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

OS Definition

- OS is a resource allocator
    - o Manages all resources
    - o Decides between conflicting requests for efficient and fair resource use
- OS is a control program
    - o Controls execution of programs to prevent errors and improper use of the computer

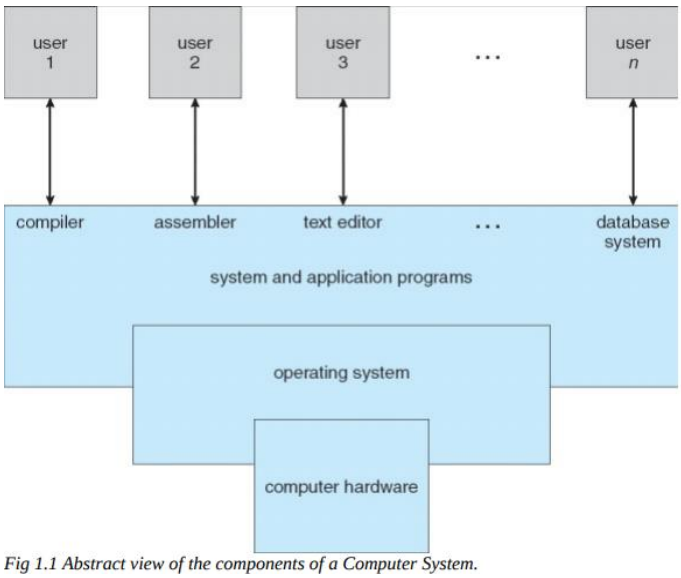


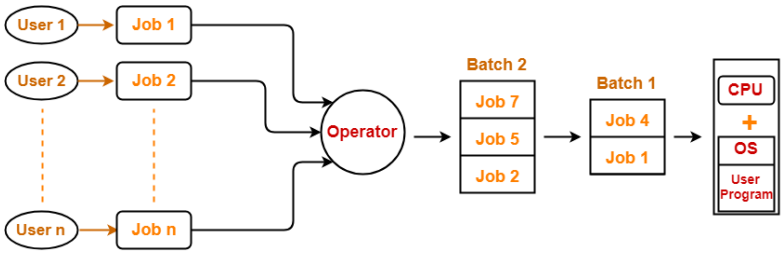
Fig 1.1 Abstract view of the components of a Computer System.

Types of Operating System

1. Mainframe System:

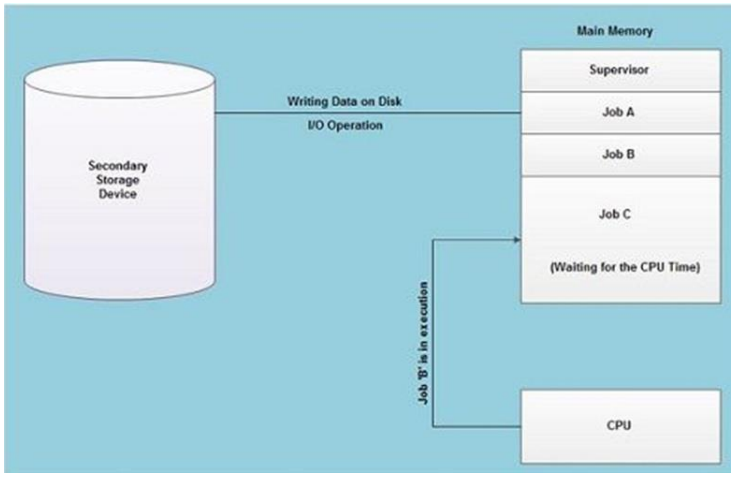
It is the system where the first computer used to handle many commercial scientific applications. The growth of mainframe systems traced from simple batch system where the computer runs one and only one application to time shared systems which allowed for user interaction with the computer system

a. **Batch /Early System:** Early computers were physically large machine. The common input devices were card readers, tape drivers. The common output devices were line printers, tape drivers and card punches. In these systems the user did not interact directly with the computer system. Instead the user preparing a job which consists of programming data and some control information and then submitted it to the computer operator after some time the output is appeared. The output in these early computer was fairly simple is main task was to transfer control automatically from one job to next. The operating system always resides in the memory. To speed up processing operators batched the jobs with similar needs and ran then together as a group. The disadvantages of batch system are that in this execution environment the CPU is often idle because the speed up of I/O devices is much slower than the CPU.



Batch Operating System

b. .**Multiprogrammed System:** Multiprogramming concept increases CPU utilization by organization jobs so that the CPU always has one job to execute the idea behind multiprogramming concept. The operating system keeps several jobs in memory simultaneously as shown in below figure.



This set of job is subset of the jobs kept in the job pool. The operating system picks and beginning to execute one of the jobs in the memory. In this environment the operating system simply switches and executes another job. When a job needs to wait the CPU is simply switched to another job and so on. The multiprogramming operating system is sophisticated because the operating system makes decisions for the user. This is known as scheduling. If several jobs are ready to run at the same time the system choose one among them. This is known as CPU scheduling. The disadvantages of the multiprogrammed system are

• It does not provide user interaction with the computer system during the program execution.

• The introduction of disk technology solved these problems rather than reading the cards from card reader into disk. This form of processing is known as spooling.

SPOOL stands for simultaneous peripheral operations online. It uses the disk as a huge buffer for reading from input devices and for storing output data until the output devices accept them. It is also use for processing data at remote sides. The remote processing is done and its own speed with no CPU intervention. Spooling overlaps the input, output one job with computation of other jobs. Spooling has a beneficial effect on the performance of the systems by keeping both CPU and I/O devices working at much higher time.

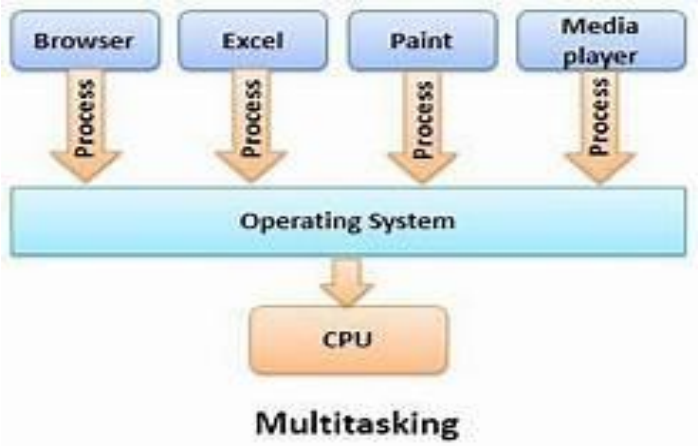
c. .**Multitasking Operating System/Time sharing operating System:** The time sharing system is also known as multi user systems. The CPU executes multiple jobs by switching among them but the switches occurs so frequently that the user can interact with each program while it is running. An interactive computer system provides direct communication between a user and system. The user gives instruction to the operating systems or to a program directly using keyboard or mouse and wait for immediate results. So the response time will be short. The time sharing system allows many users to share the computer simultaneously. Since each action in this system is short, only a little CPU time is needed for each user. The system switches rapidly from one user to the next so each user feels as if the entire computer system is dedicated to his use, even though it is being shared by many users. The disadvantages of time sharing system are:

• It is more complex than multiprogrammed operating system

• The system must have memory management & protection, since several jobs are kept in memory at the same time.

• Time sharing system must also provide a file system, so disk management is required.

• It provides mechanism for concurrent execution which requires complex CPU scheduling schemes.



Multitasking

d. **Real time Systems:** Real time system is used when there are rigid time requirements on the operation of a processor or flow of data. Sensors bring data to the computers. The computer analyzes data and adjusts controls to modify the sensors inputs. System that controls scientific experiments, medical imaging systems and some display systems are real time systems. The disadvantages of real time system are:

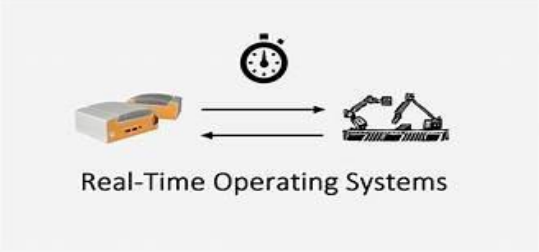
a. A real time system is considered to function correctly only if it returns the correct result within the time constraints.

b. Secondary storage is limited or missing instead data is usually stored in short term memory or ROM.

c. Advanced OS features are absent. Real time system is of two types such as:

- Hard real time systems: It guarantees that the critical task has been completed on time. The sudden task is takes place at a sudden instant of time.
- Soft real time systems: It is a less restrictive type of real time system where a critical task gets priority over other tasks and retains that priority until it computes.

These have more limited utility than hard real time systems. Missing an occasional deadline is acceptable e.g. QNX, VX works. Digital audio or multimedia is included in this category. It is a special purpose OS in which there are rigid time requirements on the operation of a processor. A real time OS has well defined fixed time constraints. Processing must be done within the time constraint or the system will fail. A real time system is said to function correctly only if it returns the correct result within the time constraint. These systems are characterized by having time as a key parameter.

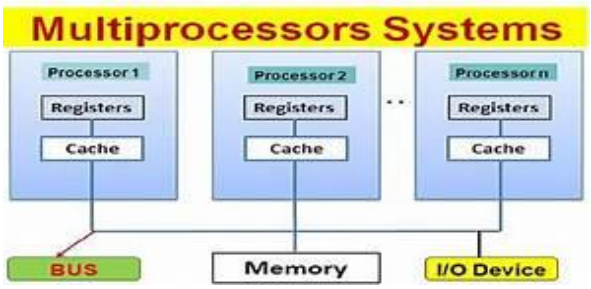


e. **Microprocessor Systems:** These Systems have more than one processor in close communications which share the computer bus, clock, memory & peripheral devices. Ex: UNIX, LINUX. Multiprocessor Systems have 3 main advantages.

