UNIT – I

OPERATING SYSTEMS OVERVIEW: Introduction-operating system operations, process management,memory management, storage management, protection and security, System structures-Operating system services, systems calls, Types of system calls, system programs (**T1: Ch-1, 2**) **(1.1-1.9, 2.1-2.5)**

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

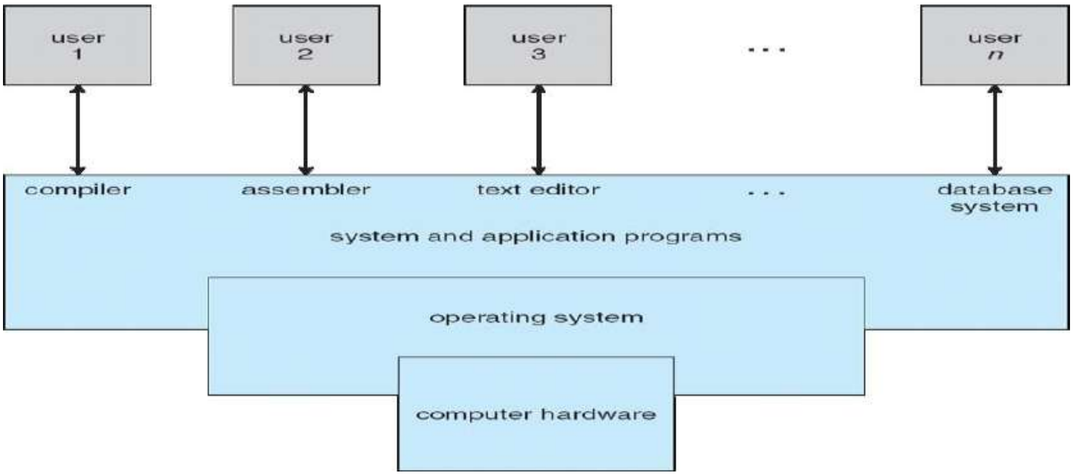
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure

Computer system can be divided into four components

- Hardware – provides basic computing resources
  CPU, memory, I/O devices
- Operating system
  Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  Word processors, compilers, web browsers, database systems, video games
- Users
  People, machines, other computers

Four Components of a Computer System



Operating System Definition

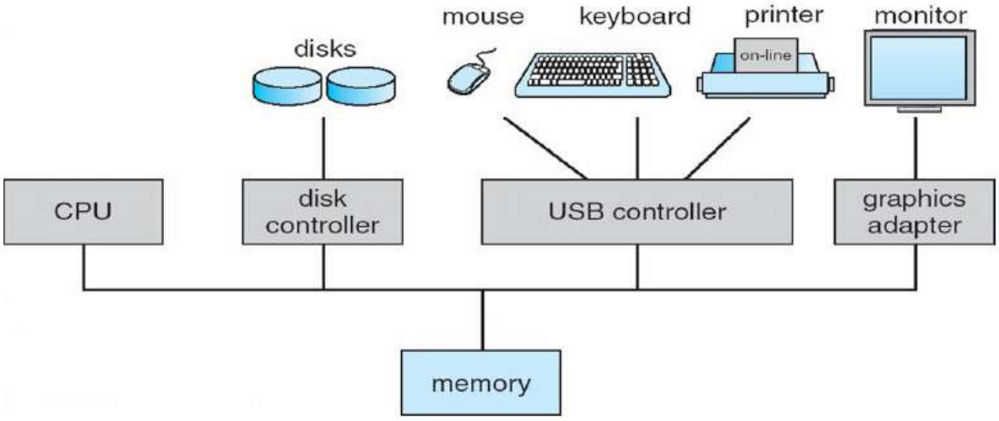
- An operating system as a resource allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources.
- Decides between conflicting requests for efficient and fair resource use
- OS is a **control program** A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.
- "OS is the one program running at all times on the computer" is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program
- **System programs** which are associated with the operating system but are not part of the kernel, and **application programs** which include all programs not associated with the operation of the system.)

Computer Startup

- **bootstrap program** is loaded at power-up or reboot
- Typically stored in ROM or EPROM, generally known as **firmware**
- Initializes all aspects of system
- Loads operating system kernel and starts execution
- Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become system processes, or system daemons that run the entire time the kernel is running. On UNIX, the first system process is "init," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

Computer System Organization

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



Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- Device controller informs CPU that it has finished its operation by causing An *interrupt*
- The occurrence of an event is usually signaled by an **Interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt executing a special operation called a **System call** (also called a **monitor call**)

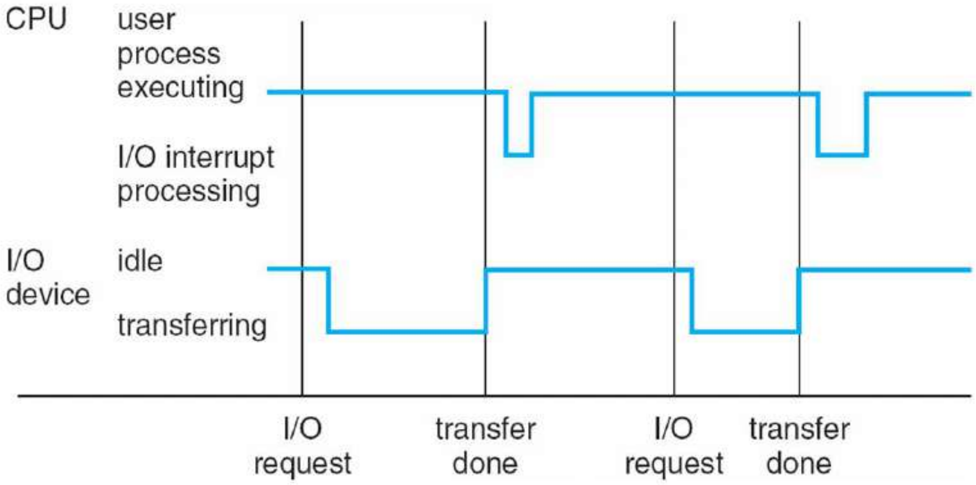
Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the **interruptvector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*. A ***trap*** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Timeline



I/O Structure

- A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the small computer-systems interface (SCSI) controller.
- A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a device driver for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.
- To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information.
- **System call** – request to the operating system to allow user to wait for I/O completion
- **Device-status table** contains entry for each I/O device indicating its type, address, and **state**

- Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt

Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewriteable memory, called main memory (also called or RAM). Main commonly is implemented in a semiconductor technology called **DRAM.**

All forms of memory provide an array of words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

1) Main memory is usually too small to store all needed programs and data permanently.
2) Main memory is a volatile storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage device is a **magnetic disk** which provides storage for both programs and data.

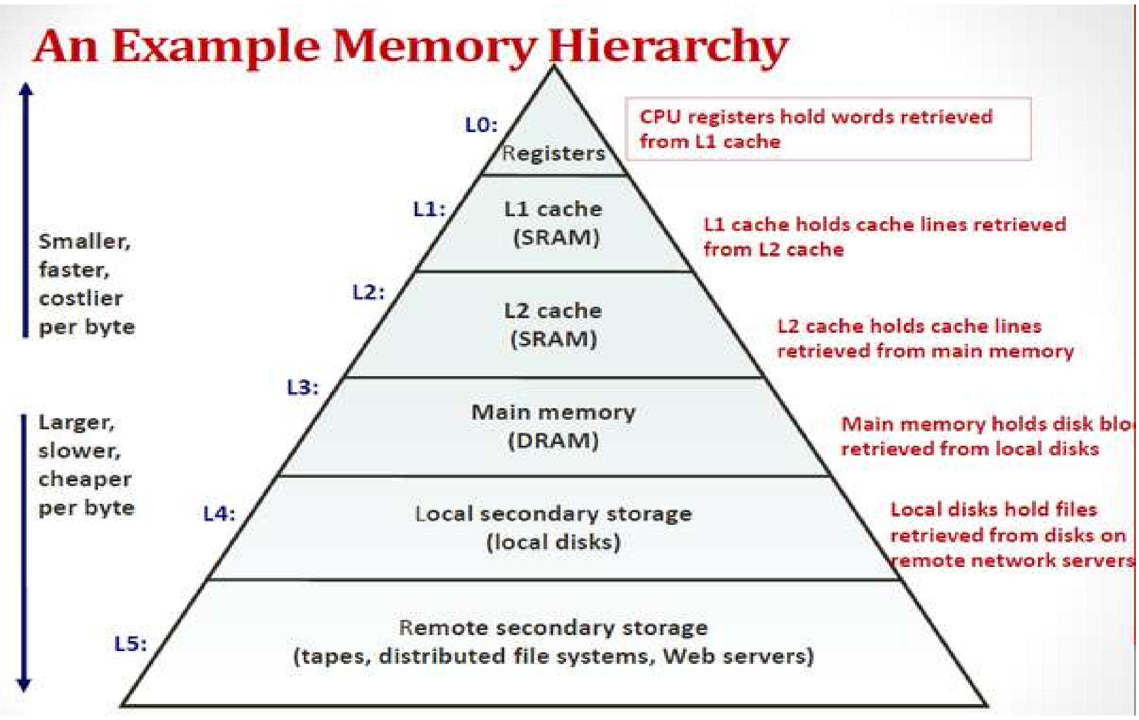
- Main memory – only large storage media that the CPU can access directly
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material

Storage Hierarchy

- Storage systems organized in hierarchy
- Speed

- Cost
- Volatility

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


An Example Memory Hierarchy
Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
L0:
Registers
CPU registers hold words retrieved from L1 cache
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from L2 cache
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from main memory
L3:
Main memory (DRAM)
Main memory holds disk blo retrieved from local disks
L4:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers
L5:
Remote secondary storage (tapes, distributed file systems, Web servers)

Computer-System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
- Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
- Also known as parallel systems, tightly-coupled systems

Advantages include

1.Increased throughput

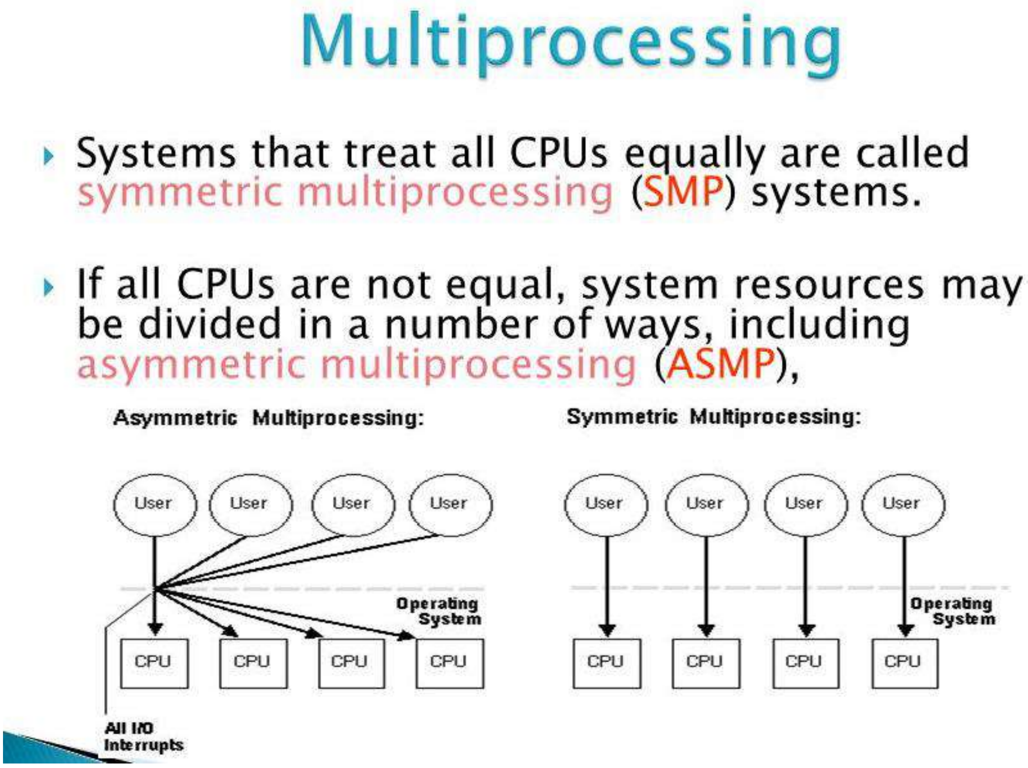
2.Economy of scale

3.Increased reliability – graceful degradation or fault tolerance

Two types

1.Asymmetric Multiprocessing

2.Symmetric Multiprocessing

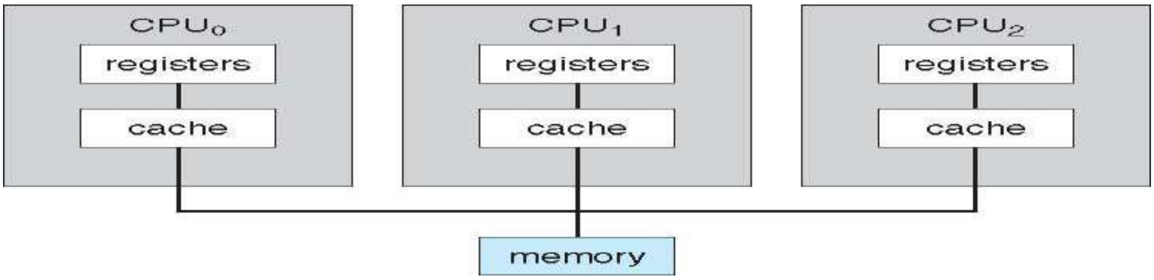


Multiprocessing

- Systems that treat all CPUs equally are called symmetric multiprocessing (SMP) systems.
- If all CPUs are not equal, system resources may be divided in a number of ways, including asymmetric multiprocessing (ASMP),

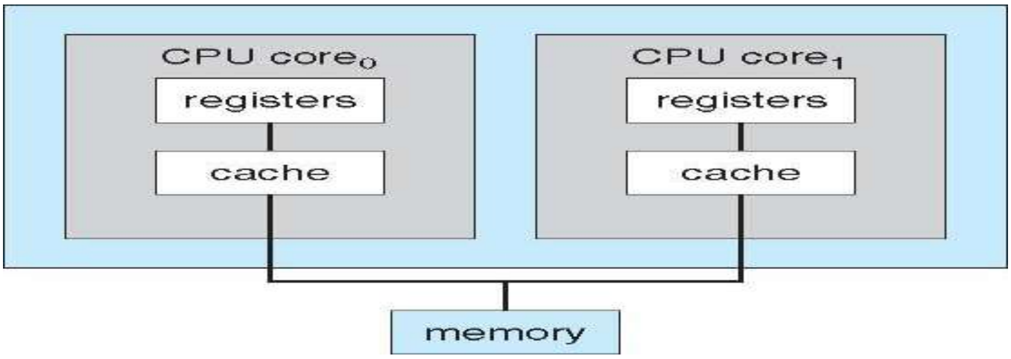
<u>How a Modern Computer Works</u>

Symmetric Multiprocessing Architecture



Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system. Either way, multiprocessing can cause a system to change its memory access model from uniform memory access to non-uniform memory access UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts, creating a performance penalty. Operating systems can minimize the NUMA penalty through resource management.

A Dual-Core Design



We show a dual-core design with two cores on the samechip. In this design, each core has its own register set as well as its own localcache; other designs might use a shared cache or a combination of local andshared caches.

Blade servers are a recent development in which multiple processor boards, I/0 boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

Clustered Systems

Another type of multiprocessor system is a clustered system, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems (loosely coupled). Clustering is usually used to provide high-availability service — that is, service will continue even if one or more systems in the cluster fail.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In symmetric **mode**, two or more hosts are running applications and are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. It does require that more than one application be available to run.

However, applications must be written to take advantage of the cluster by using a technique known as **parallelization** which consists of dividing a program into separate components that run in parallel on individual computers in the cluster.

Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters are usually accomplished by use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a is **distributed lock manager (DLM).**

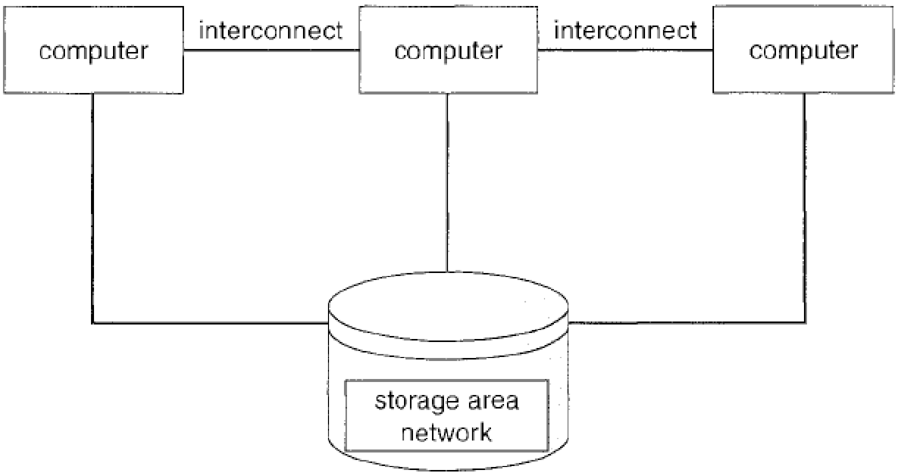
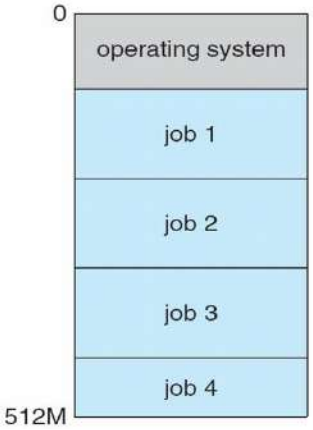


Figure 1.8 General structure of a clustered system.

Operating System Structure

- **Multiprogramming** needed for efficiency
- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to Execute a subset of total jobs in system is kept in memory
- Main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **Job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.
- The operating system picks and begins to execute one of the jobs in memory.
- One job selected and run via **job scheduling**
- When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

Memory Layout for Multi programmed System



Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an interactive computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the response time should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.A program loaded into memory and executing is called a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory,and if there is not enough room for all of them, then the system must chooseamong them. Making this decision involves **job scheduling**.

If several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**. In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk.

A more common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory

Operating-System Operations

Modern operating systems are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap (or an exception)** is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An **interrupt service routine** is provided to deal with the interrupt.

Transition from User to Kernel Mode

At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system. The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems (such as MIPS) have a specific syscall instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting.

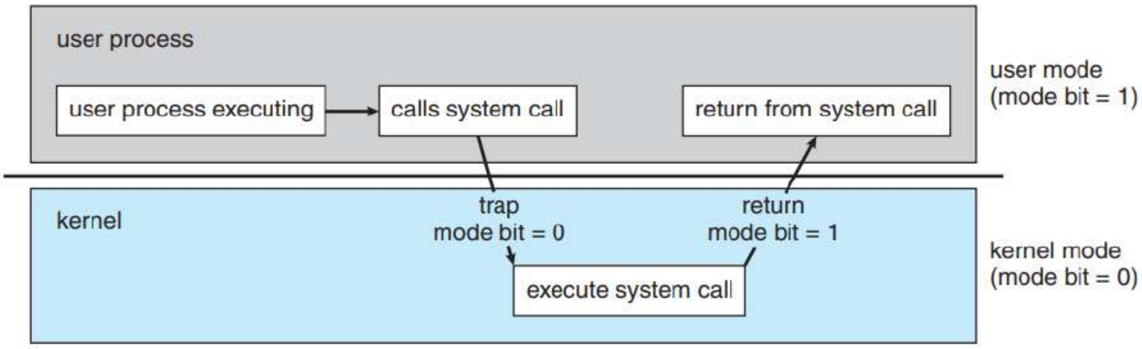


Figure 1.10 Transition from user to kernel mode.

Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond. We can use the timer to prevent a user program from running too long.

Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one).

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along.

A single-threaded process has one program counter specifying the next instruction to execute. The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code).

The operating system is responsible for the following activities in connection with process management:

Process Management Activities

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
- Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs

Memory Management

The main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.

Memory management activities

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. The operating system maps files onto physical media and accesses these files via the storage devices.

File-System Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.

OS File Management activities

- Creating and deleting files and directories
- Primitives to manipulate files and directories
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

Mass-Storage Management

The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Magnetic tape drives and their tapes and CD and DVD drives and platters are typical tertiary storage devices.

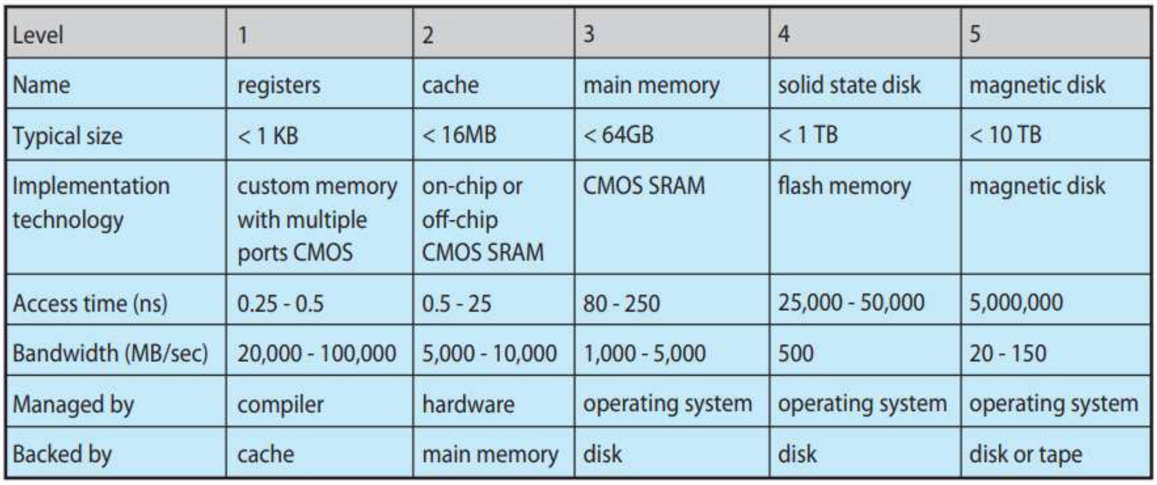
Caching

When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory.Because caches have limited size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory.



Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.11 Performance of various levels of storage.

I/O Subsystem

The I/O subsystem consists of several components:

• A memory-management component that includes buffering, caching, and spooling

• A general device-driver interface

• Drivers for specific hardware devices

Protection and Security

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system.

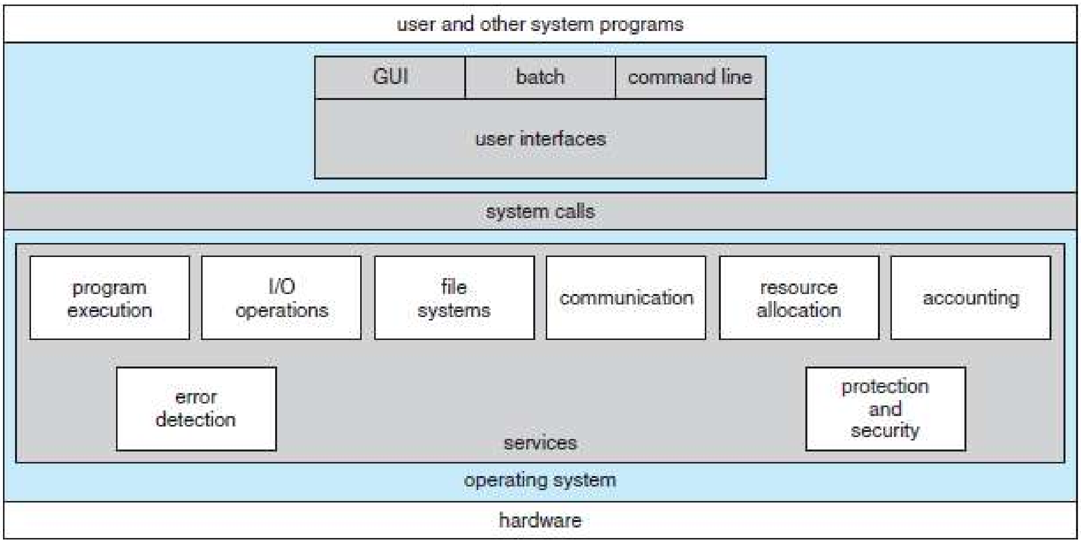
Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated user identifiers (user IDs). In Windows parlance, this is a security ID (SID). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and group identifiers. A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread.

- Systems generally first distinguish among users, to determine who can do what
- User identities (**user IDs**, security IDs) include name and associated number, one per user
- User ID then associated with all files, processes of that user to determine access control
- Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
- **Privilege escalation** allows user to change to effective ID with more rights

Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes.



A view of operating system services.

User interface. Almost all operating systems have a **user interface** (**UI**). This interface can take several forms. One is a **command-line interface** (**CLI**), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface** (**GUI**) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

- **Program execution**. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

- **File-system manipulation**. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership.
- **Communications**. There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
- **Error detection**. The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.
- **Resource allocation.** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors.
- **Accounting**. We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.
- **Protection and security**. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with

the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled.

User and Operating-System Interface

One provides a command-line interface, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI. interpreters are known as **shells**. For example, on UNIX and Linux systems, a user may choose among several different shells, including the ***Bourne shell***, ***C shell***, ***Bourne-Again shell***, ***Korn shell***, and others.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way.

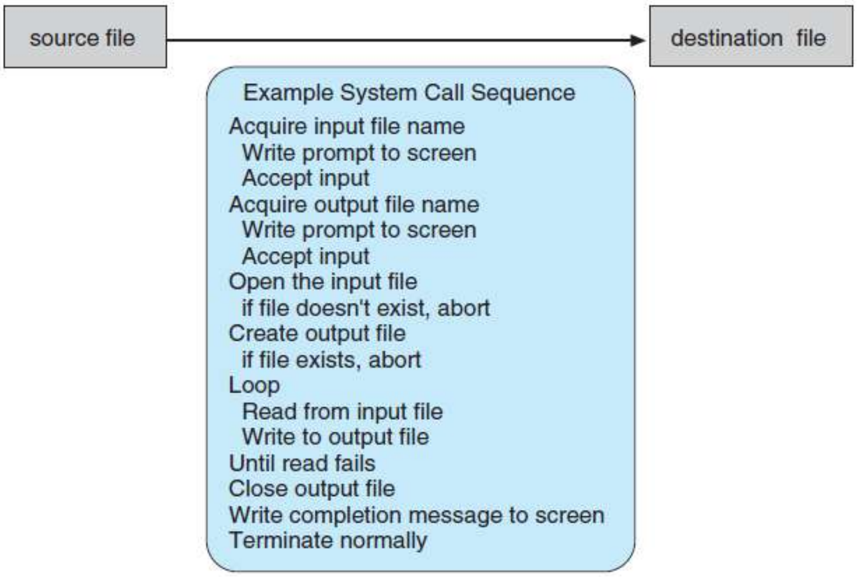
System Calls

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 (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions. 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.

Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface** **(API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OSX), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**.

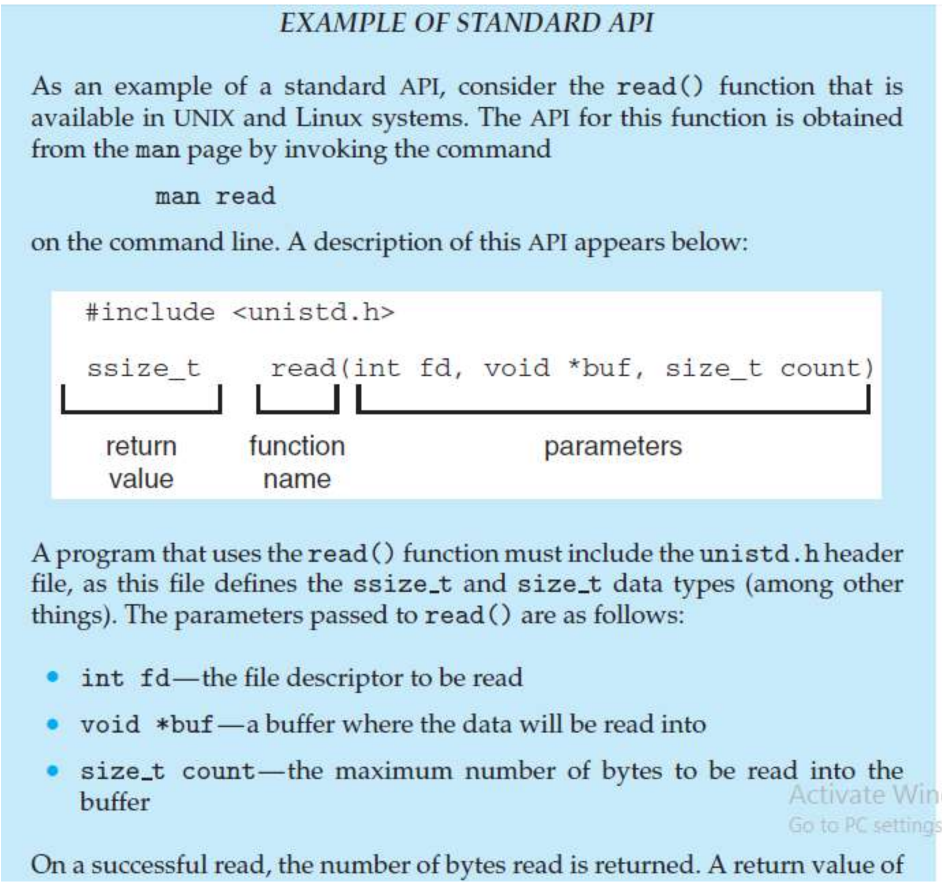
Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function

CreateProcess() (which unsurprisingly is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel.



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Example of how system calls are used.



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

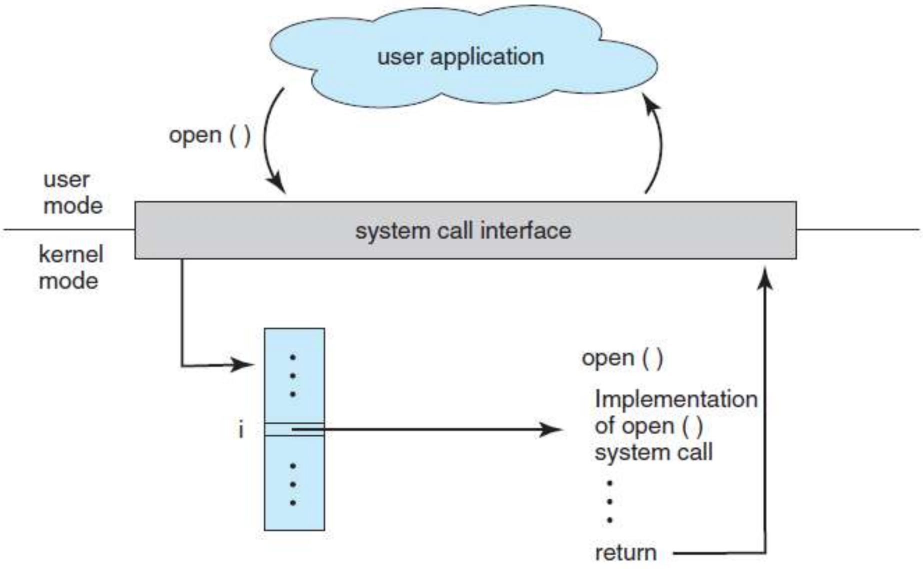
return value
function name
parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of

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



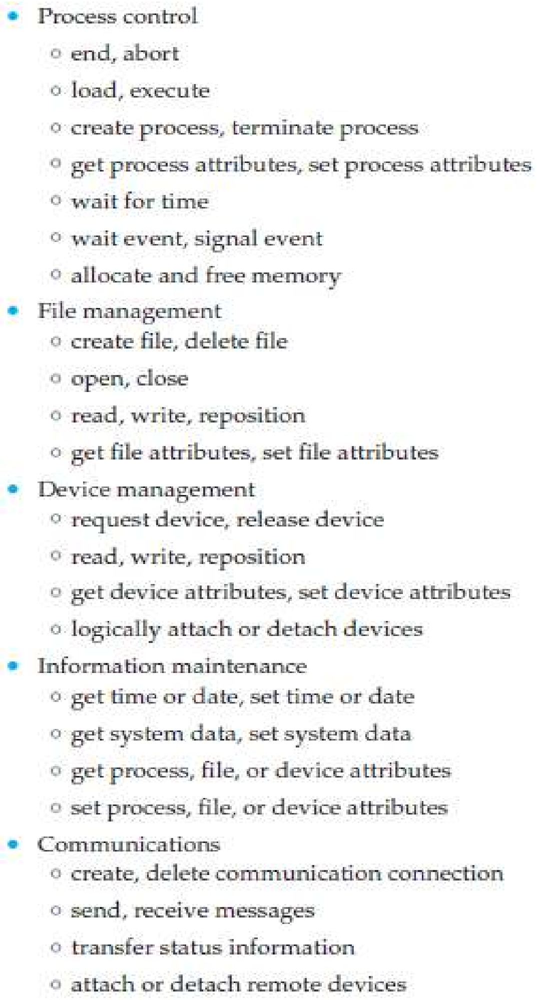
The handling of a user application invoking the open() system call.

