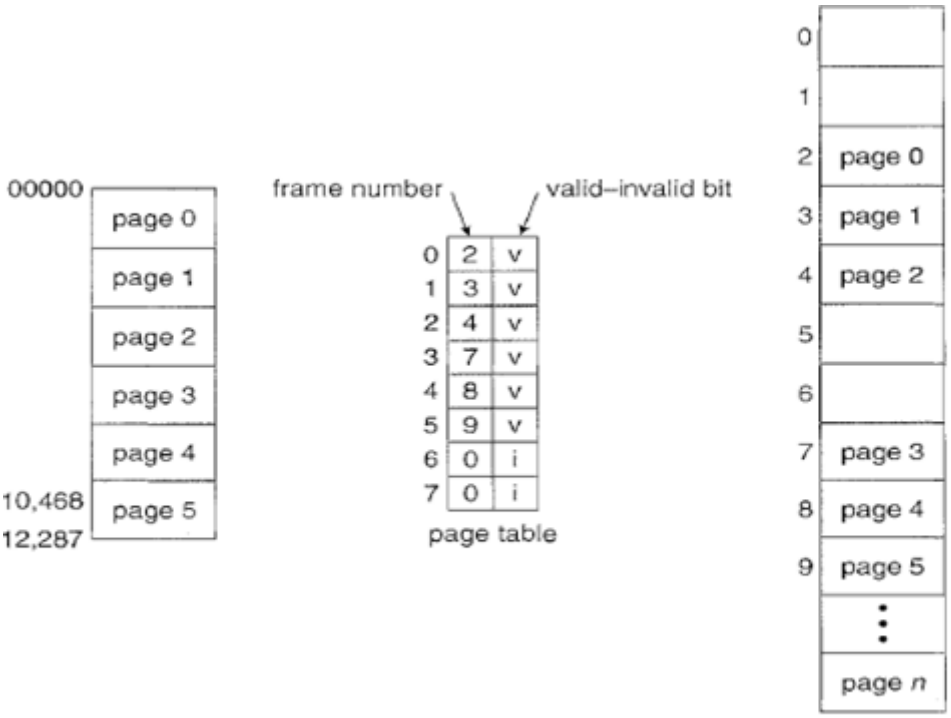
**Protection**

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

One additional bit is generally attached to each entry in the page table: **a valid or invalid** bit.

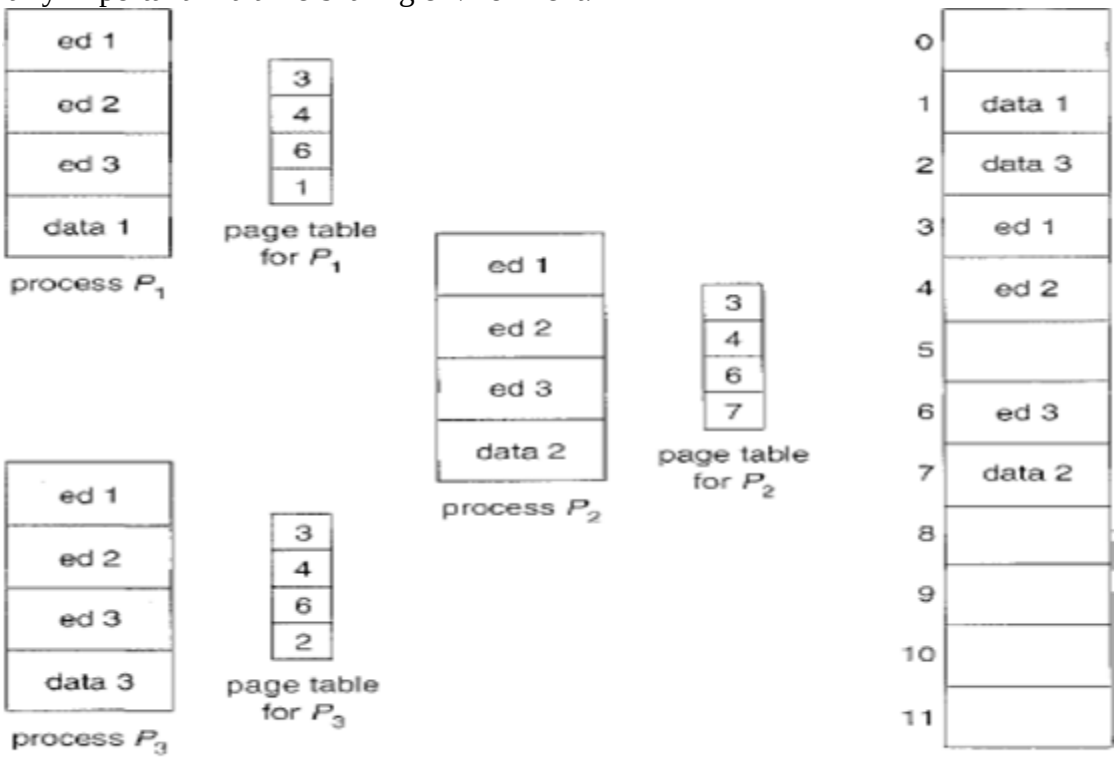
When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to "invalid," the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid -invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.




Valid (v) or invalid (i) bit in a page table.

**Shared Pages**

An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment.




Sharing of code in a paging environment.

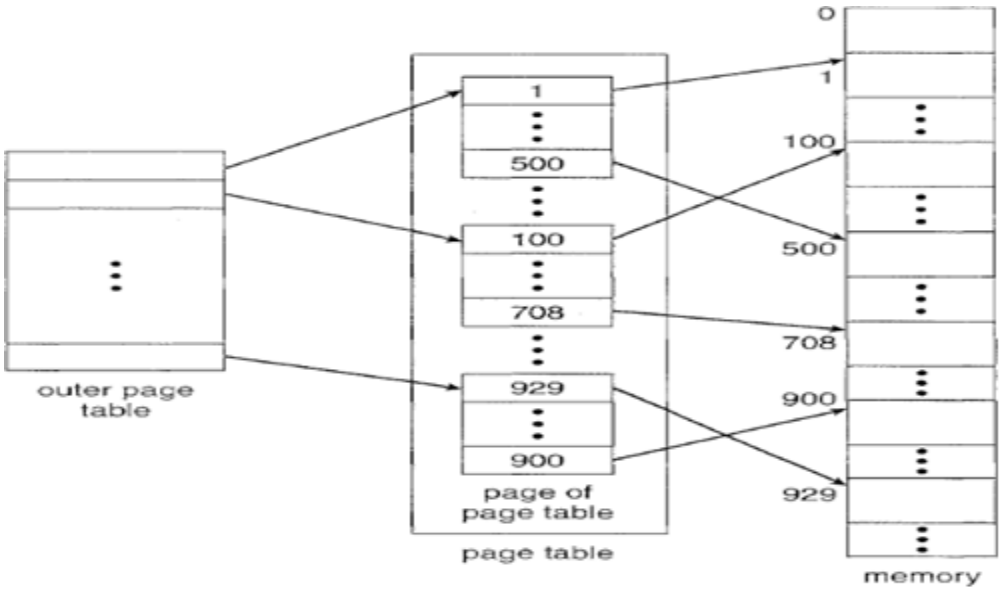
**Structure of a page table**

Techniques for structuring the page table.

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

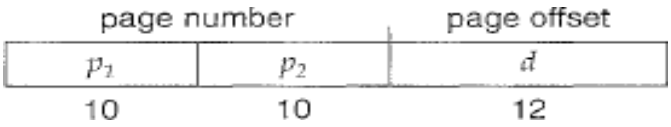
**Hierarchical Paging**

One way is to use a two-level paging algorithm, in which the page table itself is also paged

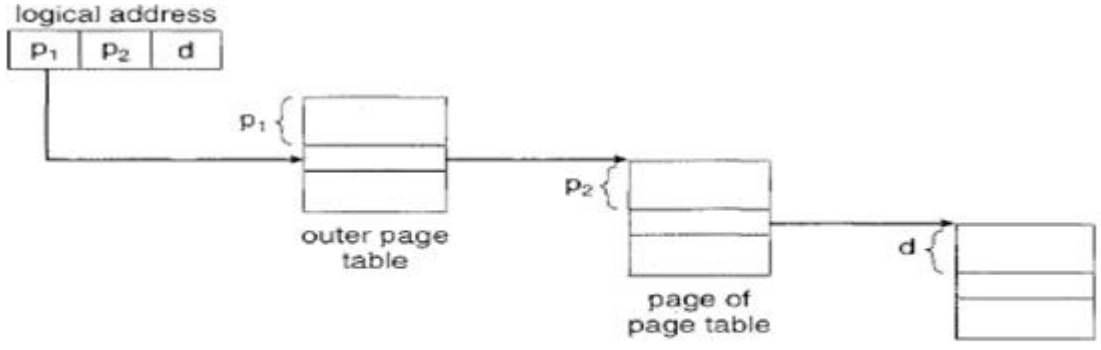



A two-level page-table scheme.

For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

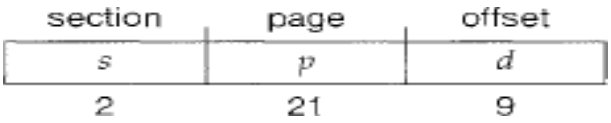



where p1 is an index into the outer page table and P2 is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure 8.15. Because address translation works from the outer page table inward, this scheme is also known as a forwarded mapping page table. This can be shown in the following figure.



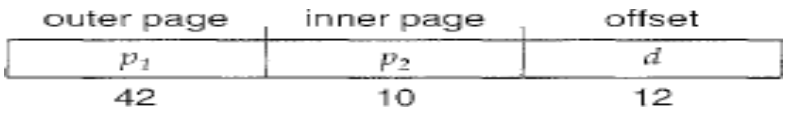

Address translation for a two-level 32-bit paging architecture.