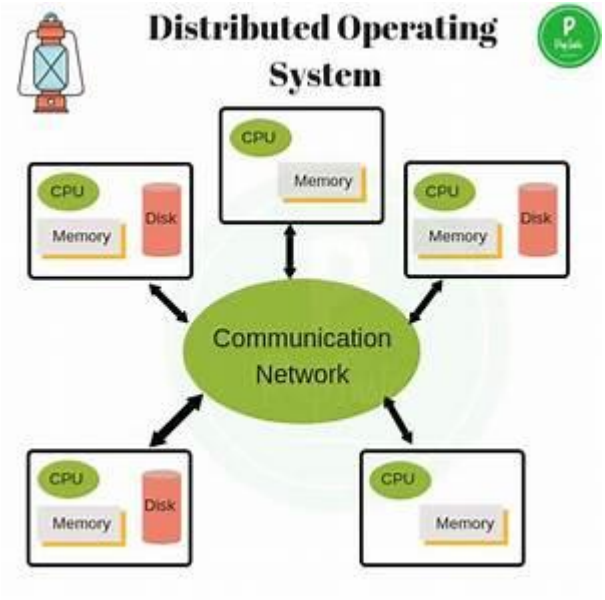
a. Increased throughput: No. of processes computed per unit time. By increasing the no. of processors move work can be done in less time. The speed up ratio with N processors is not N, but it is less than N. Because a certain amount of overhead is incurred in keeping all the parts working correctly.

b. Increased Reliability: If functions can be properly distributed among several processors, then the failure of one processor will not halt the system, but slow it down. This ability to continue to operate in spite of failure makes the system fault tolerant.

c. Economic scale: Multiprocessor systems can save money as they can share peripherals, storage & power supplies.



f. **Distributed System/Loosely Coupled Systems:** In contrast to tightly coupled systems, the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with each other by various communication lines such as high speed buses or telephone lines. Distributed systems depend on networking for their functionalities. By being able to communicate distributed systems are able to share computational tasks and provide a rich set of features to the users. Networks vary by the protocols used, the distances between the nodes and transport media. TCP/IP is the most common network protocol. The processor is a distributed system varies in size and function. It may microprocessors, work stations, minicomputer, and large general purpose computers. Network types are based on the distance between the nodes such as LAN (within a room, floor or building) and WAN (between buildings, cities or countries). The advantages of distributed system are resource sharing, computation speed up, reliability, communication.



Basic Functions of Operation System: The various functions of operating system are as follows:

1. **Process Management:**

• A program does nothing unless their instructions are executed by a CPU.A process is a program in execution. A time shared user program such as a complier 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 is also a process. A process needs certain resources including CPU time, memory files & 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. The OS is responsible for the following activities of process management.

• Creating & deleting both user & system processes.

• Suspending & resuming processes.

• Providing mechanism for process synchronization.

• Providing mechanism for process communication.

• Providing mechanism for deadlock handling.

2. **Main Memory Management:** The main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes ranging in size from hundreds of thousand to billions. Main memory stores the quickly accessible data shared by the CPU & I/O device. The central processor reads instruction from main memory during instruction fetch cycle & it both reads &writes data from main memory during the data fetch cycle. The main memory is generally the only large storage device that the CPU is able to address & access directly. For example, for the CPU to process data from disk. Those data must first be transferred to main memory by CPU generated E/O calls. Instruction must be in memory for the CPU to execute them. The OS is responsible for the following activities in connection with memory management.

• Keeping track of which parts of memory are currently being used & by whom.

• Deciding which processes are to be loaded into memory when memory space becomes available.

• Allocating &deallocating memory space as needed.

3. **File Management:** File management is one of the most important components of an OS computer can store information on several different types of physical media magnetic tape, magnetic disk & optical disk are the most common media. Each medium is controlled by a device such as disk drive or tape drive those has unique characteristics. These characteristics include access speed, capacity, data transfer rate & access method (sequential or random).For convenient use of computer system the OS provides a uniform logical view of information storage. The OS abstracts from the physical properties of its storage devices to define a logical storage unit the file. A file is collection of related information defined by its creator. The OS is responsible for the following activities of file management.

• Creating & deleting files.

• Creating & deleting directories.

• Supporting primitives for manipulating files & directories.

• Mapping files into secondary storage.

• Backing up files on non-volatile media.

4. **I/O System Management:** One of the purposes of an OS is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX the peculiarities of I/O devices are hidden from the bulk of the OS itself by the I/O subsystem. The I/O subsystem consists of:

• A memory management component that includes buffering, catching & spooling.

• A general device- driver interfaces drivers for specific hardware devices. Only the device driver knows the peculiarities of the specific device to which it is assigned.

5. Secondary Storage Management: The main purpose of computer system is to execute programs. These programs with the data they access must be in main memory during execution. As the main memory is too small to accommodate all data & programs & because the data that it holds are lost when power is lost. The computer system must provide secondary storage to back-up main memory. Most modern computer systems are disks as the storage medium to store data & program. The operating system is responsible for the following activities of disk management.

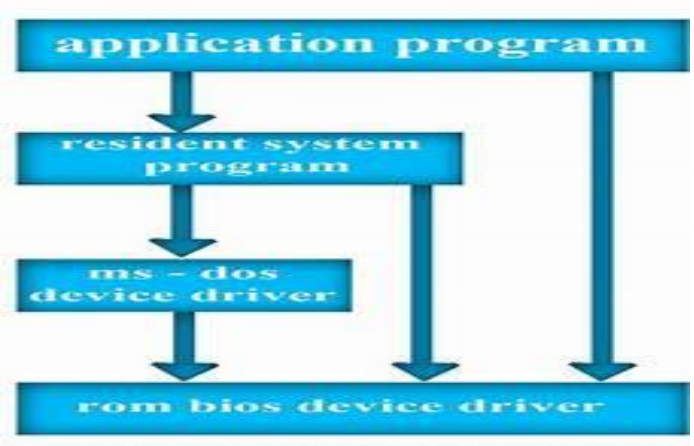
- Free space management.
- Storage allocation.
- Disk scheduling Because secondary storage is used frequently it must be used efficiently.

6. Networking: A distributed system is a collection of processors that don't share memory peripheral devices or a clock. Each processor has its own local memory & clock and the processor communicate with one another through various communication lines such as high speed buses or networks. The processors in the system are connected through communication networks which are configured in a number of different ways. The communication network design must consider message routing & connection strategies are the problems of connection & security.

7. Protection or security: If a computer system has multi users & allow the concurrent execution of multiple processes then the various processes must be protected from one another's activities. For that purpose, mechanisms ensure that files, memory segments, CPU & other resources can be operated on by only those processes that have gained proper authorization from the OS. Command interpretation: One of the most important functions of the OS is connected interpretation where it acts as the interface between the user & the OS.

System structure:

1. **Simple structure:** There are several commercial system that don't have a well- defined structure such operating systems begins as small, simple & limited systems and then grow beyond their original scope. MS-DOS is an example of such system. It was not divided into modules carefully. Another example of limited structuring is the UNIX operating system.



ms dos system structure

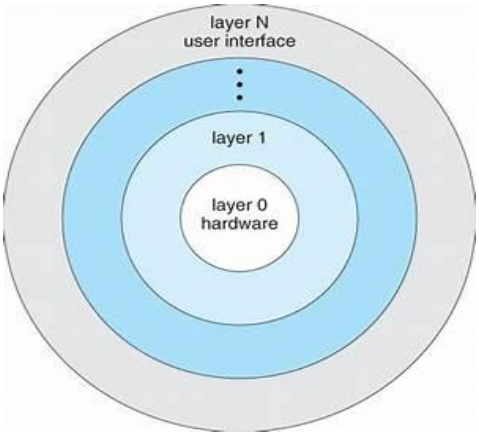
2. Layered approach: In the layered approach, the OS is broken into a number of layers (levels) each built on top of lower layers. The bottom layer (layer o) is the hardware & top most layer (layer N) is the user interface. The main advantage of the layered approach is modularity.

• The layers are selected such that each users functions (or operations) & services of only lower layer.

• This approach simplifies debugging & system verification, i.e. the first layer can be debugged without concerning the rest of the system. Once the first layer is debugged, its correct functioning is assumed while the 2nd layer is debugged & so on.

• If an error is found during the debugging of a particular layer, the error must be on that layer because the layers below it are already debugged. Thus the design & implementation of the system are simplified when the system is broken down into layers.

- Each layer is implemented using only operations provided by lower layers. A layer doesn't need to know how these operations are implemented; it only needs to know what these operations do.
- The layer approach was first used in the operating system. It was defined in six layers.

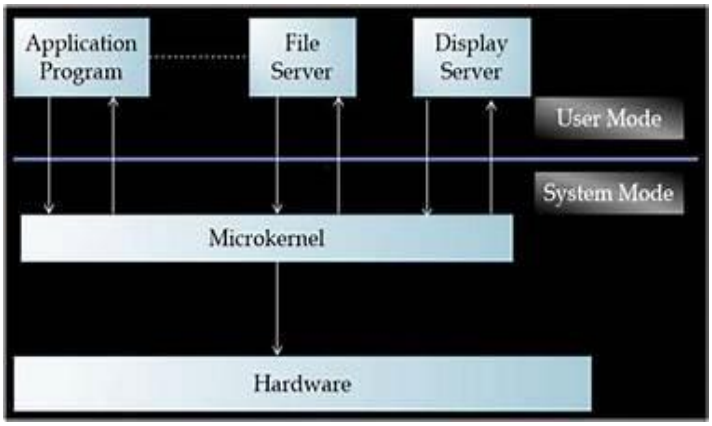


The main disadvantage of the layered approach is:

• The main difficulty with this approach involves the careful definition of the layers, because a layer can use only those layers below it. For example, the device driver for the disk space used by virtual memory algorithm must be at a level lower than that of the memory management routines, because memory management requires the ability to use the disk space.

• It is less efficient than a non layered system (Each layer adds overhead to the system call & the net result is a system call that take longer time than on a non layered system).

Micro kernels- The kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. microkernels provide minimal process and memory management, in addition to a communication facility. The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user rather than kernel processes. If a service fails, the rest of the operating system remains untouched.



Operating-System Operations

1. 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

2. A trap (or an exception) is a software-generated interrupt caused either by an error or by a specific request from a user program that an operating-system service is performed.