Types of System Calls

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

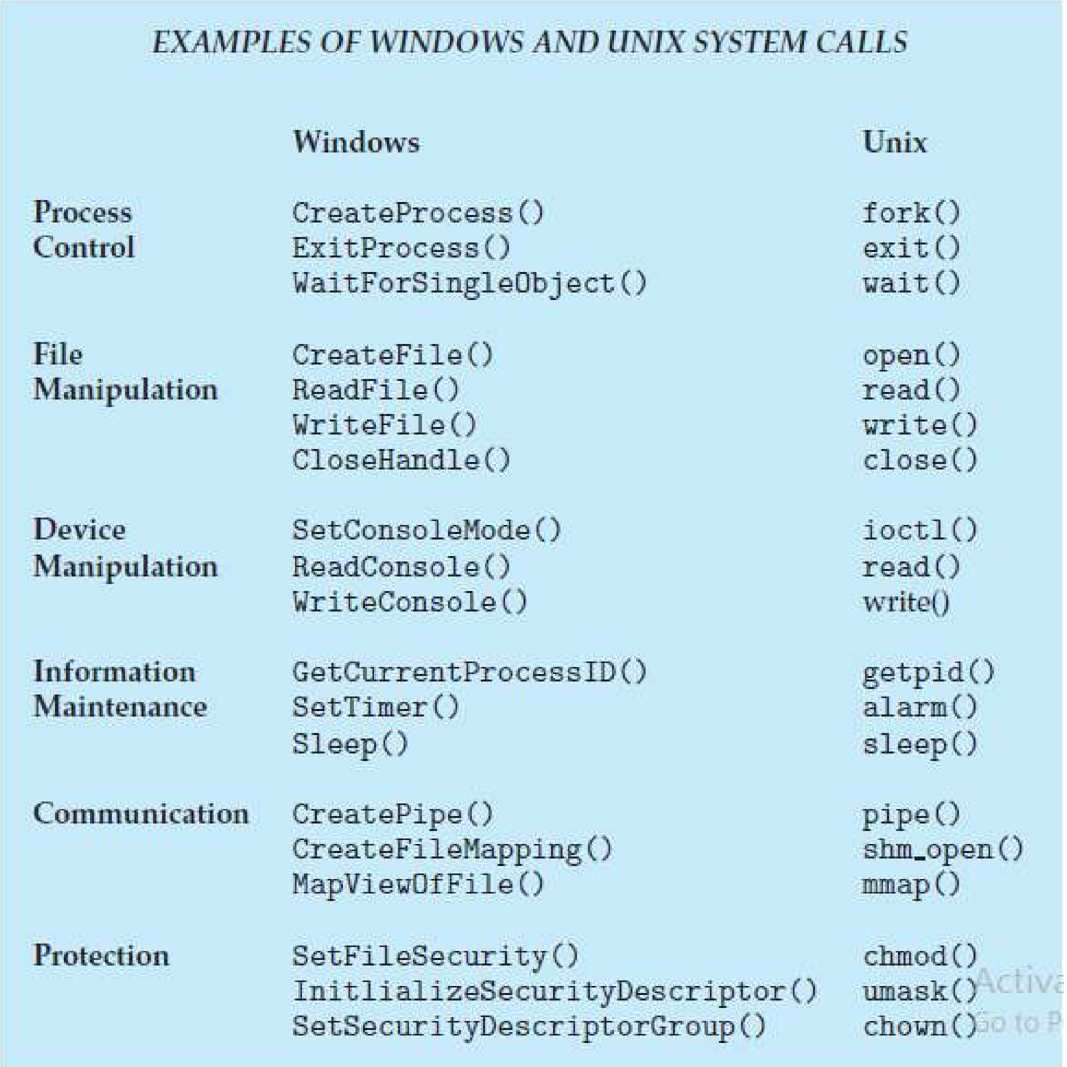
Process Control

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. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.



- 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

Figure 2.8 Types of system calls.



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

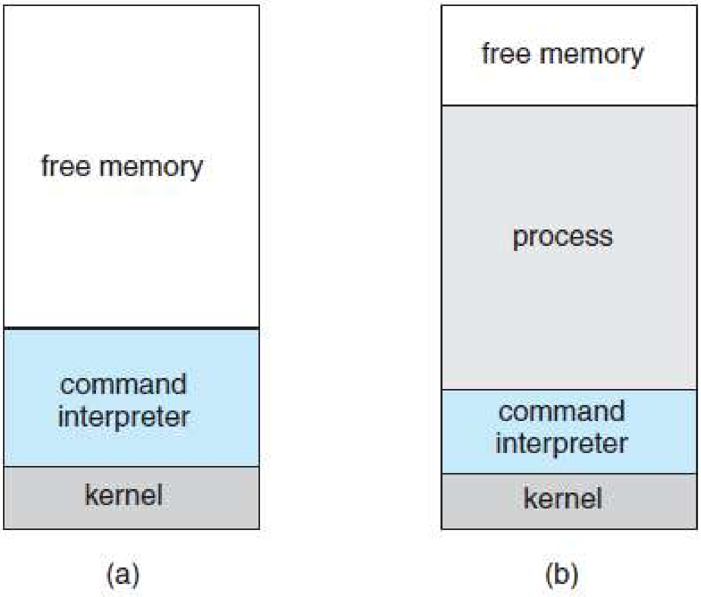
File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`



MS-DOS execution. (a) At system startup. (b) Running a program.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10).

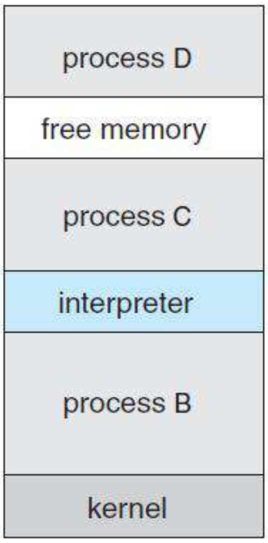


Figure 2.10 FreeBSD running multiple programs.

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background."

File Management

We first 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() (rewind or skip 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

Process may need several resources to execute—main memory, disk drives, access to 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() a device, to ensure exclusive use of it. After we are finished with the device, we release() it. Once the device has been requested (and allocated to us), 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. This provision is useful for debugging.

Communication

There are two common models of inter process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened.

Each computer in a network has a host name by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The get hostid() and get processid() system calls do this translation. The identifiers are then passed to the generalpurpose open() and close() calls provided by the file system or to specific open connection() and close connection() system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an accept connection() call.

In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes.

Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Typically, system calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as files and disks. The allow user() and deny user() system calls specify whether particular users can—or cannot—be allowed access to certain resources.

System Programs

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

- **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information:** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information.
- **File modification:** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support:** Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.
- **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Background services:** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such application programs include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

UNIT – II

PROCESS MANAGEMENT: Process concepts- Operations on processes, IPC, Process Scheduling (**T1: Ch-3).**

PROCESS COORDINATION: Process synchronization- critical section problem, Peterson's solution, synchronization hardware, semaphores, classic problems of synchronization, readers and writers problem, dining philosopher's problem, monitors **(T1: Ch-5**).

Introduction

Process:

A **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system. A system therefore consists of a collection of processes: operating system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

A process is also known as job or task. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and 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 Figure 3.1.

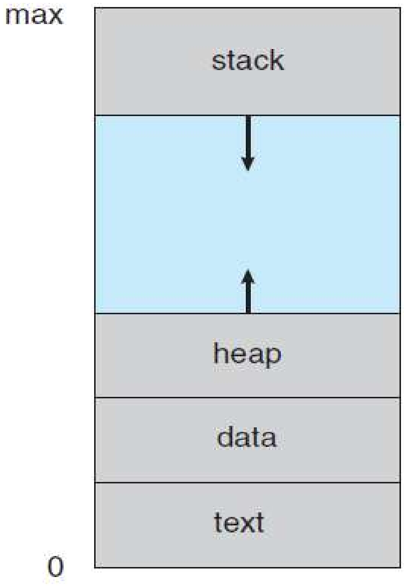


Figure 3.1 Process in memory.

We emphasize that 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**). In contrast, 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.

Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O

completion or reception of a signal).

- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting,* however. The state diagram corresponding to these states is presented in Figure 3.2.

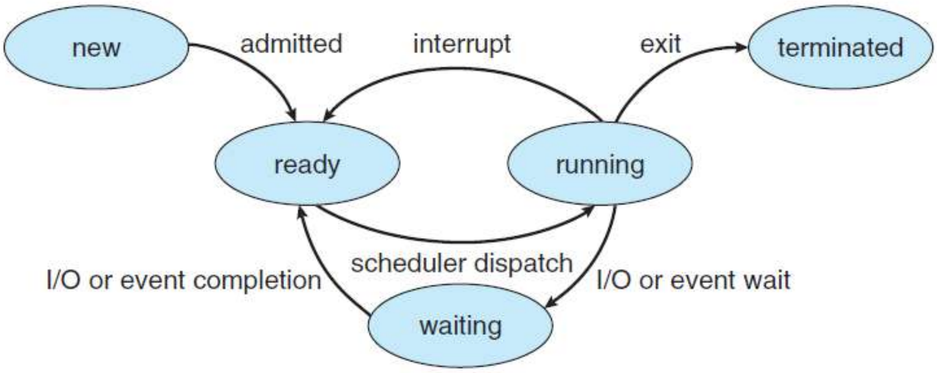


Figure 3.2 Diagram of process state.

2.3 Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

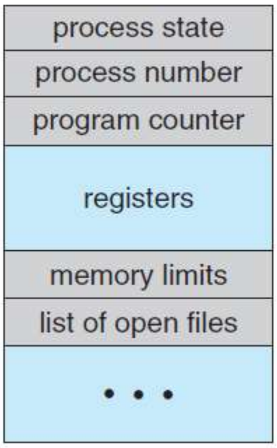


Figure 3.3 Process control block (PCB).

Process state. The state may be new, ready, running, and waiting, halted, 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 (Figure 3.4).
- **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 such items as the value of the base and limit registers and 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.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

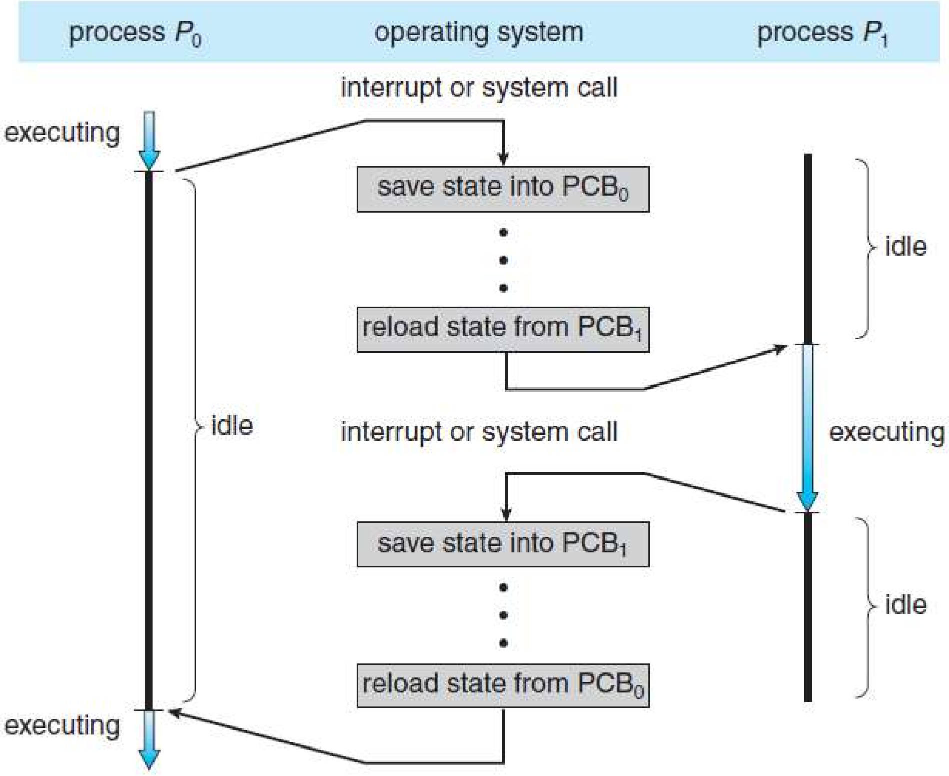
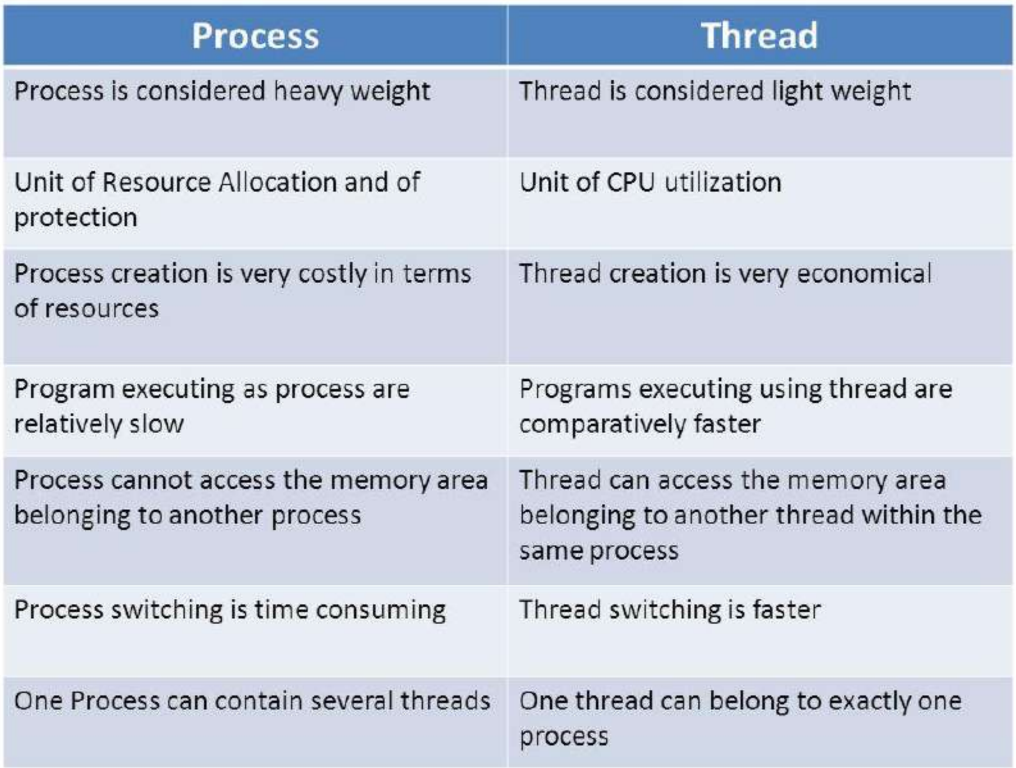


Figure 3.4 Diagram showing CPU switch from process to process.

Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.



Process	Thread
Process is considered heavy weight	Thread is considered light weight
Unit of Resource Allocation and of protection	Unit of CPU utilization
Process creation is very costly in terms of resources	Thread creation is very economical
Program executing as process are relatively slow	Programs executing using thread are comparatively faster
Process cannot access the memory area belonging to another process	Thread can access the memory area belonging to another thread within the same process
Process switching is time consuming	Thread switching is faster
One Process can contain several threads	One thread can belong to exactly one process

2.4 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically.

Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes. Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system. A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly.

2.2 Inter process Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an **inter process communication** (**IPC**) mechanism that will allow them to exchange data and information. There are two fundamental models of inter process communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

The two communications models are contrasted in Figure 3.12.

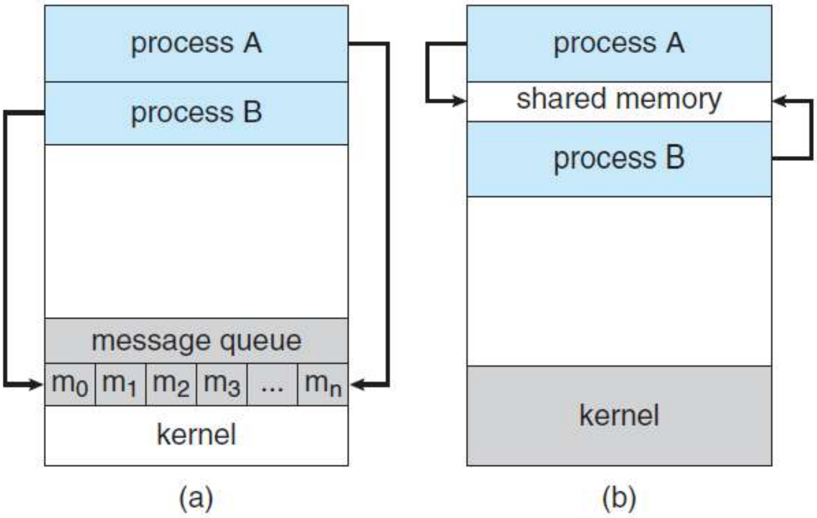
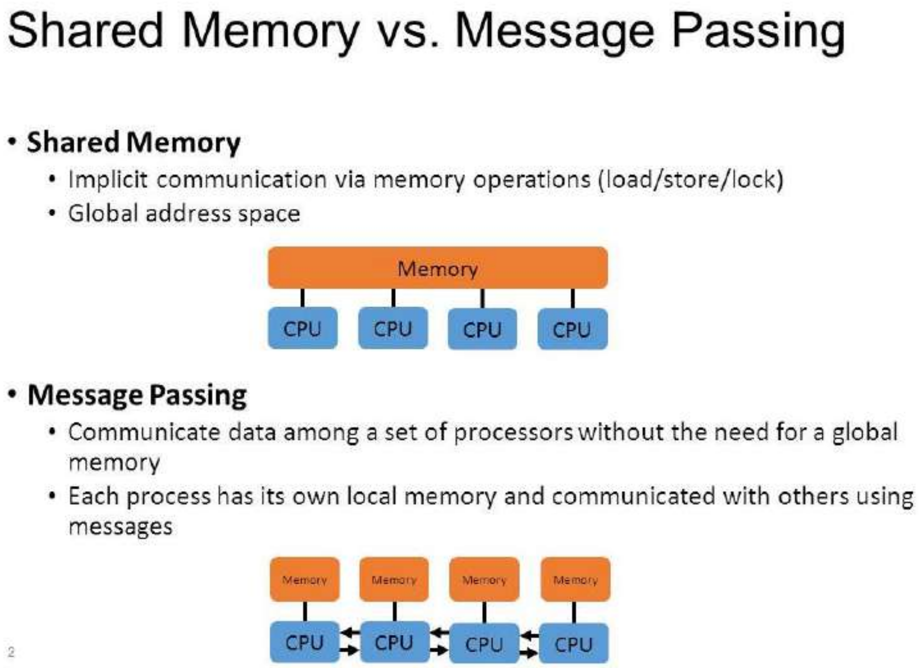


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.



Shared Memory vs. Message Passing

- Shared Memory
  - Implicit communication via memory operations (load/store/lock)
  - Global address space

- Message Passing
  - Communicate data among a set of processors without the need for a global memory
  - Each process has its own local memory and communicated with others using messages

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages. A message-passing facility provides at least two operations:

send(message) receive(message)

Messages sent by a process can be either fixed or variable in size. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

2.3 Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that 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 list of processes waiting for a particular I/O device is called a **device queue**.

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 3.6. 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 **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child'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.

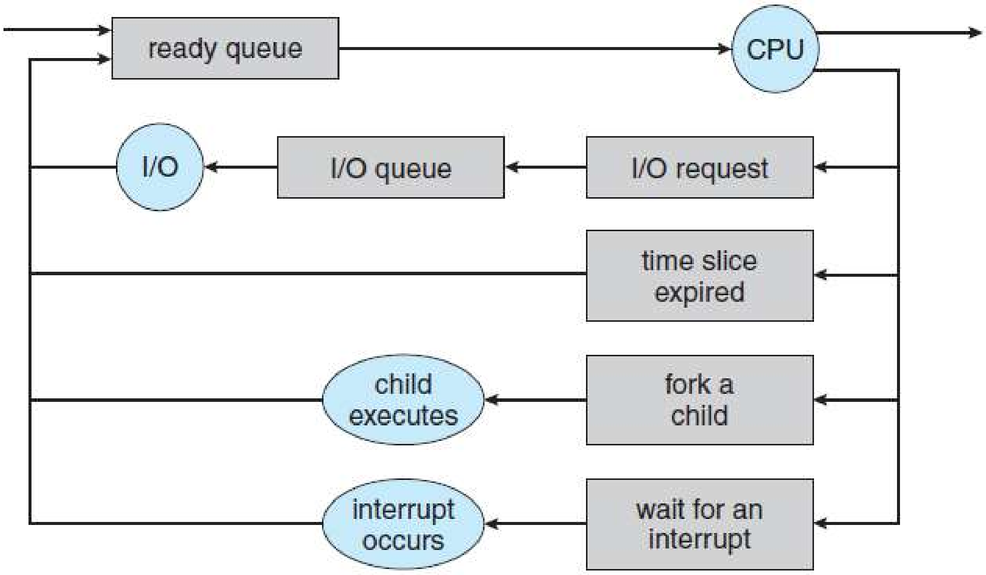


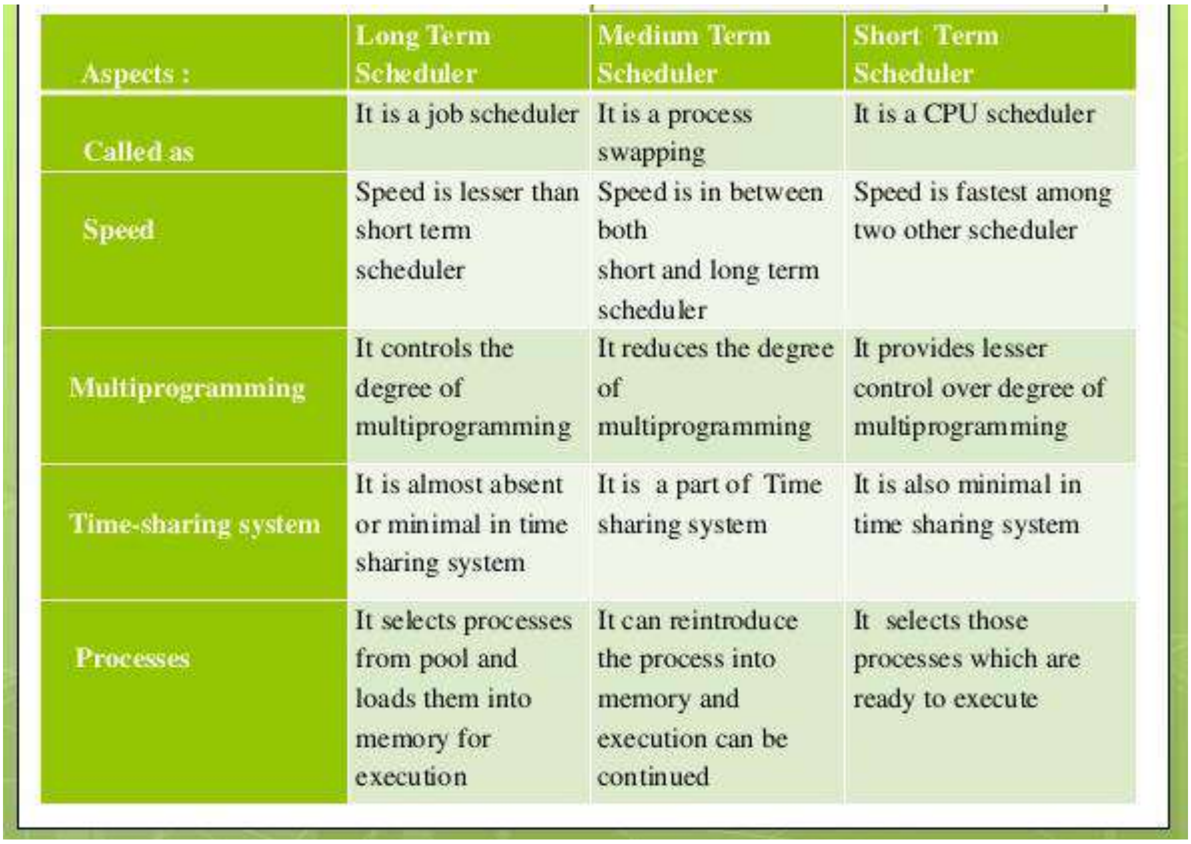
Figure 3.6 Queueing-diagram representation of process scheduling.

Schedulers

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

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, 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 long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O 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/O 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. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.



Aspects :	Long Term Scheduler	Medium Term Scheduler	Short Term Scheduler
Called as	It is a job scheduler	It is a process swapping	It is a CPU scheduler
Speed	Speed is lesser than short term scheduler	Speed is in between both short and long term scheduler	Speed is fastest among two other scheduler
Multiprogramming	It controls the degree of multiprogramming	It reduces the degree of multiprogramming	It provides lesser control over degree of multiprogramming
Time-sharing system	It is almost absent or minimal in time sharing system	It is a part of Time sharing system	It is also minimal in time sharing system
Processes	It selects processes from pool and loads them into memory for execution	It can reintroduce the process into memory and execution can be continued	It selects those processes which are ready to execute

Context Switch

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. 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.

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 times are highly dependent on hardware support.

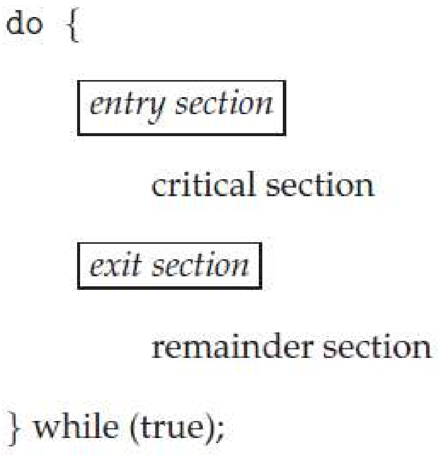
2.4 Process synchronization

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

2.4.1 The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so called critical-section problem. Consider a system consisting of *n* processes {*P*0*, P*1*, ..., Pn*−1}. 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 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 the **remainder section**. The general structure of a typical process *Pi* is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.



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

Figure 5.1 General structure of a typical process P_i.

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.

Two general approaches are used to handle critical sections in operating systems:

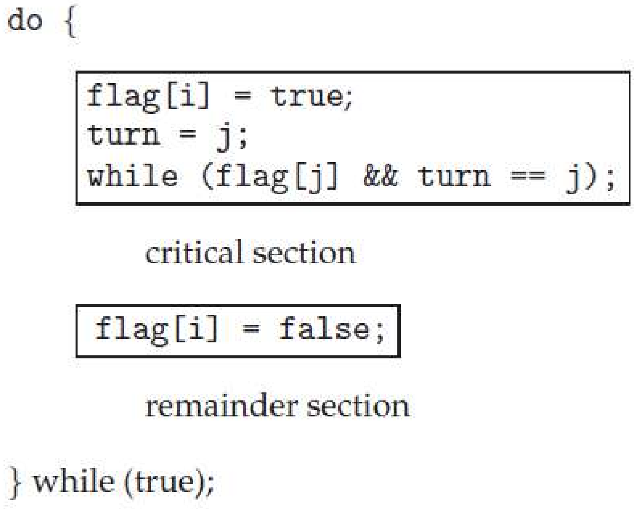
1) **Preemptive kernels** and

2) **non-Preemptive kernels**.

A preemptive (**preemption** is the act of temporarily interrupting a task being carried out by a **computer** system) kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

2.4.2 Peterson's Solution

A classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.



```
do {

     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j);

          critical section

     flag[i] = false;

          remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Peterson's Solution is a classical software based solution to the critical section problem. In Peterson's solution, we have two shared variables:

boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section

int turn : The process whose turn is to enter the critical section.

```
// code for producer (j)
// producer j is ready
// to produce an item
flag[j] = true;

// but consumer (i) can consume an item
turn = i;

// if consumer is ready to consume an item
// and if its consumer's turn
while (flag[i] == true && turn == i)
    {
// then producer will wait }
    // otherwise producer will produce
  // an item and put it into buffer (critical Section)
```

```
   // Now, producer is out of critical section
   flag[j] = false;
   // end of code for producer
   //--------------------------------------------------------
   // code for consumer i
   // consumer i is ready
   // to consume an item
   flag[i] = true;
   // but producer (j) can produce an item
   turn = j;

   // if producer is ready to produce an item
   // and if its producer's turn
   while (flag[j] == true && turn == j)
      {
// then consumer will wait }
      // otherwise consumer will consume
      // an item from buffer (critical Section)
      // Now, consumer is out of critical section
      flag[i] = false;
// end of code for consumer
```

Explanation of Peterson's algorithm –

Peterson's Algorithm is used to synchronize two processes. It uses two variables, a bool array flag of size 2 and an int variable turn to accomplish it. In the solution i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute it's critical section, it sets it's flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section.

After this the current process enters it's critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indication it does not wish to execute anymore.

Synchronization Hardware

However, as mentioned, software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to both kernel developers and application programmers. All these solutions are based on the premise of **locking** —that is, protecting critical regions through the use of locks.

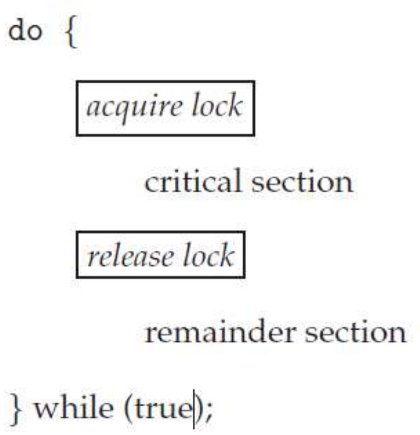
Mutex Locks

operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. (In fact, the term ***mutex*** is short for ***mut***ual ***ex***clusion.) We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

The definition of acquire() is as follows:

```
acquire() {
while (!available)
; /* busy wait */
available = false;;
}
```



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

Solution to the critical-section problem using mutex locks.

The definition of release() is as follows:

```
release() {
available = true;
}
```

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.

Semaphores

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

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)
; // busy wait
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 ≤ 0), as well as its possible modification (S--), must be executed without interruption.

Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2. Suppose we require that *S*2 be executed only after *S*1 has completed. We can implement this scheme readily by letting *P*1 and *P*2 share a common semaphore synch, initialized to 0. In process *P*1, we insert the statements

```
S1;
signal(synch);
```

In process *P*2, we insert the statements

```
wait(synch);
```

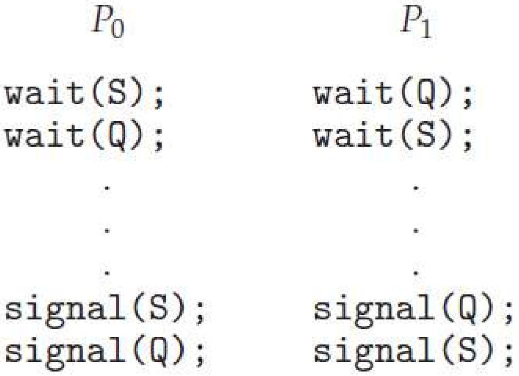
*S*2;

Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:



```
    P0              P1

wait(S);        wait(Q);
wait(Q);        wait(S);
    .               .
    .               .
    .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked. We say that 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.

Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

As an example, assume we have three processes—*L*, *M*, and *H*—whose priorities follow the order *L* < *M* < *H*. Assume that process *H* requires resource *R*, which is currently being accessed by process *L*. Ordinarily, process *H* would wait for *L* to finish using resource *R*. However, now suppose that process *M* becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*. This problem is known as **priority inversion**. It occurs only in systems with more than two priorities, so one solution is to have only two priorities.

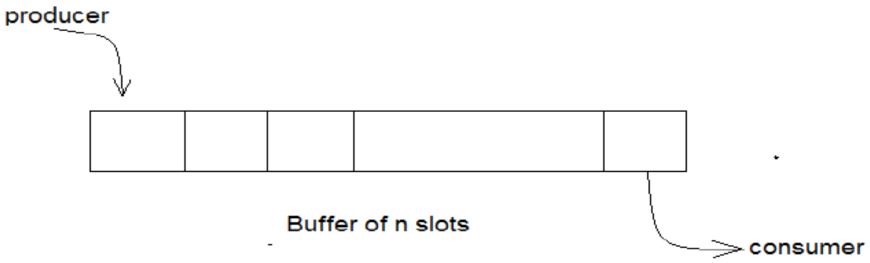
Classic Problems of Synchronization

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- m, a **binary semaphore** which is used to acquire and release the lock.
- empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do

{

  // wait until empty > 0 and then decrement 'empty'

  wait(empty);

  // acquire lock

  wait(mutex);

    /* perform the insert operation in a slot */

    // release lock

  signal(mutex);

  // increment 'full'

  signal(full);

}

while(TRUE);
```

- it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do

{

   // wait until full > 0 and then decrement 'full'

   wait(full);

   // acquire the lock

   wait(mutex);

      /* perform the remove operation in a slot */

      // release the lock

   signal(mutex);

   // increment 'empty'

   signal(empty);

}

while(TRUE);
```

- The consumer waits until there is at least one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.

- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

What is Readers Writer Problem?

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer**is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero number of readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** m and a **semaphore** w. An integer variable read_count is used to maintain the number of readers currently accessing the resource. The variable read_count is initialized to 0. A value of 1 is given initially to m and w.

Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

The code for the **writer** process looks like this:

```
while(TRUE)
{
   wait(w);
  /* perform the write operation */
  signal(w);
}
```

And, the code for the reader process looks like this:

```
while(TRUE)
```

```
{
   //acquire lock
   wait(m);
   read_count++;
   if(read_count == 1)
      wait(w);
      //release lock
   signal(m);
      /* perform the reading operation */
      // acquire lock
   wait(m);
   read_count--;
   if(read_count == 0)
      signal(w);

   // release lock
   signal(m);
}
```

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.

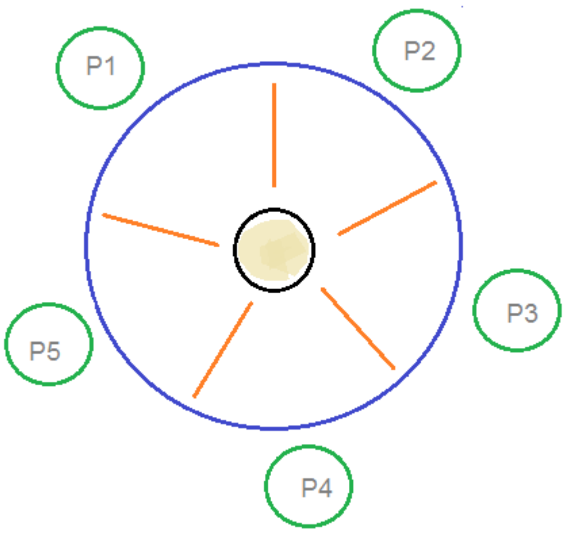
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)
{
   wait(stick[i]);
   /*
       mod is used because if i=5, next
       chopstick is 1 (dining table is circular)
   */
   wait(stick[(i+1) % 5]);
                   /* eat */
   signal(stick[i]);
     signal(stick[(i+1) % 5]);
   /* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

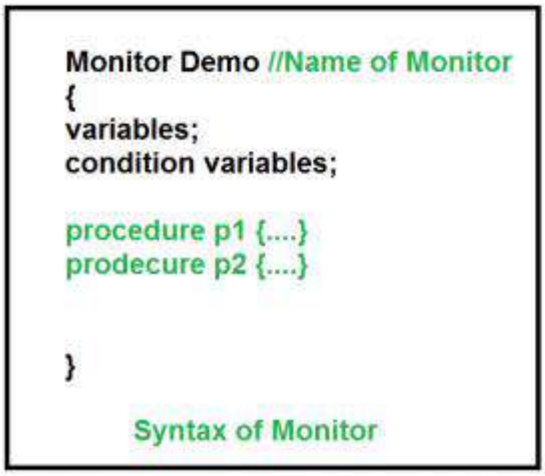
Process Synchronization | Monitors

Monitor is one of the ways to achieve Process synchronization. Monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of monitor but can call procedures of the monitor.

3. Only one process at a time can execute code inside monitors.

Syntax of Monitor



```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Condition Variables

Two different operations are performed on the condition variables of the monitor.

1. Wait.
2. signal.

let say we have 2 condition variables

condition x, y; //Declaring variable

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation

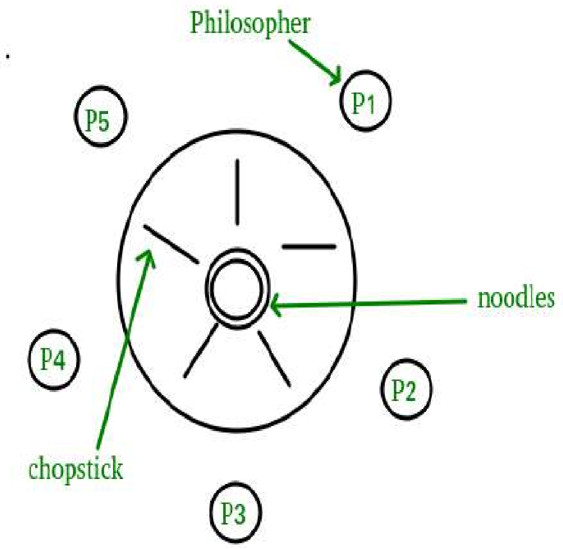
x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)

  // Ignore signal

else
```

// Resume a process from block queue.

Dining-Philosophers Solution Using Monitors

Dining-Philosophers Problem – N philosophers seated around a circular table



- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

We need an algorithm for allocating these limited resources(chopsticks) among several processes(philosophers) such that solution is free from deadlock and free from starvation.

There exist some algorithm to solve Dining – Philosopher Problem, but they may have deadlock situation. Also, a deadlock-free solution is not necessarily starvation-free. Semaphores can result in deadlock due to programming errors. Monitors alone are not sufficiency to solve this, we need monitors with *condition variables*

<u>Monitor-based Solution to Dining Philosophers</u>

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

THINKING – When philosopher doesn’t want to gain access to either fork.

HUNGRY – When philosopher wants to enter the critical section.

EATING – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).

```
// Dining-Philosophers Solution Using Monitors

monitor DP

{

        status state[5];

        condition self[5];

        // Pickup chopsticks

        Pickup(int i)

        {

                // indicate that I’m hungry

                state[i] = hungry;

                // set state to eating in test()

                // only if my left and right neighbors

                // are not eating

                test(i);

                // if unable to eat, wait to be signaled

                if (state[i] != eating)

                        self[i].wait;

        }


        // Put down chopsticks

        Putdown(int i)

        {

                // indicate that I’m thinking

                state[i] = thinking;

                // if right neighbor R=(i+1)%5 is hungry and

                // both of R’s neighbors are not eating,
            // set R’s state to eating and wake it up by
            // signaling R’s CV
            test((i + 1) % 5);
            test((i + 4) % 5);
        }
        test(int i)
        {
            if (state[(i + 1) % 5] != eating
                && state[(i + 4) % 5] != eating
                && state[i] == hungry) {
                // indicate that I’m eating
                state[i] = eating;
                // signal() has no effect during Pickup(),
                // but is important to wake up waiting
                // hungry philosophers during Putdown()
                self[i].signal();
            }
        }
        init()
        {
            // Execution of Pickup(), Putdown() and test()
            // are all mutually exclusive,
            // i.e. only one at a time can be executing
for
        i = 0 to 4
            // Verify that this monitor-based solution is
            // deadlock free and mutually exclusive in that
            // no 2 neighbors can eat simultaneously
            state[i] = thinking;
        }
```

```
} // end of monitor
```

Above Program is a monitor solution to the dining-philosopher problem.

We also need to declare

```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
        ...
        eat
        ...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.

UNIT – III

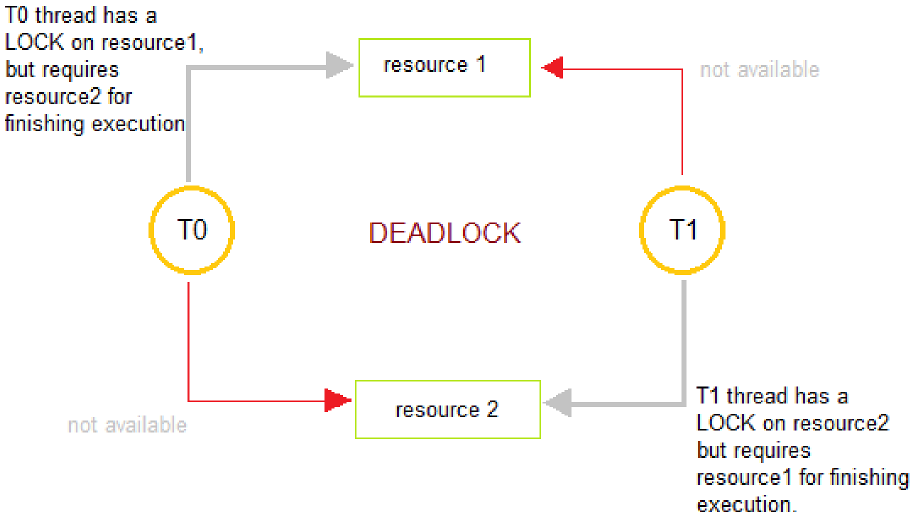
DEADLOCKS: System model, deadlock characterization, deadlock prevention, avoidance, detection and recovery from deadlock. **(T1: Ch-7**)

MEMORY MANAGEMENT: Memory management strategies-Swapping, contiguous memory allocation, paging, structure of the page table, segmentation, virtual-memory management-demand paging, page-replacement algorithms, allocation of frames, thrashing. (**T1: Ch-8, 9**)

What is a Deadlock?

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, awaiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.


T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

System Model

System consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O 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.

A process must request a resource before using it and must release there source 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:

1. **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 there source.
2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release**. The process releases the resource.

The request and release of resources may be system calls like request() and release() device, open() and close() file, and allocate() and free() memory system calls.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process *Pi* is holding the DVD and process *Pj* is holding the printer. If *Pi* requests the printer and *Pj* requests the DVD drive, a deadlock occurs.

Deadlock Characterization

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

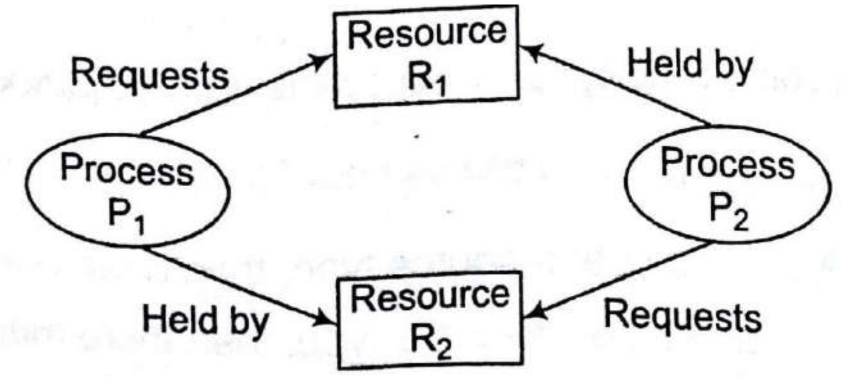
Necessary Conditions

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

1. **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.
2. **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.

3. **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.

4. **Circular wait**. A set *{P*0, *P*1, ..., *Pn}* of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., *Pn*−1 is waiting for a resource held by *Pn*, and *Pn* is waiting for a resource held by *P*0.



State diagram of circular wait condition

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices *V* is partitioned into two different types of nodes: *P* = *{P*1, *P*2, ..., *Pn}*, the set consisting of all the active processes in the system, and *R* = *{R*1, *R*2, ..., *Rm}*, the set consisting of all resource types in the system.

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

Pictorially, we represent each process *Pi* as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, were present each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *Rj* , whereas an assignment edge must also designate one of the dots in the rectangle. When process *Pi* requests an instance of resource type *Rj*, 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 in Figure 7.1 depicts the following situation.

The sets *P, R,* and *E*:

◦*P* = {*P*1, *P*2, *P*3}

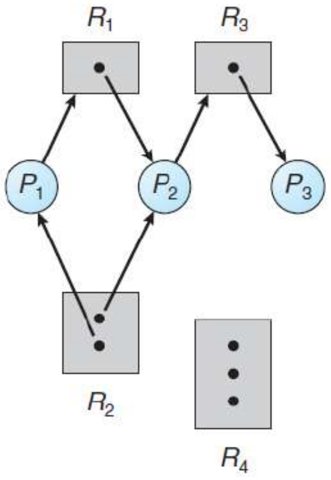


Figure 7.1 Resource-allocation graph.

R = {*R*1, *R*2, *R*3, *R*4}

◦*E* = {*P*1 → *R*1, *P*2 → *R*3, *R*1 → *P*2, *R*2 → *P*2, *R*2 → *P*1, *R*3 → *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 Figure 7.1. Suppose that process *P*3 requests an instance of resource type *R*2. Since no resource instance is currently available, we add a request edge*P*3→ *R*2 to the graph (Figure 7.2). At this point, two minimal cycles exist in the system:

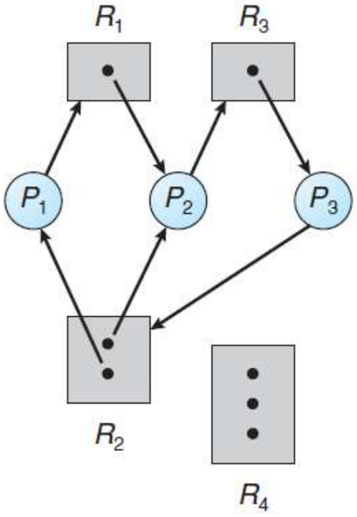


Figure 7.2 Resource-allocation graph with a deadlock.

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

Processes *P*1, *P*2, and *P*3 are deadlocked. Process *P*2 is waiting for the resource*R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process*P*2 to release resource *R*1.

Now consider the resource-allocation graph in Figure 7.3. In this example, we also have a cycle:

$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

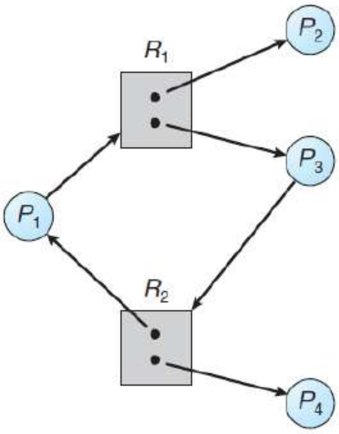


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

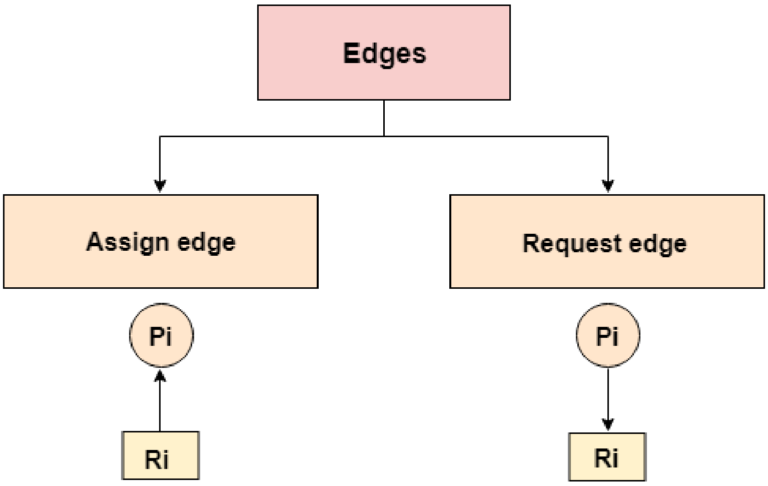
However, there is no deadlock. Observe that process *P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

Resource Allocation Graph

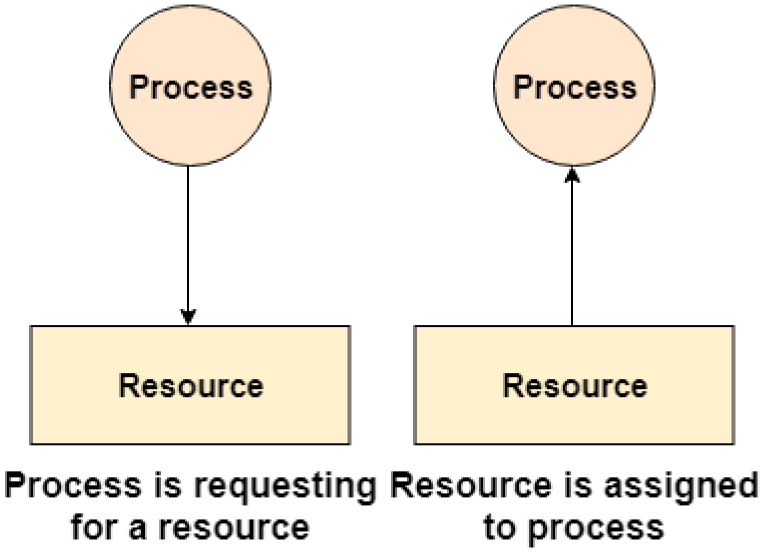
The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

It also contains the information about all the instances of all the resources whether they are available or being used by the processes. In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.

Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource. A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them. A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process. A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

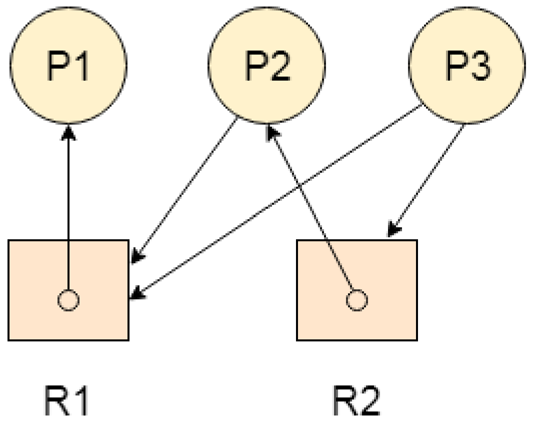

Process
Process
Resource
Resource
Process is requesting for a resource
Resource is assigned to process

Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

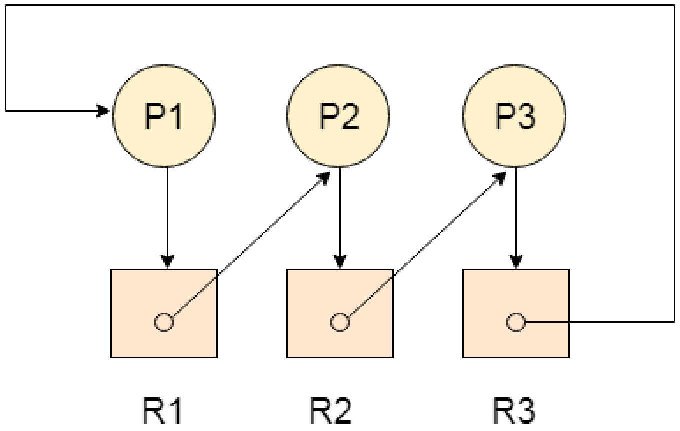
According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.



Deadlock Detection using RAG

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked. In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition. The following example contains three processes P1, P2, P3 and three resources R2, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Allocation Matrix

Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, en entry is being made in front of P1 and below R3 since R3 is assigned to P1.

Process	R1	R2	R3
P1	0	0	1

P2	1	0	0
P3	0	1	0

Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.

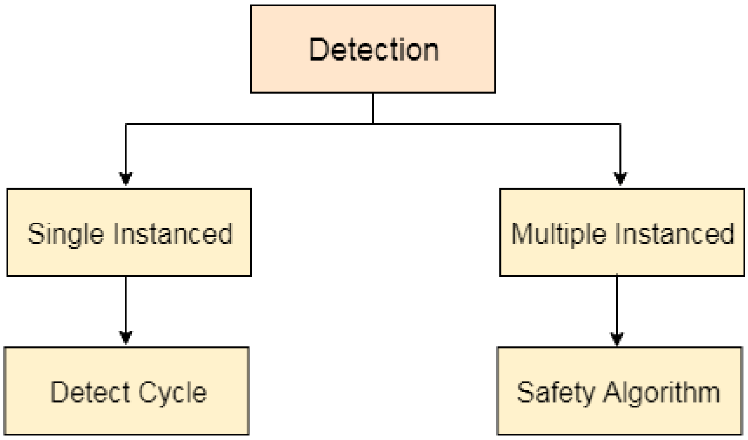
Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avial = (0,0,0)

Neither we are having any resource available in the system nor a process going to release. Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them. We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

Deadlock Detection and Recovery

In this approach, The OS doesn't apply any mechanism to avoid or prevent the deadlocks. Therefore the system considers that the deadlock will definitely occur. In order to get rid of deadlocks, The OS periodically checks the system for any deadlock. In case, it finds any of the deadlock then the OS will recover the system using some recovery techniques. The main task of the OS is detecting the deadlocks. The OS can detect the deadlocks with the help of Resource allocation graph.



In single instanced resource types, if a cycle is being formed in the system then there will definitely be a deadlock. On the other hand, in multiple instanced resource type graph, detecting a cycle is not just enough. We have to apply the safety algorithm on the system by converting the resource allocation graph into the allocation matrix and request matrix. In order to recover the system from deadlocks, either OS considers resources or processes.

For Resource

Preempt the resource

We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.

Rollback to a safe state

System passes through various states to get into the deadlock state. The operating system can rollback the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state. The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

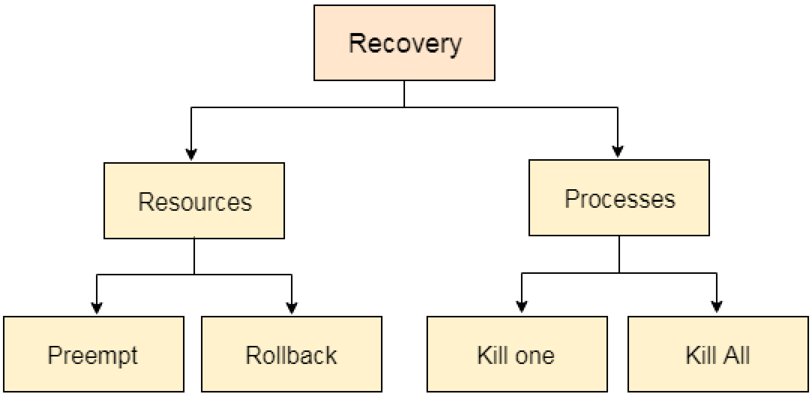
For Process

Kill a process

Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, Operating system kills a process which has done least amount of work until now.

Kill all process

This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.



Methods for Handling Deadlocks

Generally speaking, 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 Linux 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 to ensure that at least one of the necessary conditions (Section7.2.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system 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.

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.

Deadlock Prevention

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

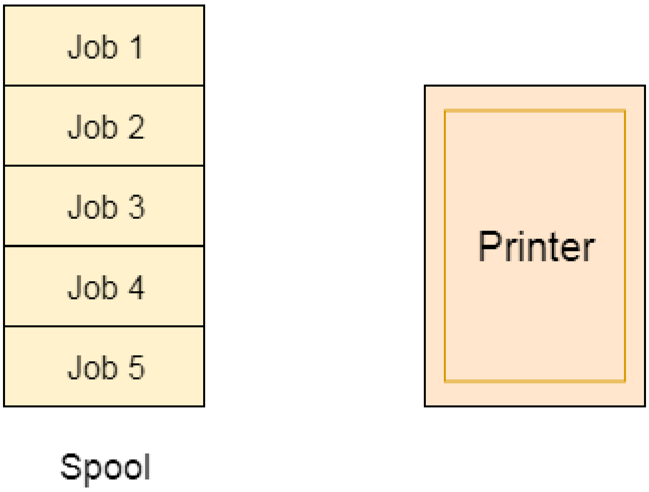
1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially. Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

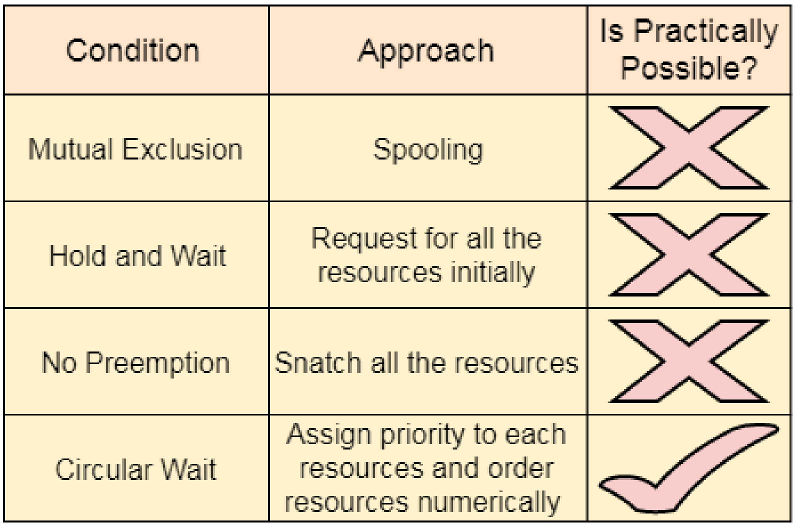
3. No Preemption

Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent. Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.



Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	✗
Hold and Wait	Request for all the resources initially	✗
No Preemption	Snatch all the resources	✗
Circular Wait	Assign priority to each resources and order resources numerically	✓

Among all the methods, violating Circular wait is the only approach that can be implemented practically.

Deadlock Avoidance

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. In this the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Deadlock avoidance algorithms:

1) Resource-Allocation-Graph Algorithm
2) Banker's Algorithm

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 P j ,with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all P j 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 +1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states. In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes. A state of a system recorded at some random time is shown below.

Resources Assigned

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

Resources still needed

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

1. E = (7 6 8 4)
2. P = (6 2 8 3)
3. A = (1 4 0 1)

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process. Table 2 shows the instances of the resources, each process still needs. Vector E is the representation of total instances of each resource in the system.

Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use. A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock. If the system cannot fulfill the request of all processes then the state of the system is called unsafe. The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

What is Banker's Algorithm?

Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Every time a loan has to be granted by the bank, it subtracts the **loan amount** from the **total money** the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers.

Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

Let us assume that there are n processes and m resource types. Some data structures that are used to implement the banker's algorithm are:

1. Available

It is an **array** of length m. It represents the number of available resources of each type. If Available[j] = k, then there are k instances available, of resource type R(j).

2. Max

It is an n x m matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process P(i) can request atmost k instances of resource type R(j).

3. Allocation

It is an n x m matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process P(i) is currently allocated k instances of resource type R(j).

4. Need

It is an n x m matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process P(i) may need k more instances of resource type R(j) to complete its task.

```
Need[i][j] = Max[i][j] - Allocation [i][j]
```

Resource Request Algorithm

This describes the behavior of the system when a process makes a resource request in the form of a request matrix. The steps are:

1. If number of requested instances of each resource is less than the need (which was declared previously by the process), go to step 2.
2. If number of requested instances of each resource type is less than the available resources of each type, go to step 3. If not, the process has to wait because sufficient resources are not available yet.
3. Now, assume that the resources have been allocated. Accordingly do,