The VAX architecture supports a variation of two-level paging. The VAX is a 32-bit machine with a page size of 512 bytes. The logical address space of a process is divided into four equal sections, each of which consists of 230 bytes. Each section represents a different part of the logical address space of a process. The first 2 high- order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the final 9 bits represent an offset in the desired page. By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them. An address on the VAX architecture is as follows:

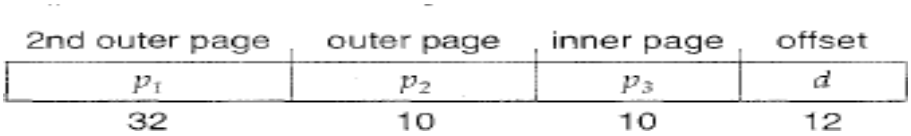



Whereas designates the section number, p is an index into the page table, and dis the displacement within the page. Even when this scheme is used, the size of a one-level page table for a VAX process using one section is 221 bits * 4 bytes per entry= 8MB. To further reduce main-memory use, the VAX pages the user-process page tables.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4 KB (212). In this case, the page table consists of up
to 252 entries. If we use a two-level paging scheme, then the iml.er page tables can conveniently be one page long, or contain 210 4-byte entries. The addresses look like this:




We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (210 entries, or 212 bytes). In this case, a 64-bit address space is still daunting:



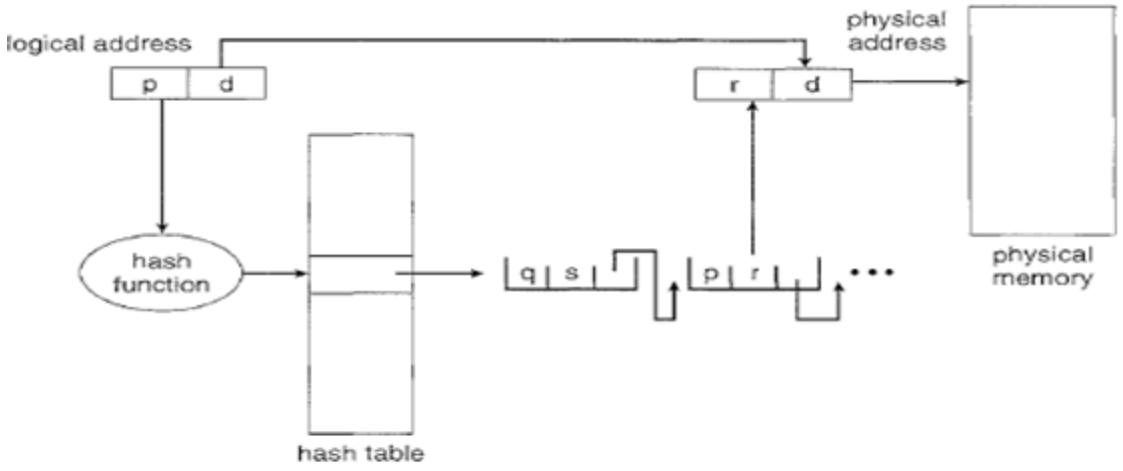
## Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a hashed page table. With the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields:

**(1)** The virtual page number
**(2)** The value of the mapped page frame
**(3)** A pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation uses **clustered page table. W**hich are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical- page frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.




Hashed page table.

## Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using. This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an page An inverted page table. Page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. The following figure shows the operation of an inverted page table.

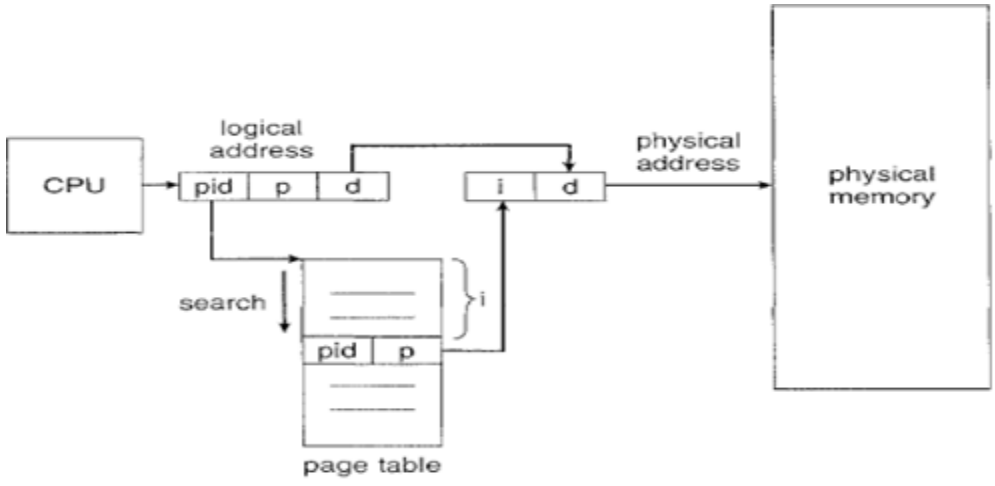
Each virtual address in the system consists of a triple:

**<process-id, page-number, offset>.**

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier.

When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>,is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found-say, at entry i-then the physical address <i, offset> is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long.



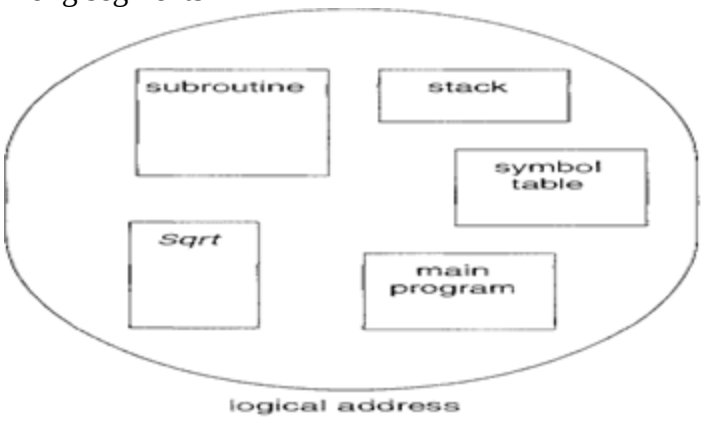

Inverted page table.

## Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory. As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

**Basic Method**

users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments




User's view of a program.

Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the stack," "the math library," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the stack is stored before or after the Sqrt () function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment segmentation is a memory-management scheme that supports this user view of memory.

A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

**<segment-number, offset>**

The user program is compiled, and the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

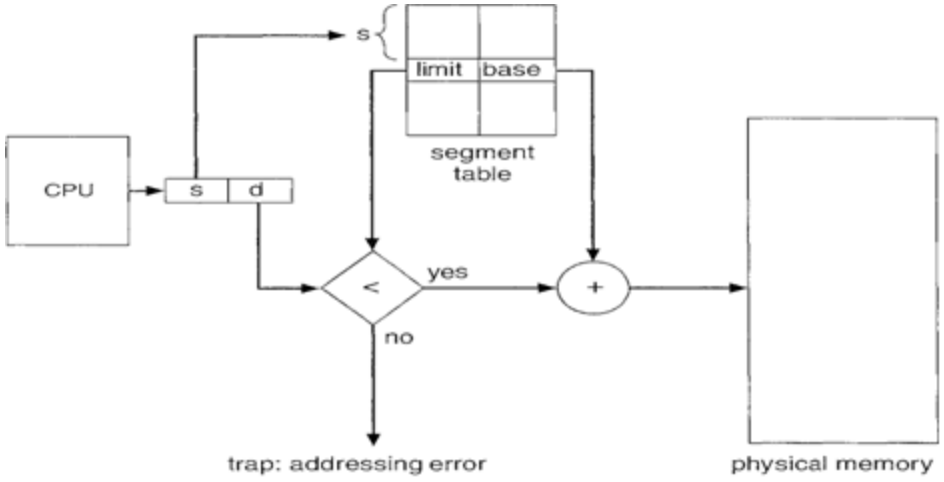
- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

## Hardware

The user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses.

This mapping is effected by a segment table. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

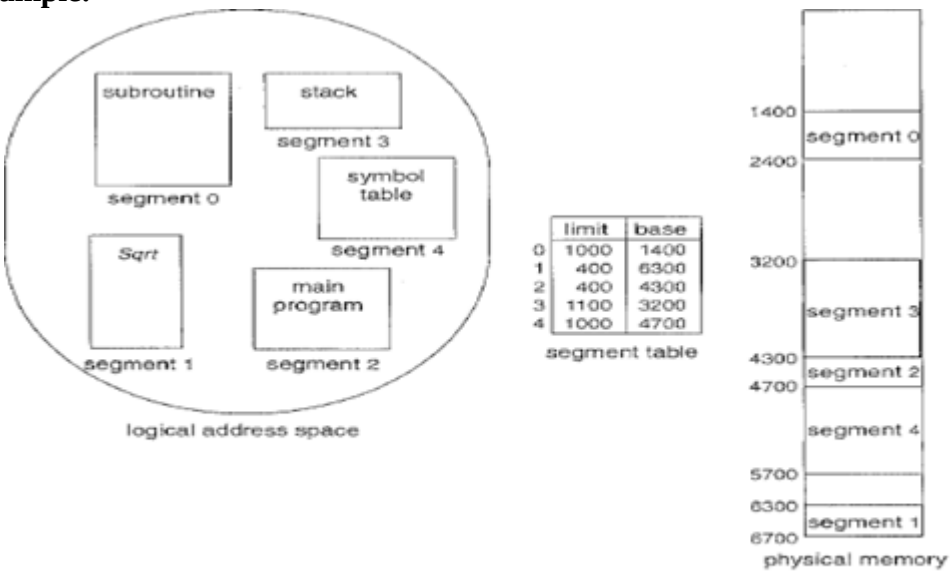



Segmentation hardware.

A logical address consists of two parts: a segment number, s, and an offset into that segment, d. The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

**Example:**

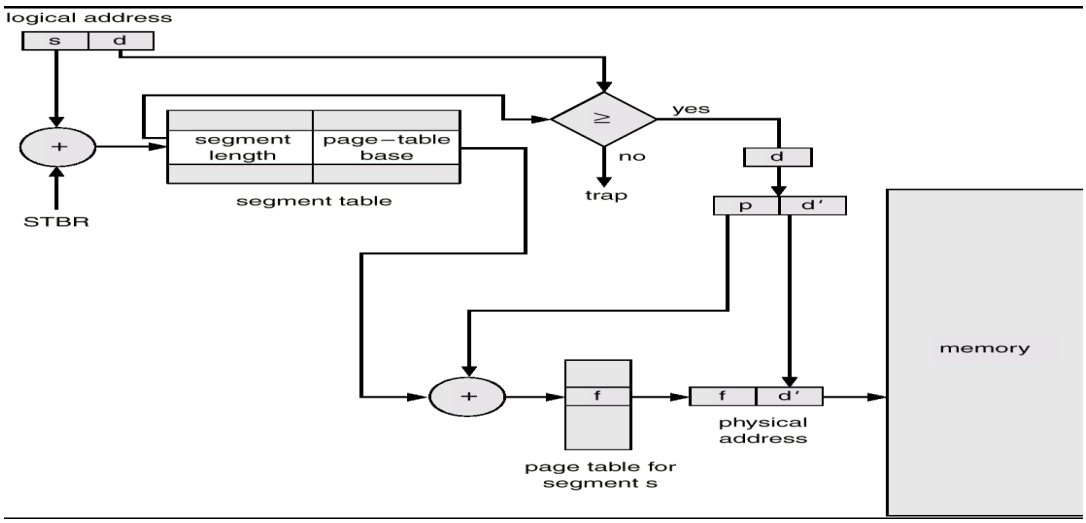



Example of segmentation.

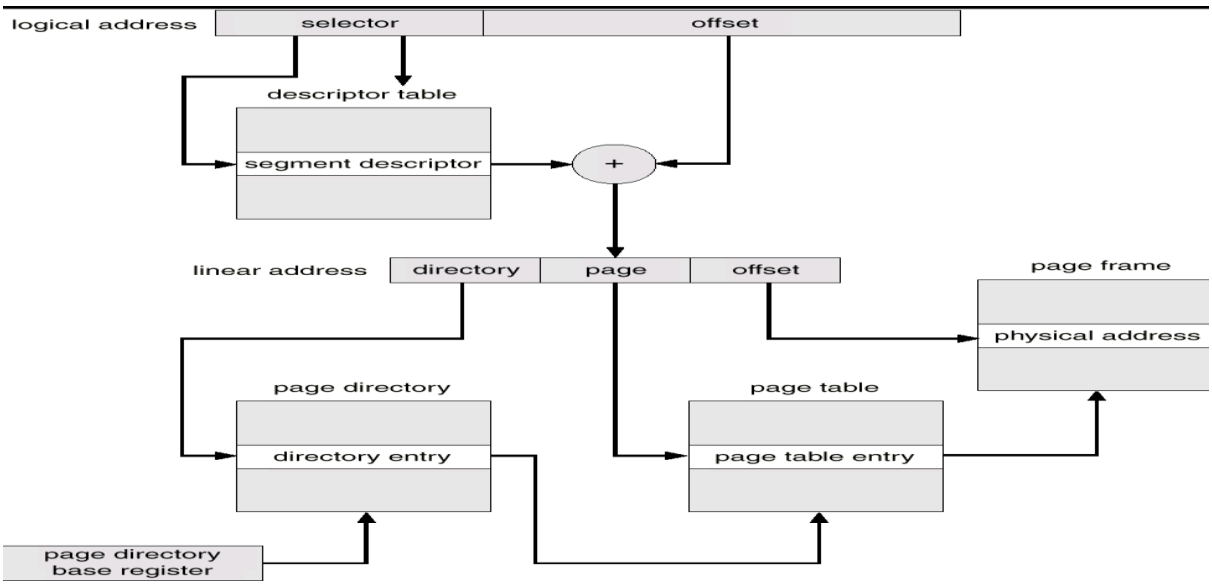
We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1OOO bytes long.