3. The interrupt-driven nature of an operating system defines that system's general structure. 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 that is responsible for dealing with the interrupt.

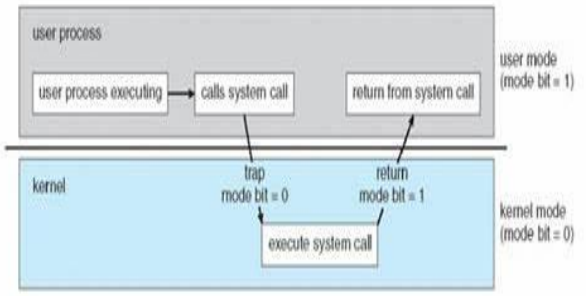
4. The operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program that was running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes.

5. Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect.

Dual-Mode Operation

Dual-mode operation allows OS to protect itself and other system components User mode and kernel mode Mode bit provided by hardware Provides ability to distinguish when system is running user code or kernel code Some instructions designated as privileged, only executable in kernel mode System call changes mode to kernel, return from call resets it to user Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call usenThree most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

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 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. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

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 (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls, such as calls for file move and copy.

Device Management- A 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, tapes), while others can be thought of as abstract or virtual devices (for example, files). If there are multiple users of the system, the system may require us to first request the device, to ensure exclusive use of it. After we are finished with the device, we release it. These functions are similar to the open and close system calls for 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 t I m e 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. In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes) .

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. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. 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 host id and get processid system calls do this translation. The identifiers are then passed to the general purpose 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.

Operating System Services-

An operating system provides an environment for the execution of the program. It provides some services to the programs. The various services provided by an operating system are as follows:

• **Program Execution:** The system must be able to load a program into memory and to run that program. The program must be able to terminate this execution either normally or abnormally.

• **I/O Operation:** A running program may require I/O. This I/O may involve a file or a I/O device for specific device. Some special function can be desired. Therefore the operating system must provide a means to do I/O.

• **File System Manipulation:** The programs need to create and delete files by name and read and write files. Therefore the operating system must maintain each and every files correctly.

• **Communication:** The communication is implemented via shared memory or by the technique of message passing in which packets of information are moved between the processes by the operating system.

• **Error detection:** The operating system should take the appropriate actions for the occurrences of any type like arithmetic overflow, access to the illegal memory location and too large user CPU time.

• **Resource Allocation:** When multiple users are logged on to the system the resources must be allocated to each of them. For current distribution of the resource among the various processes the operating system uses the CPU scheduling run times which determine which process will be allocated with the resource.

• **Accounting:** The operating system keep track of which users use how many and which kind of computer resources.

• **Protection:** The operating system is responsible for both hardware as well as software protection. The operating system protects the information stored in a multiuser computer system.

Difference between process & program:

A program by itself is not a process. A program in execution is known as a process. A program is a passive entity, such as the contents of a file stored on disk where as process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources may be shared among several process with some scheduling algorithm being used to determinate when the stop work on one process and service a different one.

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
…
    body of F;
…
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

The operation x.signal can be implemented as:

```
if (x-count > 0) {
next_count++;
signal(x_sem);
wait(next);
next_count--;
}
```

Sleeping Barber Problem-

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in [Figure 2-35]. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls. Our solution uses three semaphores, customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), barbers, the number of barbers (0 or 1) who are idle, waiting for customers, and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers. The reason for having waiting is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves. Our solution is shown [below]. When the barber shows up for work in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. The barber then goes to sleep. He stays asleep until the first customer shows up. When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released mutex. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases mutex and leaves without a haircut. If there is an available chair, the customer increments the integer variable, waiting. Then he does an Up on the semaphore customers, thus waking up the barber. At this point, the customer and the barber are both awake. When the customer releases mutex, the barber grabs it, does some housekeeping, and begins the haircut. When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, a haircut is given. If not, the barber goes to sleep. As an aside, it is worth pointing out that although the readers and writers and sleeping barber problems do not involve data transfer, they still belong to the area of IPC because they involve synchronization between multiple processes.

```
#define CHAIRS 5 /*# chairs for waiting customers    */
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customer are waiting (not being cut) */
void barber(void)
{
 while (TRUE) {
 down(&customers); /* go to sleep if # of customers is 0 */
 down(&mutex);  /* acquire access to "waiting' */
waiting = waiting - 1; /* decrement count of waiting customers */
up(&barbers); /* one barber is now ready to cut hair */
 up(&mutex);  /* release 'waiting'  */
 cut_hair(); /* cut hair (outside critical region */
```

```
        }
 }
void customer(void)
{
 down(&mutex);  /* enter critical region */
if (waiting < CHAIRS){  /* if there are no free chairs, leave */
waiting = waiting + 1; /* increment count of waiting customers */
up(&customers); /* wake up barber if necessary */
 up(&mutex);  /* release access to 'waiting' */
down(&barbers); /* go to sleep if # of free barbers is 0 */
get_haircut(); /* be seated and be served */
} else {
 up(&mutex);  /* shop is full; do not wait */
}
 }
```

Peterson's Solution

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

```
do{
      flag[0]=true;
      turn=1;
       while(flag[1]&& turn==1);
     Critical Section
     flag[0]=false;
     remainder section
}while(1);
```

```
P0                                  P1
  do                                do
     {                                 {
       flag[0]=true;                     flag[1]=true;
        turn=1;                          turn=0;
     while(flag[1]&& turn==1);         while(flag[0]&& turn==0);
          Critical Section                  Critical Section
        flag[0]=false;                    flag[1]=false;
         remainder section                 remainder section
      }while(1);                        }while(1);
```

Synchronization Hardware

1.Many systems provide hardware support for critical section code
2. Uniprocessors – could disable interrupts
3. Currently running code would execute without preemption
4. Generally too inefficient on multiprocessor systems
    a. Operating systems using this not broadly scalable
5. Modern machines provide special atomic hardware instructions
    a. Atomic = non-interruptable
6. Either test memory word and set value Or swap contents of two memory words

Solution to Critical Section Problem using Lock

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

TestAndndSet Instruction

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

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
while ( TestAndSet (&lock ))
    ; // do nothing
     // critical section
lock = FALSE;
    // remainder section
} while (TRUE);
```

Sawp Instruction

```
  void Swap (boolean *a, boolean *b)
{
```

```
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Solution using Swap

 Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

 Solution:

```
do {
      key = TRUE;
      while ( key == TRUE)
      Swap (&lock, &key );
      // critical section
   lock = FALSE;
      // remainder section
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

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

Critical section problem:- A section of code which reads or writes shared data. Race Condition

• The situation where two or more processes try to access and manipulate the same data and output of the process depends on the orderly execution of those processes is called as Race Condition.

• count++ could be implemented as register1 = count register1 = register1 + 1 count = register1

• count-- could be implemented as register2 = count register2 = register2 - 1 count = register2

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

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

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

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

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

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

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

Requirements for the Solution to Critical-Section Problem

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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting: - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

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

(1) Preemptive Kernel

(2) Non Preemptive Kernel – Preemptive Kernel allows a process to be preempted while it is running in kernel mode.

– Non Preemptive Kernel does not allow a process running in kernel mode to be preempted. (these are free from race conditions)

2 Peterson's Solution

• It is restricted to two processes that alternates the execution between their critical and remainder sections.

• Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

• The two processes share two variables: – int turn; – Boolean flag[2]

• The variable turn indicates whose turn it is to enter the critical section.

• The flag array is used to indicate if a process is ready to enter the critical section.

flag[i] = true implies that process Pi is ready!

Note:- Peterson's Solution is a software based solution.

Algorithm for Process Pi

```
while (true) { flag[i] = TRUE; turn = j;
while ( flag[j] && turn == j);
CRITICAL SECTION
```

flag[i] = FALSE;
REMAINDER SECTION }

Solution to Critical Section Problem using Locks.

```
do{
        acquire lock
            Critical Section
        release lock
             }while (True);
```

Note:- Race Conditions are prevented by protecting the critical region by the locks.

2.2.3 Synchronization Hardware

• In general we can provide any solution to critical section problem by using a simple tool called as LOCK where we can prevent the race condition.

• Many systems provide hardware support (hardware instructions available on several systems) for critical section code.

• In UniProcessor hardware environment by disabling interrupts we can solve the critical section problem. So that Currently running code would execute without any preemption .

• But by disabling interrupts on multiprocessor systems is time taking so that it is inefficient compared to UniProcessor system.

• Now a days Modern machines provide special atomic hardware instructions that allow us to either test memory word and set value Or swap contents of two memory words automatically i.e. done through an uninterruptible unit.

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
- Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
while S <= 0;
S--;
}
```

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

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

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

Semaphore Implementation

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

- Could now have busy waiting in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied

 Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

 With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:

  - value (of type integer)
  - pointer to next record in the list

 Two operations:

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

 Implementation of wait:

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
}
}
```

 Implementation of signal:

```
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
```

Classical Problems of Synchronization

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

Bounded-Buffer Problem

The pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- The structure of the producer process

```
do {
// produce an item in nextp
wait (empty);
wait (mutex);
// add the item to the buffer
signal (mutex);
signal (full);
} while (TRUE);
```

 The structure of the consumer process

```
do {
wait (full);
wait (mutex);
// remove an item from buffer to nextc
signal (mutex);
signal (empty);
// consume the item in nextc
        } while (TRUE);
```

2 Process Solution-

The general structure for Process Pi-

```
do{
Entry Section
   Critical Section
Exit Section
   Remainder Section
}while(true);
```

Algo1-

```
do {
     while (turn == j);
           critical section
           turn = j;
      remainder section
} while (true);
```

Above algo satisfy mutual exclusion but it does not satisfy Progress.

Algo2-

```
Do
{
  flag[0]=true;
   while(flag(1));
    Critical Section
   Flag[0]=false;
    Remainder Section
}while(1)
```

Above algo also satisfy mutual exclusion but it does not satisfy progress.

Cooperating Process

- Independent process cannot affect or be affected by the execution of another
- Process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
    - o Information sharing
    - o Computation speed-up
    - o Modularity

Producer Consumer Problem

 Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

- *unbounded-buffer* places no practical limit on the size of the buffer
- *bounded-buffer* assumes that there is a fixed buffer size