```
Available = Available - Requesti
Allocation(i) = Allocation(i) + Request(i)
Need(i) = Need(i) - Request(i)
```

This step is done because the system needs to assume that resources have been allocated. So there will be less resources available after allocation. The number of allocated instances will increase. The need of the resources by the process will reduce. That's what is represented by the above three operations.

After completing the above three steps, check if the system is in safe state by applying the safety algorithm. If it is in safe state, proceed to allocate the requested resources. Else, the process has to wait longer.

Safety Algorithm

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,
2. Work = Available
3. Finish[i] =false for i = 0, 1, ... , n - 1.

   This means, initially, no process has finished and the number of available resources is represented by the **Available** array.
4. Find an index **i** such that both
5. Finish[i] ==false
6. Needi <= Work

   If there is no such i present, then proceed to step 4.

   It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.
7. Perform the following:
8. Work = Work + Allocation;
9. Finish[i] = true;

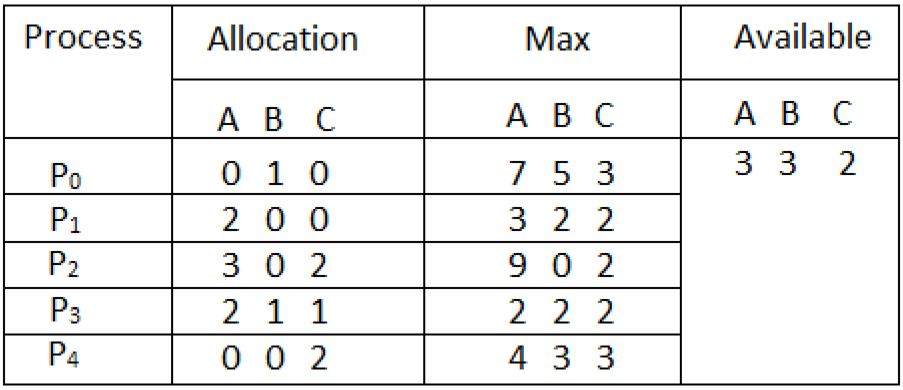
   Go to step 2.

   When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.
10. If Finish[i] == true for all i, then the system is in a safe state.

    That means if all processes are finished, then the system is in safe state.

Example:

Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:



Process	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	

Question1. What will be the content of the Need matrix?

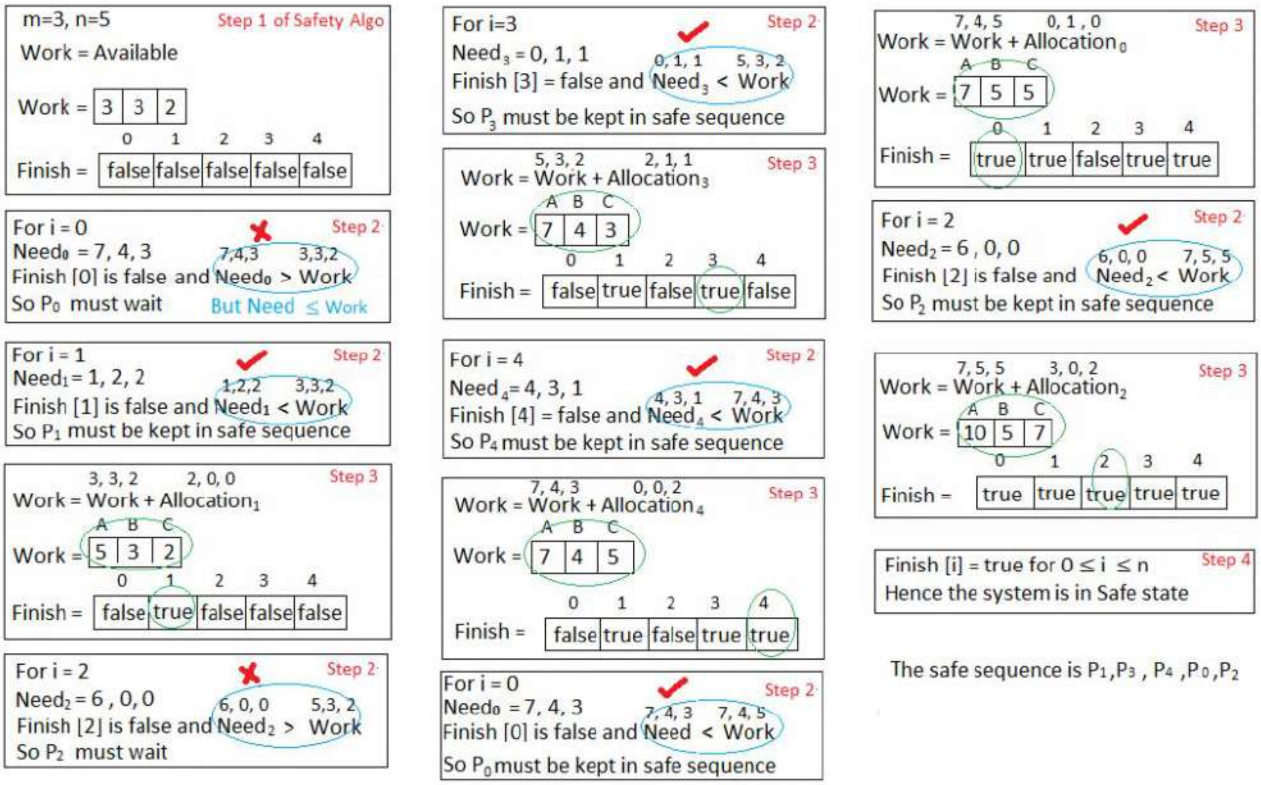
Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

Process	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

Question2. Is the system in safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,



m=3, n=5 Step 1 of Safety Algo

Work = Available

Work =	3	3	2

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 ✗ Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$

So P_0 must wait But Need ≤ Work

For i = 1 ✓ Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$

So P_1 must be kept in safe sequence

Step 3

Work = $\overset{3, 3, 2}{Work}$ + $\overset{2, 0, 0}{Allocation_1}$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 ✗ Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6, 0, 0}{Need_2} > \overset{5,3, 2}{Work}$

So P_2 must wait

For i=3 ✓ Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0, 1, 1}{Need_3} < \overset{5, 3, 2}{Work}$

So P_3 must be kept in safe sequence

Step 3

Work = $\overset{5, 3, 2}{Work}$ + $\overset{2, 1, 1}{Allocation_3}$

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 ✓ Step 2

$Need_4$= 4, 3, 1

Finish [4] = false and $\overset{4, 3, 1}{Need_4} < \overset{7, 4, 3}{Work}$

So P_4 must be kept in safe sequence

Step 3

Work = $\overset{7, 4, 3}{Work}$ + $\overset{0, 0, 2}{Allocation_4}$

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 ✓ Step 2

$Need_0$ =7, 4, 3

Finish [0] is false and $\overset{7, 4, 3}{Need} < \overset{7, 4, 5}{Work}$

So P_0 must be kept in safe sequence

Step 3

Work = $\overset{7, 4, 5}{Work}$ + $\overset{0, 1, 0}{Allocation_0}$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 ✓ Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6, 0, 0}{Need_2} < \overset{7, 5, 5}{Work}$

So P_2 must be kept in safe sequence

Step 3

Work = $\overset{7, 5, 5}{Work}$ + $\overset{3, 0, 2}{Allocation_2}$

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

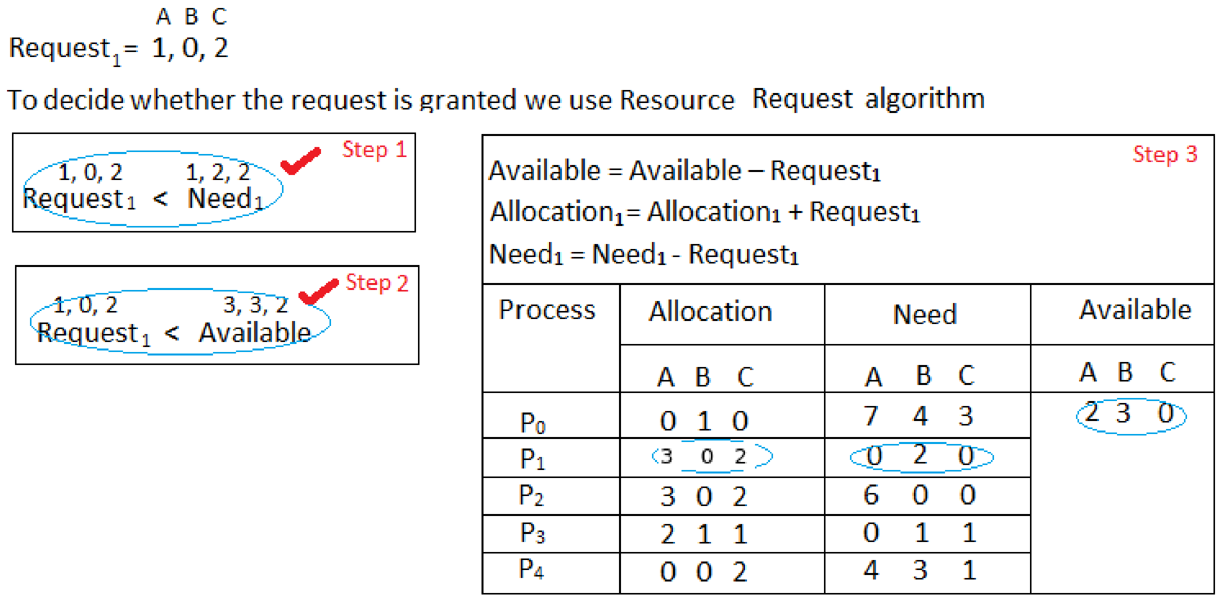
Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

The safe sequence is P_1,P_3 , P_4 ,P_0,P_2

Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?



A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

1, 0, 2 1, 2, 2

$Request_1 < Need_1$

Step 2

1, 0, 2 3, 3, 2

$Request_1 < Available$

Step 3

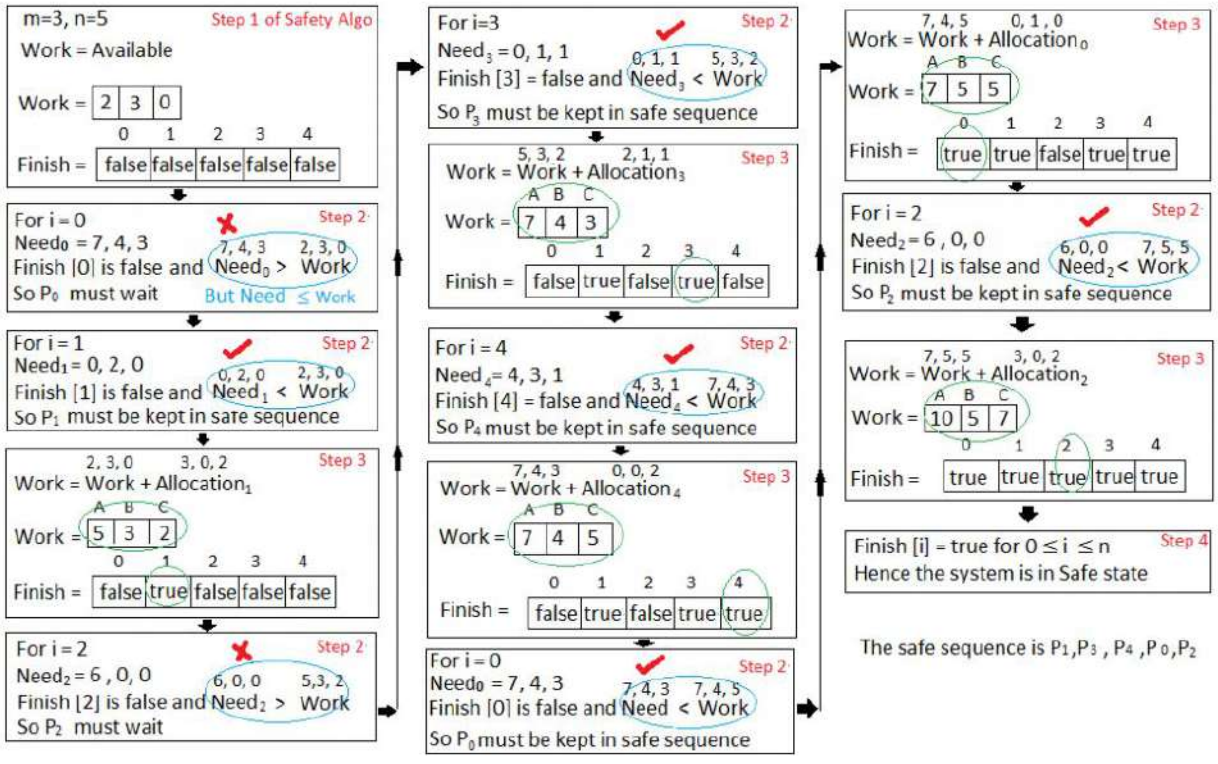
$Available = Available – Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	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 again execute Safety algorithm on the above data structures.



m=3, n=5 Step 1 of Safety Algo

Work = Available

Work = 2 3 0

Finish = false false false false false (0 1 2 3 4)

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0 > Work$ (7, 4, 3 2, 3, 0)

So P_0 must wait But Need ≤ Work

For i = 1 Step 2

$Need_1$ = 0, 2, 0

Finish [1] is false and $Need_1 < Work$ (0, 2, 0 2, 3, 0)

So P_1 must be kept in safe sequence

Step 3

$Work = Work + Allocation_1$ (2, 3, 0 3, 0, 2)

Work = 5 3 2 (A B C)

Finish = false true false false false (0 1 2 3 4)

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2 > Work$ (6, 0, 0 5,3, 2)

So P_2 must wait

For i=3 Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3 < Work$ (0, 1, 1 5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

$Work = Work + Allocation_3$ (5, 3, 2 2, 1, 1)

Work = 7 4 3 (A B C)

Finish = false true false true false (0 1 2 3 4)

For i = 4 Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4 < Work$ (4, 3, 1 7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

$Work = Work + Allocation_4$ (7, 4, 3 0, 0, 2)

Work = 7 4 5 (A B C)

Finish = false true false true true (0 1 2 3 4)

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

$Work = Work + Allocation_0$ (7, 4, 5 0, 1 , 0)

Work = 7 5 5 (A B C)

Finish = true true false true true (0 1 2 3 4)

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2 < Work$ (6, 0, 0 7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

$Work = Work + Allocation_2$ (7, 5, 5 3, 0, 2)

Work = 10 5 7 (A B C)

Finish = true true true true true (0 1 2 3 4)

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Hence the new system state is safe, so we can immediately grant the request for process **P_1**.

Example2

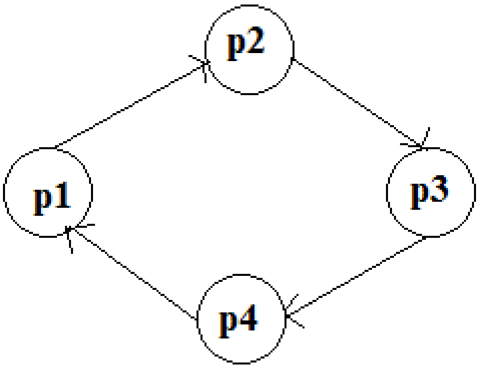
Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. *Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger quantities of each resource.*