**Segmentation with Paging – MULTICS!**

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



As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

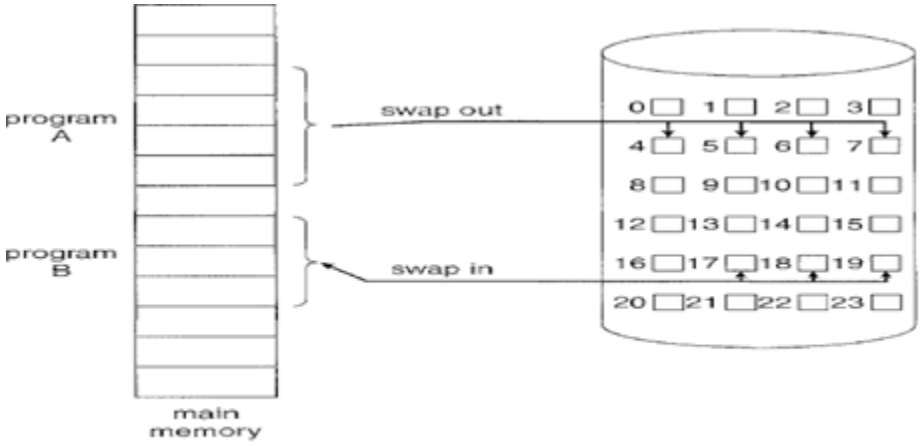



## Demand paging

Consider how an executable program might be loaded from disk into memory.

One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping



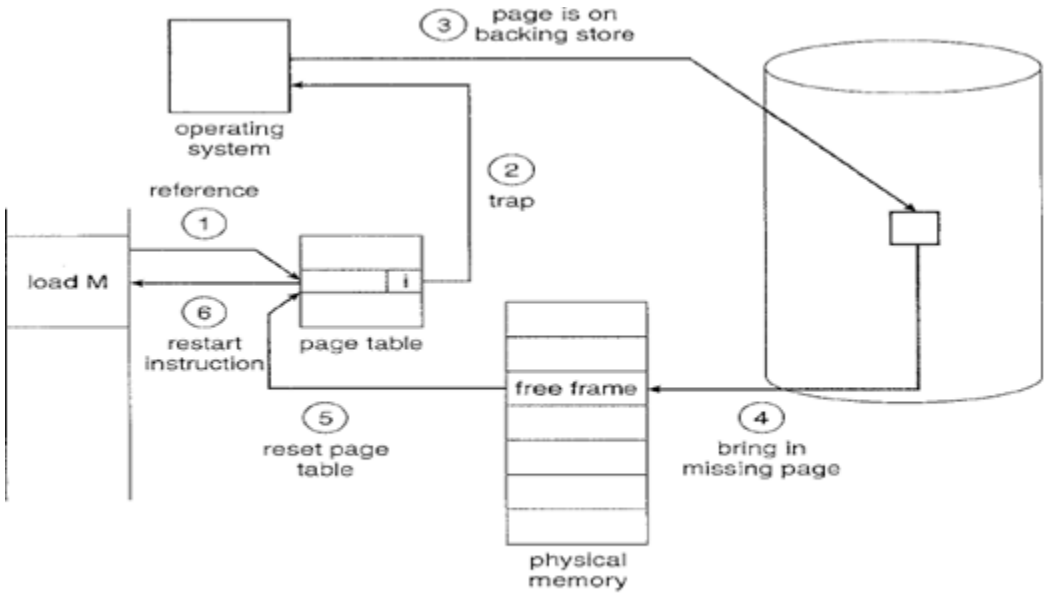

Transfer of a paged memory to contiguous disk space.

Where processes reside in secondary memory (usually a disk).When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term swapper is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

**Basic Concepts**

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid -invalid bit scheme described can be used for this purpose. This time, however, when this bit is set to "valid' the associated page is both legal and in n1.emory. If the bit is set to "invalid' the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.




Steps in handling a page fault.

The procedure for handling this page fault is straightforward (above Figure):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point it can execute with no more faults. This scheme is pure demand paging: never bring a page into memory until it is required.

The hardware to support demand paging is the same as the hardware for paging and swapping.

## Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the effective access time for a demand-paged memory. For most computer systems, the memory-access time, denoted ma, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.

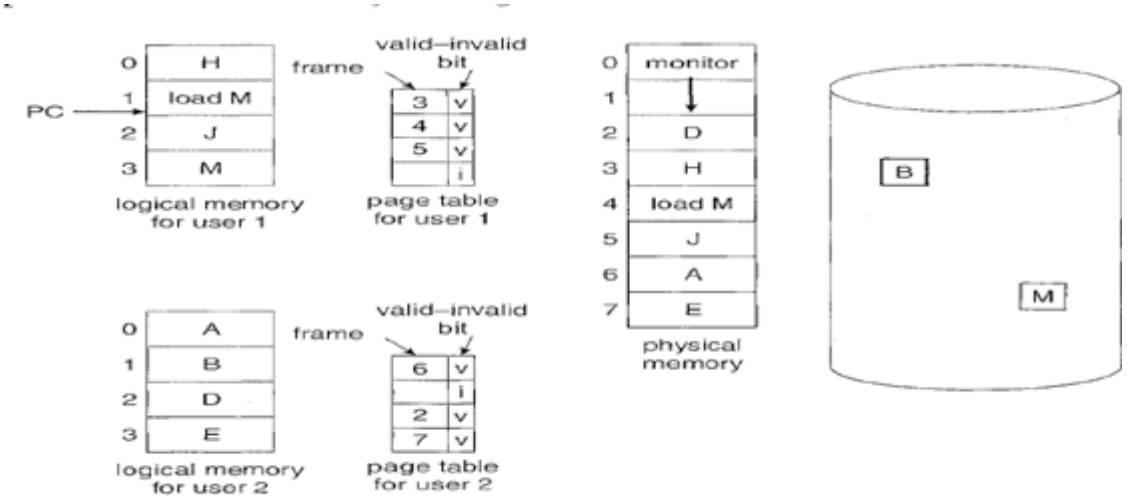
Let p be the probability of a page fault. We would expect p to be close to zero-that is, we would expect to have only a few page faults. The effective access is then effective access time= (1 - p) x ma + p x page fault time.

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    - Wait in a queue for this device until the read request is serviced.
    - Wait for the device seek and/ or latency time.
    - Begin the transfer of the page to a free frame.

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/0 subsystem (I/0 completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

## Page replacement




Need for page replacement.

While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use (Figure 9.9).

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. So this option is not the best choice.

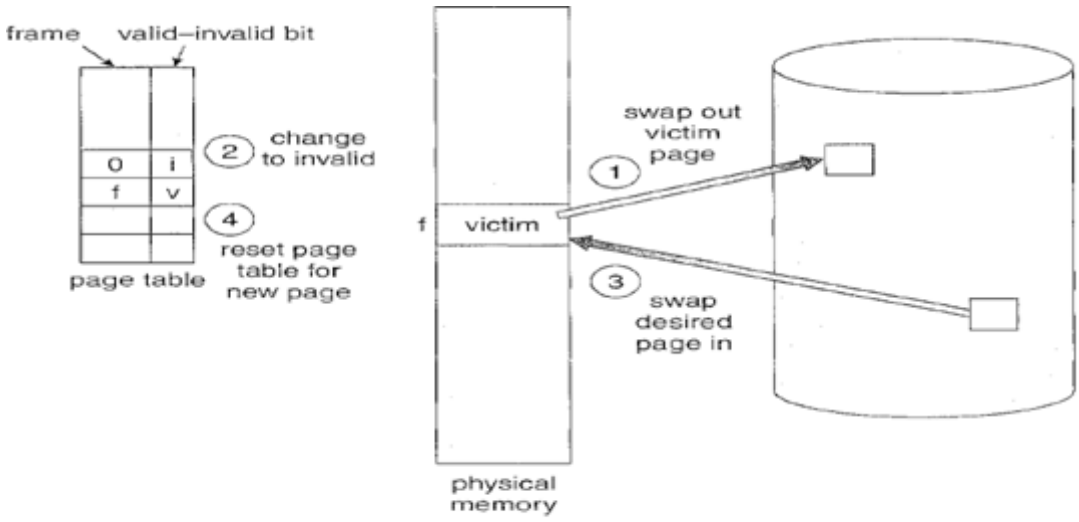
The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.

### Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.10). We can now use the freed frame to hold the page for which the process faulted.

We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame
    c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.




Page replacement.

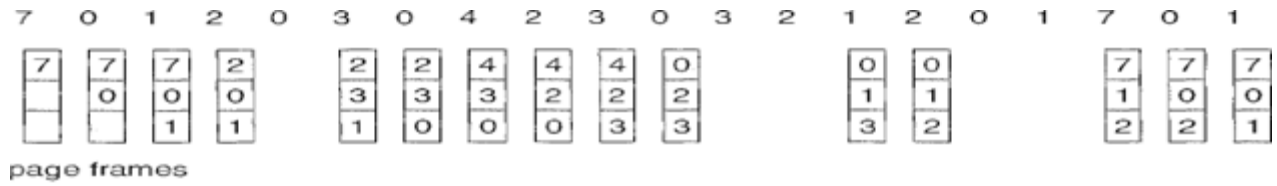
**Page replacement algorithms**

1. **FIFO Page Replacement**
2. **Optimal Page Replacement**
3. **LRU Page Replacement**
4. **LRU-Approximation Page Replacement**
5. **Counting-Based Page Replacement**

**FIFO Page Replacement**

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

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since reference string



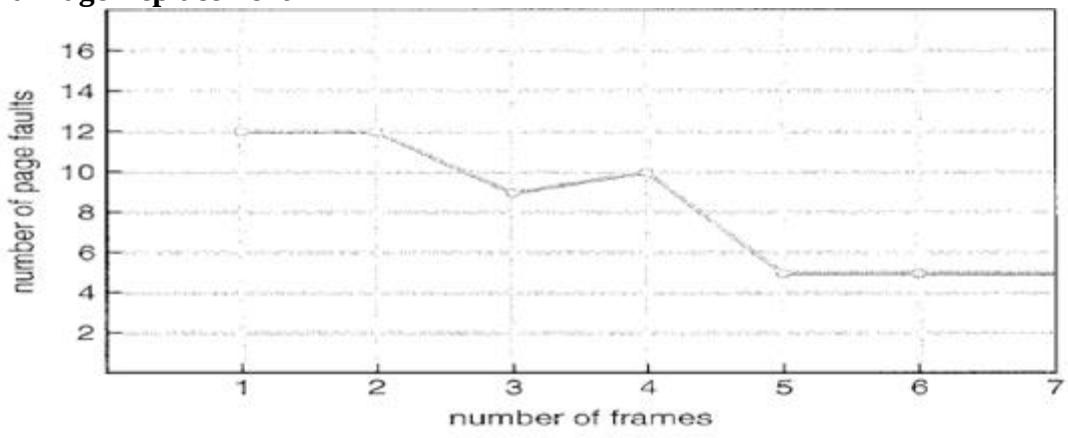

FIFO page-replacement algorithm.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string:

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

The curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as belady's anomaly for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

**Optimal Page Replacement**

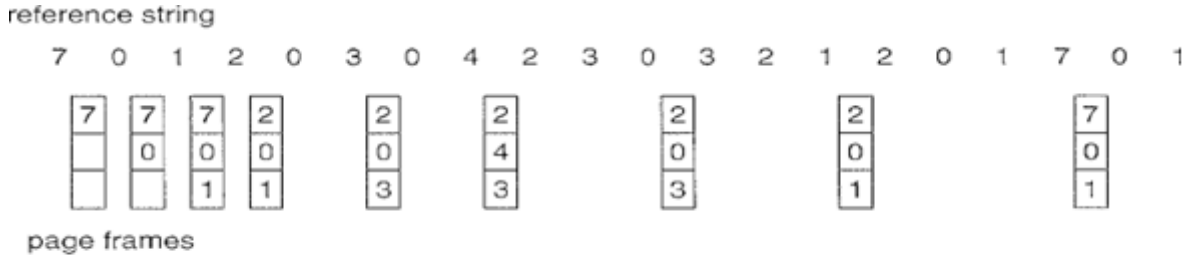



Page-fault curve for FIFO replacement on a reference string.

Belady's anomaly is a problem in FIFO page replacement algorithm so as a result optimal page-replacement algorithm was introduced which has the lowest page-fault rate of all algorithms and will never suffer from belady's anomaly**.**

It follows that

**Replace the page that will not be used for the longest period of time**




Optimal page-replacement algorithm.

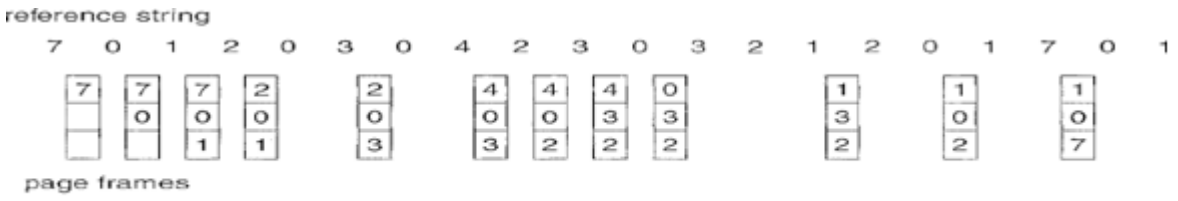
The optimal page-replacement algorithm would yield nine page faults The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

**LRU Page Replacement**

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal

page-replacement algorithm looking backward in time, rather than forward.




LRU page-replacement algorithm.

The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.

When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen. The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page- replacement algorithm may require substantial hardware assistance.

### LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

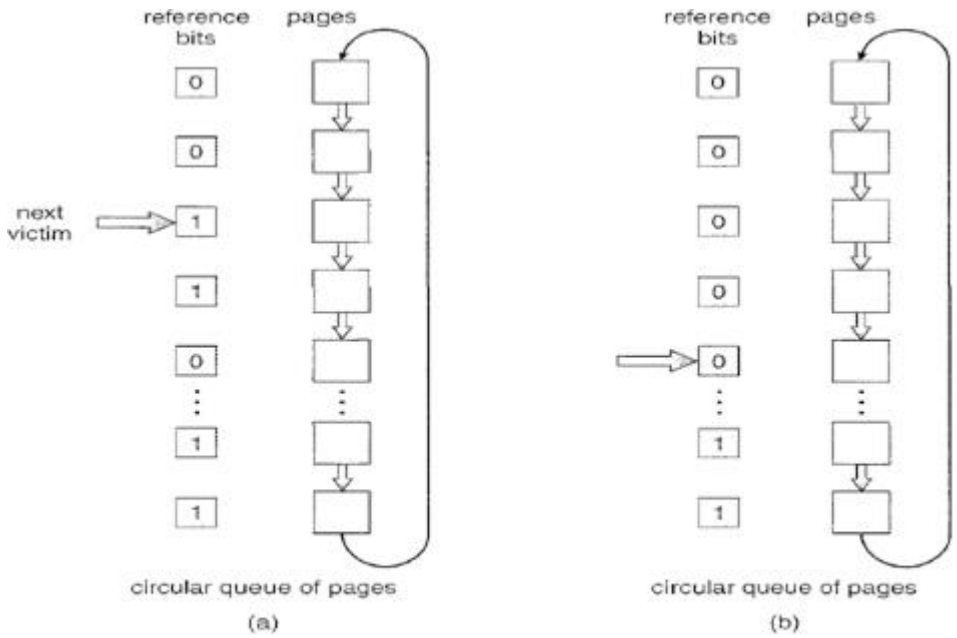
Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use.

### Additional-Reference-Bits Algorithm

By recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

## Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.




Second-chance (clock) page-replacement algorithm.

**Enhanced Second-Chance Algorithm**

We can enhance the second-chance algorithm by considering the reference bit and the modify bit (described in Section 9.4.1) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) Neither recently used nor modified -best page to replace.
2. (0, 1) not recently used but modified.
3. (1, 0) recently used but clean-probably will be used again soon
4. (1, 1) recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

**Counting-Based Page Replacement**

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

**The least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

**The most frequently used (MFU) page-replacement algorithm** is based on the argument that the page with the smallest count was probably just brought in and has yet to be used. As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive.

# File system

### The concept of a file

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.
Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
A file is a named collection of related information that is recorded on secondary storage.

From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.
Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly.
In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general the information in a file is defined by its creator.
Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
A file has a certain defined which depends on its type.

A **text** file is a sequence of characters organized into lines (and possibly pages).

**A source file** is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.

**An object file** is a sequence of bytes organized into blocks understandable by the system's linker.

**An executable file** is a series of code sections that the loader can bring into memory and execute.

## File Attributes

**Name:** The symbolic file name is the only information kept in human readable form. **Identifier.** This unique tag, usually a number, identifies the file within the file system; it is then on-human-readable name for the file.
**Type:** This information is needed for systems that support different types of files. **Location.** This information is a pointer to a device and to the location of the file on that device.
**Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
**Protection:** Access-control information determines who can do reading, writing, executing and so on.
**Time, date and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

## File Operations

- Create
- Write
- Read
- Reposition within file – file seek
- Delete
- Truncate
- 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.
- Appending
- Rename
- Copy

**File types**

| 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 |

Common file types.


**Sequential:** The simplest access method is sequential access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

A read operation-read next-reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation-write next appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward n records for some integer n-perhaps only for n = 1.

## Direct access or relative access

Another method is direct access or relative access. A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

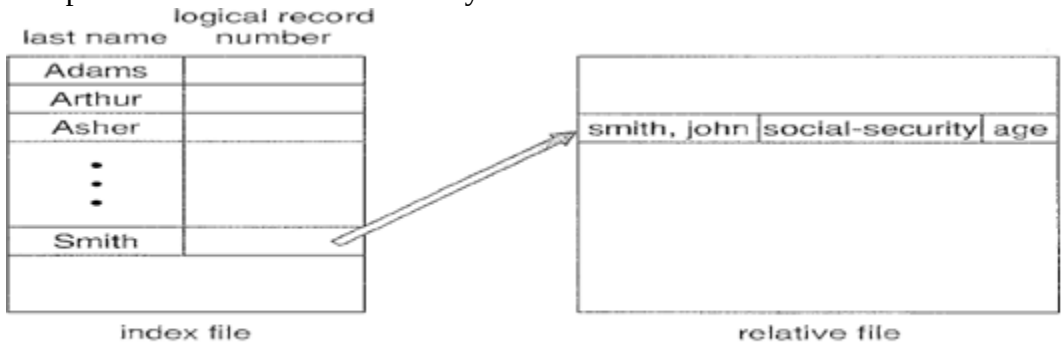
For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and ·write n rather than write next. An alternative approach is to retain read next and write next, as with sequential.

| sequential access | implementation for direct access |
|---|---|
| *reset* | *cp* = 0; |
| *read next* | *read cp*;<br>*cp* = *cp* + 1; |
| *write next* | *write cp*;<br>*cp* = *cp* + 1; |

Simulation of sequential access on a direct-access file.

**Other access methods**

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index like an index in the back of a book contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.




Example of index and relative files.

## Directory and disk structure

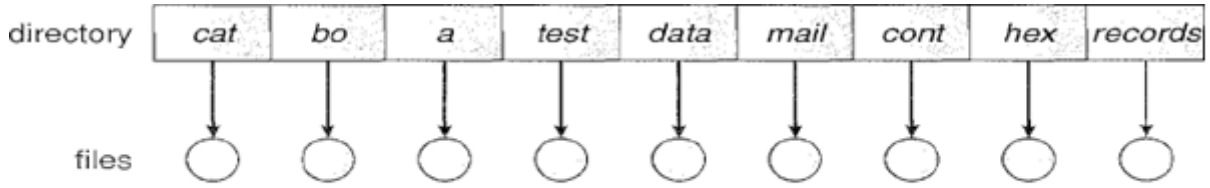
**Directory Overview**

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

## Single-level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file test, then the unique-name rule is violated.

The MS-DOS operating system allows only11-character file names; UNIX, in contrast, allows 255 characters. Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

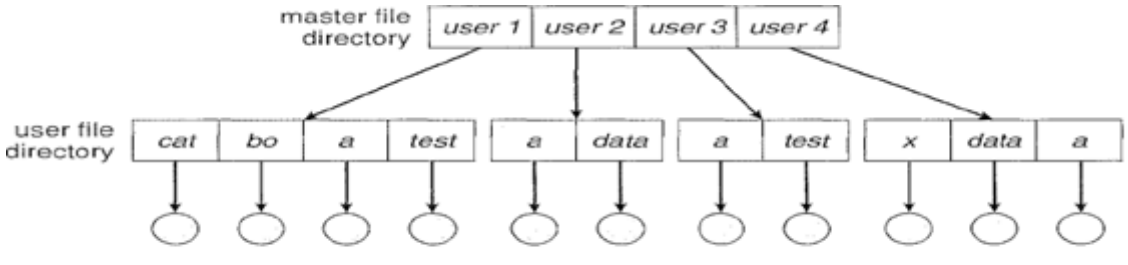



Single-level directory.

## Two-Level Directory

a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **Master File Directory**(MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.




Two-level directory structure.