```
while (true) {
/* Produce an item */
while (((in = (in + 1) % BUFFER SIZE count) == out)
; /* do nothing -- no free buffers */
buffer[in] = item;
in = (in + 1) % BUFFER SIZE;
}
```

Producer Process

```
while (true) {
while (in == out)
; // do nothing -- nothing to consume
// remove an item from the buffer
item = buffer[out];
out = (out + 1) % BUFFER SIZE;
```

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Initialization of semaphores–

Mutex = 1 Full =0 //Initially, all slots are empty. Thus full slot sare 0

Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –

```
do{

wait(full);
wait(mutex);

// remove item from buffer

signal(mutex);
signal(empty);

// consumes item

}while(true)
```

As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

Banker's Algorithm

Assumptions

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structure for Bankers' Algorithm

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

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type Rj available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process Pi may request at most k instances of resource type Rj
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then Pi is currently allocated k instances of Rj
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then Pi may need k more instances of Rj to complete its task

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

Safety Algorithm

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

Work = Available

Finish $[i]$ = *false* for $i = 0, 1, \ldots, n-1$

2. Find and i such that both:

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

(b) *Needi* □ *Work*

If no such i exists, go to step 4

3. *Work = Work + Allocationi*

Finish$[i]$ = *true*

go to step 2

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

Resource Request Algorithm

Request = request vector for process *Pi*. If *Requesti* $[j] = k$ then process *Pi* wants k instances of resource type *Rj*

1. If *Requesti* □ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If *Requesti* □ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available

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

Available = Available – Request;

Allocationi = Allocationi + Requesti;

Needi = Needi – Requesti;

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

Deadlock Detection

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

Recovery from Deadlock

A. Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated

B. Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Process Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

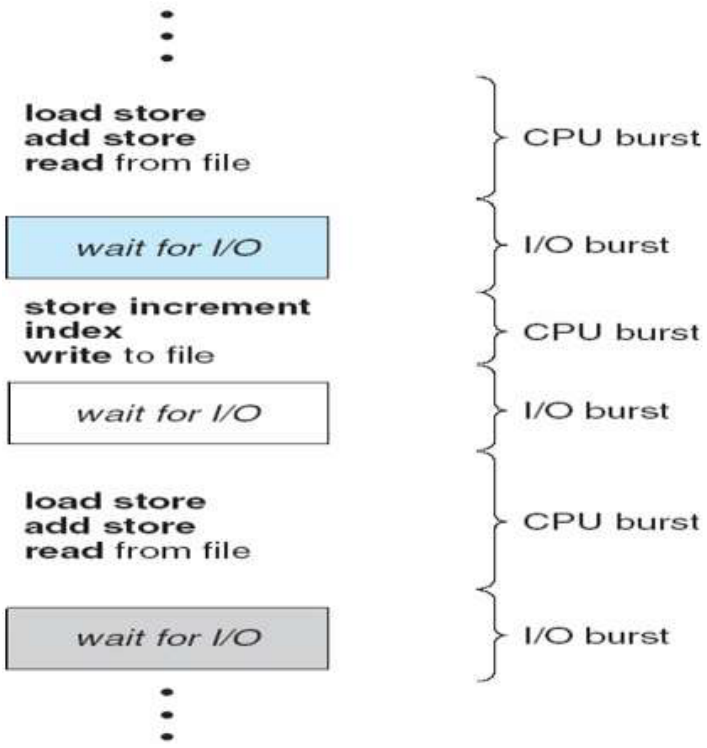


Fig: CPU burst and I/O burst

CPU Scheduler

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

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

- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

 Dispatcher module gives control of the CPU to the process selected by the shortterm scheduler; this involves:

o switching context

o switching to user mode

o jumping to the proper location in the user program to restart that program

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

CPU Scheduling Criteria

 Max CPU utilization

 Max throughput

 Min turnaround time

 Min waiting time

 Min response time

CPU Scheduling Algorithms

A. First Come First Serve Scheduling

 Schedule the task first which arrives first

 Non preemptive In nature

B. Shortest Job First Scheduling

 Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

 SJF is optimal – gives minimum average waiting time for a given set of processes

o The difficulty is knowing the length of the next CPU request

Priority Scheduling

 A priority number (integer) is associated with each process

 The CPU is allocated to the process with the highest priority (smallest integer highest priority)

o Preemptive

o nonpreemptive

 SJF is a priority scheduling where priority is the predicted next CPU burst time

 Problem **Starvation** – low priority processes may never execute

 Solution **Aging** – as time progresses increase the priority of the process

Round Robin Scheduling

 Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

 If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

Performance

o *q* large FIFO

o *q* small *q* must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue Scheduling

 Ready queue is partitioned into separate queues:

foreground (interactive)

background (batch)

 Each queue has its own scheduling algorithm

o foreground – RR

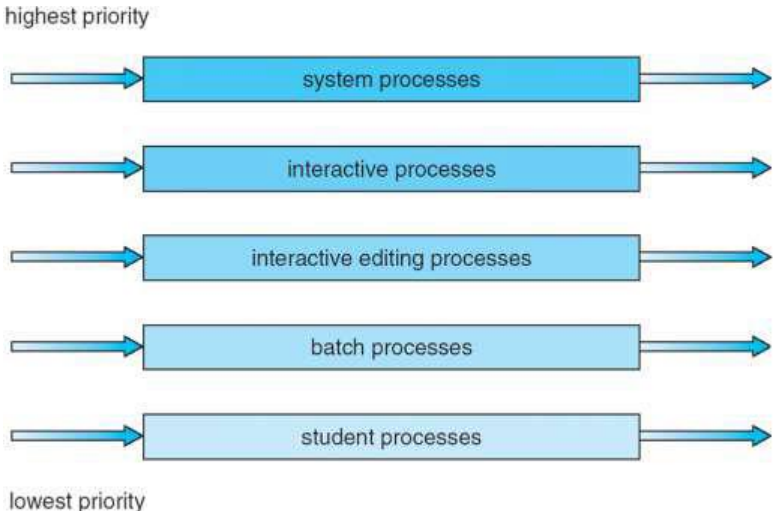
o background – FCFS

Scheduling must be done between the queues

o Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.

o Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR

o 20% to background in FCFS



Multilevel Feedback Queue Scheduling