```
Total resources in system:
A B C D
6 5 7 6
Available system resources are:
A B C D
3 1 1 2
Processes (currently allocated resources):
   A B C D
P1 1 2 2 1
P2 1 0 3 3
P3 1 2 1 0
Processes (maximum resources):
   A B C D
P1 3 3 2 2
P2 1 2 3 4
P3 1 3 5 0
Need= maximum resources - currently allocated resources
Processes (need resources):
   A B C D
P1 2 1 0 1
P2 0 2 0 1
P3 0 1 4 0
```

Deadlock Detection

If deadlock prevention and avoidance are not done properly, as deadlock may occur and only things left to do is to detect the recover from the deadlock.

If all resource types has only single instance, then we can use a graph called wait-for-graph, which is a variant of resource allocation graph. Here, vertices represent processes and a directed edge from P1 to P2 indicate that P1 is waiting for a resource held by P2. Like in the case of resource allocation graph, a cycle in a wait-for-graph indicate a deadlock. So the system can maintain a wait-for-graph and check for cycles periodically to detect any deadlocks.



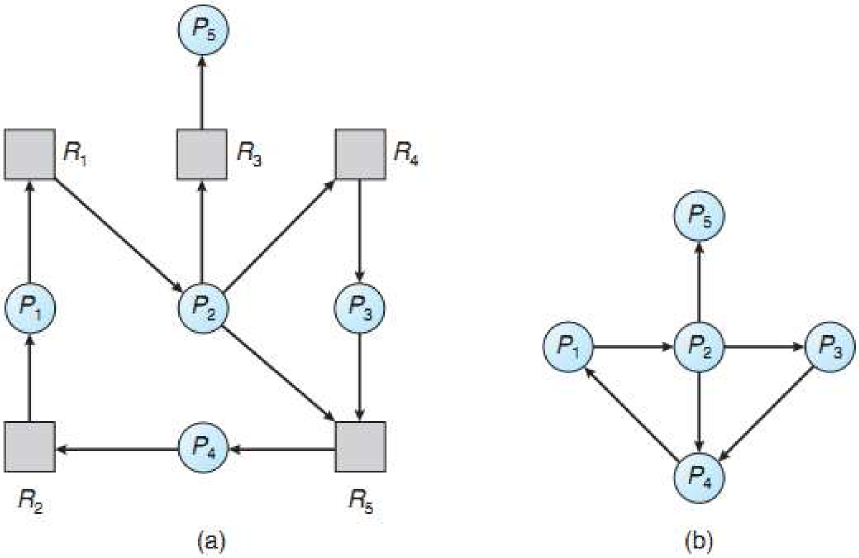


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

The wait-for-graph is not much useful if there are multiple instances for a resource, as a cycle may not imply a deadlock. In such a case, we can use an algorithm similar to Banker's algorithm to detect deadlock. We can see if further allocations can be made on not based on current allocations. You can refer to any operating system text books for details of these algorithms.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alter-natives 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

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.

MEMORY MANAGEMENT

In computer each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 8.1.The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

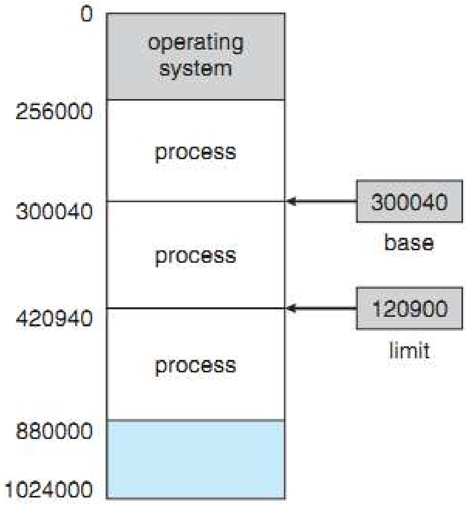
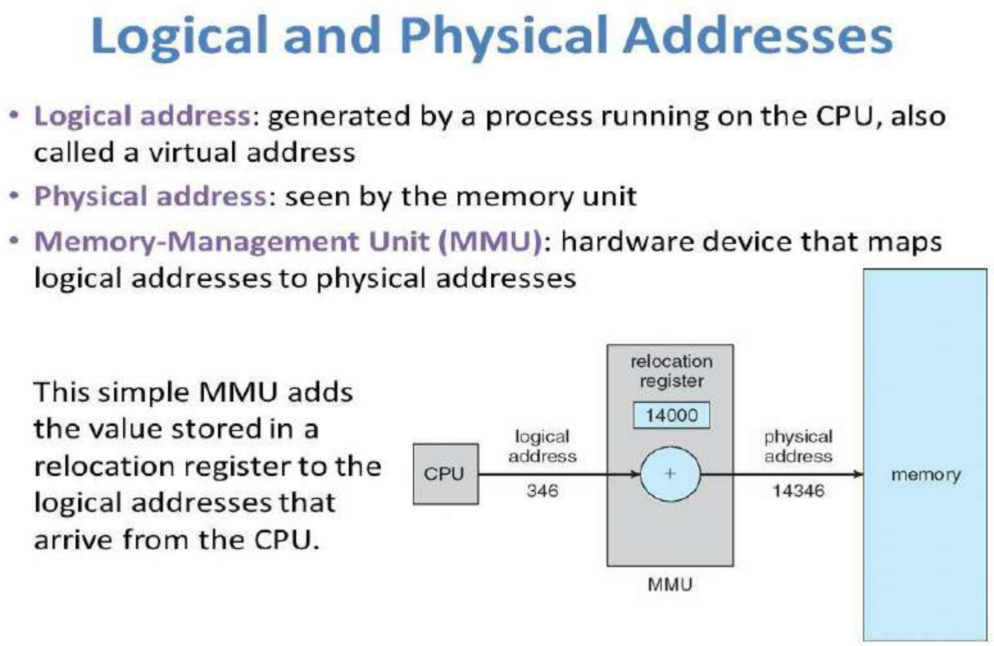


Figure 8.1 A base and a limit register define a logical address space.

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Process Address Space

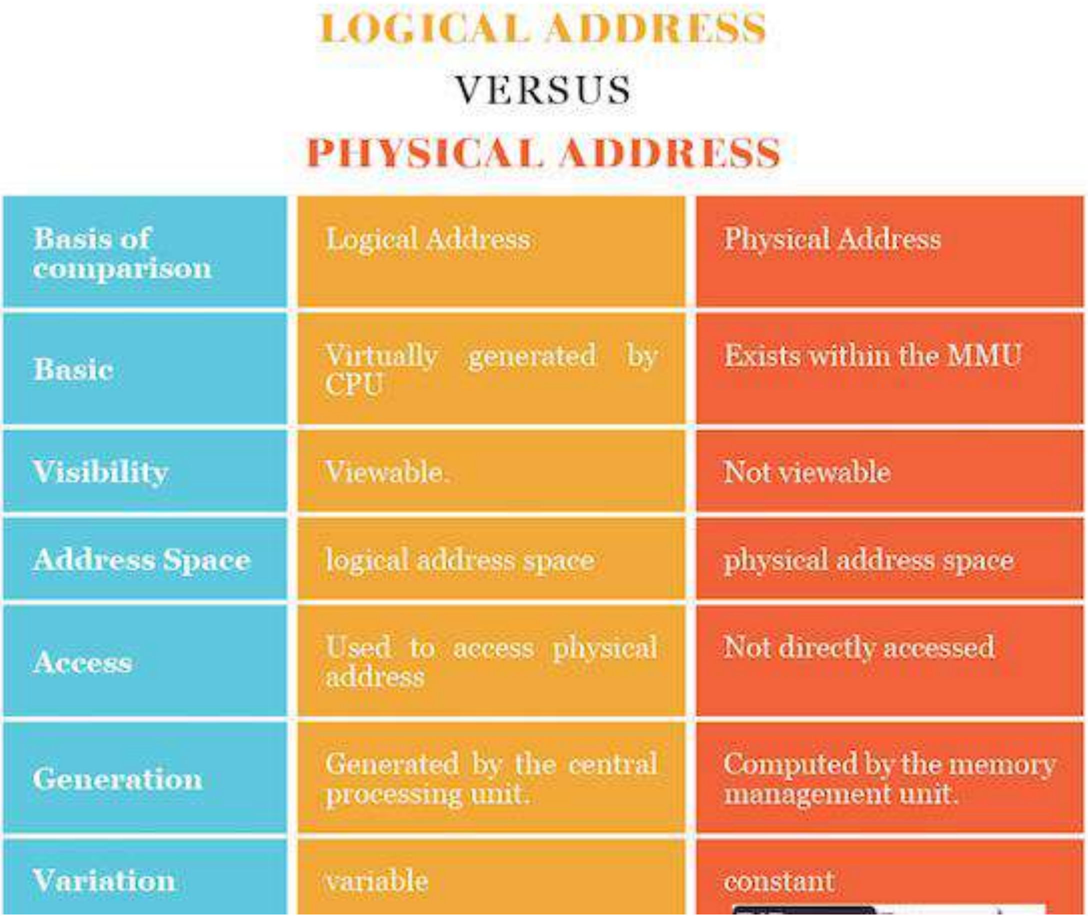
The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program.



Logical and Physical Addresses

- Logical address: generated by a process running on the CPU, also called a virtual address
- Physical address: seen by the memory unit
- Memory-Management Unit (MMU): hardware device that maps logical addresses to physical addresses

This simple MMU adds the value stored in a relocation register to the logical addresses that arrive from the CPU.



LOGICAL ADDRESS
VERSUS
PHYSICAL ADDRESS

Basis of comparison	Logical Address	Physical Address
Basic	Virtually generated by CPU	Exists within the MMU
Visibility	Viewable.	Not viewable
Address Space	logical address space	physical address space
Access	Used to access physical address	Not directly accessed
Generation	Generated by the central processing unit.	Computed by the memory management unit.
Variation	variable	constant

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme. The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.

- The user program deals with virtual addresses; it never sees the real physical addresses.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution. At the time of loading, with static loading, the absolute program (and data) is loaded into memory in order for execution to start. If you are using dynamic loading, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue.

In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 8.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically binds these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

• Compile time. If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.

• Load time. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

• Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

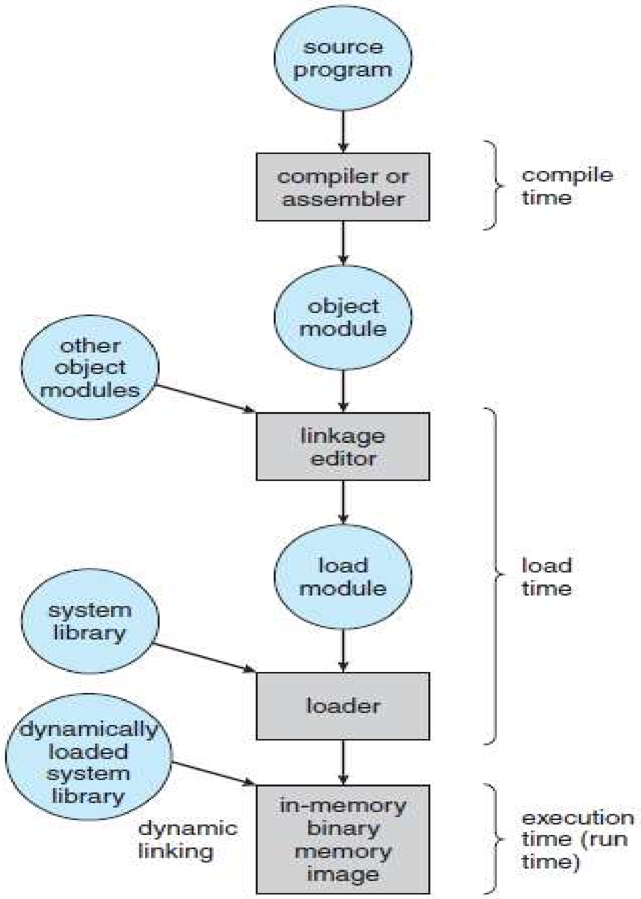


Figure 8.3 Multistep processing of a user program.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory. Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.

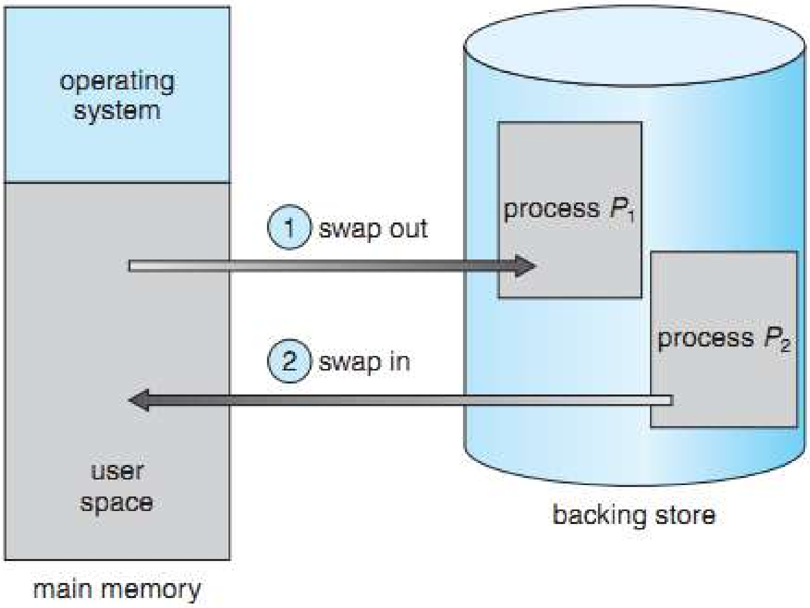
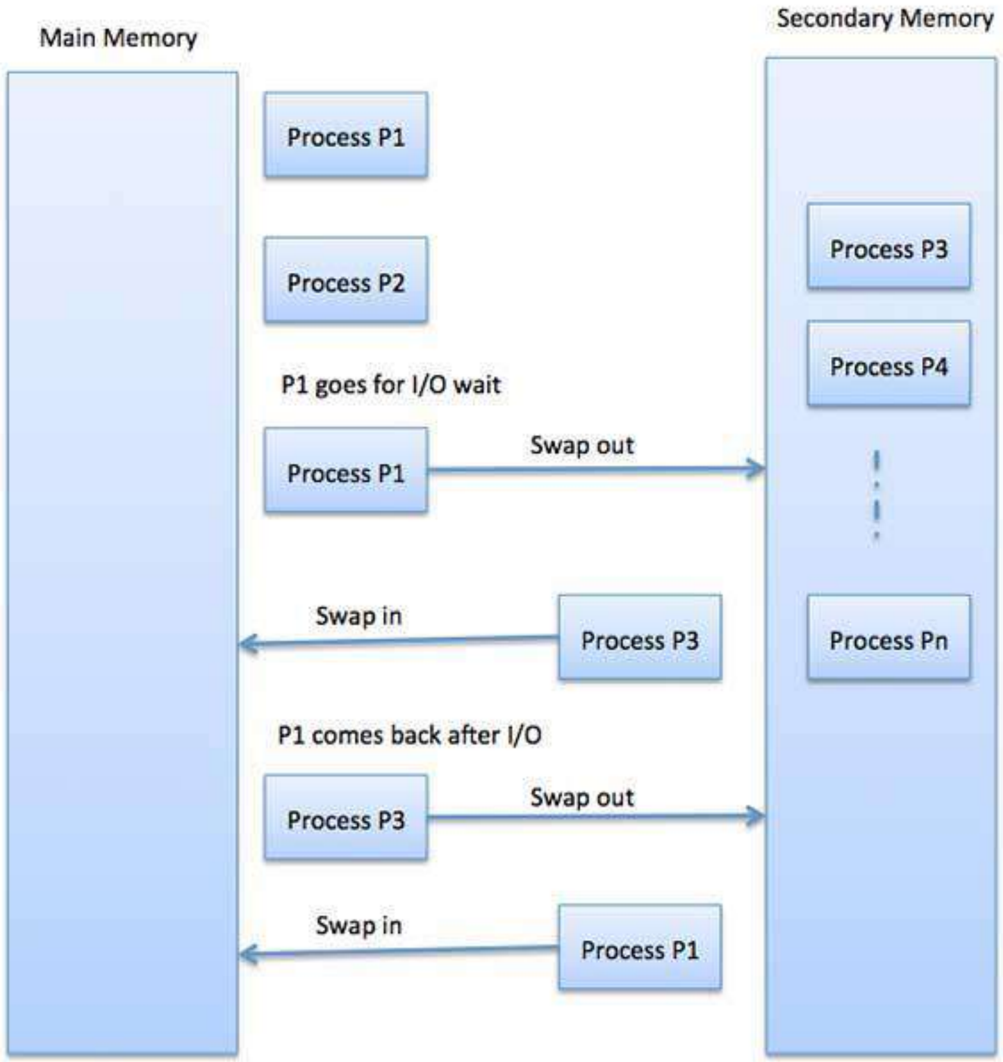


Figure 8.5 Swapping of two processes using a disk as a backing store.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions –

- **Low Memory** – Operating system resides in this memory.
- **High Memory** – User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. 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.

Contiguous Memory Allocation

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process. When a partition is free, a process is selected from the input queue and loaded into it. The free blocks of memory are known as holes. The set of holes is searched to determine which hole is best to allocate.

Memory Protection

Memory protection is a phenomenon by which we control memory access rights on a computer. The main aim of it is to prevent a process from accessing memory that has not been allocated to it. Hence prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the disturbing process, generally killing of process.

Memory Allocation

Memory allocation is a process by which computer programs are assigned memory or space. It is of three types :

First Fit: The first hole that is big enough is allocated to program.

Best Fit: The smallest hole that is big enough is allocated to program.

Worst Fit: The largest hole that is big enough is allocated to program.

The contiguous memory allocation scheme can be implemented in operating systems with the help of two registers, known as the base and limit registers. When a process is executing in main memory, its base register contains the starting address of the memory location where the process is executing, while the amount of bytes consumed by the process is stored in the limit register.

A process does not directly refer to the actual address for a corresponding memory location. Instead, it uses a relative address with respect to its base register. All addresses referred by a program are considered as virtual addresses. The CPU generates the logical or virtual address, which is converted into an actual address with the help of the memory management unit (MMU). The base address register is used for address translation by the MMU. Thus, a physical address is calculated as follows:

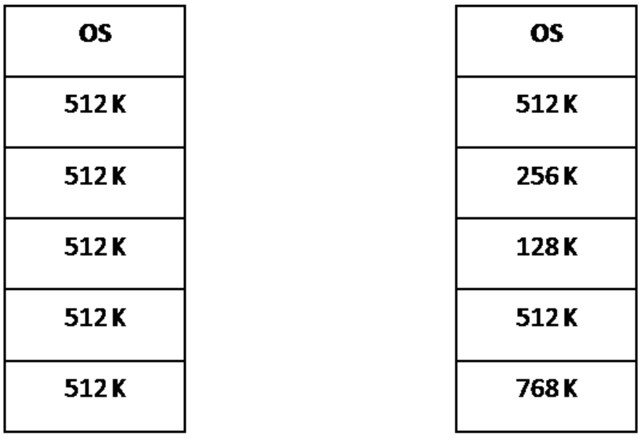
Physical Address = Base register address + Logical address/Virtual address

The address of any memory location referenced by a process is checked to ensure that it does not refer to an address of a neighboring process. This processing security is handled by the underlying operating system. One disadvantage of contiguous memory allocation is that the degree of multiprogramming is reduced due to processes waiting for free memory.

Operating System keeps track of available free memory areas. There are three approaches to select a free partition from the set of available blocks.

First Fit:

It allocates the first free large area whose size is >= program size. Searching may start from either beginning of the list or where previous first-fit search ended. Limitation of this technique is that it may split a free area repeatedly and produce smaller size of blocks that may consider as external fragmentation.



(a) equal-size

(b) Unequal-size

Example of fixed size partitioning

Best Fit:

It allocates the smallest free area with size >= program size. We have to search the entire free list to find out the smallest free hole so it has higher allocation cost. Limitation of this technique is that in long run it too may produce numerous unusable small free areas. It also suffers from higher allocation cost because it has to process entire free list at every allocation.

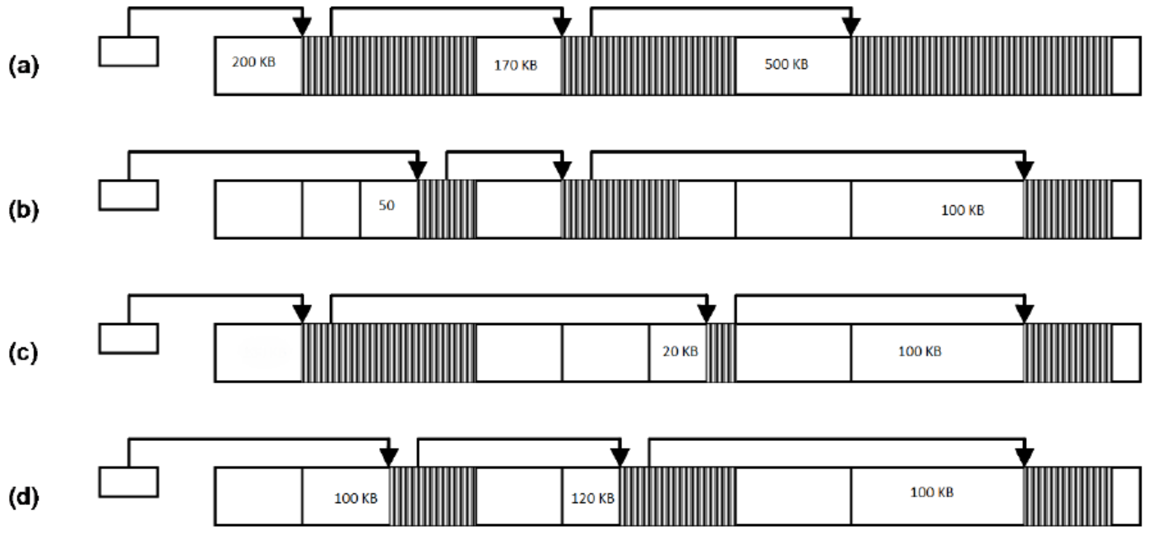
Worst Fit (Next Fit):

The worst fit technique is a compromise between these two techniques. It remembers the entry of last allocation. It searches the free list starting from the previous allocation for the first free area of size >= program size. The first fit technique is better than best fit. Both first fit and next fit performs better than best fit.

Example:A free list contains three free areas of size 200, 170 and 500 bytes respectively (figure a). Processes sends allocation requests for 100, 50 and 400 bytes.

The first fit technique will allocate 100 and 50 bytes from the first free area leaving a free area of 50 bytes. It allocates 400 bytes from the third free area.

The best fit technique will allocate 100 and 50 bytes from the second free area leaving a free area of 20 bytes. The next fit technique allocates 100, 50 and 400 bytes from the three free areas.



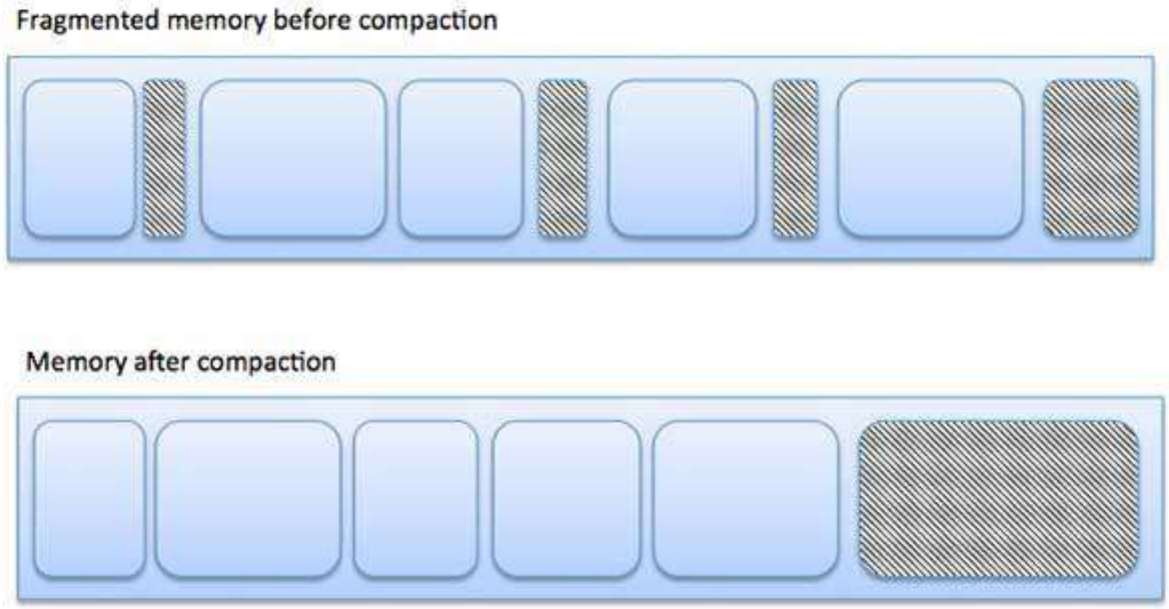
Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types –

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –



External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic. The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Difference between Internal and External fragmentation

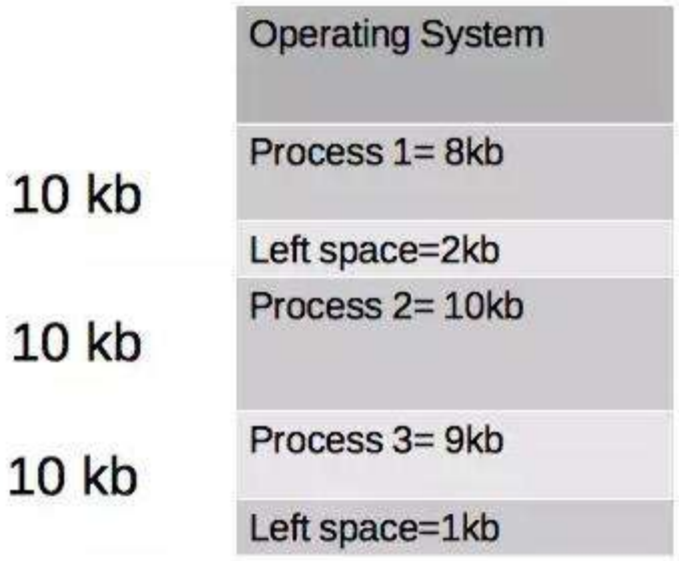
Internal Fragmentation

1. When a process is allocated more memory than required, few space is left unused and this is called as **INTERNAL FRAGMENTATION.**
2. It occurs when memory is divided into fixed-sized partitions.
3. It can be cured by allocating memory dynamically or having partitions of different sizes.

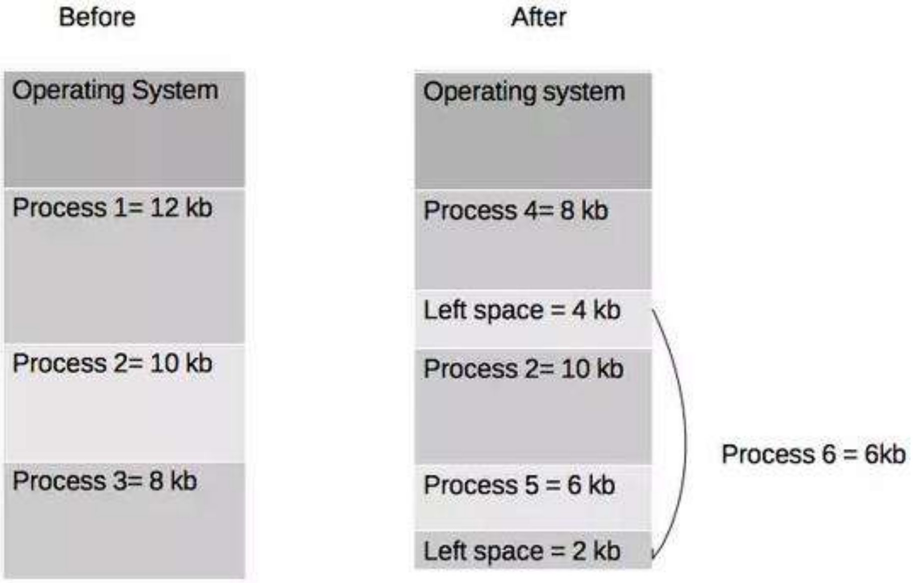
External Fragmentation

1. After execution of processes when they are swapped out of memory and other smaller processes replace them, many small non contiguous(adjacent) blocks of unused spaces are formed which can serve a new request if all of them are put together but as they are not adjacent to each other a new request can't be served and this is known as **EXTERNAL FRAGMENTATION**.
2. It occurs when memory is divided into variable-sized partitions based on size of process.
3. It can be cured by Compaction, Paging and Segmentation.

Internal Fragmentation



External Fragmentation

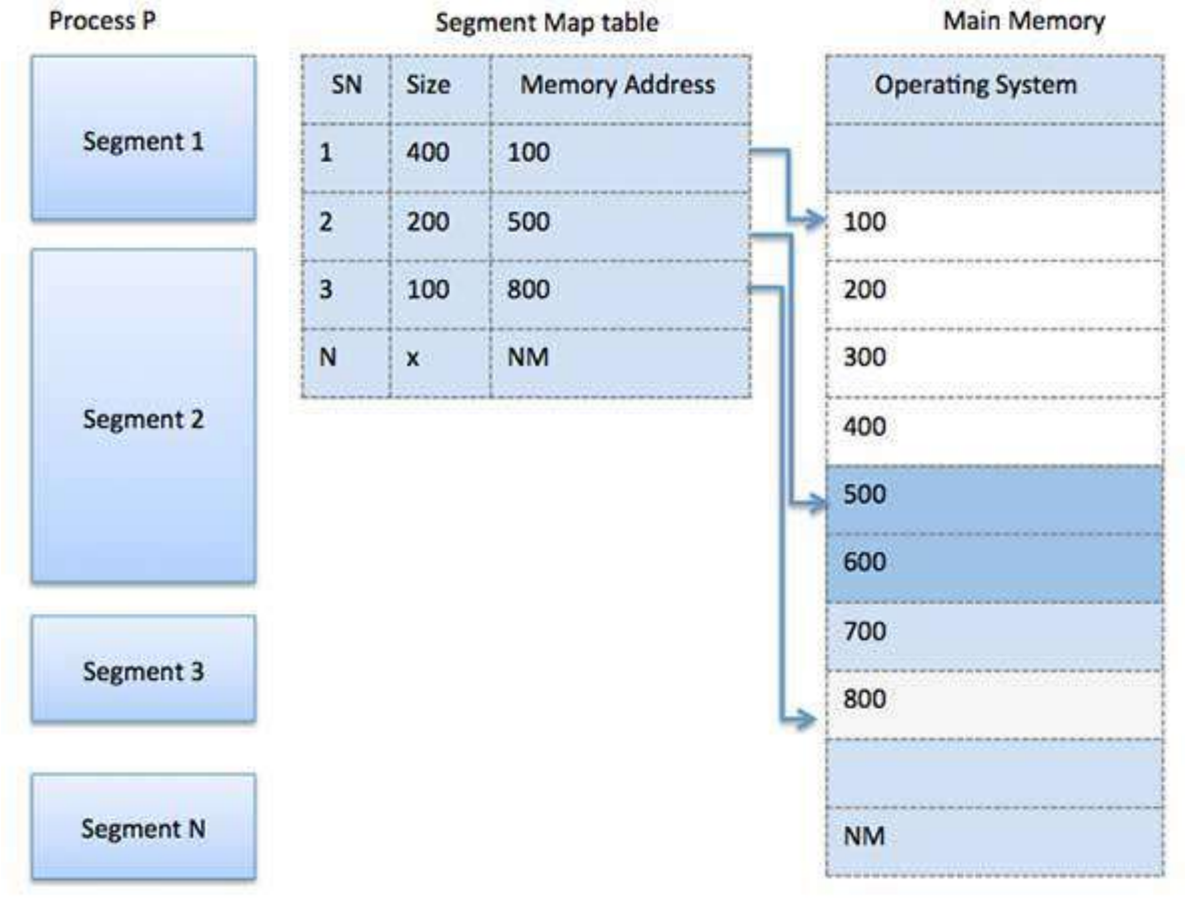


Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory. Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Segment Map table

SN	Size	Memory Address
1	400	100
2	200	500
3	100	800
N	x	NM

Segmentation Hardware

Although the programmer 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.

The use of a segment table is illustrated in Figure 8.8. 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.

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 to3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment0 would result in a trap to the operating system, as this segment is only 1,000bytes long.

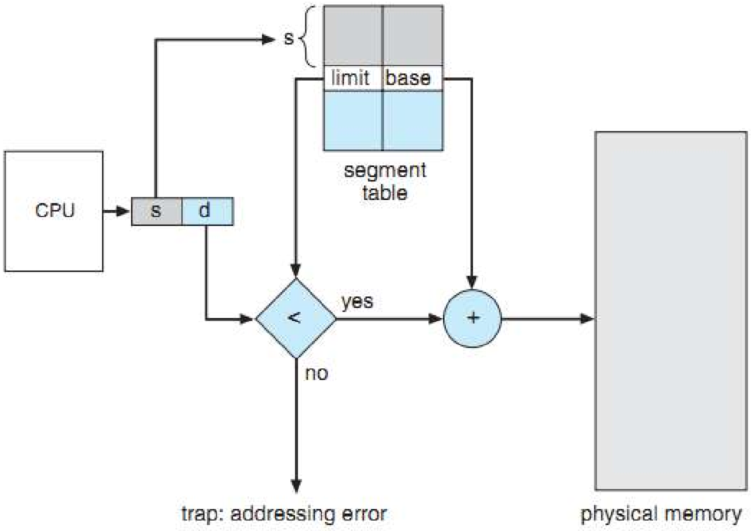
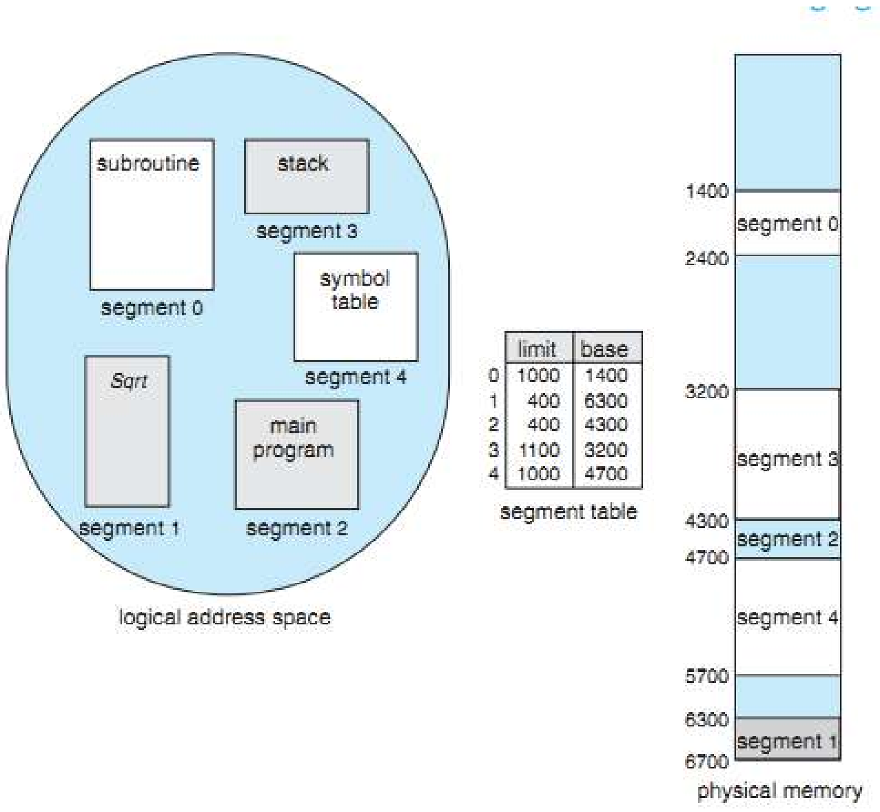


Figure 8.8 Segmentation hardware.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

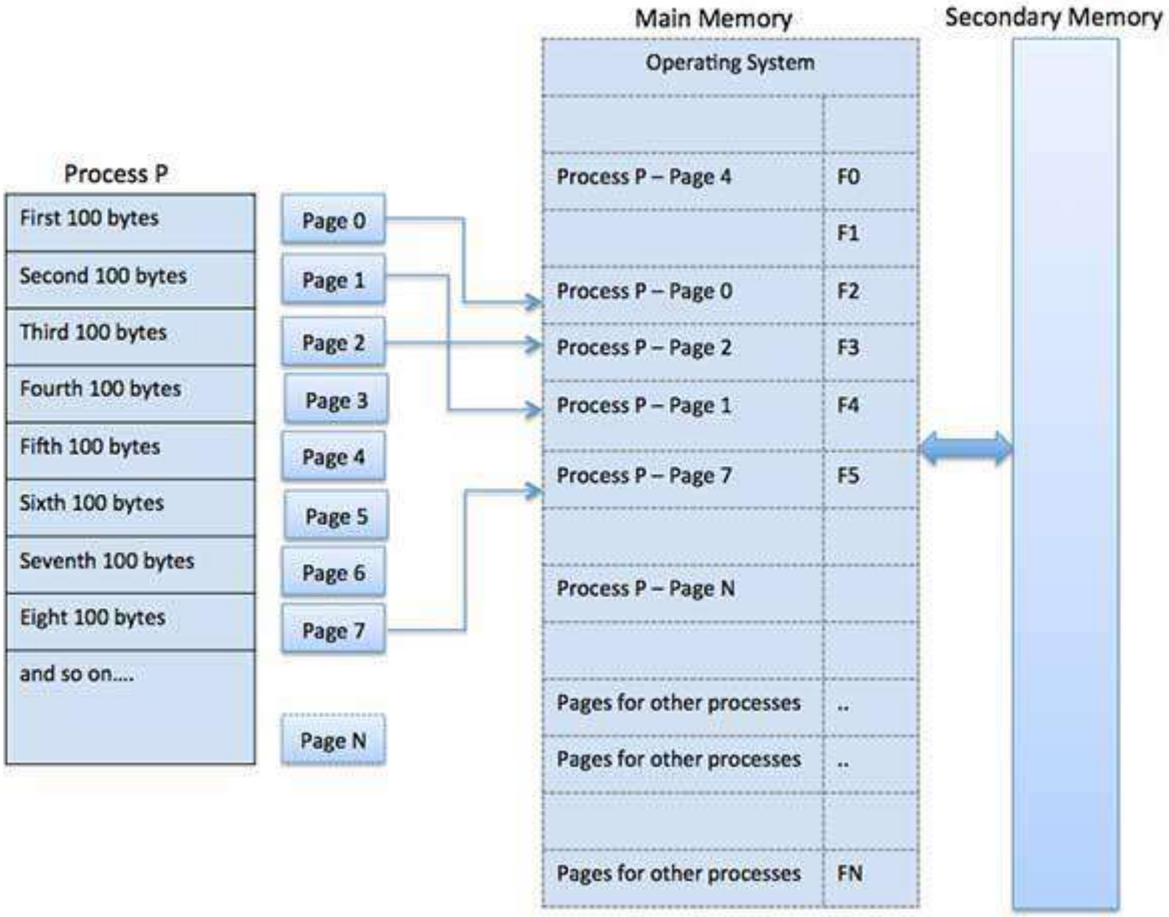
Figure 8.9 Example of segmentation.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

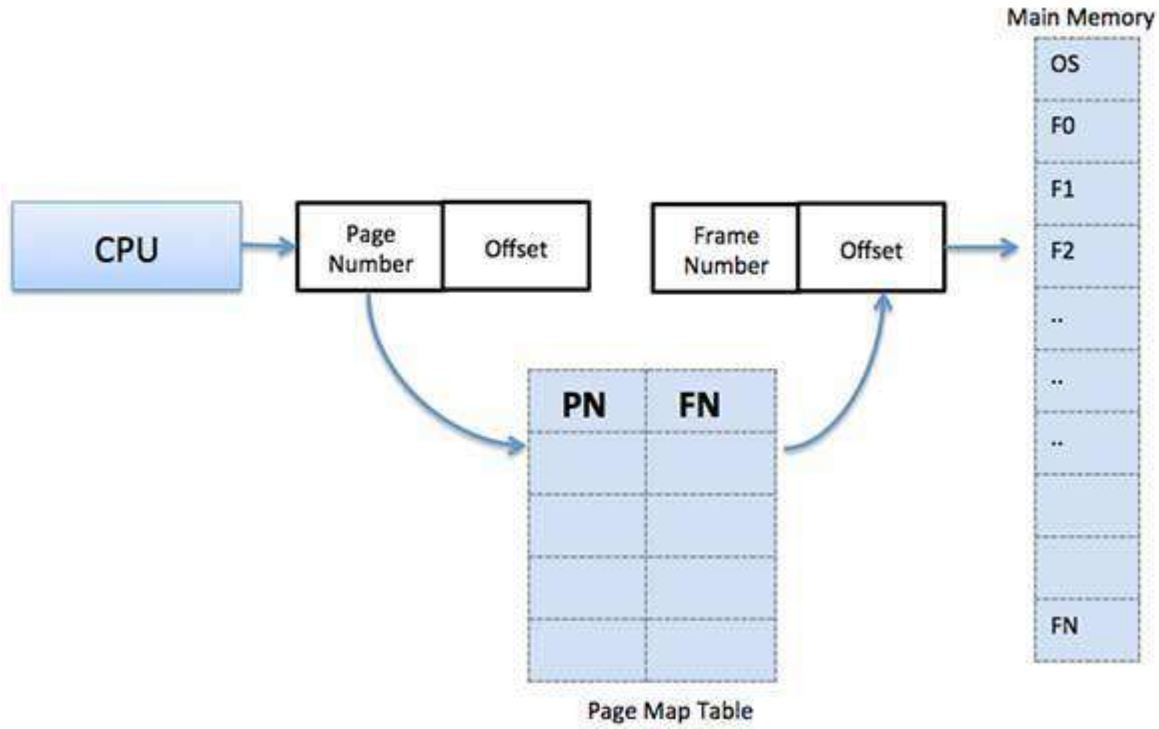
Page address is called **logical address** and represented by **page number**and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging –

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.

- Page table requires extra memory space, so may not be good for a system having small RAM.

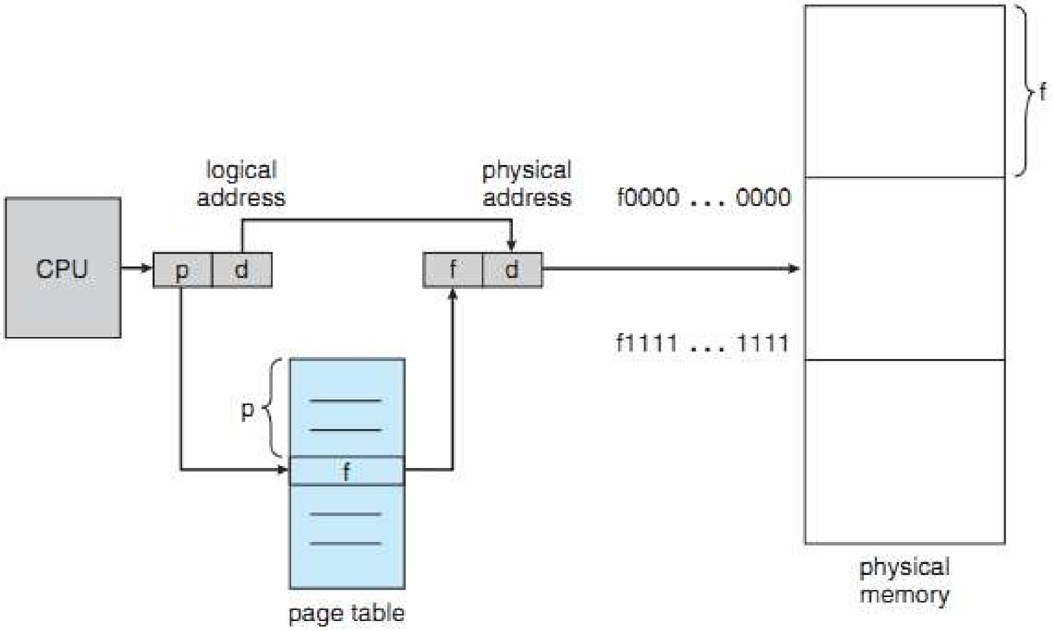


Figure 8.10 Paging hardware.

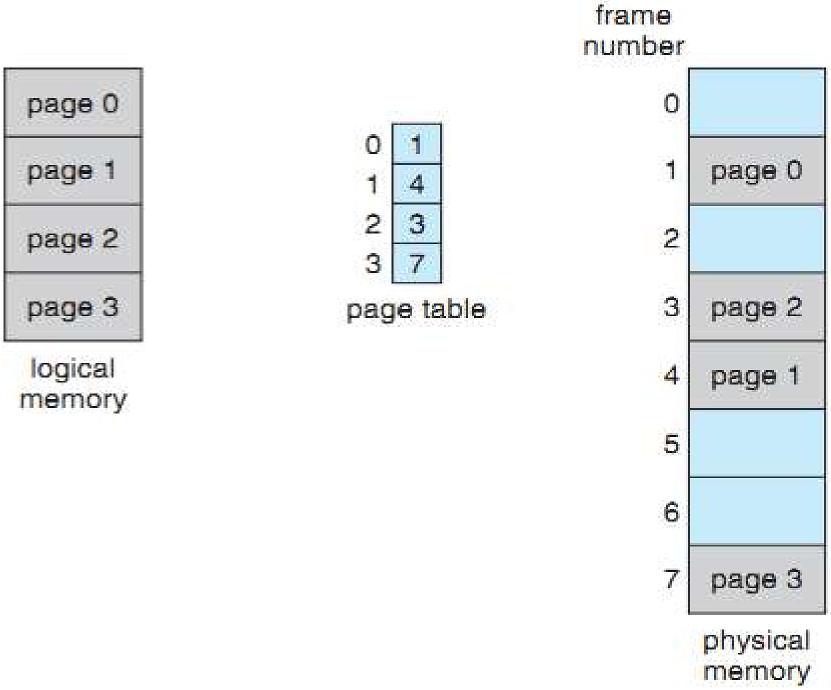


Figure 8.11 Paging model of logical and physical memory.

Paging VS Segmentation

S No.	Paging	Segmentation
1	Non-Contiguous memory allocation	Non-contiguous memory allocation
2	Paging divides program into fixed size pages.	Segmentation divides program into variable size segments.
3	OS is responsible	Compiler is responsible.
4	Paging is faster than segmentation	Segmentation is slower than paging
5	Paging is closer to Operating System	Segmentation is closer to User
6	It suffers from internal fragmentation	It suffers from external fragmentation
7	There is no external fragmentation	There is no external fragmentation
8	Logical address is divided into page number and page offset	Logical address is divided into segment number and segment offset
9	Page table is used to maintain the page information.	Segment Table maintains the segment information
10	Page table entry has the frame number and some flag bits to represent details about pages.	Segment table entry has the base address of the segment and some protection bits for the segments.

Virtual Memory

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM.

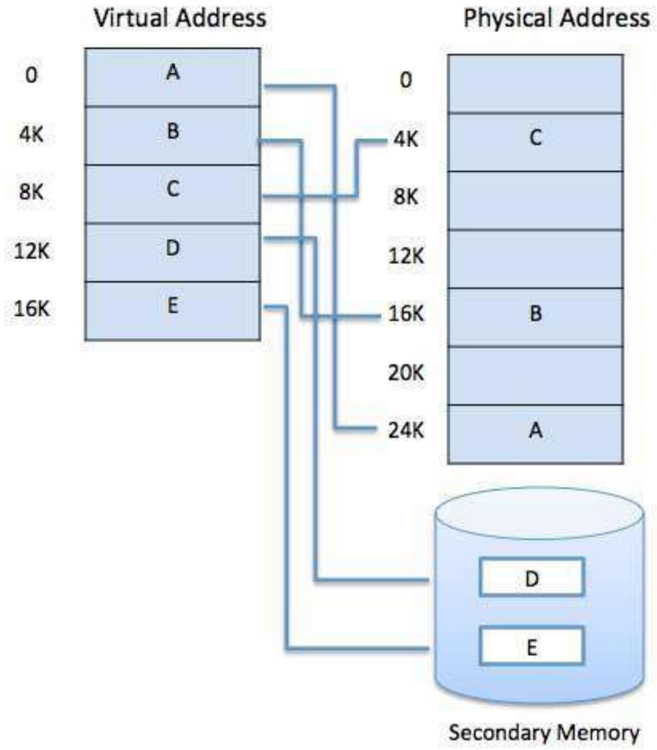
The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.

- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

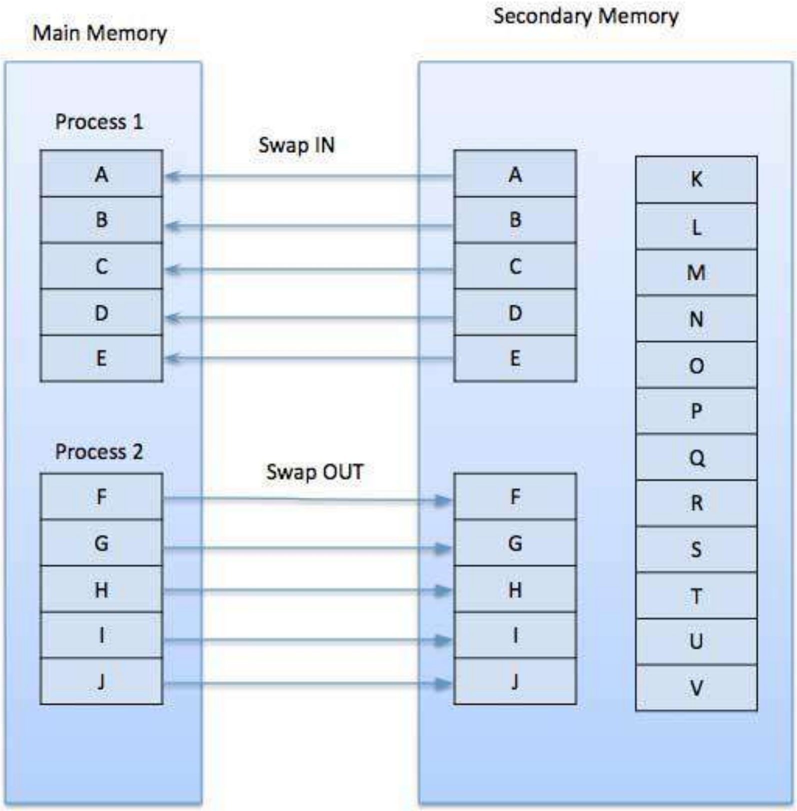
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below –

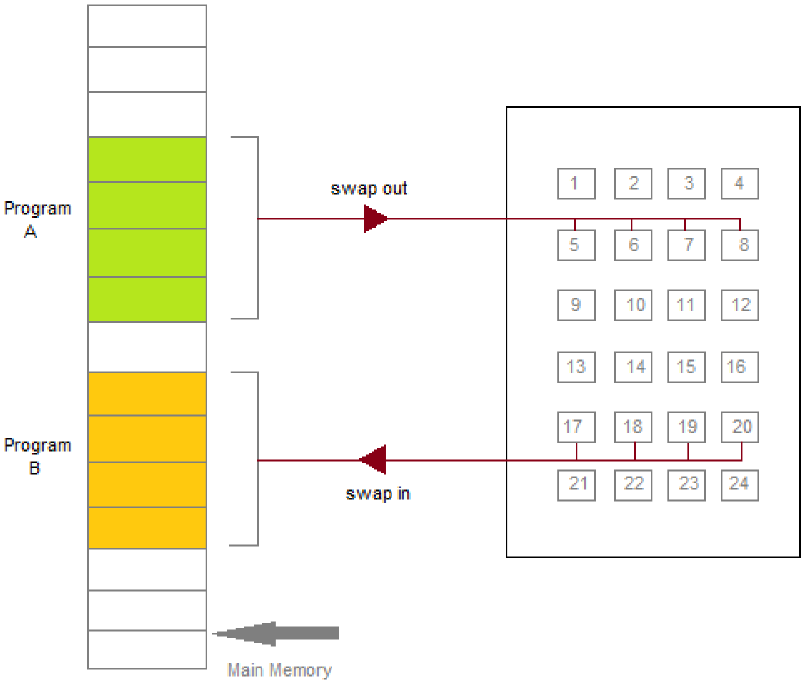


Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.





While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging –

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement Algorithms

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

For a given page size, we need to consider only the page number, not the entire address. If we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

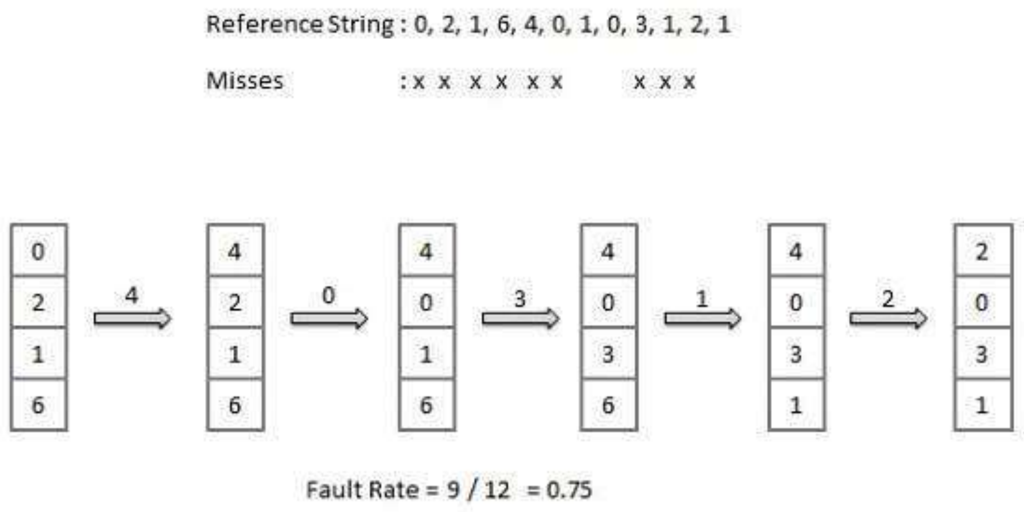
For example, consider the following sequence of addresses – 123,215,600,1234,76,96

If page size is 100, then the reference string is 1,2,6,12,0,0

First In First Out (FIFO) algorithm

Oldest page in main memory is the one which will be selected for replacement.

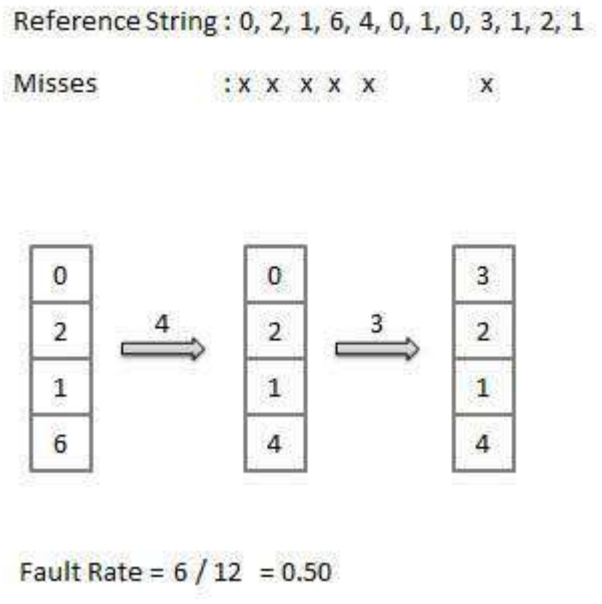
Easy to implement, keep a list, replace pages from the tail and add new pages at the head.


Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Misses :x x x x x x x x x
0 2 1 6
4
4 2 1 6
0
4 0 1 6
3
4 0 3 6
1
4 0 3 1
2
2 0 3 1
Fault Rate = 9 / 12 = 0.75

Optimal Page algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.

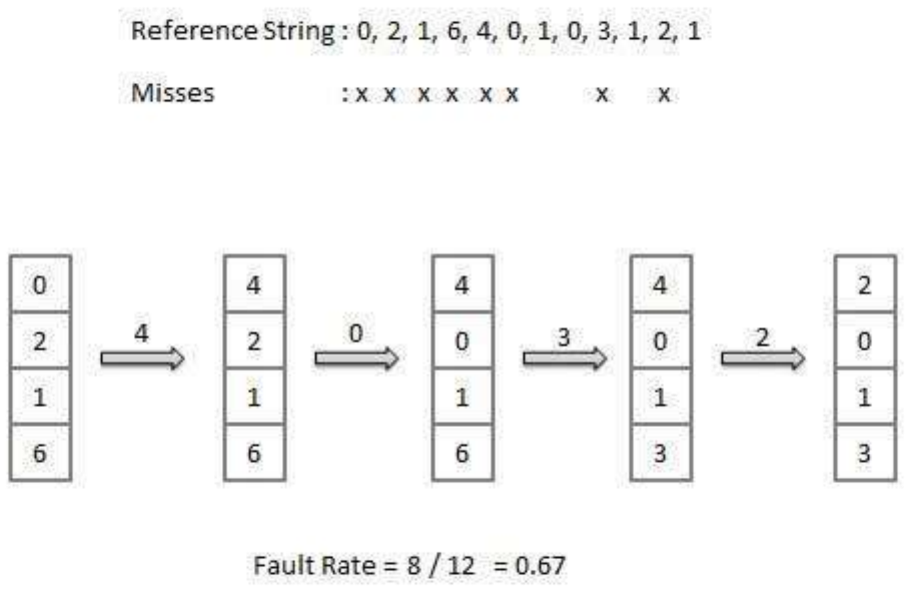
Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.


Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Misses :x x x x x x
0 2 1 6
4
0 2 1 4
3
3 2 1 4
Fault Rate = 6 / 12 = 0.50

Least Recently Used (LRU) algorithm

Page which has not been used for the longest time in main memory is the one which will be selected for replacement.

Easy to implement, keep a list, replace pages by looking back into time.



Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x

Fault Rate = 8 / 12 = 0.67

What is a Page Fault?

If the referred page is not present in the main memory then there will be a miss and the concept is called Page miss or page fault. The CPU has to access the missed page from the secondary memory. If the number of page fault is very high then the effective access time of the system will become very high.

What is Thrashing?

If the number of page faults is equal to the number of referred pages or the number of page faults are so high so that the CPU remains busy in just reading the pages from the secondary memory then the effective access time will be the time taken by the CPU to read one word from the secondary memory and it will be so high. The concept is called thrashing.

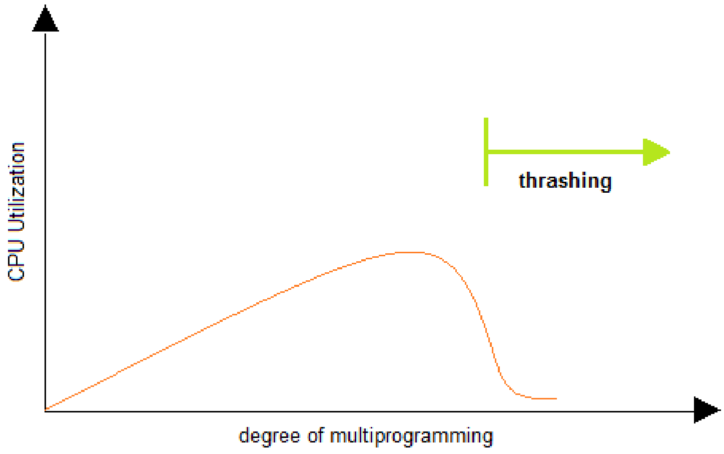
If the page fault rate is PF %, the time taken in getting a page from the secondary memory and again restarting is S (service time) and the memory access time is ma then the effective access time can be given as;

EAT = PF X S + (1 - PF) X (ma)

Thrashing

A process that is spending more time paging than executing is said to be thrashing. In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the proccesses are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.



To prevent thrashing we must provide processes with as many frames as they really need "right now".