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

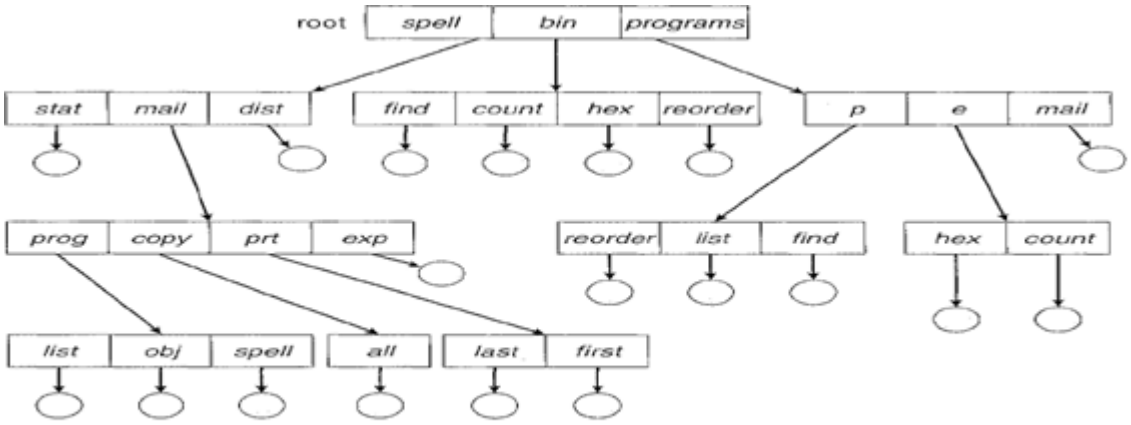
The user directories themselves must be created and deleted as necessary.

A special system program is run with the appropriate user name and account information. Although the two-level directory structure solves the name-collision problem.

## Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height.

This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.



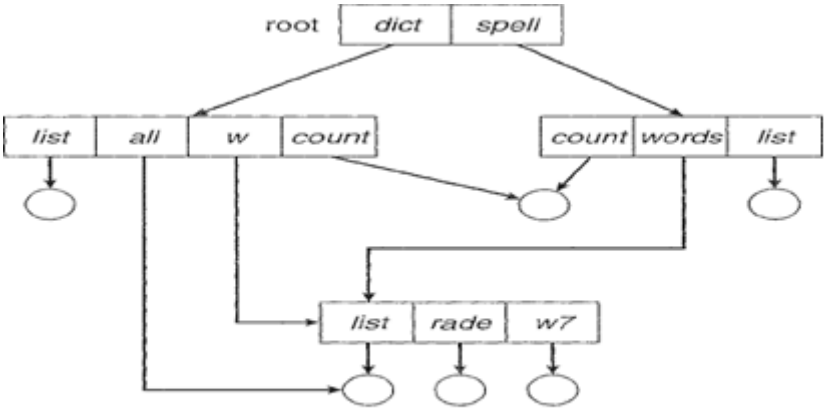

Tree-structured directory structure.

### Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be shared. A shared directory or file will exist in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An acyclic graph-that is, a graph with no cycles-allows directories to share subdirectories and files (Figure 10.11). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

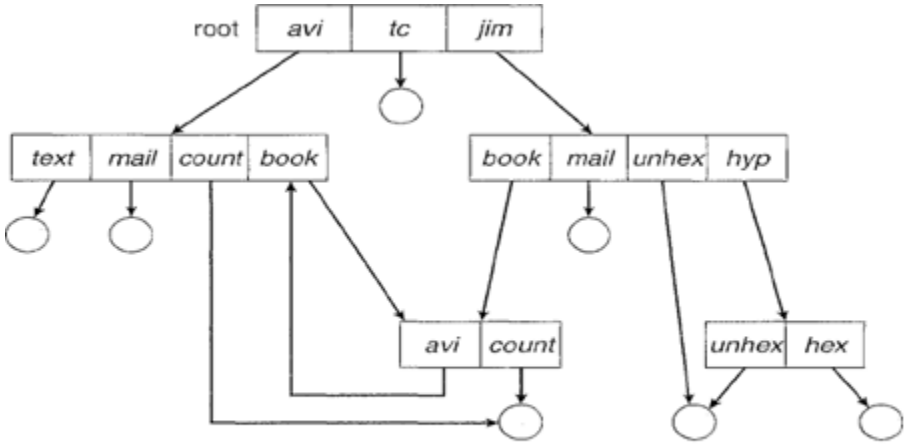
It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories when people are working as a team, all the files they want to share can be put into one directory.




Acyclic-graph directory structure.

**General Graph Directory**

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree- structured nature. However when we add links, the tree structure is destroyed, resulting in a simple graph structure




General graph directory.

**Acyclic Graph Directories:**

- Allow directories to link to one another, allow multiple directories to contain same file i.e., only one copy of the file exists and any change in the file can be viewed by all directories in which it is contained.
- Results in Acyclic graphs
- Two users can name same file or same directory
- Duplicate paths may complicate task of backing up periodically

**General Graph Directories:**

- Allow cycles
- More flexible
- More costly
- Need garbage collection (circular structures)
- There could be a subset of directories all with reference count > 0, but none of these are reachable from the root

**File system structure**

File structure

✦ Logical storage unit

✦ Collection of related information

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

## Allocation Method

The allocation method defines how the files are stored in the disk blocks. The direct access nature of the disks gives us the flexibility to implement the files. In many cases, different files or many files are stored on the same disk. The main problem that occurs in the operating system is that how we allocate the spaces to these files so that the utilization of disk is efficient and the quick access to the file is possible. There are mainly three methods of file allocation in the disk. Each method has its advantages and disadvantages. Mainly a system uses one method for all files within the system.

- Contiguous allocation
- Linked allocation
- Indexed allocation

The main idea behind contiguous allocation methods is to provide

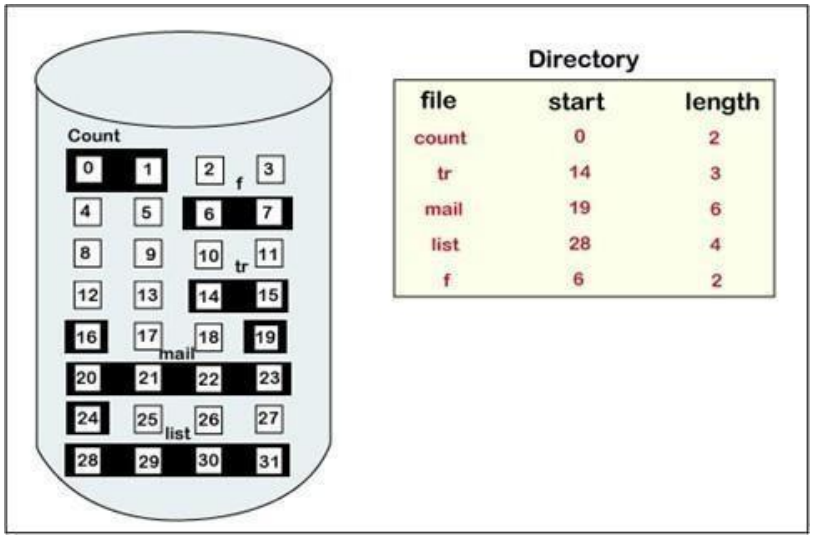
- Efficient disk space utilization
- Fast access to the file blocks

**Contiguous Allocation: -** Contiguous allocation is one of the most used methods for allocation. Contiguous allocation means we allocate the block in such a manner, so that in the hard disk, all the blocks get the contiguous physical block.

We can see in the below figure that in the directory, we have three files. In the table, we have mentioned the starting block and the length of all the files. We can see in the table that for each file, we allocate a contiguous block.

**Example of contiguous allocation**

We can see in the given diagram, that there is a file. The name of the file is 'mail.' The file starts from the 19th block and the length of the file is 6. So, the file occupies 6 blocks in a contiguous manner. Thus, it will hold blocks 19, 20, 21, 22, 23, 24.




The advantages of contiguous allocation are:

1. The contiguous allocation method gives excellent read performance.
2. Contiguous allocation is easy to implement.
3. The contiguous allocation method supports both types of file access methods that are sequential access and direct access.

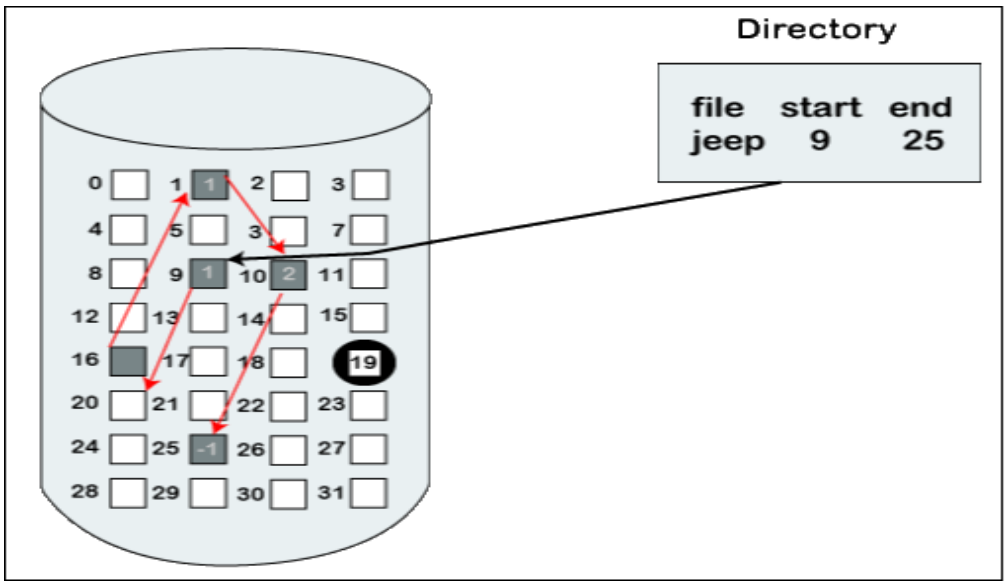
The disadvantages of contiguous allocation method are:

1. In the contiguous allocation method, sometimes disk can be fragmented.
2. In this method, it is difficult to increase the size of the file due to the availability of the contiguous memory block.

## Linked List Allocation

The linked list allocation method overcomes the drawbacks of the contiguous allocation method. In this file allocation method, each file is treated as a linked list of disks blocks. In the linked list allocation method, it is not required that disk blocks assigned to a specific file are in the contiguous order on the disk. The directory entry comprises of a pointer for starting file block

and also for the ending file block. Each disk block that is allocated or assigned to a file consists of a pointer, and that pointer point the next block of the disk, which is allocated to the same file.




**Advantages of Linked list allocation**

1. In liked list allocation, there is no external fragmentation. Due to this, we can utilize the memory better.
2. In linked list allocation, a directory entry only comprises of the starting block address.
3. The linked allocation method is flexible because we can quickly increase the size of the file because, in this to allocate a file, we do not require a chunk of memory in a contiguous form.

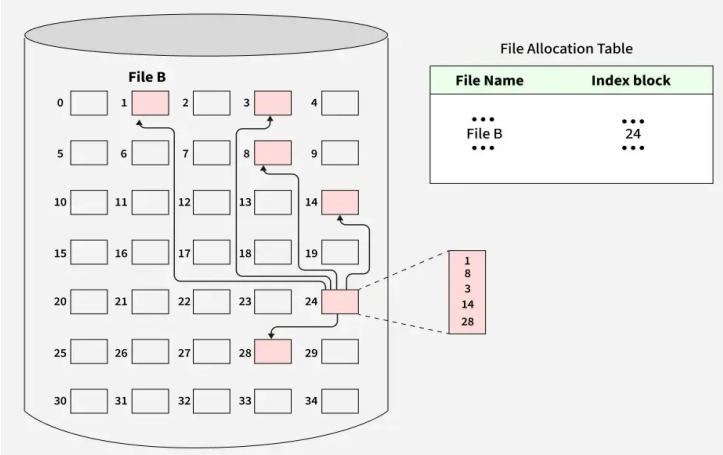
**Disadvantages of Linked list Allocation**

1. Linked list allocation does not support direct access or random access.
2. In linked list allocation, we need to traverse each block.
3. If the pointer in the linked list break in linked list allocation, then the file gets corrupted.
4. In the disk block for the pointer, it needs some extra space.