 A process can move between the various queues; aging can be implemented this way

 Multilevel-feedback-queue scheduler defined by the following parameters:

o number of queues

o scheduling algorithms for each queue

o method used to determine when to upgrade a process

o method used to determine when to demote a process

o method used to determine which queue a process will enter when that process needs service

Scheduling Criteria-

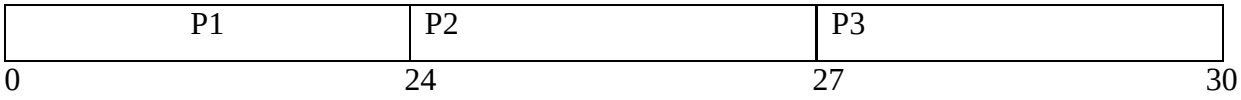
- CPU Utilization-keep the CPU as busy as possible
- Throughput-of process that complete their execution per unit time
- Turnaround Time- amount of time to execute a particular process
- Waiting Time- amount of time a process has been waiting in the ready queue.
- Response Time-amount of time it takes from when a request was submitted until the first response is produced, not output(for time-sharing environment)
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

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

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: P1,P2,P3

The grant chart for the schedule is:



Waiting time for P1=0;P2=24;P3=27

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

B. Shortest Job First Scheduling

 Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

 SJF is optimal – gives minimum average waiting time for a given set of processes

o The difficulty is knowing the length of the next CPU request

Priority Scheduling

 A priority number (integer) is associated with each process The CPU is allocated to the process with the highest priority (smallest integer= highest priority)

o Preemptive

o nonpreemptive

 SJF is a priority scheduling where priority is the predicted next CPU burst time

 Problem **Starvation** – low priority processes may never execute

 Solution **Aging** – as time progresses increase the priority of the process

Round Robin Scheduling

 Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

 If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

 Performance

o *q* large FIFO

o *q* small *q* must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue Scheduling

 Ready queue is partitioned into separate queues: foreground (interactive) background (batch)

 Each queue has its own scheduling algorithm

o foreground – RR

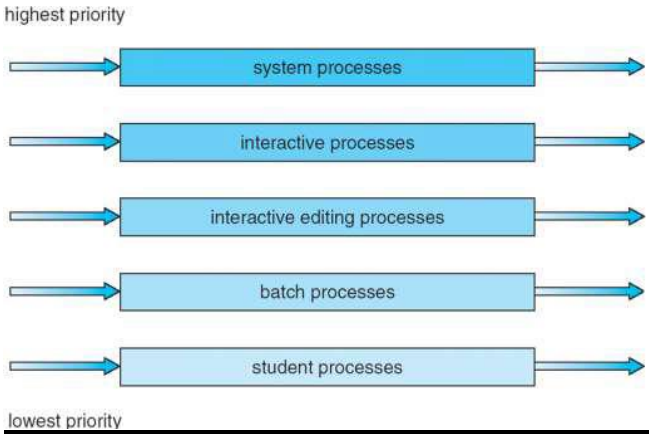
o background – FCFS

Scheduling must be done between the queues

o Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.

o Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR

o 20% to background in FCFS



Multilevel Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:

o number of queues

o scheduling algorithms for each queue

o method used to determine when to upgrade a process

o method used to determine when to demote a process

o method used to determine which queue a process will enter when that process needs service

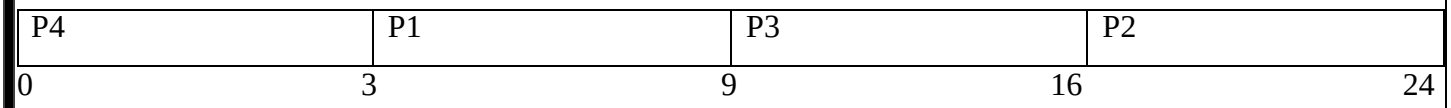
B. Shortest Job First Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Process	Arrival Time	Burst Time
P1	0	6
P2	2	8
P3	4	7
P4	5	3

SJF Scheduling Chart

Average waiting time=(3+16+9+0)/4 =7 the length of the next CPU request



Priority Scheduling-

- A priority number is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer o highest priority)
- Preemptive
- Nonpreemptive
- SJF is priority scheduling where priority is the predicted next CPU burst time.
- Problem –starvation low priority processes may never execute.
- Solution –aging –as time progress increase the priority of the process .

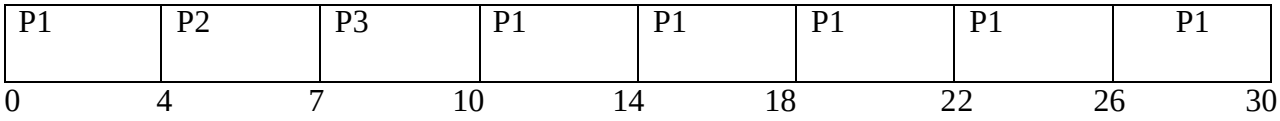
Round Robin-

- Each process gets a small unit of CPU time(time quantum) usually 10-100 miliseconds . After this time has elapsed , the process is pre-empted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process get 1/n of the CPU time in chunks of atmost q time units at once . No process waits more than(n-1)qtime units.
- Performance

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

Time quantum=4

The grant chart is



Typically higher average turnaround than SJF , but better response.

Deadlock In Operating System

Deadlock is the situation in which two processes are each waiting for the other in order to complete the present process before going to the other one. The result is that both the processes are hanging. The situation of Deadlock generally occurred in multitasking and client-server environment. A deadlock is also called a deadly embrace. This term is most commonly used in the Country Europe. Here in this post, lecture notes in computer science on Deadlock in **Operating System** including description of necessary conditions for Deadlock ,Deadlock handling, prevention and avoidance.

Deadlock in Operating System:

In an operating system, a deadlock is the occurs when a process enters into a waiting state because a resource request is being made by the other waiting process, which in turn become a waiting status for the other resource. When the process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system in a deadlock state.

So this was a general description about deadlock, technically we can define deadlock as a state where certain process rather we can say the inter related process wait for the resources which is used by the other process so the state where every process waits for the resources is called as the deadlock.

So there are certain conditions due to which in a system deadlock occur

1) **Mutual Exclusion:** In this condition, at least two resources are must be non-sharable and only one process can use the resource any given instant of time.

2) **Hold and Wait:** This is also known as Resource holding, a process in which at least one resource is in holding state and can request more resources which are being held by other processes.

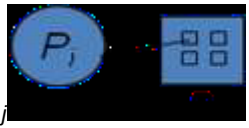
3) **No preemption:** In this condition the other process is not able to request the resources until the previous process is not complete. These resources must be released by the holding process voluntarily.

4) **Circular Wait:** This is the last but not the least condition in which one process waits for the resource held other process which is waiting for the nest process to release the resources. These are the four necessary conditions for Deadlock which is named as Coffman conditions. These conditions are enough to preclude a deadlock from occurring.

Resource Allocation Graph

- A set of vertices V and a set of edges E.
- V is partitioned into two types:

1. $P = \{P1, P2, \ldots, Pn\}$, the set consisting of all the processes in the system
2. $R = \{R1, R2, \ldots, Rm\}$, the set consisting of all resource types in the system

- request edge – directed edge $P1 \rightarrow Rj$
- assignment edge – directed edge $Rj \rightarrow Pi$
- Process 
- Resource Type with 4 instances 
- Pi requests instance of Rj 
- P_i is holding an instance of R_j 

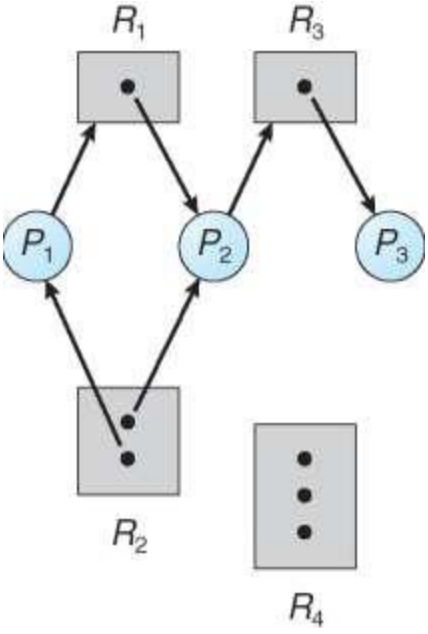


Fig: RAG

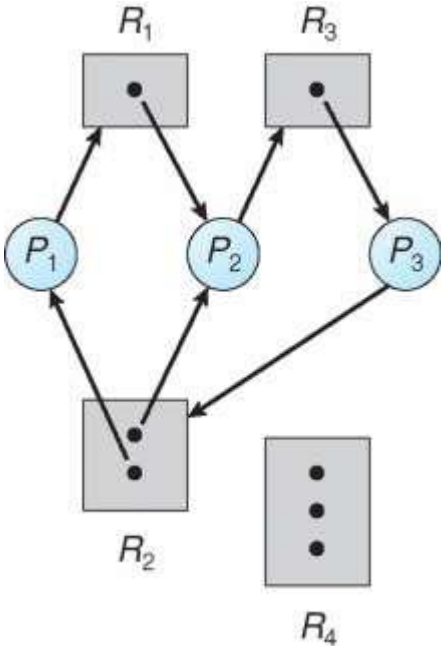


Fig: RAG with a deadlock

- If graph contains no cycles ⇨ no deadlock
- If graph contains a cycle ⇨

1) if only one instance per resource type, then deadlock
2) if several instances per resource type, possibility of deadlock

Methods for Handling Deadlock

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

Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

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

- **No Preemption** –

1) If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
2) Preempted resources are added to the list of resources for which the process is waiting
3) Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

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

Memory Management

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

Logical vs Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management

1. **Logical address** – generated by the CPU; also referred to as **virtual address**
2. **Physical address** – address seen by the memory unit

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

Address Binding

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

1. **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
2. **Load time**: Must generate **relocatable code** if memory location is not known at compile time
3. **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Memory Management Unit

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

Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded

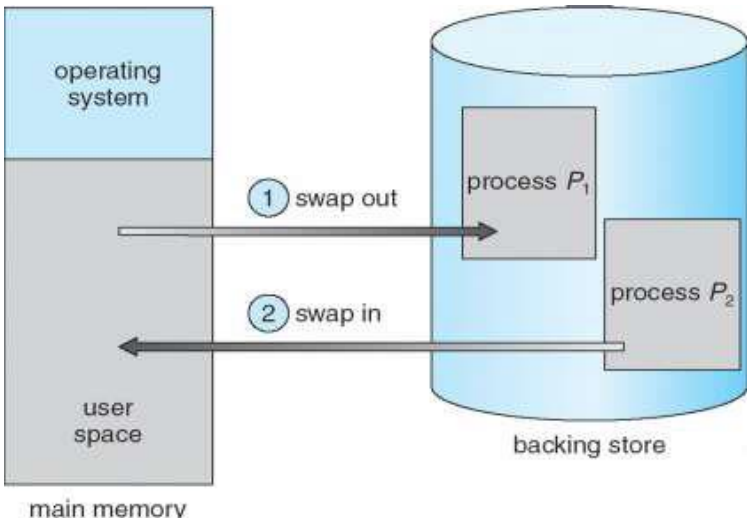
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk



Contiguous Allocation

- Main memory usually into two partitions:

1. Resident operating system, usually held in low memory with interrupt vector
2. User processes then held in high memory

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data

1. Base register contains value of smallest physical address
2. Limit register contains range of logical addresses – each logical address must be less than the limit register
3. MMU maps logical address *dynamically*

- Multiple-partition allocation

1. Hole – block of available memory; holes of various size are scattered throughout memory
2. When a process arrives, it is allocated memory from a hole large enough to accommodate it
3. Operating system maintains information about:

   a) allocated partitions b) free partitions (hole)

Dynamic Storage Allocation Problem

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  1. Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  2. Produces the largest leftover hole

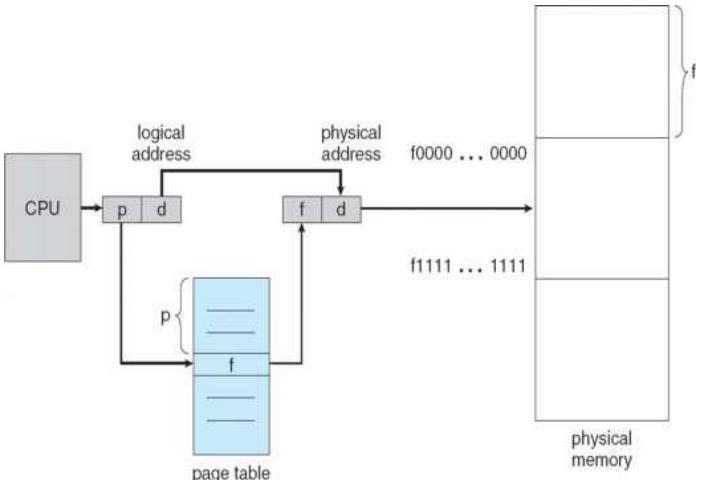
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  i) Shuffle memory contents to place all free memory together in one large block