UNIT – IV

STORAGE MANAGEMENT: File system-Concept of a file, access methods, directory structure, file system mounting, file sharing, protection. (**T1: Ch-10**) **SECONDARY-STORAGE STRUCTURE:** Overview of mass storage structure, disk structure, disk attachment, disk scheduling algorithms, swap space management, stable storage implementation, and tertiary storage structure (**T1: Ch-12**).

File Concept

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between 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. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.

The information in a file is defined by its creator. Many different types of information maybe stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, 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 functions, each of which is further organized as declarations followed by executable statements. An **executable file** is a series of code sections that the loader can bring into memory and execute.

File Attributes

file's attributes vary from one operating system to another but typically consist of these:

- **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 the non-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

Creating a file.

Writing a file: The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

Reading a file: system needs to keep a **read pointer** to the location in the file where the next read is to take place. the current operation location can be kept as a per-process **current file-position pointer**.

Repositioning within a file: This file operation is also known as a file **seek**.

Deleting a file

Truncating a file

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open() system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files.

Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each close() decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

In summary, several pieces of information are associated with an open file.

File pointer.

File-open count

Disk location of the file.

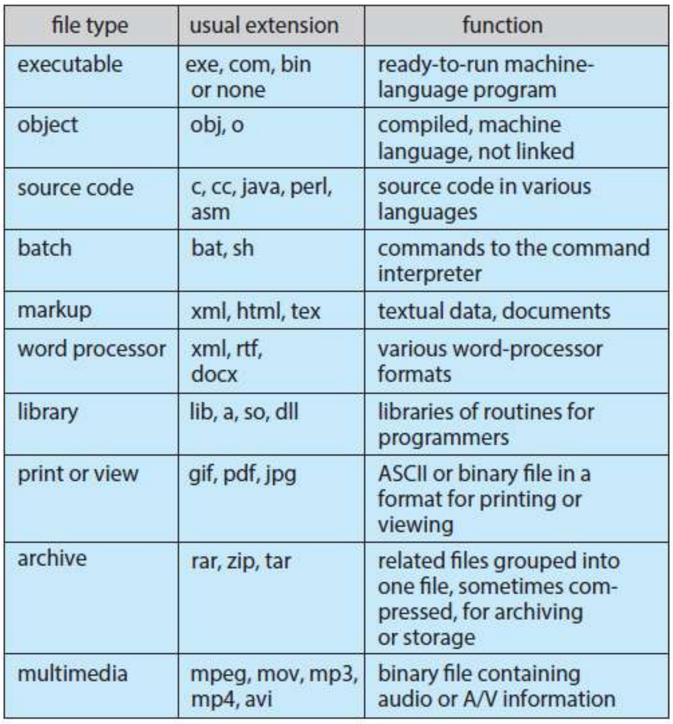
Access rights

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file.

File Types

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period.



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, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 11.3 Common file types.

Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. while others support many access methods, and choosing the right one for a particular application is a major design problem.

There are three ways to access a file into computer system: Sequential Access, Direct Access, Index sequential Method.

Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. Reads and writes make up the bulk of the operations on a file. 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).

Key points:

1. Data is accessed one record right after another record in an order.
2. When we use read command, it move ahead pointer by one
3. When we use write command, it will allocate memory and move the pointer to the end of the file
4. Such a method is reasonable for tape.

Direct Access

Another method is **direct access** (or **relative access**). Here, 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. 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().

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is1, and so on. When file is used, information is read and accessed into computer memory and there are several ways to accesses these information of the file.

Index sequential method –

It is the other method of accessing a file. Which is built on the top of the direct access method. these methods we construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index and then by the help of pointer we access the file directly.

Key points:

It is built on top of Sequential access.

It control the pointer by using index.

Directory and Disk Structure

For example, a disk can be partitioned into quarters, and each quarter can hold a separate file system.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space. A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a **volume**. The volume may be a subset of a device, a whole device, or multiple devices linked together into a RAID set. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a device directory or volume table of contents. The device directory (more commonly known simply as the directory) records information—such as name, location, size, and type—for all files on that volume. Figure 11.7 shows a typical file-system organization.

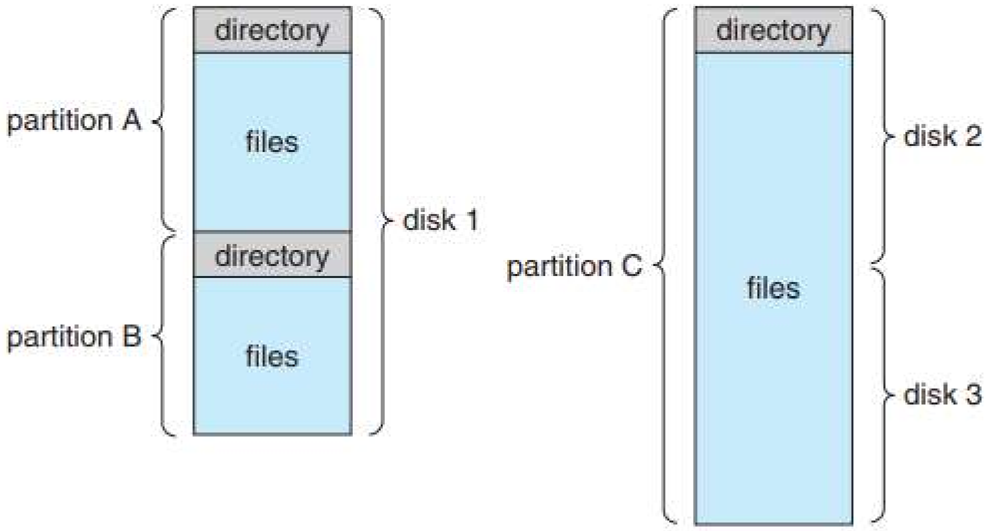


Figure 11.7 A typical file-system organization.

Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

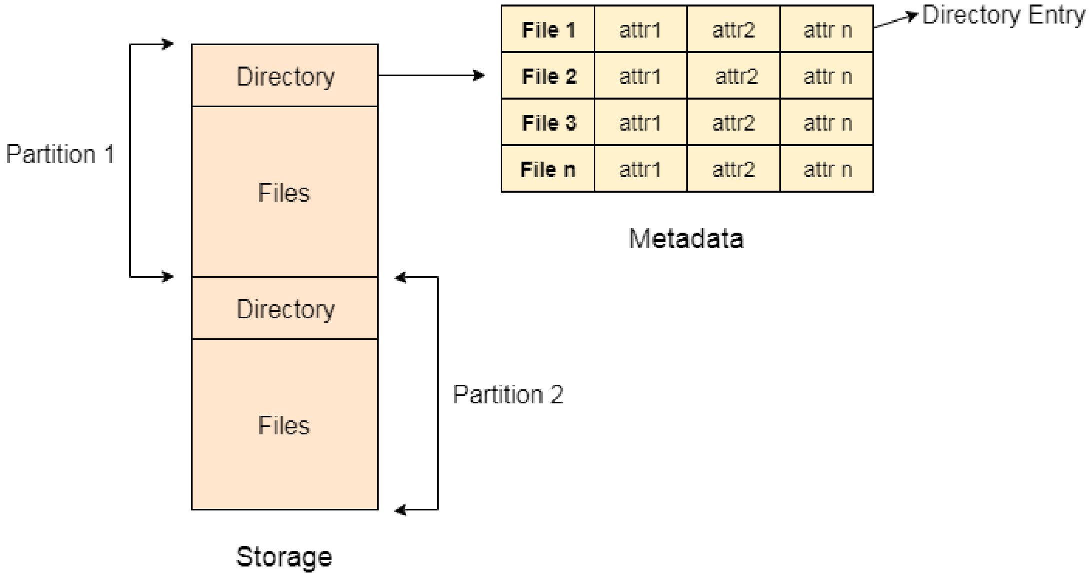
Operations that are to be performed on a directory:

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

What is a directory?

Directory can be defined as the listing of the related files on the disk. The directory may store some or the entire file attributes. To get the benefit of different file systems on the different operating systems, A hard disk can be divided into the number of partitions of different sizes. The partitions are also called volumes or mini disks.

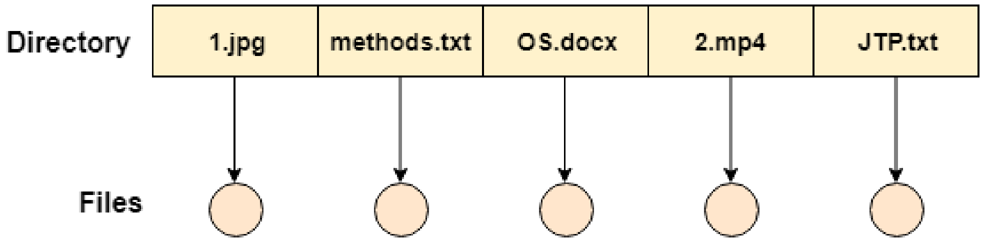
Each partition must have at least one directory in which, all the files of the partition can be listed. A directory entry is maintained for each file in the directory which stores all the information related to that file.


Partition 1
Directory
Files
Directory
Files
Partition 2
Storage
File 1 attr1 attr2 attr n
File 2 attr1 attr2 attr n
File 3 attr1 attr2 attr n
File n attr1 attr2 attr n
Directory Entry
Metadata

A directory can be viewed as a file which contains the Meta data of the bunch of files.

Single Level Directory

The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.



Single Level Directory

This type of directories can be used for a simple system.

Advantages

1. Implementation is very simple.
2. If the sizes of the files are very small then the searching becomes faster.
3. File creation, searching, deletion is very simple since we have only one directory.

Disadvantages

1. We cannot have two files with the same name.
2. The directory may be very big therefore searching for a file may take so much time.
3. Protection cannot be implemented for multiple users.
4. There are no ways to group same kind of files.
5. Choosing the unique name for every file is a bit complex and limits the number of files in the system because most of the Operating System limits the number of characters used to construct the file name.

Two Level Directory

In two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.

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. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

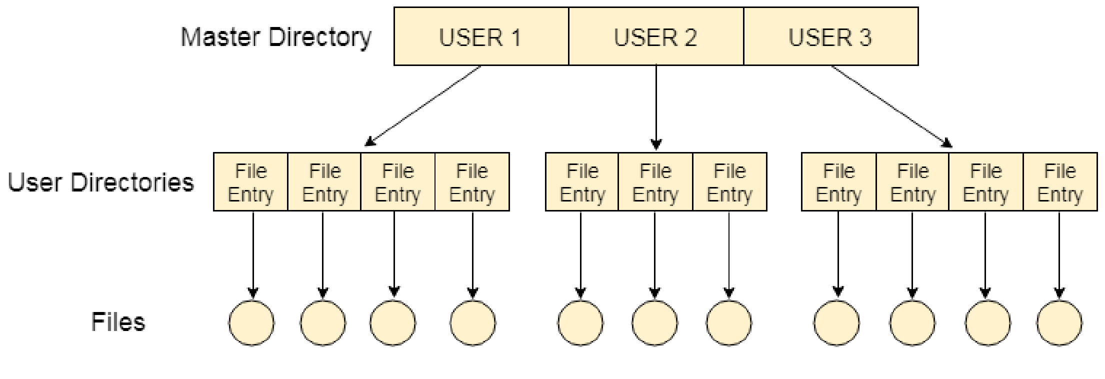
- There are two ways to specify a file path:

Absolute Path

- In this path we can reach to a specified file from the main or root directory.
- In this case current directory is not involved; file path is specified starting from the root directory.

Relative Path

- The user working in any directory that directory is called current directory.
- To reach to a specified file we have to search from the current directory.



Two Level Directory

Characteristics of two level directory system

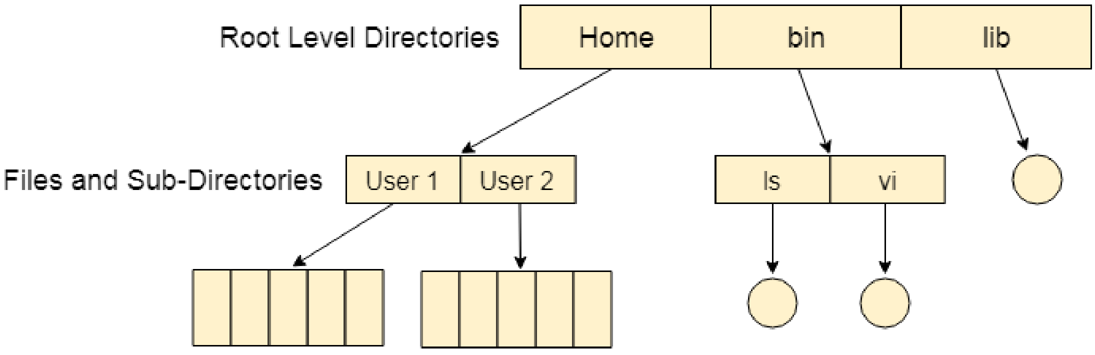
1. Each files has a path name as */User-name/directory-name/*
2. Different users can have the same file name.
3. Searching becomes more efficient as only one user's list needs to be traversed.
4. The same kind of files cannot be grouped into a single directory for a particular user.

Tree Structured Directory

In Tree structured directory system, any directory entry can either be a file or sub directory. Tree structured directory system overcomes the drawbacks of two level directory system. The similar kind of files can now be grouped in one directory.

Each user has its own directory and it cannot enter in the other user's directory. However, the user has the permission to read the root's data but he cannot write or modify this. Only administrator of the system has the complete access of root directory.

Searching is more efficient in this directory structure. The concept of current working directory is used. A file can be accessed by two types of path, either relative or absolute. In tree structured directory systems, the user is given the privilege to create the files as well as directories.



The Structured Directory System

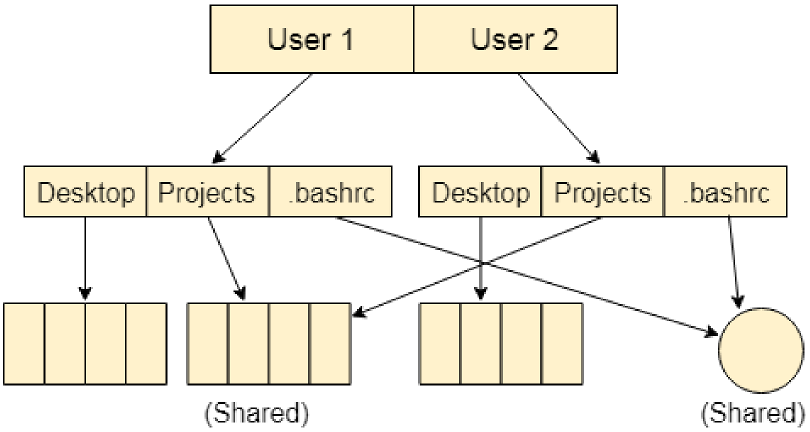
Acyclic-Graph Structured Directories

The tree structured directory system doesn't allow the same file to exist in multiple directories therefore sharing is major concern in tree structured directory system. We can provide sharing by making the directory an acyclic graph. In this system, two or more directory entry can point to the same file or sub directory. That file or sub directory is shared between the two directory entries.

These kinds of directory graphs can be made using links or aliases. We can have multiple paths for a same file. Links can either be symbolic (logical) or hard link (physical).

If a file gets deleted in acyclic graph structured directory system, then

1. In the case of soft link, the file just gets deleted and we are left with a dangling pointer.
2. In the case of hard link, the actual file will be deleted only if all the references to it gets deleted.



Acyclic-Graph Structured Directory System

File Systems

File system is the part of the operating system which is responsible for file management. It provides a mechanism to store the data and access to the file contents including data and programs. Some Operating systems treats everything as a file for example Ubuntu.

The File system takes care of the following issues

- **File Structure**

  We have seen various data structures in which the file can be stored. The task of the file system is to maintain an optimal file structure.
- **Recovering Free space**

  Whenever a file gets deleted from the hard disk, there is a free space created in the disk. There can be many such spaces which need to be recovered in order to reallocate them to other files.
- **disk space assignment to the files**

  The major concern about the file is deciding where to store the files on the hard disk.
- **tracking data location**

  A File may or may not be stored within only one block. It can be stored in the non contiguous blocks on the disk. We need to keep track of all the blocks on which the part of the files reside.

File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a file system to mount and a ***mount point*** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.
- File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted.
- File systems may be mounted read-only, or have other restrictions imposed.

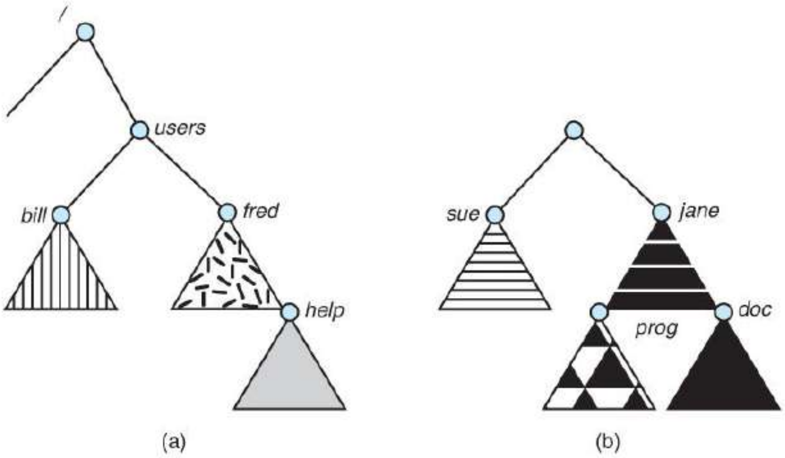


Figure 11.14 - File system. (a) Existing system. (b) Unmounted volume.

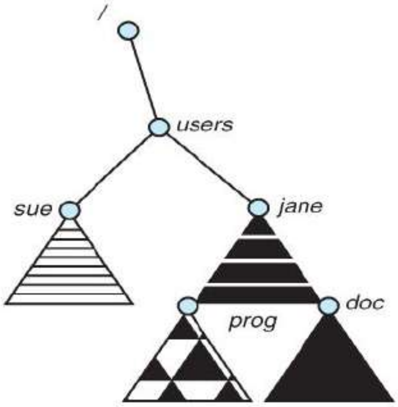


Figure 11.15 - Mount point.

- The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.
- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow file systems to be mounted to any directory in the file system, much like UNIX.

File Sharing

Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner (user) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner (**U**ser), the **G**roup, and to the rest of the world (the universe, a.k.a. **O**thers.)

  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
  - The original method was FTP (File Transfer Protocol), allowing individual files to be transported across systems as needed. FTP can be either account or password controlled, or ***anonymous***, not requiring any user name or password.
  - Various forms of ***distributed file systems*** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands.
  - The WWW has made it easy once again to access files on remote systems without mounting their file systems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically owns the files acts as a ***server***, and the system which mounts them is the ***client.***
- User IDs and group IDs must be consistent across both systems for the system to work properly
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)
- There are a number of security concerns involved in this model:
  - Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
  - Servers may restrict remote access to read-only.
  - Servers restrict which file systems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.
- The NFS (Network File System) is a classic example of such a system.

Protection

- Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies.
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

Access Control

In access-control list (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This technique has two undesirable consequences:

• Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.

• The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

• Owner. The user who created the file is the owner.

• Group. A set of users who are sharing the file and need similar access is a group, or work group.

• Universe. All other users in the system constitute the universe.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named book.tex. The protection associated with this file is as follows:

• Sara should be able to invoke all operations on the file.

• Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.

• All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

Types of Access

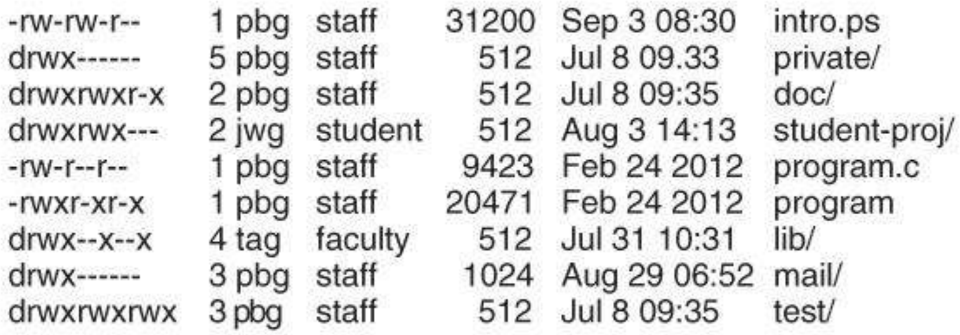
- The following low-level operations are often controlled:

  - Read - View the contents of the file
  - Write - Change the contents of the file.
  - Execute - Load the file onto the CPU and follow the instructions contained therein.
  - Append - Add to the end of an existing file.
  - Delete - Remove a file from the system.
  - List -View the name and other attributes of files on the system.
- Higher-level operations, such as copy, can generally be performed through combinations of the above.
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	**Files**	**Directories**
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write (change) file contents.	Change directory contents. Required to create or delete files.
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

- In addition there are some special bits that can also be applied:
  - The set user ID (SUID) bit and/or the set group ID (SGID) bits applied to executable files temporarily change the identity of whoever runs the program to match that of the owner / group of the executable program. This allows users running specific programs to have access to files (***while running that program***) to which they would normally be unable to access. Setting of these two bits is usually restricted to root, and must be done with caution, as it introduces a potential security leak.

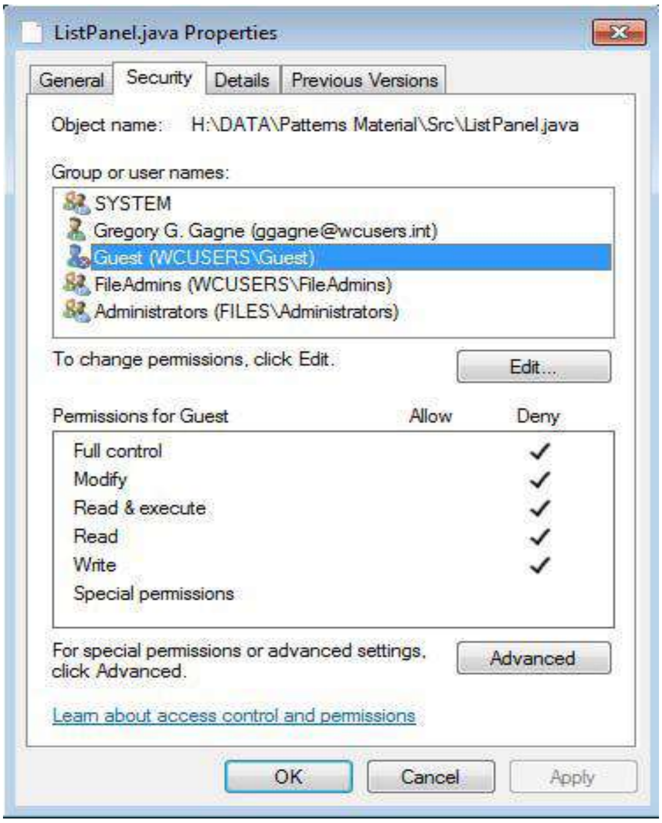
- The sticky bit on a directory modifies write permission, allowing users to only delete files for which they are the owner. This allows everyone to create files in /tmp, for example, but to only delete files which they have created, and not anyone else's.
- The SUID, SGID, and sticky bits are indicated with an S, S, and T in the positions for execute permission for the user, group, and others, respectively. If the letter is lower case, (s, s, t), then the corresponding execute permission is not also given. If it is upper case, (S, S, T), then the corresponding execute permission IS given.
- The numeric form of chmod is needed to set these advanced bits.