## Indexed Allocation

The Indexed allocation method is another method that is used for file allocation. In the index allocation method, we have an additional block, and that block is known as the index block. For each file, there is an individual index block. In the index block, the i th entry holds the disk address

of the i th file block. We can see in the below figure that the directory entry comprises of the address of the index block.




**Advantages of Index Allocation**

1. The index allocation method solves the problem of external fragmentation.
2. Index allocation provides direct access.

**Disadvantages of Index Allocation**

1. In index allocation, pointer overhead is more.
2. We can lose the entire file if an index block is not correct.
3. It is totally a wastage to create an index for a small file.

## Disk Scheduling Algorithm

A Process makes the I/O requests to the operating system to access the disk. Disk Scheduling Algorithm manages those requests and decides the order of the disk access given to the requests.

## Why Disk Scheduling Algorithm is needed

Disk Scheduling Algorithms are needed because a process can make multiple I/O requests and multiple processes run at the same time. The requests made by a process may be located at different sectors on different tracks. Due to this, the seek time may increase more. These algorithms help in minimizing the seek time by ordering the requests made by the processes.

## Important Terms related to Disk Scheduling Algorithms

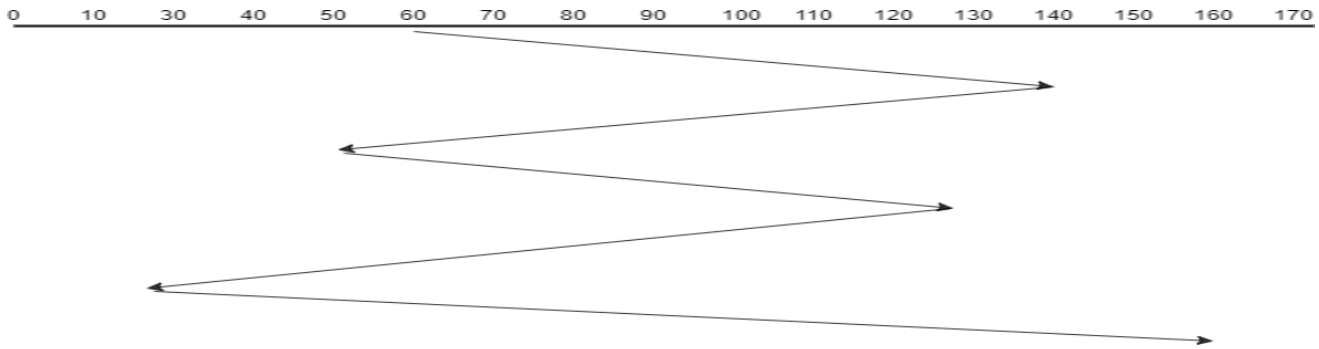
- **Seek Time -** It is the time taken by the disk arm to locate the desired track.
- **Rotational Latency -** The time taken by a desired sector of the disk to rotate itself to the position where it can access the Read/Write heads is called Rotational Latency.
- **Transfer Time -** It is the time taken to transfer the data requested by the processes.
- **Disk Access Time -** Disk Access time is the sum of the Seek Time, Rotational Latency, and Transfer Time.

## Disk Scheduling Algorithms

### First Come First Serve (FCFS)

In this algorithm, the requests are served in the order they come. Those who come first are served first. This is the simplest algorithm.

Eg. Suppose the order of requests are 70, 140, 50, 125, 30, 25, 160 and the initial position of the Read-Write head is 60.

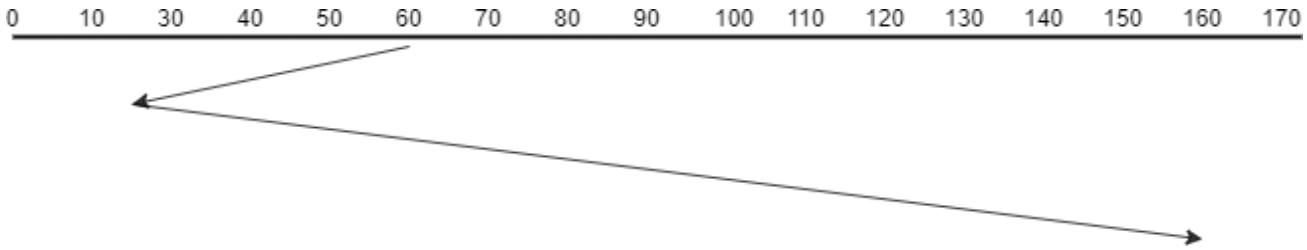



Seek Time = Distance Moved by the disk arm = (140-70)+(140-50)+(125-50)+(125-30)+(30-25)+(160-25)=480

### Shortest Seek Time First (SSTF)

In this algorithm, the shortest seek time is checked from the current position and those requests which have the shortest seek time is served first. In simple words, the closest request from the disk arm is served first.

Eg. Suppose the order of requests are 70, 140, 50, 125, 30, 25, 160 and the initial position of the Read-Write head is 60.

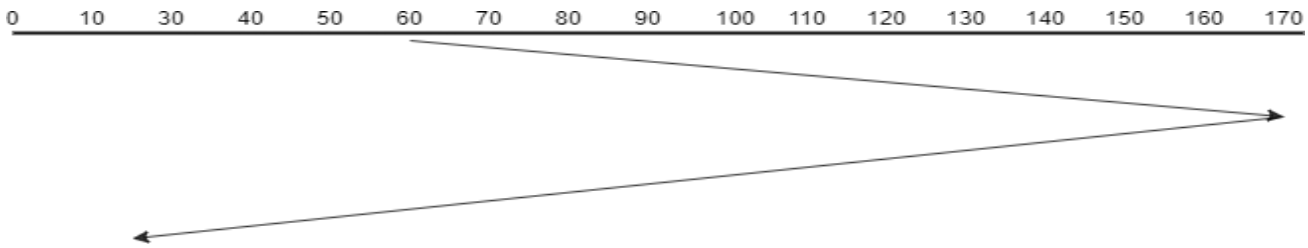


Seek Time = Distance Moved by the disk arm = (60-50)+(50-30)+(30-25)+(70-25)+(125-70)+(140-125)+(160-125)=270

**SCAN (Elevator)**

. In this algorithm, the disk arm moves in a particular direction till the end and serves all the requests in its path, then it returns to the opposite direction and moves till the last request is found in that direction and serves all of them.

Eg. Suppose the order of requests are 70, 140, 50, 125, 30, 25, 160 and the initial position of the Read-Write head is 60. And it is given that the disk arm should move towards the larger value

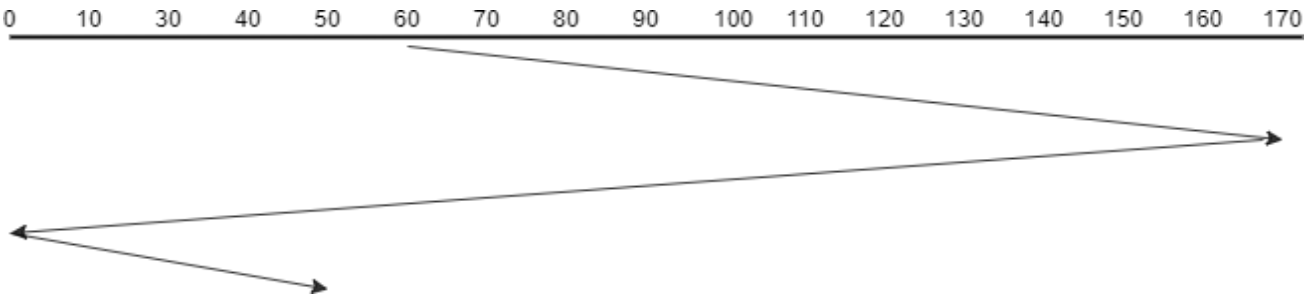



Seek Time = Distance Moved by the disk arm = (170-60)+(170-25)=255

**C-SCAN**

This algorithm is the same as the SCAN algorithm. The only difference between SCAN and C-SCAN is, it moves in a particular direction till the last and serves the requests in its path. Then, it returns in the opposite direction till the end and doesn't serve the request while returning. Then, again reverses the direction and serves the requests found in the path. It moves circularly.

Eg. Suppose the order of requests are 70, 140, 50, 125, 30, 25, 160 and the initial position of the Read-Write head is 60. And it is given that the disk arm should move towards the larger value.



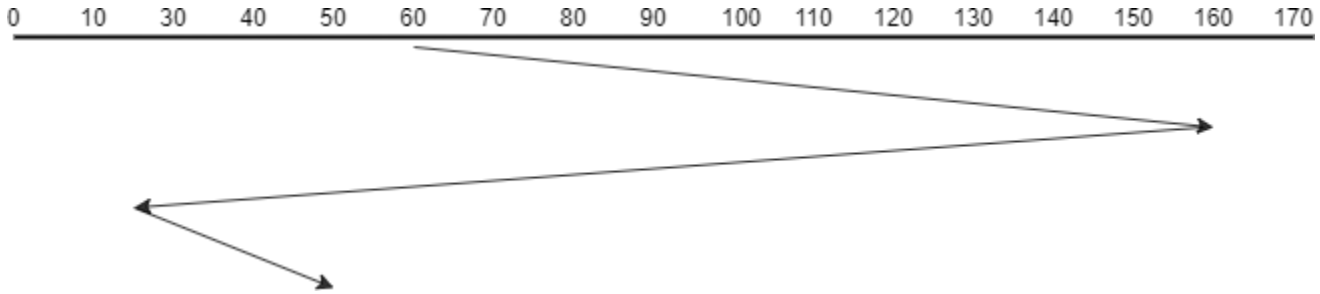
Seek Time = Distance Moved by the disk arm = (170-60)+(170-0)+(50-0)=330

**C-LOOK**

This algorithm is also the same as the LOOK algorithm. The only difference between LOOK and C-LOOK is, it moves in a particular direction till the last request is found and serves the requests in its path. Then, it returns in the opposite direction till the last request is found in that direction and doesn't serve the request while returning. Then, again reverses the direction and serves the requests found in the path. It also moves circularly.

Eg. Suppose the order of requests are 70, 140, 50, 125, 30, 25, 160 and the initial

position of the Read-Write head is 60. And it is given that the disk arm should move towards the larger value.




Seek Time = Distance Moved by the disk arm = (160-60)+(160-25)+(50-25)=260

# UNIT – V

**PROTECTION:** System Protection- Goals of Protection, Principles of Protection, Access Matrix
**SECURITY:** The Security Problem, Program Threats, System and Network Threats, User Authentication, Firewalling.

**Protection** refers to a mechanism which controls the access of programs, processes, or users to the resources defined by a computer system. We can take protection as a helper to multi programming operating system, so that many users might safely share a common logical name space such as directory or files.