  ii) Compaction is possible *only* if relocation is dynamic, and is done at execution time
  iii) I/O problem
- Latch job in memory while it is involved in I/O
  i) Do I/O only into OS buffers

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation
- Address generated by CPU is divided into:

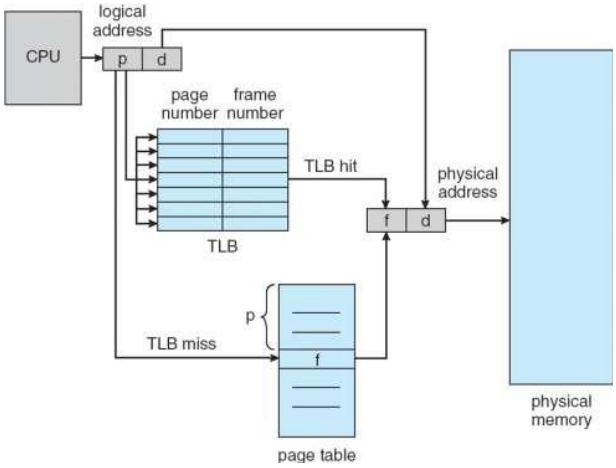
1) **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
2) **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit


logical address
physical address
CPU
p
d
f
d
f0000 ... 0000
f1111 ... 1111
p
f
page table
physical memory

Implementation of Page table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses.
  One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fastlookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Paging with TLB



Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:

1. "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
2. "invalid" indicates that the page is not in the process' logical address space

Shared Pages

- **Shared code**

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

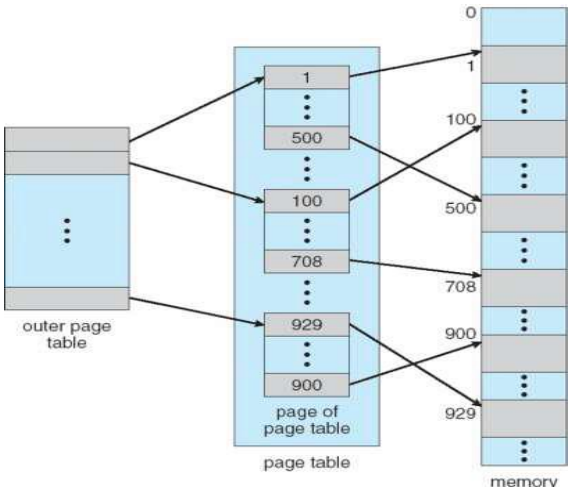
- **Private code and data**

1. Each process keeps a separate copy of the code and data
2. The pages for the private code and data can appear anywhere in the logical address space

Structure of Page table

Hierarchical Paging

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

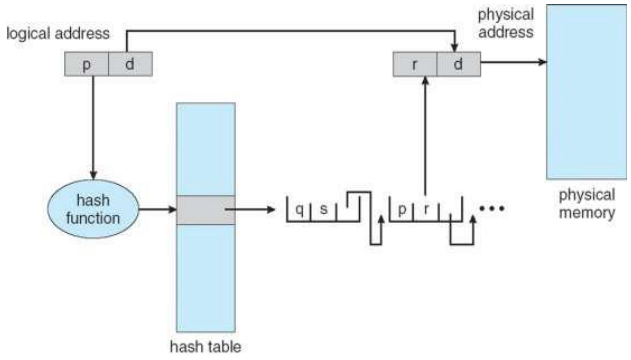


Hashed Page Tables

- The virtual page number is hashed into a page table

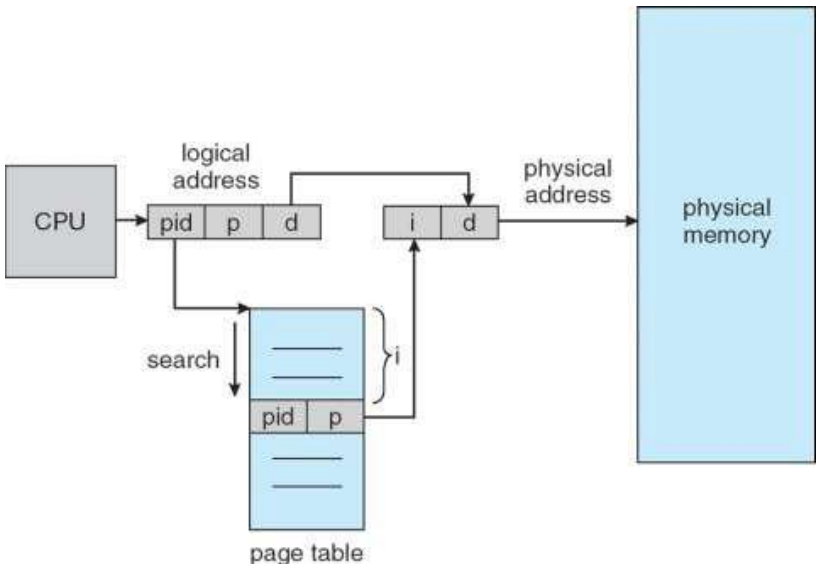
1. This page table contains a chain of elements hashing to the same location

- Virtual page numbers are compared in this chain searching for a match
  1. If a match is found, the corresponding physical frame is extracted



Inverted Page Tables

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries



Segmentation

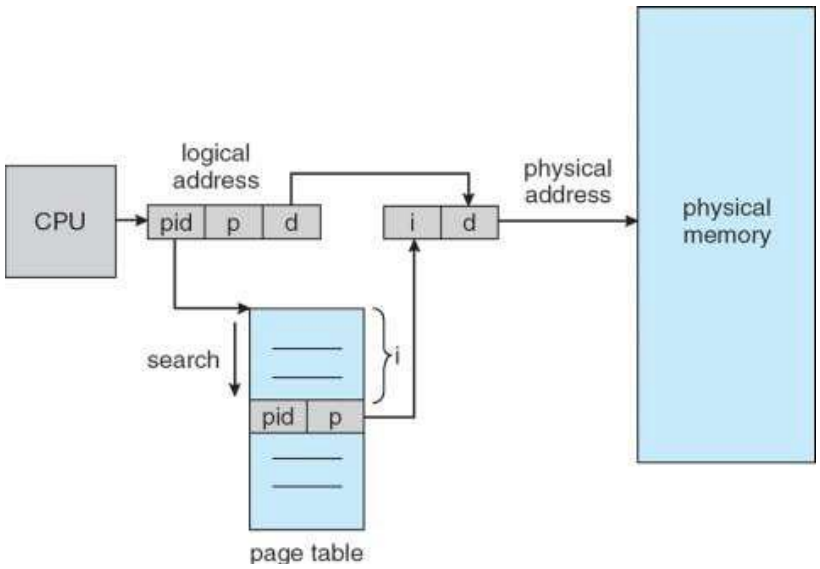
- Memory-management scheme that supports user view of memory
- A program is a collection of segments

1. A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays

- Logical address consists of a two tuple:
  1. <segment-number, offset>,
- Segment table – maps two-dimensional physical addresses; each table entry has:
  1. base – contains the starting physical address where the segments reside in memory
  2. limit – specifies the length of the segment

Inverted Page Tables

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries



Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments

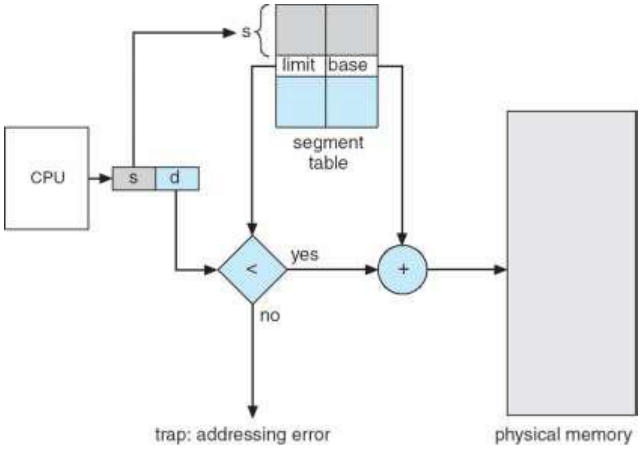
1. A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays

- Logical address consists of a two tuple:
  1. <segment-number, offset>,
- Segment table – maps two-dimensional physical addresses; each table entry has:
  1. base – contains the starting physical address where the segments reside in memory
  2. limit – specifies the length of the segment

- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;
  1. segment number *s* is legal if *s* < STLR
- Protection

o With each entry in segment table associate:
  2. validation bit = 0 □ illegal segment
  3. read/write/execute privileges

- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storageallocation problem
- A segmentation example is shown in the following diagram



Virtual Memory Management

- **Virtual memory** – separation of user logical memory from physical memory.

1. Only part of the program needs to be in memory for execution
2. Logical address space can therefore be much larger than physical address space
3. Allows address spaces to be shared by several processes
4. Allows for more efficient process creation

- Virtual memory can be implemented via:

1. Demand paging
2. Demand segmentation

Demand Paging

- Bring a page into memory only when it is needed

1. Less I/O needed
2. Less memory needed
3. Faster response
4. More users

- Page is needed reference to it

1. invalid reference ⇨ abort
2. not-in-memory ⇨bring to memory

- **Lazy swapper** – never swaps a page into memory unless page will be needed

1. Swapper that deals with pages is a **pager**

- With each page table entry a valid–invalid bit is associated

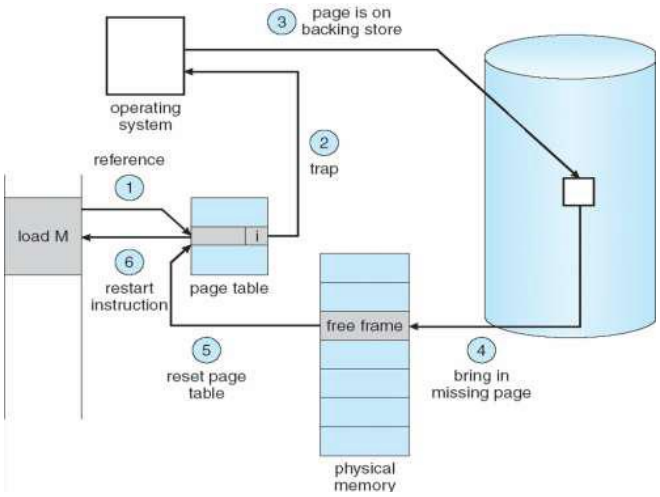
  (**v**⇨ in-memory, **i**⇨not-in-memory)

- Initially valid–invalid bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry is **I** ⇨page fault

Page Fault

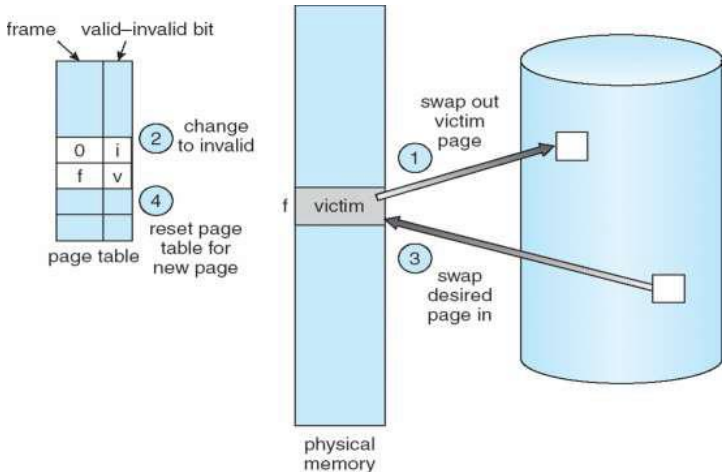
If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at another table to decide:
   - l Invalid reference □□abort
   - l Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault



Page Replacement

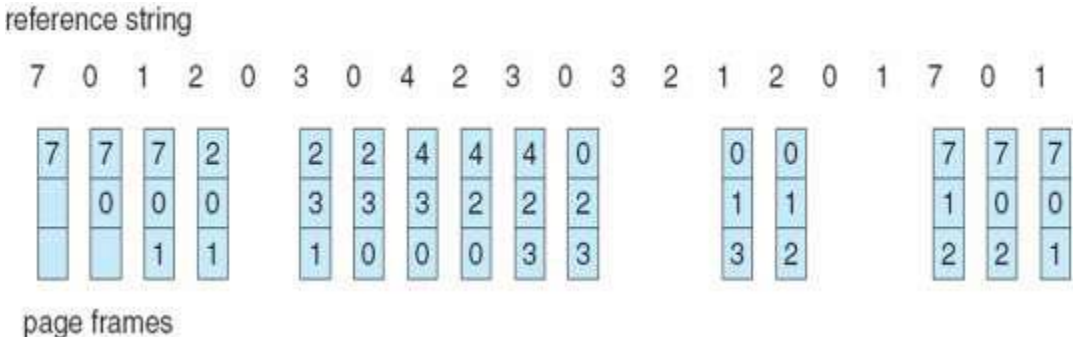
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- Find the location of the desired page on disk
- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame
- Bring the desired page into the (newly) free frame; update the page and frame tables



Page Replacement algorithm

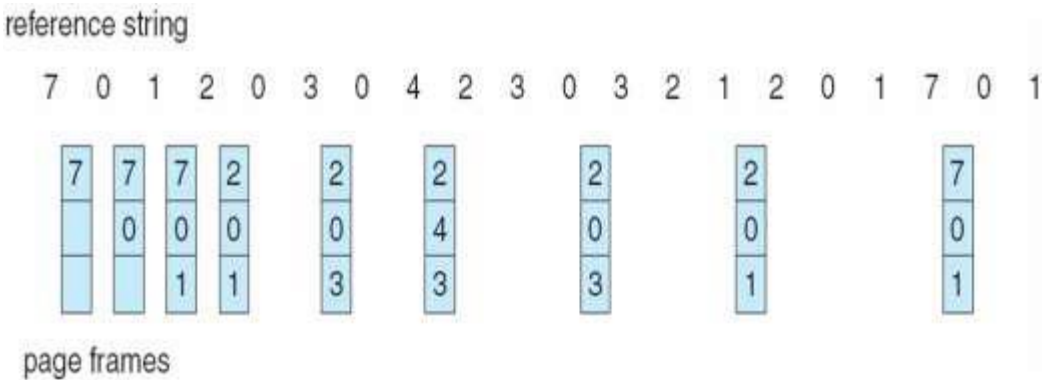
FIFO (First-in-First-Out)

- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- **Belady's Anomaly: more frames more page faults (** for some pagereplacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases.)
- **Ex-**



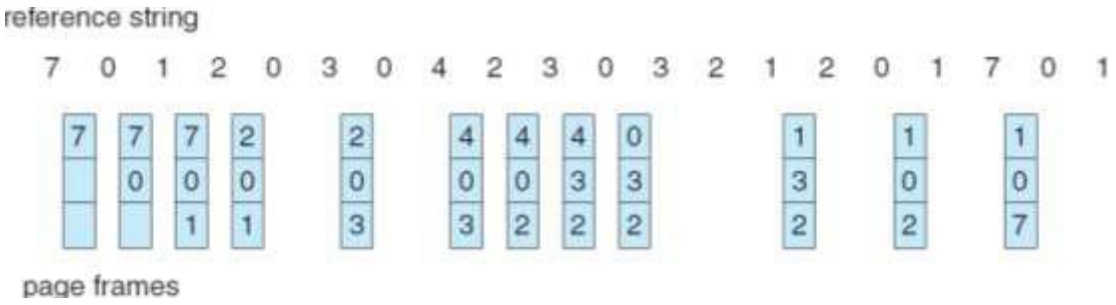
OPTIMAL PAGE REPLACEMENT

- Replace page that will not be used for longest period of time
- Ex-



LRU (LEAST RECENTLY USED)

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- Ex-



Allocation of Frames

- Each process needs *minimum* number of pages
- Two major allocation schemes

1. fixed allocation
2. priority allocation

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

MODULE-III

File System

File

- Contiguous logical address space
- Types:
    - Data
        - numeric
        - character
        - binary
    - Program

File Structure

- None - sequence of words, bytes
- Simple record structure
    - Lines
    - Fixed length
    - Variable length
- Complex Structures
    - Formatted document
    - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
    - Operating system
    - Program

File Attribute

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

File Types

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

File Operations

- **Create, Write, Read, Reposition within file, Delete, Truncate**
- *Open(F_i)* – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close (F_i)* – move the content of entry F_i in memory to directory structure on disk

File Access Methods

n **Sequential Access**

read next

write next

reset

no read after last write

(rewrite)

n **Direct Access**

read *n*

write *n*

position to *n*

read next

write next

rewrite *n*

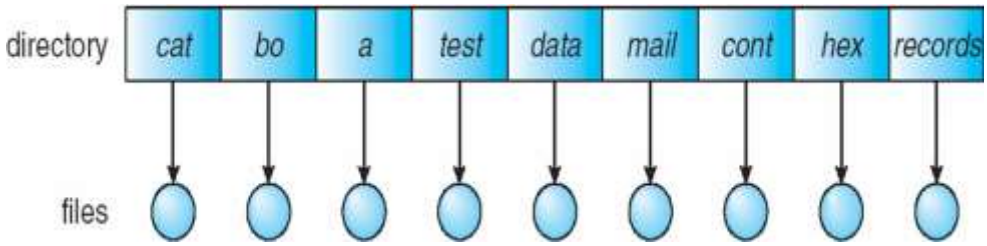
n = relative block number

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

Directory Structure

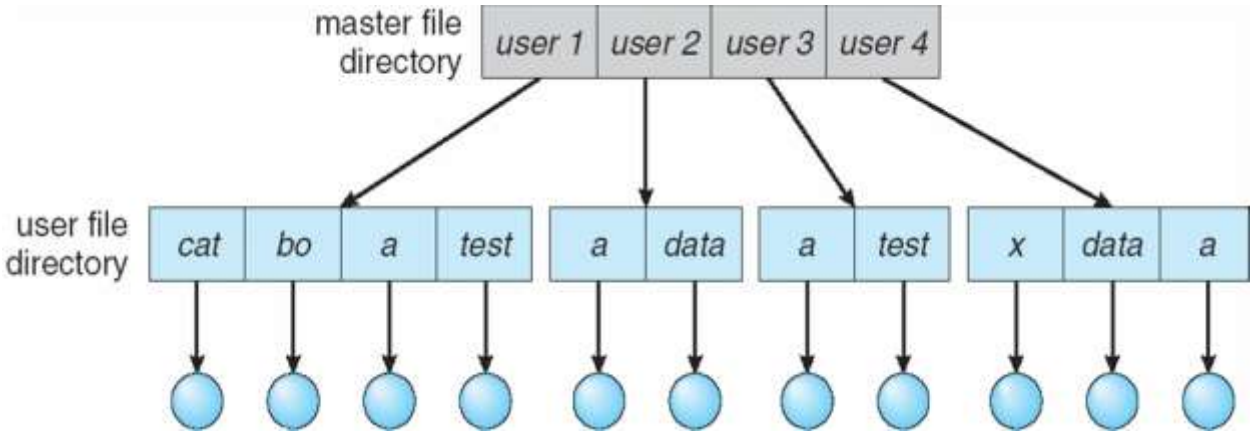
A. Single Level Directory

- A single directory for all users
- Naming problem
- Grouping problem



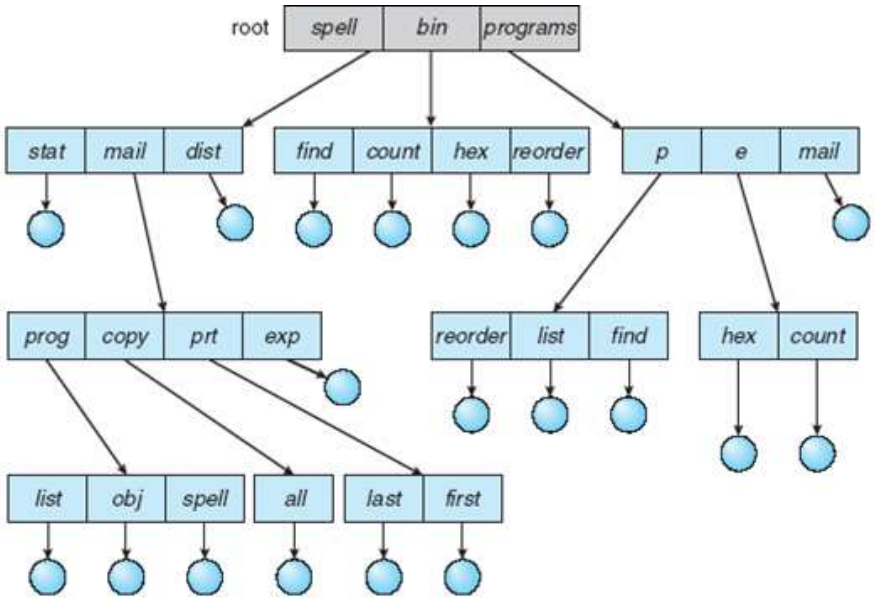
B. Two Level Directory

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

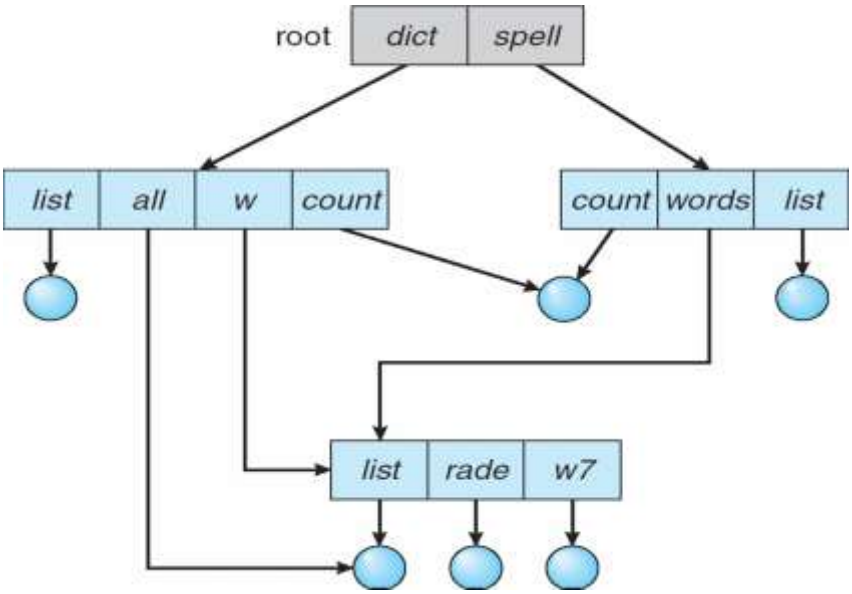


C. Tree Structure Directory

- Efficient searching
- Grouping Capability



D. Acyclic Graph Directories



- Have shared subdirectories and files

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights
- Uses networking to allow file system access between systems
    - Manually via programs like FTP
    - Automatically, seamlessly using **distributed file systems**
    - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
    - Server can serve multiple clients
    - Client and user-on-client identification is insecure or complicated
    - **NFS** is standard UNIX client-server file sharing protocol
    - **CIFS** is standard Windows protocol
    - Standard operating system file calls are translated into remote calls
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
    - Similar to Ch 7 process synchronization algorithms
        - Tend to be less complex due to disk I/O and network latency (for remote file systems
    - Andrew File System (AFS) implemented complex remote file sharing semantics
    - Unix file system (UFS) implements:
        - Writes to an open file visible immediately to other users of the same open file
        - Sharing file pointer to allow multiple users to read and write concurrently
    - AFS has session semantics
        - Writes only visible to sessions starting after the file is closed

File System Structure

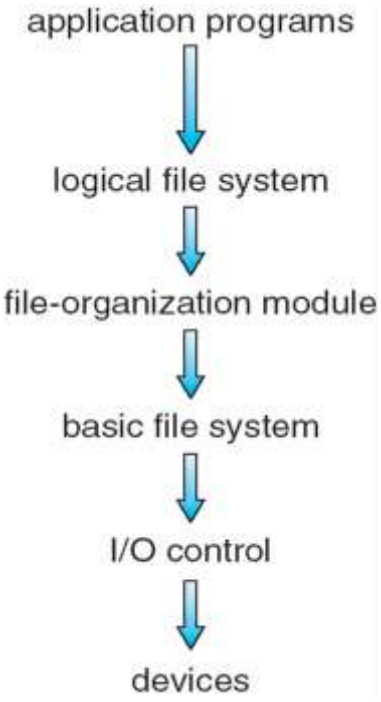
- File structure
    - Logical storage unit
    - Collection of related information

n File system resides on secondary storage (disks)

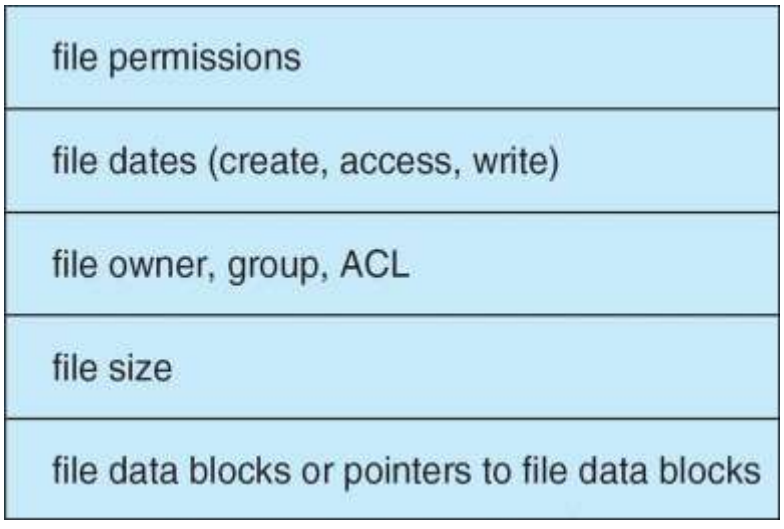
n File system organized into layers

n **File control block** – storage structure consisting of information about a file

Layered File System



File Control Block



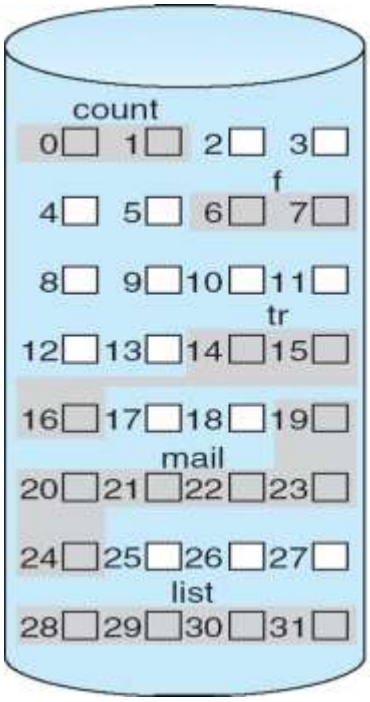
File Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

A. Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk

- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow

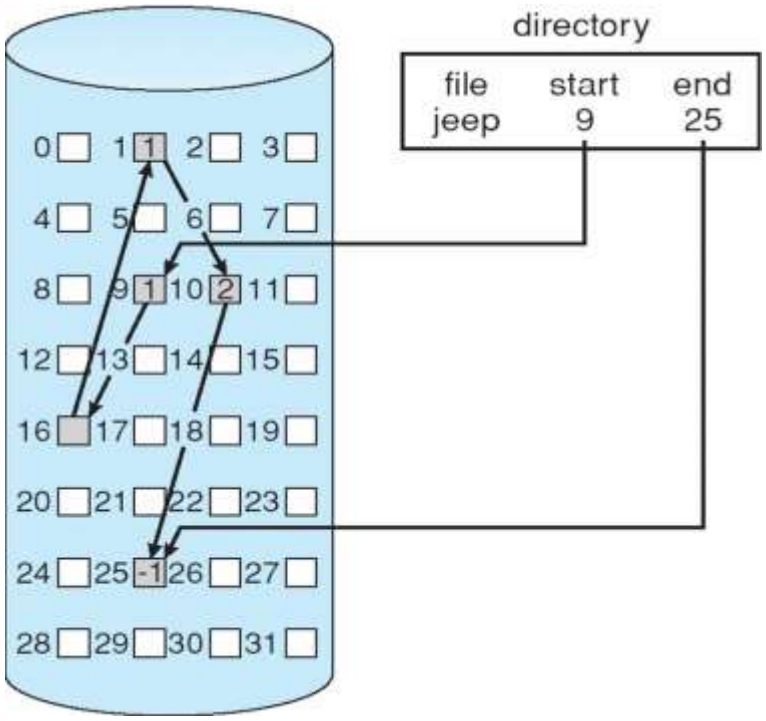


directory

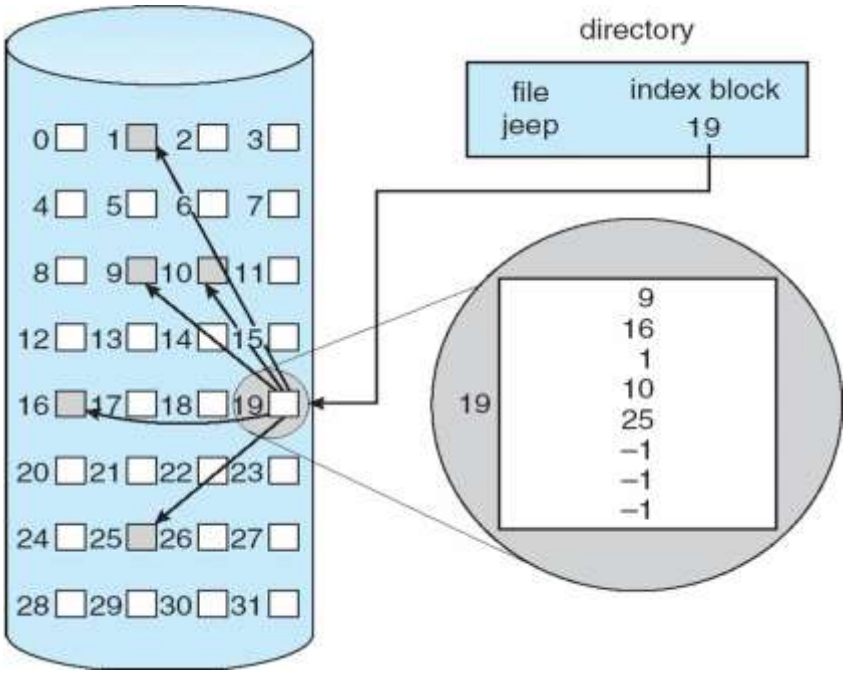
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

B. Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping



C. Indexed Allocation



- Brings all pointers together into the *index block.*
- Need index table
- Random access

- Dynamic access without external fragmentation, but have overhead of index block.

Secondary Storage Structure

Magnetic Disk

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