```
-rw-rw-r--    1 pbg  staff    31200  Sep 3 08:30   intro.ps
drwx------    5 pbg  staff      512  Jul 8 09.33   private/
drwxrwxr-x    2 pbg  staff      512  Jul 8 09:35   doc/
drwxrwx---    2 jwg  student    512  Aug 3 14:13   student-proj/
-rw-r--r--    1 pbg  staff     9423  Feb 24 2012   program.c
-rwxr-xr-x    1 pbg  staff    20471  Feb 24 2012   program
drwx--x--x    4 tag  faculty    512  Jul 31 10:31  lib/
drwx------    3 pbg  staff     1024  Aug 29 06:52  mail/
drwxrwxrwx    3 pbg  staff      512  Jul 8 09:35   test/
```

Sample permissions in a UNIX system.

- Windows adjusts files access through a simple GUI:


ListPanel.java Properties
General
Security
Details
Previous Versions
Edit...
Advanced
Learn about access control and permissions
OK
Cancel
Apply

Object name: H:\DATA\Patterns Material\Src\ListPanel.java

Group or user names:

- SYSTEM
- Gregory G. Gagne (ggagne@wcusers.int)
- Guest (WCUSERS\Guest)
- FileAdmins (WCUSERS\FileAdmins)
- Administrators (FILES\Administrators)

To change permissions, click Edit.

Permissions for Guest	Allow	Deny
Full control		✓
Modify		✓
Read & execute		✓
Read		✓
Write		✓
Special permissions		

For special permissions or advanced settings, click Advanced.

Figure - Windows 7 access-control list management.

Overview of mass storage structure

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 10.1). Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

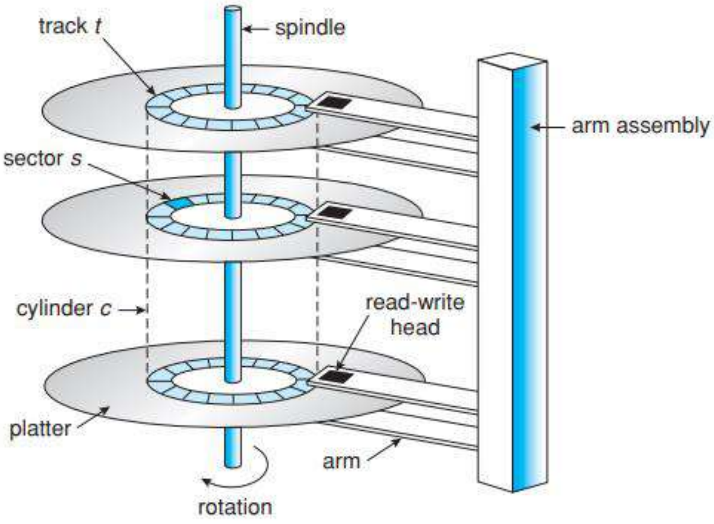


Figure 10.1 Moving-head disk mechanism.

A read–write head "flies" just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. The set of tracks that are at one arm position makes up a cylinder. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of **rotations per minute (RPM)**. Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Disk speed has two parts. The transfer rate is the rate at which data flow between the drive and the computer. The positioning time, or random-access time, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds Other forms of removable disks include CDs, DVDs, and Blu-ray discs as well as removable flash-memory devices known as flash drives (which are a type of solid-state drive).

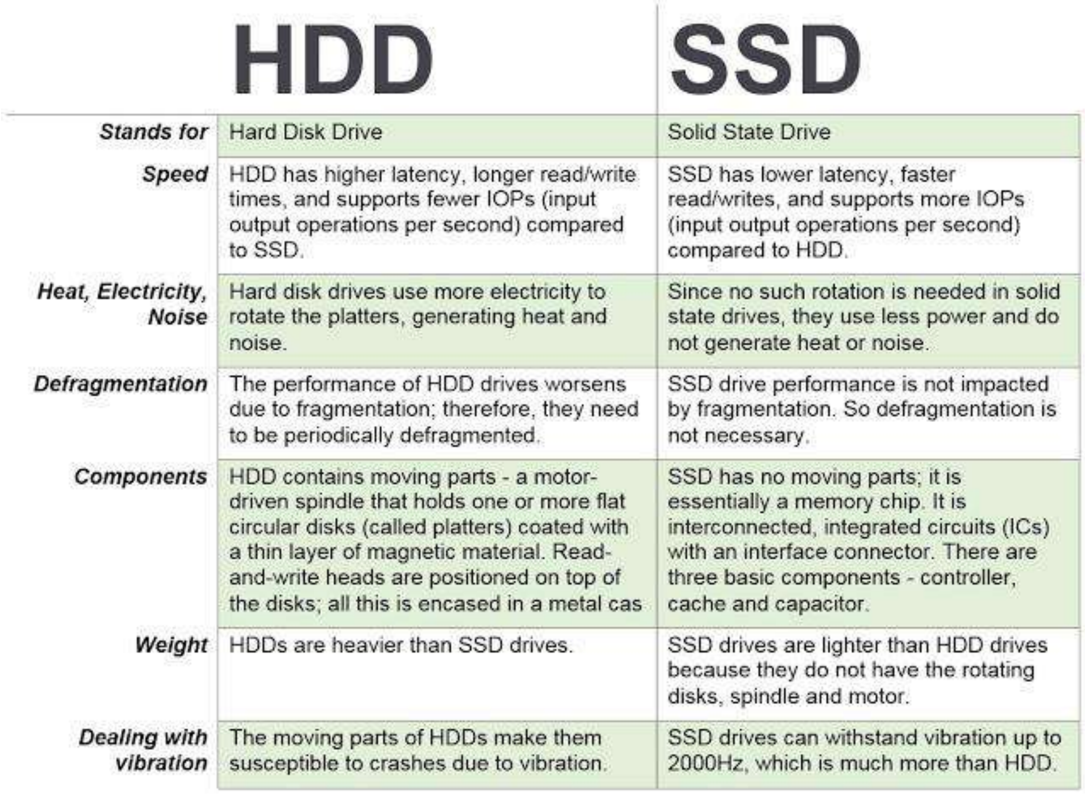
A disk drive is attached to a computer by a set of wires called an I/O bus. Several kinds of buses are available, including advanced technology attachment (ATA), serial ATA (SATA), eSATA, universal serial bus (USB), and fibre channel (FC). The data transfers on a bus are carried out by special electronic processors called controllers. The host controller is the controller at the computer end of the bus. A disk controller is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports.

Solid-State Disks

Sometimes old technologies are used in new ways as economics change or the technologies evolve. An example is the growing importance of solid-state disks, or SSDs. Simply described, an SSD is nonvolatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips.

SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power. However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited.

SSDs are also used in some laptop computers to make them smaller, faster, and more energy-efficient. Because SSDs can be much faster than magnetic disk drives, standard bus interfaces can cause a major limit on throughput.



	HDD	SSD
Stands for	Hard Disk Drive	Solid State Drive
Speed	HDD has higher latency, longer read/write times, and supports fewer IOPs (input output operations per second) compared to SSD.	SSD has lower latency, faster read/writes, and supports more IOPs (input output operations per second) compared to HDD.
Heat, Electricity, Noise	Hard disk drives use more electricity to rotate the platters, generating heat and noise.	Since no such rotation is needed in solid state drives, they use less power and do not generate heat or noise.
Defragmentation	The performance of HDD drives worsens due to fragmentation; therefore, they need to be periodically defragmented.	SSD drive performance is not impacted by fragmentation. So defragmentation is not necessary.
Components	HDD contains moving parts - a motor-driven spindle that holds one or more flat circular disks (called platters) coated with a thin layer of magnetic material. Read-and-write heads are positioned on top of the disks; all this is encased in a metal cas	SSD has no moving parts; it is essentially a memory chip. It is interconnected, integrated circuits (ICs) with an interface connector. There are three basic components - controller, cache and capacitor.
Weight	HDDs are heavier than SSD drives.	SSD drives are lighter than HDD drives because they do not have the rotating disks, spindle and motor.
Dealing with vibration	The moving parts of HDDs make them susceptible to crashes due to vibration.	SSD drives can withstand vibration up to 2000Hz, which is much more than HDD.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another

Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-5 and SDLT.

Disk Structure

Modern magnetic disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes.

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

Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM 10.3 Disk Attachment 471 and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

Disk Attachment

Computers access disk storage in two ways. One way is via I/O ports (or host-attached storage); this is common on small systems. The other way is via a remote host in a distributed file system; this is referred to as network-attached storage.

Host-Attached Storage

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies. The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. A newer, similar protocol that has simplified cabling is SATA.

High-end workstations and servers generally use more sophisticated I/O architectures such as fibre channel (FC), a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. It has two variants. One is a large switched fabric having a 24-bit address space. This variant is expected to dominate in the future and is the basis of storage-area networks (SANs), because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication.

The other FC variant is an arbitrated loop (FC-AL) that can address 126 devices (drives and controllers). A wide variety of storage devices are suitable for use as host-attached storage. Among these are hard disk drives, RAID arrays, and CD, DVD, and tape drives.

Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.
- NAS can be implemented using SCSI cabling, or ISCSI uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.

- NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

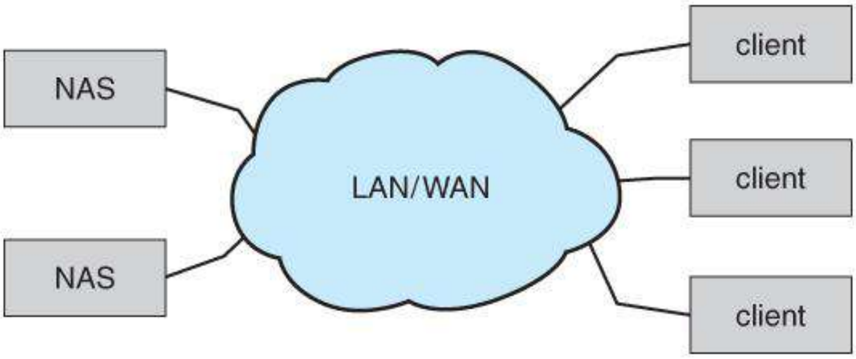
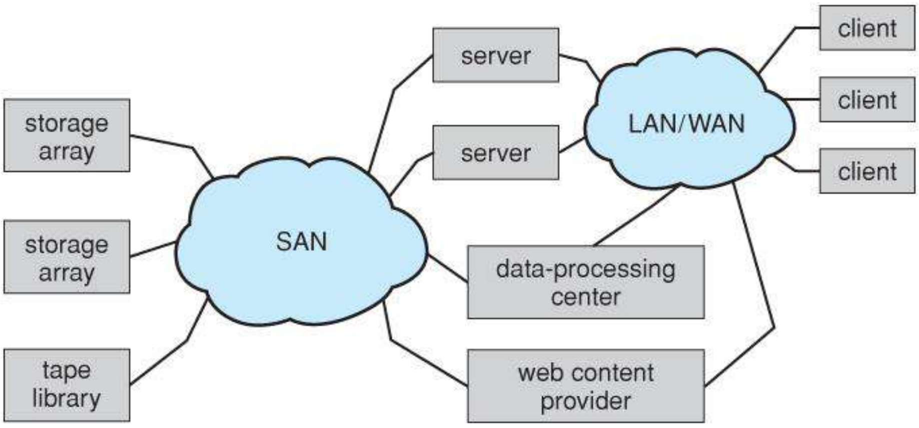


Figure - Network-attached storage.

Storage-Area Network

- A Storage-Area Network, SAN, connects computers and storage devices in a network, using storage protocols instead of network protocols.
- One advantage of this is that storage access does not tie up regular networking bandwidth.
- SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.
- SAN is also controllable, allowing restricted access to certain hosts and devices.



Disk Scheduling Algorithms

As we know, a process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.

However, the operating system must be fare enough to satisfy each request and at the same time, operating system must maintain the efficiency and speed of process execution.

The technique that operating system uses to determine the request which is to be satisfied next is called disk scheduling.

Let's discuss some important terms related to disk scheduling.

Seek Time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational Latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

Transfer Time

It is the time taken to transfer the data.

Disk Access Time

Disk access time is given as,

Disk Access Time = Rotational Latency + Seek Time + Transfer Time

Disk Response Time

It is the average of time spent by each request waiting for the IO operation.

Purpose of Disk Scheduling

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

Goal of Disk Scheduling Algorithm

- Fairness
- High throughout
- Minimal traveling head time

Disk Scheduling Algorithms

The list of various disks scheduling algorithm is given below. Each algorithm is carrying some advantages and disadvantages. The limitation of each algorithm leads to the evolution of a new algorithm.

- FCFS scheduling algorithm
- SSTF (shortest seek time first) algorithm
- SCAN scheduling
- C-SCAN scheduling
- LOOK Scheduling
- C-LOOK scheduling

FCFS Scheduling Algorithm

It is the simplest Disk Scheduling algorithm. It services the IO requests in the order in which they arrive. There is no starvation in this algorithm, every request is serviced.

Disadvantages

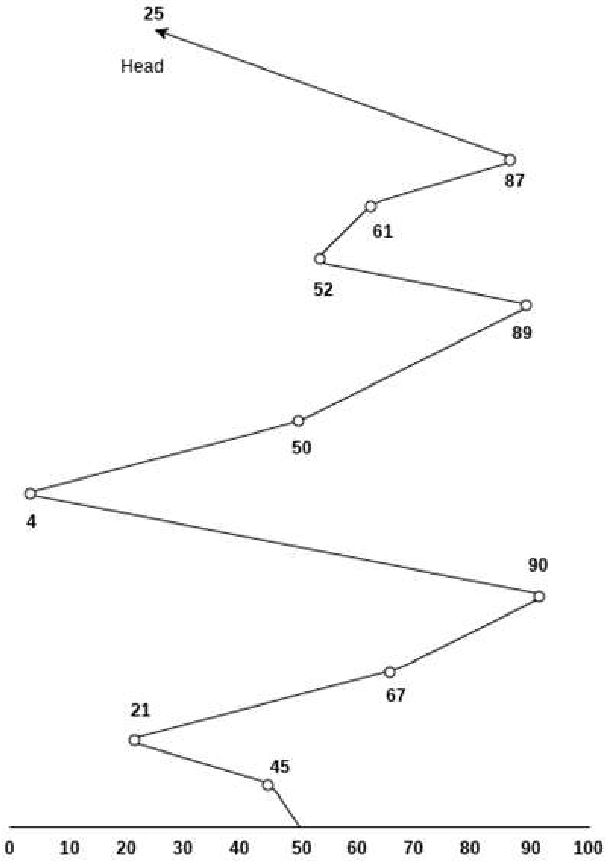
- The scheme does not optimize the seek time.
- The request may come from different processes therefore there is the possibility of inappropriate movement of the head.

Example

Consider the following disk request sequence for a disk with 100 tracks 45, 21, 67, 90, 4, 50, 89, 52, 61, 87, 25

Head pointer starting at 50 and moving in left direction. Find the number of head movements in cylinders using FCFS scheduling.

Solution



Number of cylinders moved by the head

= (50-45)+(45-21)+(67-21)+(90-67)+(90-4)+(50-4)+(89-50)+(61-52)+(87-61)+(87-25)

= 5 + 24 + 46 + 23 + 86 + 46 + 49 + 9 + 26 + 62

= 376

SSTF Scheduling Algorithm

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

It allows the head to move to the closest track in the service queue.

Disadvantages

- It may cause starvation for some requests.
- Switching direction on the frequent basis slows the working of algorithm.
- It is not the most optimal algorithm.

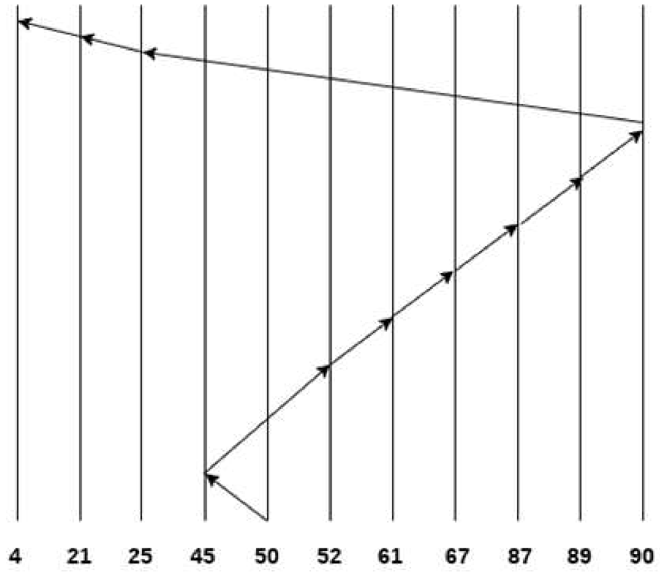
Example

Consider the following disk request sequence for a disk with 100 tracks

45, 21, 67, 90, 4, 89, 52, 61, 87, 25

Head pointer starting at 50. Find the number of head movements in cylinders using SSTF scheduling.

Solution:



Number of cylinders = 5 + 7 + 9 + 6 + 20 + 2 + 1 + 65 + 4 + 17 = 136

Scan Algorithm

It is also called as Elevator Algorithm. In this algorithm, the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path.

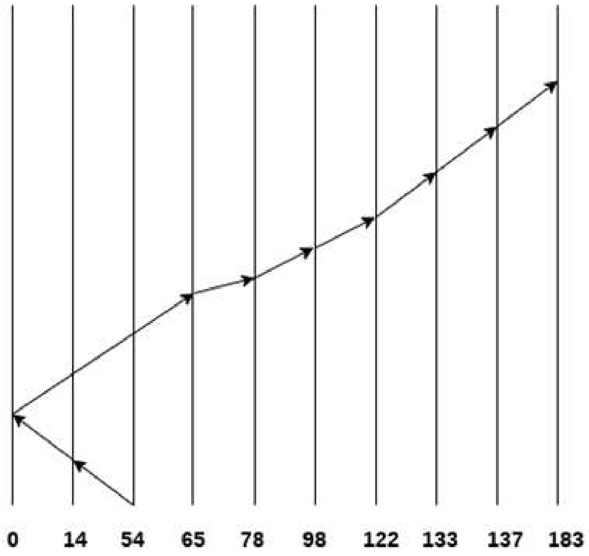
It works in the way an elevator works, elevator moves in a direction completely till the last floor of that direction and then turns back.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using SCAN scheduling.



Number of Cylinders = 40 + 14 + 65 + 13 + 20 + 24 + 11 + 4 + 46 = 237

C-SCAN algorithm

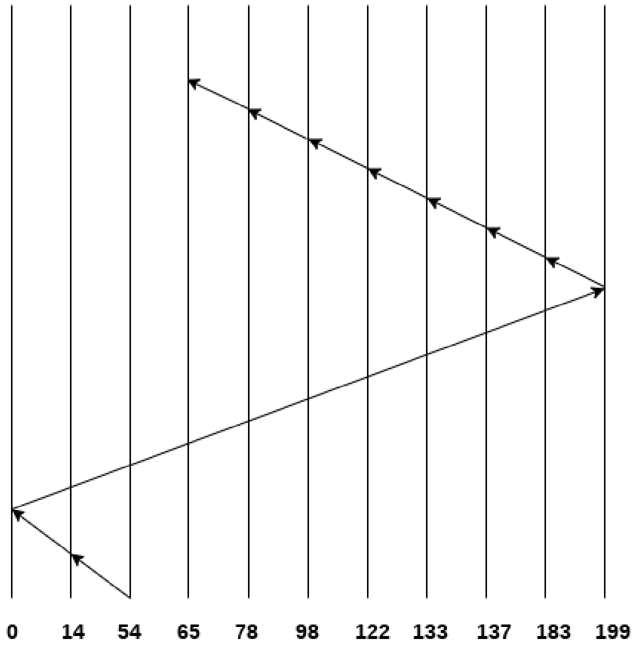
In C-SCAN algorithm, the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C-SCAN scheduling.



No. of cylinders crossed = 40 + 14 + 199 + 16 + 46 + 4 + 11 + 24 + 20 + 13 = 387

Look Scheduling

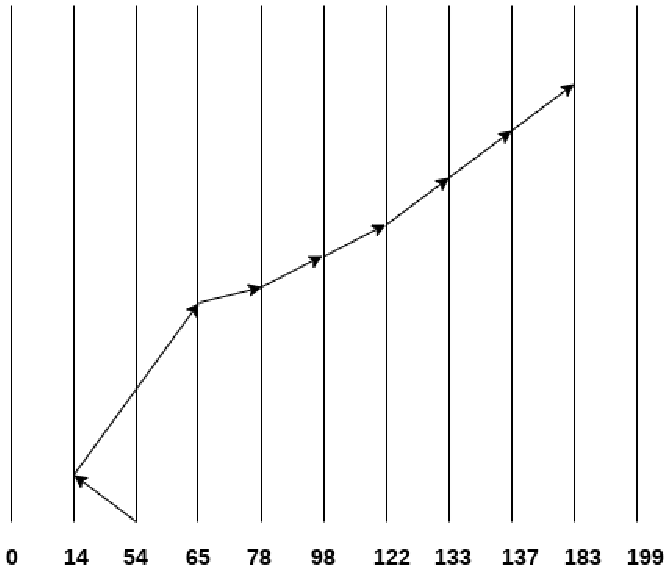
It is like SCAN scheduling Algorithm to some extant except the difference that, in this scheduling algorithm, the arm of the disk stops moving inwards (or outwards) when no more request in that direction exists. This algorithm tries to overcome the overhead of SCAN algorithm which forces disk arm to move in one direction till the end regardless of knowing if any request exists in the direction or not.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using LOOK scheduling.



Number of cylinders crossed = 40 + 51 + 13 + +20 + 24 + 11 + 4 + 46 = 209

C Look Scheduling

C Look Algorithm is similar to C-SCAN algorithm to some extent. In this algorithm, the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.

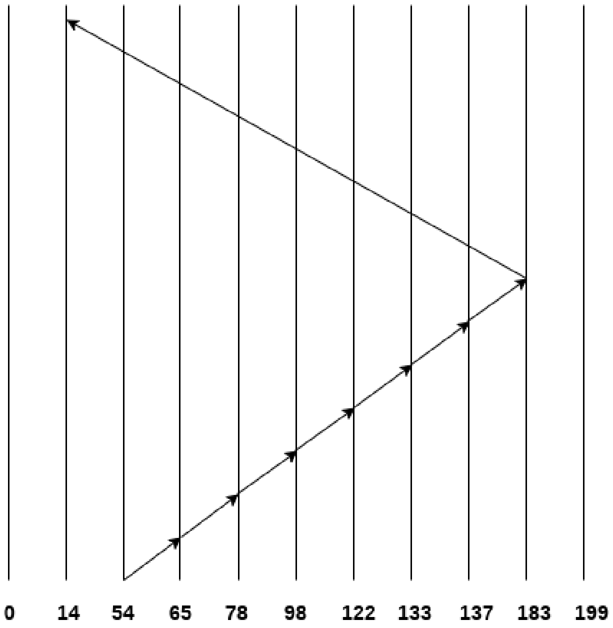
It is different from C SCAN algorithm in the sense that, C SCAN force the disk arm to move till the last cylinder regardless of knowing whether any request is to be serviced on that cylinder or not.

Example

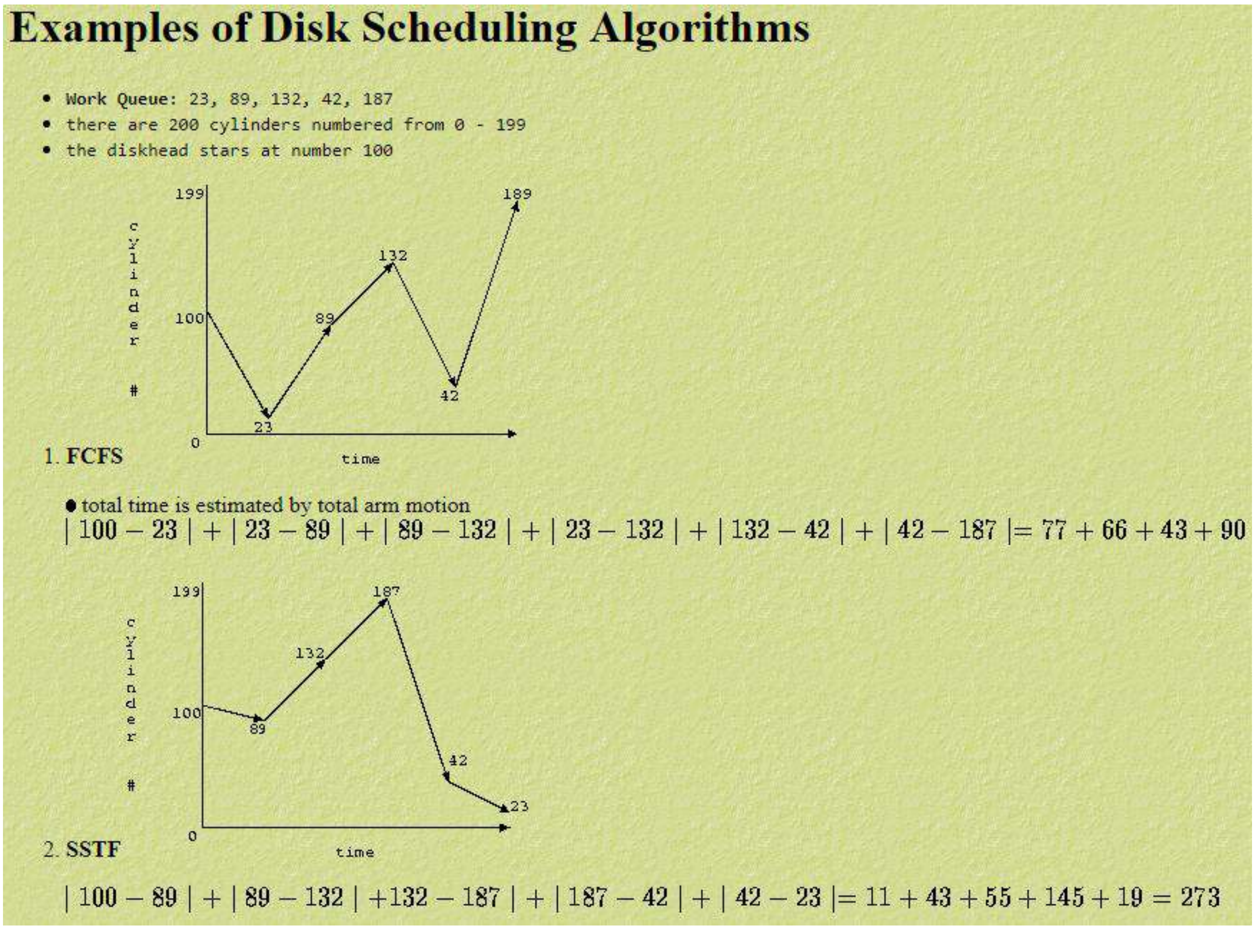
Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C LOOK scheduling.



Number of cylinders crossed = 11 + 13 + 20 + 24 + 11 + 4 + 46 + 169 = 298



Examples of Disk Scheduling Algorithms

- Work Queue: 23, 89, 132, 42, 187
- there are 200 cylinders numbered from 0 - 199
- the diskhead stars at number 100

199
189
132
cylinder #
100
89
42
23
0
time

1. **FCFS**

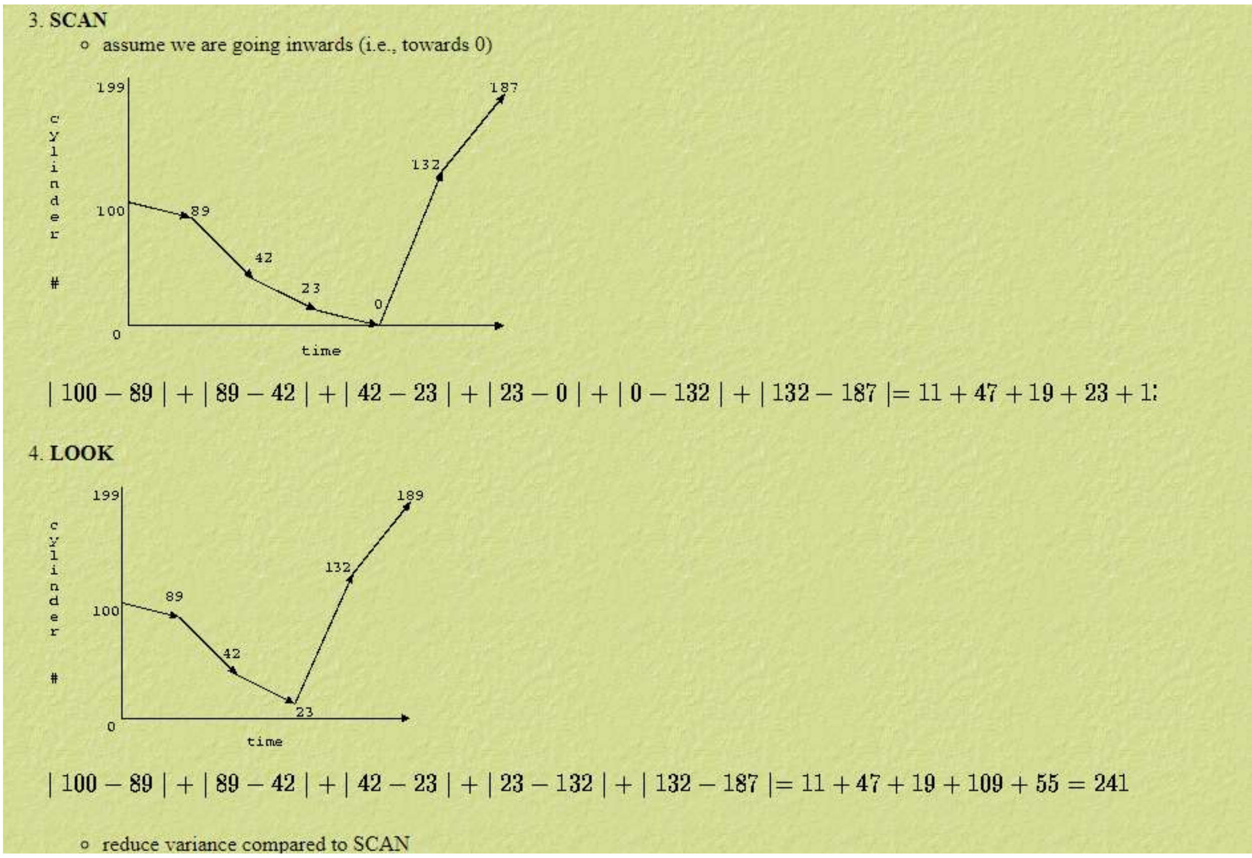
● total time is estimated by total arm motion

$| 100 - 23 | + | 23 - 89 | + | 89 - 132 | + | 23 - 132 | + | 132 - 42 | + | 42 - 187 | = 77 + 66 + 43 + 90$

199
187
132
cylinder #
100
89
42
23
0
time

2. **SSTF**

$| 100 - 89 | + | 89 - 132 | +132 - 187 | + | 187 - 42 | + | 42 - 23 | = 11 + 43 + 55 + 145 + 19 = 273$



3. **SCAN**

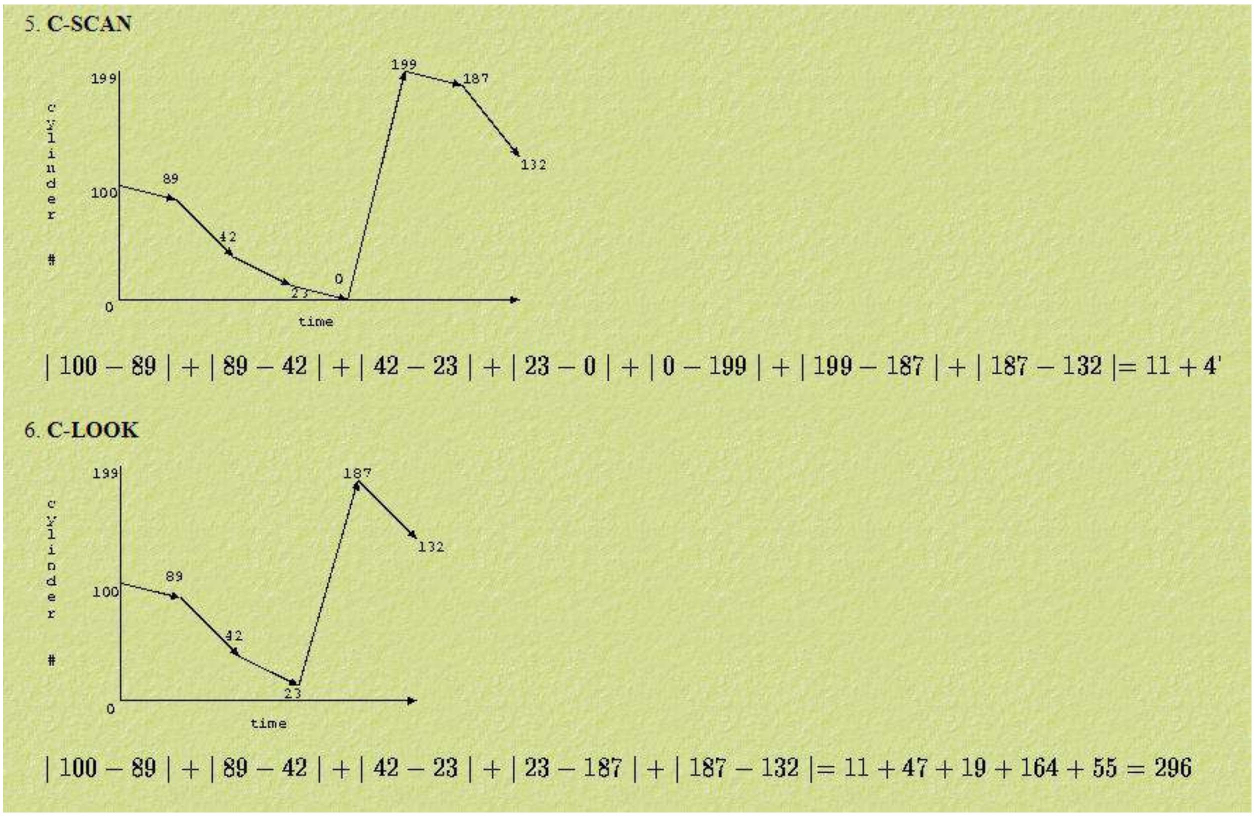
- assume we are going inwards (i.e., towards 0)

$| 100 - 89 | + | 89 - 42 | + | 42 - 23 | + | 23 - 0 | + | 0 - 132 | + | 132 - 187 | = 11 + 47 + 19 + 23 + 1$

4. **LOOK**

$| 100 - 89 | + | 89 - 42 | + | 42 - 23 | + | 23 - 132 | + | 132 - 187 | = 11 + 47 + 19 + 109 + 55 = 241$

- reduce variance compared to SCAN



5. **C-SCAN**

$| 100 - 89 | + | 89 - 42 | + | 42 - 23 | + | 23 - 0 | + | 0 - 199 | + | 199 - 187 | + | 187 - 132 | = 11 + 4$

6. **C-LOOK**

$| 100 - 89 | + | 89 - 42 | + | 42 - 23 | + | 23 - 187 | + | 187 - 132 | = 11 + 47 + 19 + 164 + 55 = 296$

Swap space management

A computer has sufficient amount of physical memory but most of times we need more so we swap some memory on disk. Swap space is a space on hard disk which is a substitute of physical memory. It is used as virtual memory which contains process memory image. Whenever our computer run short of physical memory it uses it's virtual memory and stores information in memory on disk. Swap space helps the computer's operating system in pretending that it have more RAM than it actually has. It is also called as swap file.This interchange of data between virtual memory and real memory is called as swapping and space on disk as "swap space".

Virtual memory is a combination of RAM and disk space that running processes can use. **Swap space** is the **portion of virtual memory** that is on the hard disk, used when RAM is full.

Swap space can be useful to computer in various ways:

- It can be used as a single contiguous memory which reduces i/o operations to read or write a file.
- Applications which are not used or are used less can be kept in swap file.
- Having sufficient swap file helps the system keep some physical memory free all the time.
- The space in physical memory which has been freed due to swap space can be used by OS for some other important tasks.

In operating systems such as Windows, Linux, etc systems provide a certain amount of swap space by default which can be changed by users according to their needs. If you don't want to use virtual memory you can easily disable it all together but in case if you run out of memory then kernel will kill some of the processes in order to create a sufficient amount of space in physical memory.

So it totally depends upon user whether he wants to use swap space or not.Alternatively, swap space can be created in a separate raw partition. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems (when it is used).

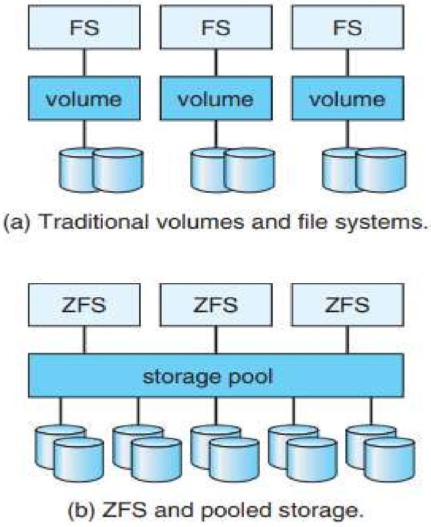
Stable-Storage Implementation

By definition, information residing in stable storage is never lost. To implement such storage, we need to replicate the required information on multiple storage devices (usually disks) with independent failure modes. We also need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state and that, when we are recovering from a failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery. A disk write results in one of three outcomes:

1. Successful completion. The data were written correctly on disk.
2. Partial failure. A failure occurred in the midst of transfer, so only some of the sectors were written with the new data, and the sector being written during the failure may have been corrupted.
3. Total failure. The failure occurred before the disk write started, so the previous data values on the disk remain intact.

Whenever a failure occurs during writing of a block, the system needs to detect it and invoke a recovery procedure to restore the block to a consistent state. To do that, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. Declare the operation complete only after the second write completes successfully.



(a) Traditional volumes and file systems.

(b) ZFS and pooled storage.

(a) Traditional volumes and file systems. (b) A ZFS pool and file systems.

During recovery from a failure, each pair of physical blocks is examined. If both are the same and no detectable error exists, then no further action is necessary. If one block contains a detectable error then we replace its contents with the value of the other block. If neither block contains a detectable error, but the blocks differ in content, then we replace the content of the first block with that of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change.

Tertiary storage structure

- ➢ Low cost is the defining characteristic of tertiary storage
- ➢ Generally, tertiary storage is built using removable media
- ➢ Common examples of removable media are floppy disks and CD-ROMs; other types are available

Removable Disks

- ➢ Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case

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

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

- ➢ A magneto-optic disk records data on a rigid platter coated with magnetic material

λ Laser heat is used to amplify a large, weak magnetic field to record a bit λ Laser light is also used to read data (Kerr effect)

λ The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes

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

WORM Disks

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

ν WORM (“Write Once, Read Many Times”) disks can be written only once νThin aluminum film sandwiched between two glass or plastic platters

ν To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered v Very durable and reliable v Read-only disks, such ad CD-ROM and DVD, come from the factory with the data pre-recorded

Tapes

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

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

v Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library

λstacker – library that holds a few tapes

λsilo – library that holds thousands of tapes

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

VARDHAMAN COLLEGE OF ENGINEERING

(AUTONOMOUS)

Shamshabad, Hyderabad – 501 218

DEPARTMENT OF INFORMATION TECHNOLOGY

B. Tech IV Semester (VCE-R19 Regulations)

IMPORTANT QUESTIONS

1) List the attributes of the file and discuss it
2) How operations are performed on the files? Explain each operation in detail
3) a) Identify the importance of the extension associated with the files.

   b) List the common file extensions associated with the same group of files
4) Explain file type and their function in detail
5) Compare and contrast of different file access methods
6) What is a directory? List the various operations performed on the directories?
7) Outline the advantages and disadvantages of single level directory
8) Compare and contrast two level directory and tree structured directory
9) What are the issues are associated with the file systems? Explain in detail

UNIT – V

PROTECTION: System protection-Goals of protection, principles of protection, domain of protection access matrix, implementation of access matrix, access control, revocation of access rights. (**T1: Ch-13**)

SECURITY: System security-The security problem, program threats, system and network threats, implementing security defenses, firewalling to protect systems(**T1: Ch -18**).

Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs.
- 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.

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

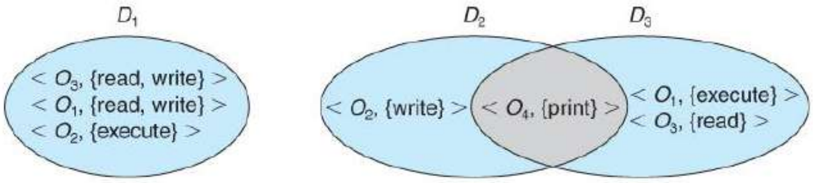
Note that ***mechanisms*** are distinct from ***policies***. Mechanisms determine ***how*** something will be done; policies decide ***what*** will be done.

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.

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.



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.

A domain can be realized in a variety of ways:

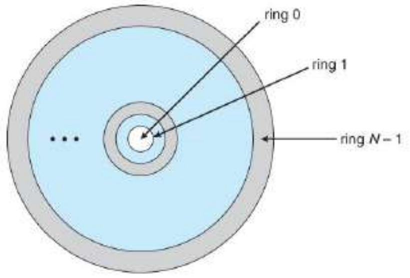
- Each ***user*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.
- Each ***process*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each ***procedure*** may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

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.

Example: MULTICS

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

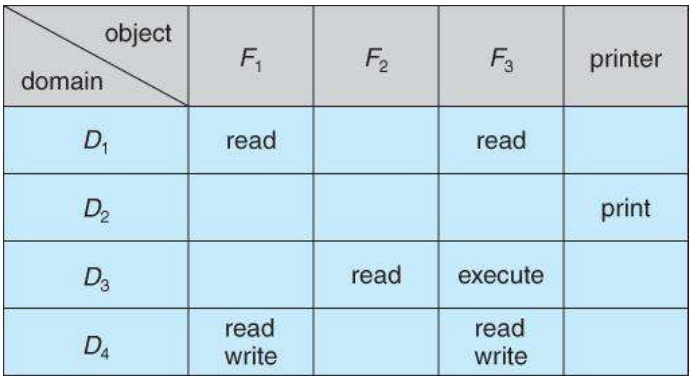


MULTICS ring structure

- 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.

Access Matrix

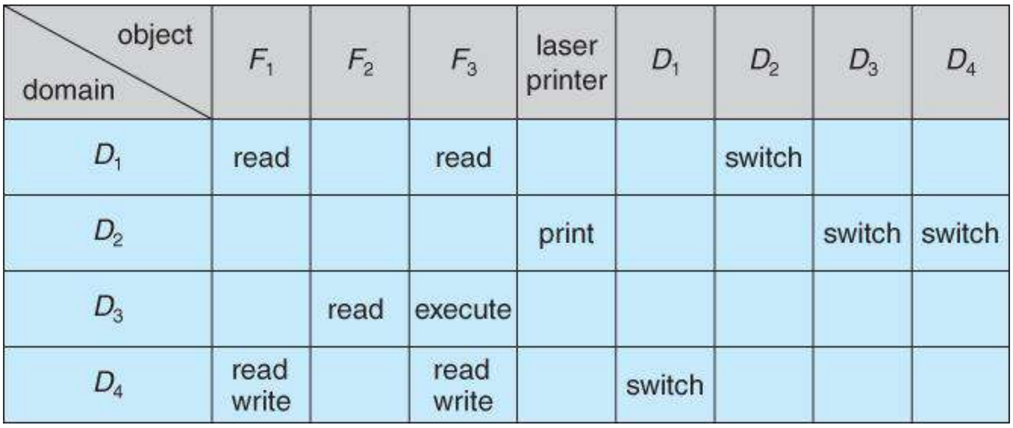
- The model of protection that we have been discussing can be viewed as an ***access matrix,*** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.



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

Access matrix

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

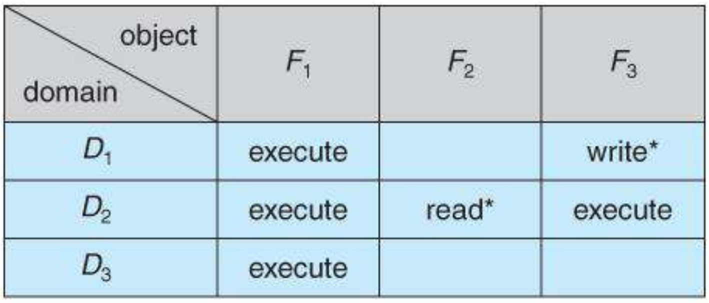


object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access matrix of Figure 14.3 with domains as objects

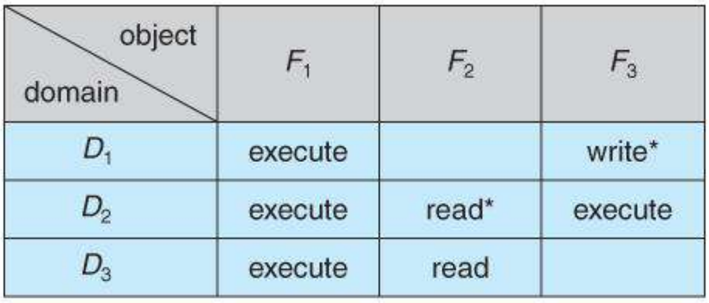
- The ability to ***copy*** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:

- 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 14.5 below:



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

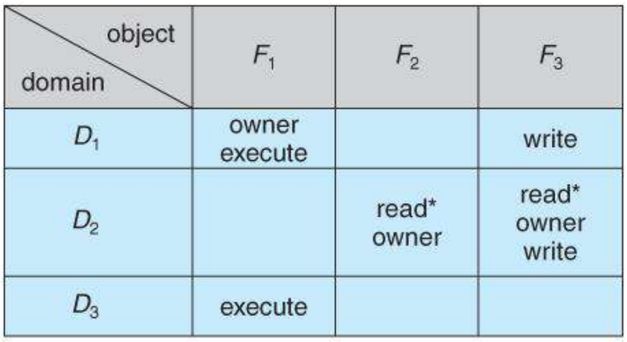


object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

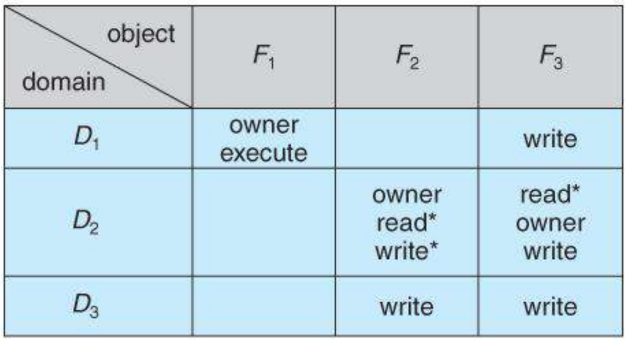
Access matrix with *copy* rights

- 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)

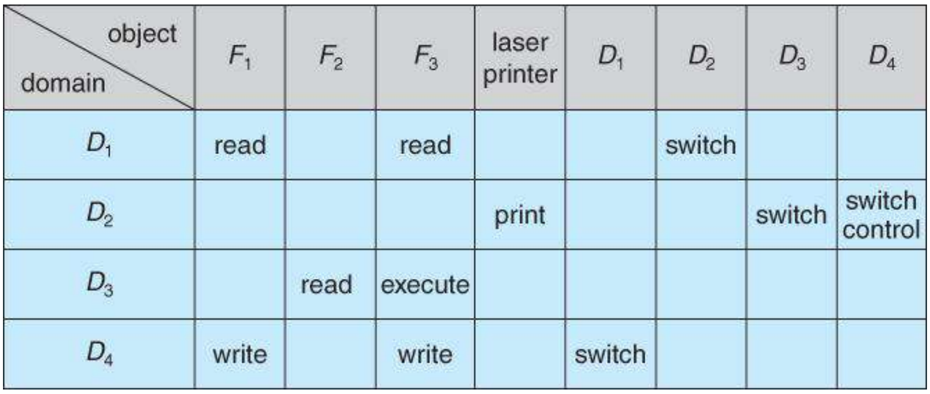