**Need of Protection:**

- To prevent the access of unauthorized users and
- To ensure that each active programs or processes in the system uses resources only as the stated policy,
- To improve reliability by detecting latent errors.

**Role of Protection:**

The role of protection is to provide a mechanism that implement policies which defines the uses of resources in the computer system. Some policies are defined at the time of design of the system, some are designed by management of the system and some are defined by the users of the system to protect their own files and programs.

Every application has different policies for use of the resources and they may change over time so protection of the system is not only concern of the designer of the operating system. Application programmer should also design the protection mechanism to protect their system against misuse.

Policy is different from mechanism. Mechanisms determine how something will be done and policies determine what will be done. Policies are changed over time and place to place. Separation of mechanism and policy is important for the flexibility of the system.

## 5.1 Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

## 5.2 Principles of Protection

- The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.

- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.

- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

## 5.3 Domain of Protection

- A computer can be viewed as a collection of *processes* and *objects* ( both HW & SW ).
- The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.

### 5.3.1 Domain Structure

- A ***protection domain*** specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An ***access right*** is the ability to execute an operation on an object.
- A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.

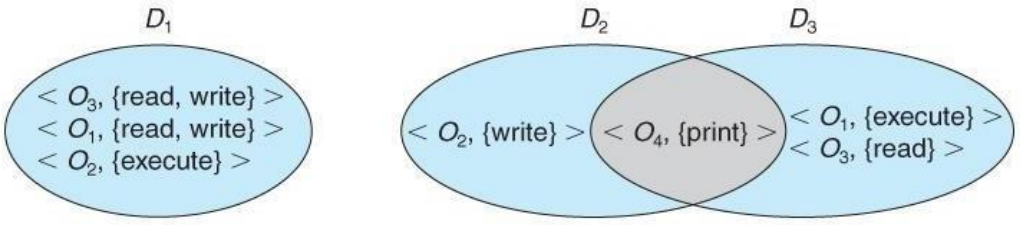



Figure 5.1 - System with three protection domains.

- The association between a process and a domain may be *static* or *dynamic.*
    - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
    - If the association is dynamic, then there needs to be a mechanism for ***domain switching.***
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

### 5.3.2 An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. ( and similarly for the SGID bit. ) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.
- Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

### 5.3.3 An Example: MULTICS

- The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:

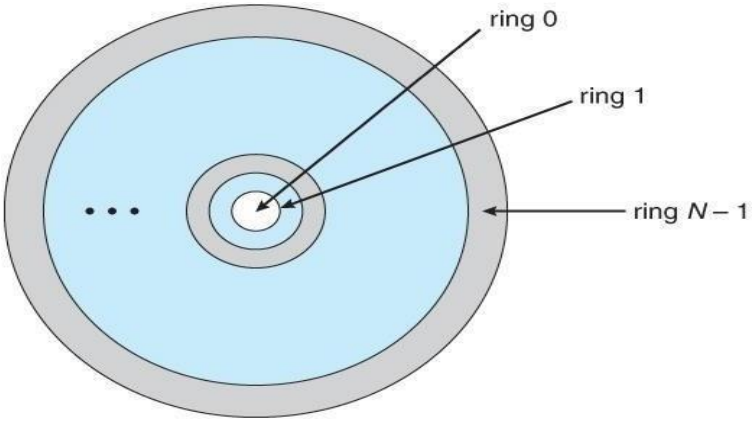



Figure 5.2 - MULTICS ring structure

- Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
- Each file is a memory segment, and each segment description includes an entry that indicates the ring number associated with that segment, as well as read, write, and execute privileges.
- Each process runs in a ring, according to the *current-ring-number*, a counter associated with each process.
- A process operating in one ring can only access segments associated with higher ( farther out ) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.
- Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:
    - An ***access bracket***, defined by integers b1 <= b2.
    - A ***limit*** b3 > b2

    - A ***list of gates,*** identifying the entry points at which the segments may be called.
- If a process operating in ring i calls a segment whose bracket is such that b1 <= i <= b2, then the call succeeds and the process remains in ring i.
- Otherwise a trap to the OS occurs, and is handled as follows:
    - If i < b1, then the call is allowed, because we are transferring to a procedure with fewer privileges. However if any of the parameters being passed are of segments below b1, then they must be copied to an area accessible by the called procedure.
    - If i > b2, then the call is allowed only if i <= b3 and the call is directed to one of the entries on the list of gates.
- Overall this approach is more complex and less efficient than other protection schemes.

## 5.4 Access Matrix

- The model of protection that we have been discussing can be viewed as an ***access matrix,*** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

| domain \ object | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

**Figure 5.3 - Access matrix.**

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

| domain \ object | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | read write | | read write | | switch | | | |

**Figure 5.4 - Access matrix of Figure 5.3 with domains as objects.**

- The ability to ***copy*** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:

- If the asterisk is removed from the original access right, then the right is ***transferred,*** rather than being copied. This may be termed a ***transfer*** right as opposed to a ***copy*** right.
- If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a ***limited copy*** right, as shown in Figure 5.5 below:

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | execute | | write* |
| $D_2$ | execute | read* | execute |
| $D_3$ | execute | | |

(a)

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | execute | | write* |
| $D_2$ | execute | read* | execute |
| $D_3$ | execute | read | |

(b)

**Figure 5.5 - Access matrix with *copy* rights.**

- The ***owner*** right adds the privilege of adding new rights or removing existing ones:

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | owner execute | | write |
| $D_2$ | | read* owner | read* owner write |
| $D_3$ | execute | | |

(a)

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | owner execute | | write |
| $D_2$ | | owner read* write* | read* owner write |
| $D_3$ | | write | write |

(b)

**Figure 5.6 - Access matrix with *owner* rights.**

- Copy and owner rights only allow the modification of rights within a column. The addition of ***control rights***, which only apply to domain objects, allow a

process operating in one domain to affect the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

| object / domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch control |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | write | | write | | switch | | | |

**Figure 5.7 - Modified access matrix of Figure 5.4**

## 5.5 Implementation of Access Matrix

### 5.5.1 Global Table

- The simplest approach is one big global table with < domain, object, rights > entries.
- Unfortunately this table is very large ( even if sparse ) and so cannot be kept in memory ( without invoking virtual memory techniques. )
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

### 5.5.2 Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

### 5.5.3 Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
    - A ***tag,*** possibly hardware implemented, distinguishing this special type of data. ( other types may be floats, pointers, Booleans, etc. )
    - The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program

itself, and used by the operating system for maintaining the process's access right capability list.

### 5.5.4 A Lock-Key Mechanism

- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

### 5.5.5 Comparison

- Each of the methods here has certain advantages or disadvantages, depending on the particular situation and task at hand.
- Many systems employ some combination of the listed methods.

## 5.6 Access Control

- ***Role-Based Access Control, RBAC,*** assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls.
- RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to SUID or SGID programs.

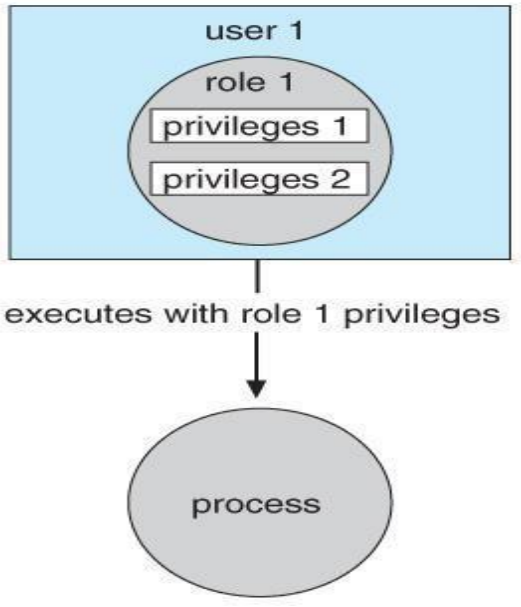



Figure 5.8 - Role-based access control in Solaris

# Security

Security refers to providing a protection system to computer system resources such as CPU, memory, disk, software programs and most importantly data/information stored in the computer system. If a computer program is run by an unauthorized user, then he/she may cause severe damage to computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc.

### 5.7 The Security Problem

- Protection dealt with protecting files and other resources from accidental misuse by cooperating users sharing a system, generally using the computer for normal purposes.

- Security deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.

- Some of the most common types of ***violations*** include:
    - ***Breach of Confidentiality*** - Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.
    - ***Breach of Integrity*** - Unauthorized ***modification*** of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
    - ***Breach of Availability*** - Unauthorized ***destruction*** of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.
    - ***Theft of Service*** - Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.
    - ***Denial of Service, DOS*** - Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.

- One common attack is ***masquerading,*** in which the attacker pretends to be a trusted third party. A variation of this is the ***man-in-the-middle,*** in which the attacker masquerades as both ends of the conversation to two targets.

- A ***replay attack*** involves repeating a valid transmission. Sometimes this can be the entire attack, ( such as repeating a request for a money transfer ), or other times the content of the original message is replaced with malicious content.

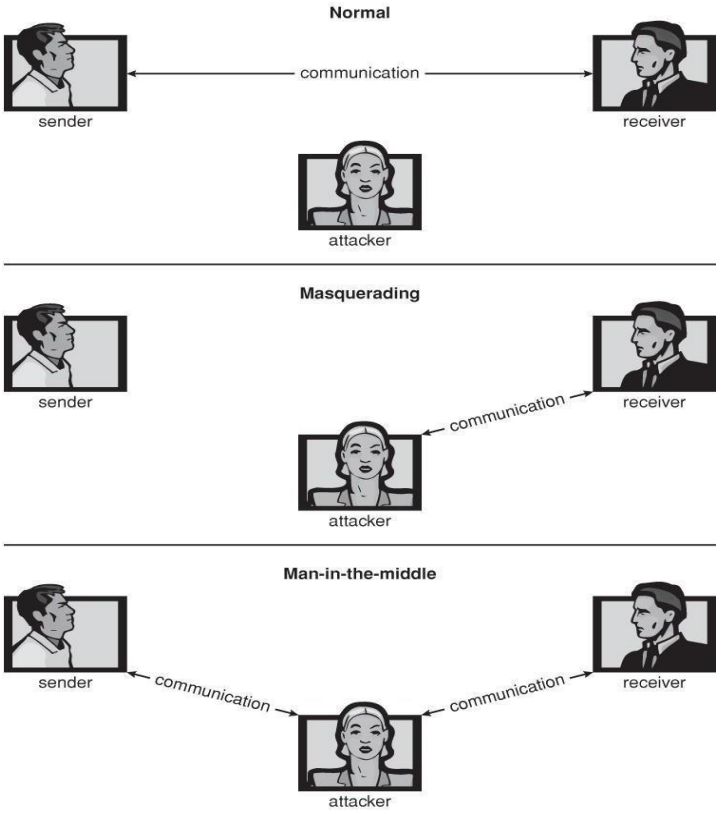



Figure 5.9 - Standard security attacks.

- There are four levels at which a system must be protected:
    1. **Physical -** The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high- speed networking environment provides more and more opportunities for remote attacks.
    2. **Human** - There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via ***social engineering,*** which basically means fooling trustworthy people into accidentally breaching security.
        - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.
        - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. ( Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station. )
        - **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. ( Note: "Good" passwords should involve

a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary ( in any language ), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password. )

3. **Operating System -** The OS must protect itself from security breaches, such as runaway processes ( denial of service ), memory- access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.