object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

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.



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

Modified access matrix of above Figure

Implementation of Access Matrix

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.

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.

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.

A Lock-Key Mechanism

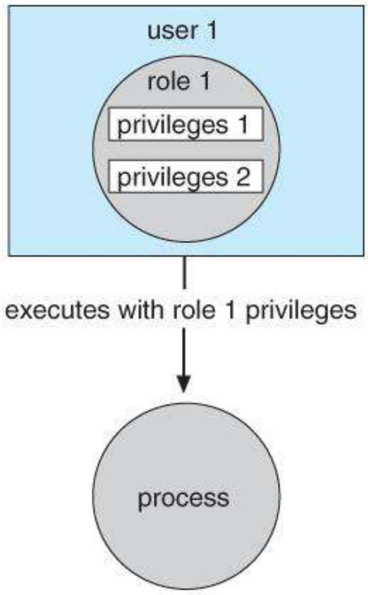
- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

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.

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.



Role-based access control in Solaris 10

Revocation of Access Rights

- The need to revoke access rights dynamically raises several questions:
    - Immediate versus delayed - If delayed, can we determine when the revocation will take place?
    - Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
    - Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
    - Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?
- With an access list scheme revocation is easy, immediate, and can be selective, general, partial, total, temporary, or permanent, as desired.
- With capabilities lists the problem is more complicated, because access rights are distributed throughout the system. A few schemes that have been developed include:
    - Reacquisition - Capabilities are periodically revoked from each domain, which must then re-acquire them.
    - Back-pointers - A list of pointers is maintained from each object to each capability which is held for that object.
    - Indirection - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry,

which may affect multiple processes, which must then re-acquire access rights to continue.

- Keys - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process.
    - A master key is associated with each object.
    - When a capability is created, its key is set to the object's master key.
    - As long as the capability's key matches the object's key, then the capabilities remain valid.
    - The object master key can be changed with the set-key command, thereby invalidating all current capabilities.
    - More flexibility can be added to this scheme by implementing a ***list*** of keys for each object, possibly in a global table.

Capability-Based Systems (Optional)

Example: Hydra

- Hydra is a capability-based system that includes both system-defined ***rights*** and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be
- Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities.
- The names of user-defined procedures must be identified to the protection system if it is to deal with user-defined rights.
- When an object is created, the names of operations defined on that object become ***auxiliary rights,*** described in a capability for an ***instance*** of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.
- Hydra also allows ***rights amplification,*** in which a process is deemed to be ***trustworthy,*** and thereby allowed to act on any object corresponding to its parameters.
- Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

Example: Cambridge CAP System

- The CAP system has two kinds of capabilities:

○ ***Data capability,*** used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.

○ ***Software capability,*** is protected but not interpreted by the CAP microcode.

▪ Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.

▪ When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability.

▪ This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.

▪ Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.

▪ Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

Language-Based Protection (Optional)

○ As systems have developed, protection systems have become more powerful, and also more specific and specialized.

    ○ To refine protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

Compiler-Based Enforcement

- In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.
- This approach has several advantages:
    1. Protection needs are simply declared, as opposed to a complex series of procedure calls.

2. Protection requirements can be stated independently of the support provided by a particular OS.
3. The means of enforcement need not be provided directly by the developer.
4. Declarative notation is natural, because access privileges are closely related to the concept of data types.

- Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.
- Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments cannot be modified, data segments can't be executed.)
- There are several areas in which compiler-based protection can be compared to kernel-enforced protection:
    - **Security.** Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.
    - **Flexibility.** A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.
    - **Efficiency.** The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that many checks can be made off-line, at compile time, rather that during execution.

The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:

0. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.
1. Specifying the ***type*** of operations a process may execute on a resource, such as reading or writing.
2. Specifying the ***order*** in which operations are performed on the resource, such as opening before reading.

Security

The Security Problem

We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved.

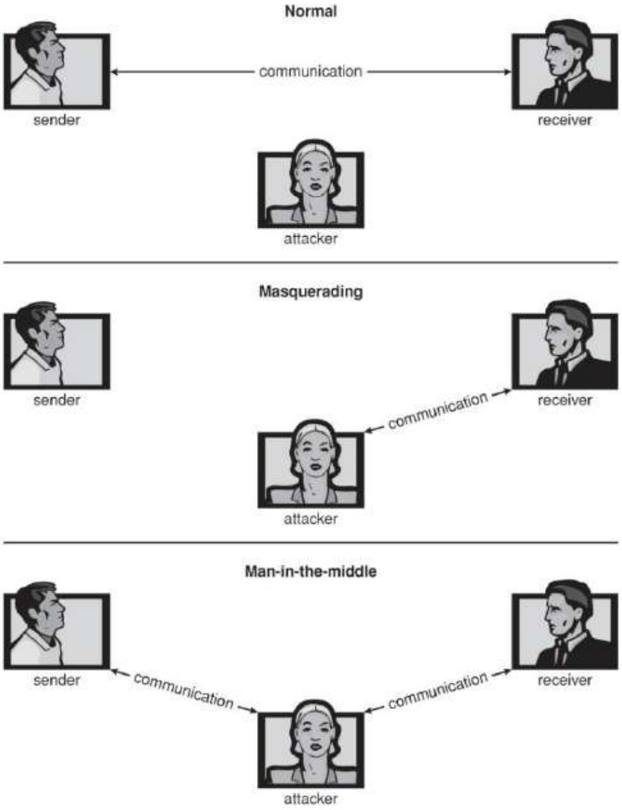
Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of protection from accidents. The following list includes several forms of accidental and malicious security violations. We should note that in our discussion of security, we use the terms **intruder** and **cracker** for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is the attempt to break security.

- **Breach of confidentiality**. This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.
- **Breach of integrity**. This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.
- **Breach of availability**. This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Website defacement is a common example of this type of security breach.
- **Theft of service**. This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.

- **Denial of service**. This violation involves preventing legitimate use of the system. **Denial-of-service (DOS)** attacks are sometimes accidental.

Methods to breach security

- 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.



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 user's 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.

Program Threats

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

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.
- 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.

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.

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.

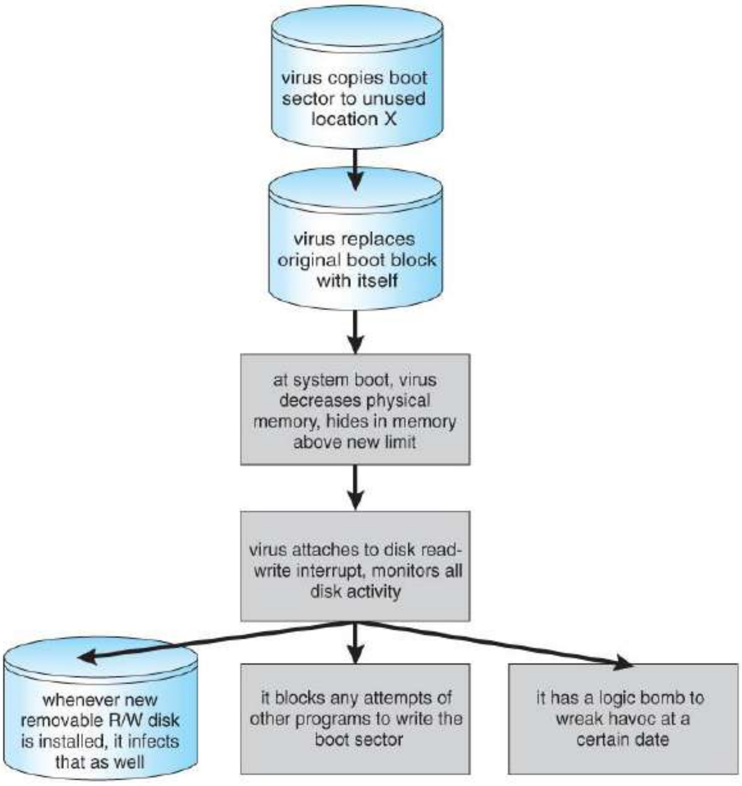
Stack and Buffer Overflow

- 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.)

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:


virus copies boot sector to unused location X
virus replaces original boot block with itself
at system boot, virus decreases physical memory, hides in memory above new limit
virus attaches to disk read-write interrupt, monitors all disk activity
whenever new removable R/W disk is installed, it infects that as well
it blocks any attempts of other programs to write the boot sector
it has a logic bomb to wreak havoc at a certain date

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.

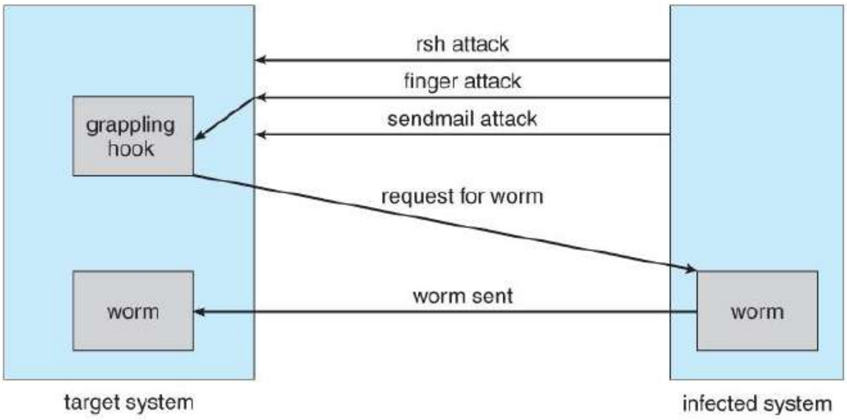
System and Network Threats

- 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.

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.



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".

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*** (http://www.insecure.org/nmap) and ***nessus*** (http://www.nessus.org). 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.

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. :-)

Implementing Security Defenses

Security Policy

The first step toward improving the security of any aspect of computing is to have a **security policy**. Policies vary widely but generally include a statement of what is being secured. For example, a policy might state that all outside accessible applications must have a code review before being deployed, or that users should not share their passwords, or that all connection points between a company and the outside must have port scans run every six months. Without a policy in place, it is impossible for users and administrators to know what is permissible, what is required, and what is not allowed. The policy is a road map to security, and if a site is trying to move from less secure to more secure, it needs a map to know how to get there. Once the security policy is in place, the people it affects should know it well. It should be their guide. The policy should also be a **living document** that is reviewed and updated periodically to ensure that it is still pertinent and still followed.

Vulnerability Assessment

How can we determine whether a security policy has been correctly implemented? The best way is to execute a vulnerability assessment. Such assessments can cover broad ground, from social engineering through risk assessment to port scans. **Risk assessment**, for example, attempts to value the assets of the entity in question (a program, a management team, a system, or a facility) and determine the odds that a security incident will affect the entity and decrease its value. When the odds of suffering a loss and the amount of the potential loss are known, a value can be placed on trying to secure the entity. The core activity of most vulnerability assessments is a **penetration test**, in which the entity is scanned for known vulnerabilities. A scan within an individual system can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized privileged programs, such as setuid programs
- Unauthorized programs in system directories
- Unexpectedly long-running processes

- Improper directory protections on user and system directories
- Improper protections on system data files, such as the password file, device drivers, or the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 15.2.1)
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

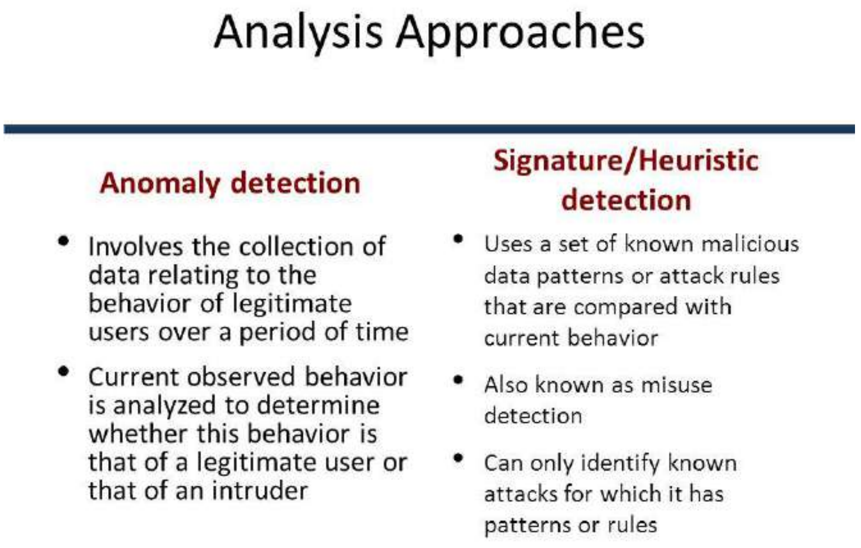
Any problems found by a security scan can be either fixed automatically or reported to the managers of the system. Networked computers are much more susceptible to security attacks than are standalone systems.

Intrusion(Obstruction) Detection

Securing systems and facilities is intimately linked to intrusion detection. **Intrusion detection**, as its name suggests, strives to detect attempted or successful intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion detection encompasses a wide array of techniques that vary on a number of axes, including the following:

- The time at which detection occurs. Detection can occur in real time (while the intrusion is occurring) or after the fact.
- The types of inputs examined to detect intrusive activity. These may include user-shell commands, process system calls, and network packet headers or contents. Some forms of intrusion might be detected only by correlating information from several such sources.
- The range of response capabilities. Simple forms of response include alerting an administrator to the potential intrusion or somehow halting the potentially intrusive activity.

These degrees of freedom in the design space for detecting intrusions have yielded a wide range of solutions, known as **intrusion-detection systems (IDSs)** and **intrusion-prevention systems (IDPs).** IDS systems raise an alarm when an intrusion is detected, while IDP systems act as routers, passing traffic unless an intrusion is detected.



Analysis Approaches

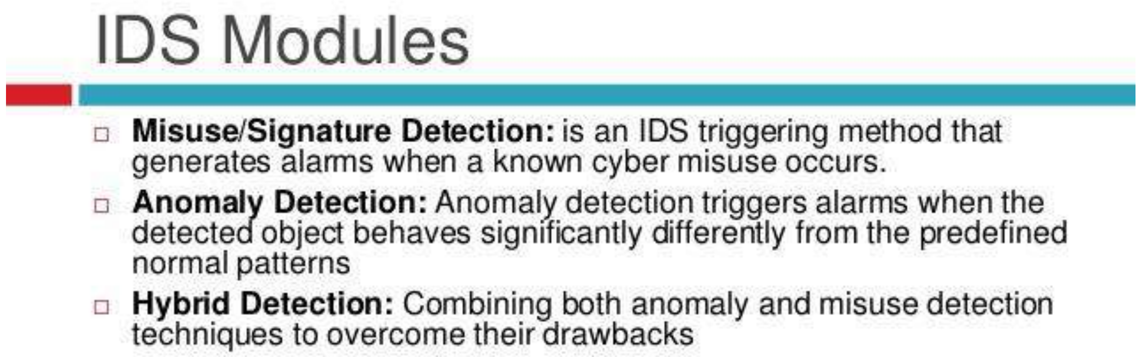
Anomaly detection

- Involves the collection of data relating to the behavior of legitimate users over a period of time
- Current observed behavior is analyzed to determine whether this behavior is that of a legitimate user or that of an intruder

Signature/Heuristic detection

- Uses a set of known malicious data patterns or attack rules that are compared with current behavior
- Also known as misuse detection
- Can only identify known attacks for which it has patterns or rules

Anomaly detection can find previously unknown methods of intrusion (so-called **zero-day attacks**). Signature-based detection, in contrast, will identify only known attacks that can be codified in a recognizable pattern.



IDS Modules

□ **Misuse/Signature Detection:** is an IDS triggering method that generates alarms when a known cyber misuse occurs.

□ **Anomaly Detection:** Anomaly detection triggers alarms when the detected object behaves significantly differently from the predefined normal patterns

□ **Hybrid Detection:** Combining both anomaly and misuse detection techniques to overcome their drawbacks

Digital signature

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document. The digital equivalent of a handwritten signature or stamped seal, a digital signature offers far more inherent security, and it is intended to solve the problem of tampering and impersonation in digital communications.

Digital signatures can provide the added assurances of evidence of origin, identity and status of an electronic document, transaction or message and can acknowledge informed consent by the signer.

How digital signatures work

Digital signatures are based on public key cryptography, also known as asymmetric cryptography. Using a public key algorithm, such as RSA, one can generate two keys that are mathematically linked: one private and one public.

Digital signatures work because public key cryptography depends on two mutually authenticating cryptographic keys. The individual who is creating the digital signature uses their own private key to encrypt signature-related data; the only way to decrypt that data is with the signer's public key. This is how digital signatures are authenticated.

How to create a digital signature

To create a digital signature, signing software -- such as an email program -- creates a one-way hash of the electronic data to be signed. The private key is then used to encrypt the hash. The encrypted hash -- along with other information, such as the hashing algorithm -- is the digital signature.

Virus Protection

As we have seen, viruses can and do wreak havoc on systems. Protection from viruses thus is an important security concern. Antivirus programs are often used to provide this protection. Some of these programs are effective against only particular known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus. When they find a known pattern, they remove the instructions, **disinfecting** the program. Antivirus programs may have catalogs of thousands of viruses for which they search.

Both viruses and antivirus software continue to become more sophisticated. Some viruses modify themselves as they infect other software to avoid the basic pattern-match approach of antivirus programs. Antivirus programs in turn now look for families of patterns rather than a single pattern to identify a virus. In fact, some antivirus programs implement a variety of detection algorithms. They can decompress compressed viruses before checking for a signature. Some also look for process anomalies. A process opening an executable file for writing is suspicious, for example, unless it is a compiler. Another popular technique is to run a program in a **sandbox**, which is a controlled or emulated section of the system. The antivirus software analyzes the behavior of the code in the sandbox before letting it run unmonitored. Some antivirus programs also put up a complete shield rather than just scanning files within a file system. They search boot sectors, memory, inbound and outbound e-mail, files as they are downloaded, files on removable devices or media, and so on. The best protection against computer viruses is prevention, or the practice of **safe computing**.

Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or disk exchange offer the safest route to preventing infection. For macro viruses, one defense is to exchange Microsoft Word documents in an alternative file format called **rich text format** **(RTF)**. Unlike the native Word format, RTF does not include the capability to attach macros.

Another defense is to avoid opening any e-mail attachments from unknown users. Another safeguard, although it does not prevent infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack.

Firewalling to Protect Systems and Networks

A **firewall** is a computer, appliance, or router that sits between the trusted and the untrusted. A network firewall limits network access between the two **security domains** and monitors and logs all connections. It can also limit connections based on source or destination address, source or destination port, or direction of the connection. For instance, web servers use HTTP to communicate with web browsers. A firewall therefore may allow only HTTP to pass from all hosts outside the firewall to the web server within the firewall.

In fact, a network firewall can separate a network into multiple domains. A common implementation has the Internet as the untrusted domain; a semi trusted and semi secure network, called the **demilitarized zone** **(DMZ)**, as another domain; and a company's computers as a third domain (Figure 15.10). Connections are allowed from the Internet to the DMZ computers and from the company computers to the Internet but are not allowed from the Internet or DMZ computers to the company computers. Optionally, controlled communications maybe allowed between the DMZ and one company computer or more. For instance, a web server on the DMZ may need to query a database server on the corporate network. With a firewall, however, access is contained, and any DMZ systems that are broken into still are unable to access the company computers.

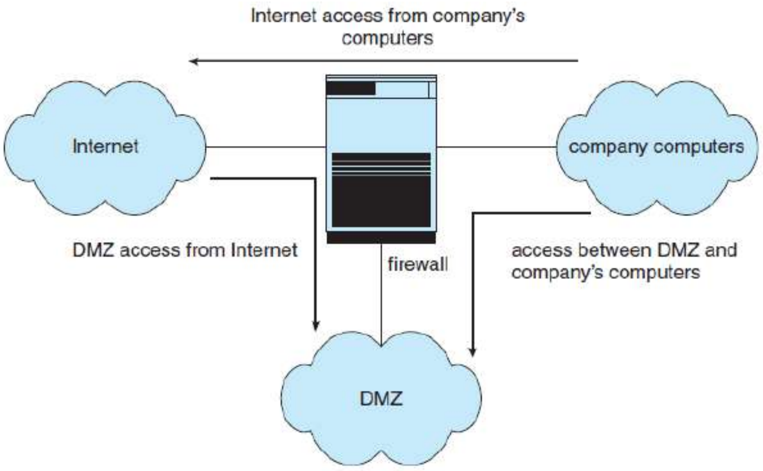

Internet access from company's computers
Internet
company computers
DMZ access from Internet
firewall
access between DMZ and company's computers
DMZ

Figure 15.10 Domain separation via firewall.

Firewalls do not prevent attacks that **tunnel**, or travel within protocols or connections that the firewall allows. A buffer-overflow attack to a web server will not be stopped by the firewall, for example, because the HTTP connection is allowed; Likewise, denial of- service attacks can affect firewalls as much as any other machines. Another vulnerability of firewalls is **spoofing**, in which an unauthorized host pretends to be an authorized host by meeting some authorization criterion. In addition to the most common network firewalls, there are other, newer kinds of firewalls, each with its pros and cons. A **personal firewall** is a software layer either included with the operating system or added as an application. Rather than limiting communication between security domains, it limits communication to (and possibly from) a given host.

An **application proxy firewall** understands the protocols that applications speak across the network. For example, SMTP is used for mail transfer. An application proxy accepts a connection just as an SMTP server would and then initiates a connection to the original destination SMTP server. It can monitor the traffic as it forwards the message, watching for and disabling illegal commands, attempts to exploit bugs, and so on. Some firewalls are designed for one specific protocol.

An **XML firewall**, for example, has the specific purpose of analyzing XML traffic and blocking disallowed or malformed XML.

System-call firewalls sit between applications and the kernel, monitoring system-call execution. For example, in Solaris 10, the "least privilege" feature implements a list of more than fifty system calls that processes may or may not be allowed to make.

UNIT-5

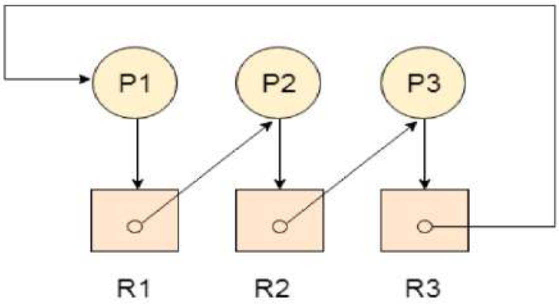
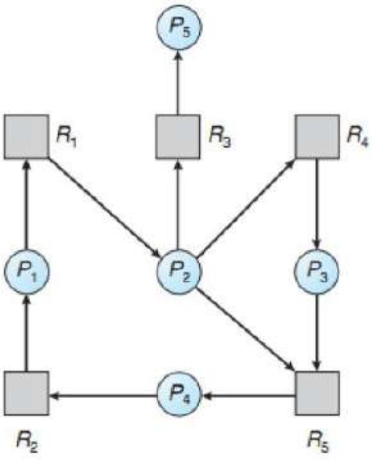
QNO	QUESTION	COURSE OUTCOME	BLOOM'S LEVEL
1	What is a deadlock? Identify the necessary conditions which create a deadlock.	**III**	**III**
2	Apply the resource allocation graph to identify the deadlock situation in the operating system with an example	**III**	**III**
3	How do you represent the following resource allocation graph with allocation matrix and request matrix 	**III**	**IV**
4	Explain how a deadlock can be recovered in the operating system	**III**	**II**
5	Draw the corresponding wait-for-graph for the following resource allocation graph  P_5 R_1 R_3 R_4 P_1 P_2 P_3 P_4 R_2 R_5	**III**	**V**
6	Compare logical address and physical address. Illustrate importance of the memory management unit with	**III**	**IV**

	diagram.		
7	"Swapping is also known as a technique for memory compaction" justify your answer.	**III**	**IV**
8	Classify the types of memory techniques to select a free partition from the set of available blocks	**III**	**IV**
9	What is fragmentation? Compare fragmentation techniques used in the operating system	**III**	**IV**
10	What is segmentation? Explain segmentation hardware with its neat diagram	**III**	**II**
11	How paging is implemented in the operating system? List the advantages and disadvantages of paging	**III**	**II**

12) Discuss the strengths and weaknesses of implementing an access matrix using capabilities that are associated with domains
13) List the various goals of protection and principles of protection
14) How a system can be protected with domain structure? Explain with MULTICS ring structure
15) Illustrate the process of access matrix implementation with an example