4. **Network -** As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. ( Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network. ) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

## 5.8 Program Threats

- There are many common threats to modern systems. Only a few are discussed here.

### 5.8.1 Trojan Horse

- A ***Trojan Horse*** is a program that secretly performs some maliciousness in addition to its visible actions.
- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with ***viruses,*** ( see below. )
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory ( "." ) as part of the path. If a dangerous program having the same name as a legitimate program ( or a common mis-spelling, such as "sl" instead of "ls" ) is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again ( with a proper login prompt ), logs in successfully, and doesn't realize that their information has been stolen.
- **( Special Note to UIC students: Beware that someone has registered the domain name of uic.EU ( without the "D" ), and is running an ssh server which will accept requests to any machine in the domain, and gladly accept your login and password information, without, of course, actually logging you in. Access to this site is blocked from campus, but you are on your own off campus. )**
- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such as Control-Alt-Delete in order to log in. ( This is why

modern Windows systems require the Control-Alt-Delete sequence to commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system. )

- ***Spyware*** is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. ( This is an example of ***covert channels,*** in which surreptitious communications occur. ) Another common task of spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

### 5.8.2 Trap Door

- A ***Trap Door*** is when a designer or a programmer ( or hacker ) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

### 5.8.3 Logic Bomb

- A ***Logic Bomb*** is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.
- A classic example is the ***Dead-Man Switch***, which is designed to check whether a certain person ( e.g. the author ) is logging in every day, and if they don't log in for a long time ( presumably because they've been fired ), then the logic bomb goes off and either opens up security holes or causes other problems.

### 5.8.4 Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[ 1 ] exceeds 256 characters:
    - The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
    - ( The problem could be avoided using str***n***cpy, with a limit of 255 characters copied plus room for the null byte. )

```
#include
#define BUFFER_SIZE 256
int main( int argc, char * argv[ ] )
{
   char buffer[ BUFFER_SIZE ];
   if( argc < 2 )
      return -1;
   else {
      strcpy( buffer, argv[ 1 ] );
      return 0;
   }
```

Figure 5.10 - C program with buffer-overflow condition

- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
  - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
  - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses.
    i.e. an array "grows" towards the bottom of the stack.
  - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.

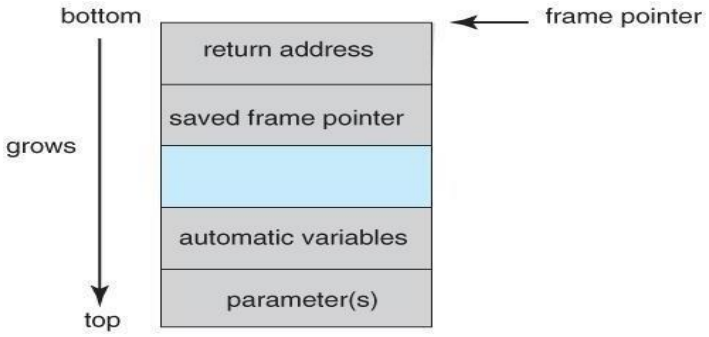



Figure 5.11- The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec( /bin/sh )". To do this requires compiling a program that contains this instruction, and then

using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.

- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location. ( Note, however, that neither the bad code or the padding can contain null bytes, which would terminate the strcpy. )
- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.

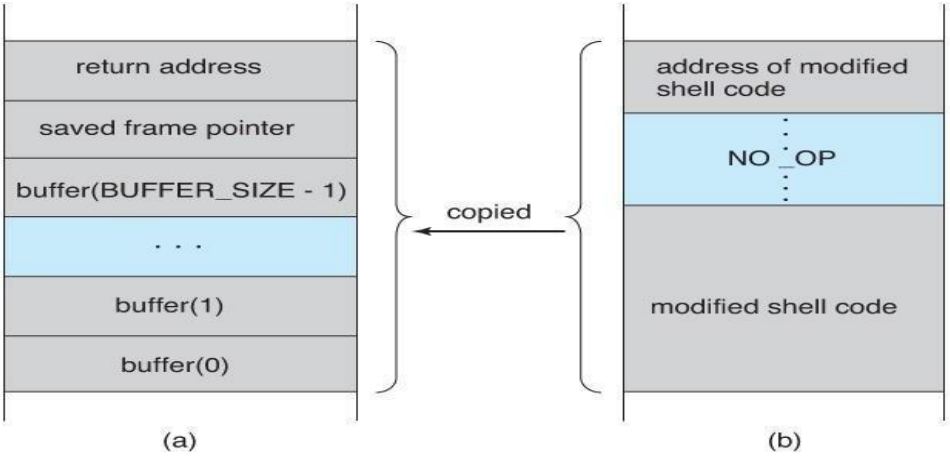



Figure 5.12 - Hypothetical stack frame for Figure 5.10, (a) before and (b) after.

- Unfortunately famous hacks such as the buffer overflow attack are well published and well known, and it doesn't take a lot of skill to follow the instructions and start attacking lots of systems until the law of averages eventually works out. ( ***Script Kiddies*** are those hackers with only rudimentary skills of their own but the ability to copy the efforts of others. )
- Fortunately modern hardware now includes a bit in the page tables to mark certain pages as non-executable. In this case the buffer-overflow attack would work up to a point, but as soon as it "returns" to an address in the data space and tries executing statements there, an exception would be thrown crashing the program.

### 5.8.5 Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself ( by infecting other programs ), and ( eventually ) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures ( such as the boot block. )
- Viruses are delivered to systems in a ***virus dropper,*** usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.

- Viruses take many forms ( see below. ) Figure 15.5 shows typical operation of a boot sector virus:

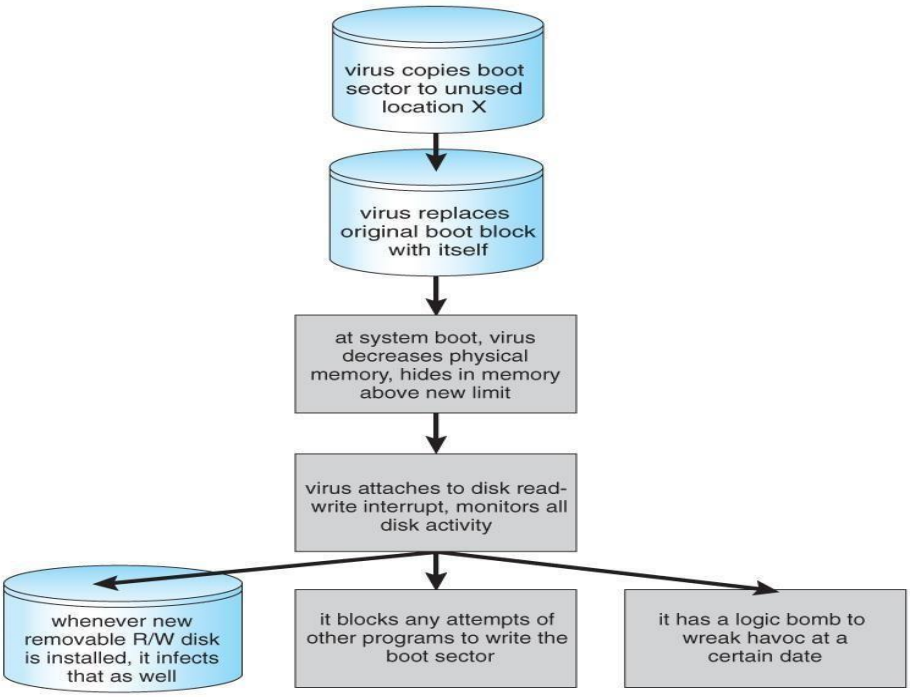



Figure 5.13 - A boot-sector computer virus.

- Some of the forms of viruses include:
  - **File -** A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed ***parasitic,*** because they do not leave any new files on the system, and the original program is still fully functional.
  - **Boot -** A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as ***memory viruses***, because in operation they reside in memory, and do not appear in the file system.
  - **Macro** - These viruses exist as a macro ( script ) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
  - **Source code** viruses look for source code and infect it in order to spread.
  - **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their ***signature,*** by which virus checkers recognize them.
  - **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.
  - **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read( )

system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
    - **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
    - **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
    - **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".
- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers ( including many trusted sites ) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a ***keystroke logger,*** which records users keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a ***monoculture,*** in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

## 5.9 System and Network Threats

- Most of the threats described above are termed ***program threats***, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

### 5.9.1 Worms

- A ***worm*** is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
    1. A small program called a ***grappling hook,*** which was deposited on the target system through one of three vulnerabilities, and

2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

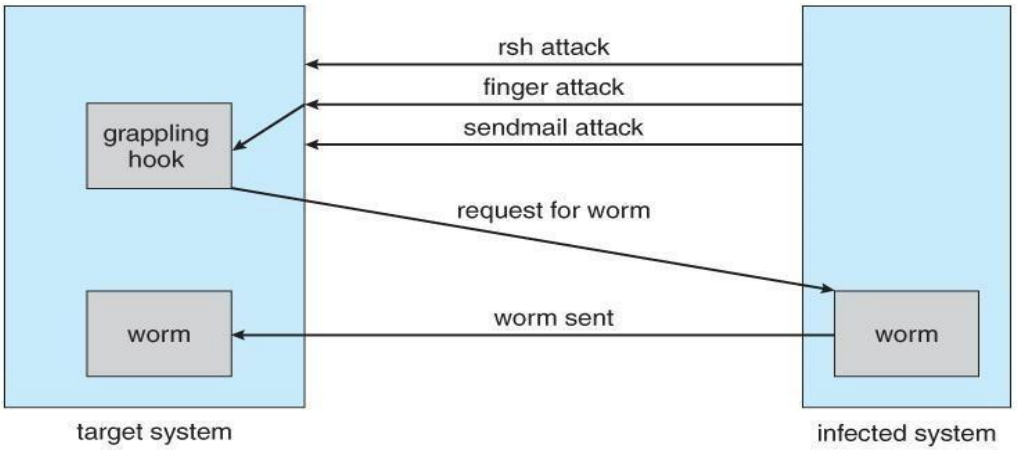



Figure 5.14 - The Morris Internet worm.

- The three vulnerabilities exploited by the Morris Internet worm were as follows:
    1. **rsh ( remote shell )** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers ( with the same account name on both systems ), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that ***any*** user ( except root ) on system A could access the same account on system B without providing a password.
    2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon ( which ran with system privileges ) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.
    3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.
- Once in place, the worm undertook systematic attacks to discover user passwords:
    1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".

2. Then it would try an internal dictionary of 432 favorite password choices. ( I'm sure "password", "pass", and blank passwords were all on the list. )
3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.

- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. ( The seventh was to prevent the worm from being stopped by fake copies. )
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line of the infection- carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

### 5.9.2 Port Scanning

- ***Port Scanning*** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known ( or common or possible ) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.
- Because port scanning is easily detected and traced, it is usually launched from ***zombie systems,*** i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.
- There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are ***nmap*** and ***nessus*** The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

### 5.9.3 Denial of Service

- ***Denial of Service ( DOS )*** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. ( Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing. )
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.
- Sometimes DOS is not the result of deliberate maliciousness. Consider for example:
    - A web site that sees a huge volume of hits as a result of a successful advertising campaign.
    - CNN.com occasionally gets overwhelmed on big news days, such as Sept 11, 2001.
    - CS students given their first programming assignment involving fork( ) often quickly fill up process tables or otherwise completely consume system resources.