

**B.Sc. COMPUTER SCIENCE
SEMESTER V
OPERATING SYSTEMS**

UNIT – I

Introduction - History of operating system- Different kinds of operating system - Operation system concepts - System Calls-Operating system structure.

UNIT – II

Processes and Threads: Processes - threads - thread model and usage - inter process communication.

UNIT – III

Scheduling - Memory Management: Memory Abstraction - Virtual Memory - page replacement algorithms.

UNIT – IV

Deadlocks: Resources- introduction to deadlocks - deadlock detection and recovery - deadlocks avoidance - deadlock prevention. Multiple processor system: multiprocessors - multi computers.

UNIT – V

Input / Output: principles of I/O hardware - principles of I/O software. Files systems: Files - directories - files systems implementation - File System Management and Optimization

TEXT BOOK

1. Andrew S. Tanenbaum, "Modern Operating Systems", 2ndEdition, PHI private Limited, New Delhi, 2008.

REFERENCE BOOKS

1. William Stallings, "Operating Systems - Internals & Design Principles", 5thEdition, Prentice - Hall of India private Ltd, New Delhi, 2004.

2. Sridhar Vaidyanathan, "Operating System", 1st Edition, Vijay Nicole Publications, 2014

Introduction to operating System

An operating system (OS) is a set of programs that control the execution of application programs and act as an intermediary between a user of a computer and the computer hardware. OS is software that manages the computer hardware as well as providing an environment for application programs to run.

Examples of OS are: Windows, Windows/NT, OS/2 and MacOS.

Operating system Objectives

The objectives of OS are:

- (1) To make the computer system convenient and easy to use for the user.
- (2) To use the computer hardware in an efficient way.
- (3) To execute user programs and make solving user problems easier.

Computer System

A computer system can be divided into four components: the hardware, the operating system, the application programs and the users. The abstract view of system components is shown in figure 1.

- 1. Hardware:** such as CPU, memory and I/O devices.
- 2. Operating system:** provides the means of proper use of the hardware in the operations of the computer system, it is similar to government.
- 3. Application programs:** solve the computing problems of the user, such as : compilers, database systems and web browsers.
- 4. Users:** peoples, machine, or other computer.

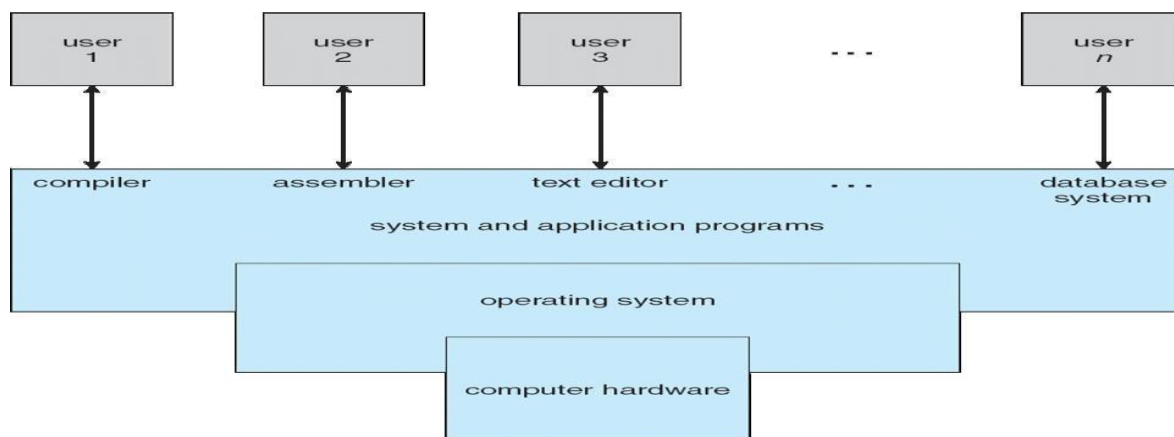


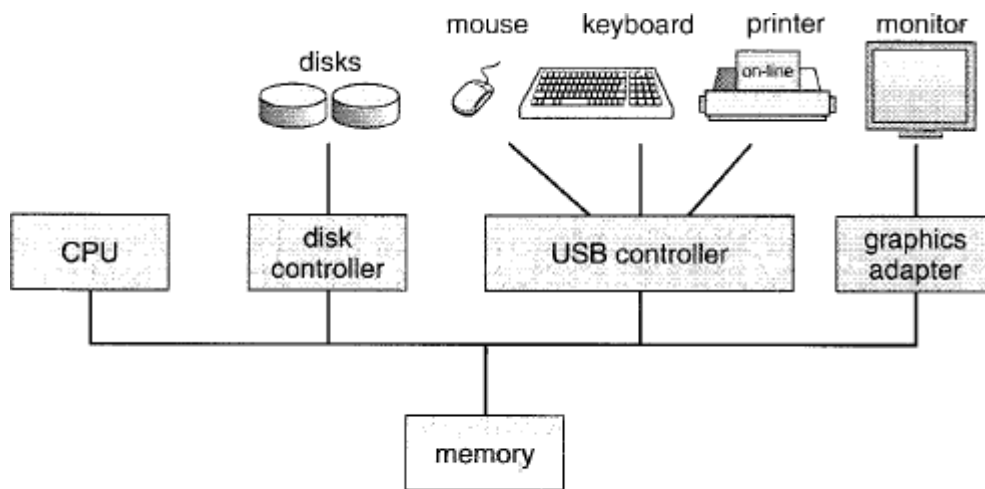
Fig:1 Computer System

1. Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is synchronizing access to the memory.

For a computer to start running-for instance, when it is powered up or rebooted- it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM), known by the general term **firmware**, within

the computer hardware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and to start executing that system. To accomplish this goal, the bootstrap program must locate and load into memory the operating system kernel. The operating system then starts executing the first process, such as "init," and waits for some event to occur.



A modern computer system.

Figure 2

2. Storage Structure

Computer programs must be in main memory (also called RAM) to be executed. Main memory is the only large storage area that the processor can access directly. It forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.

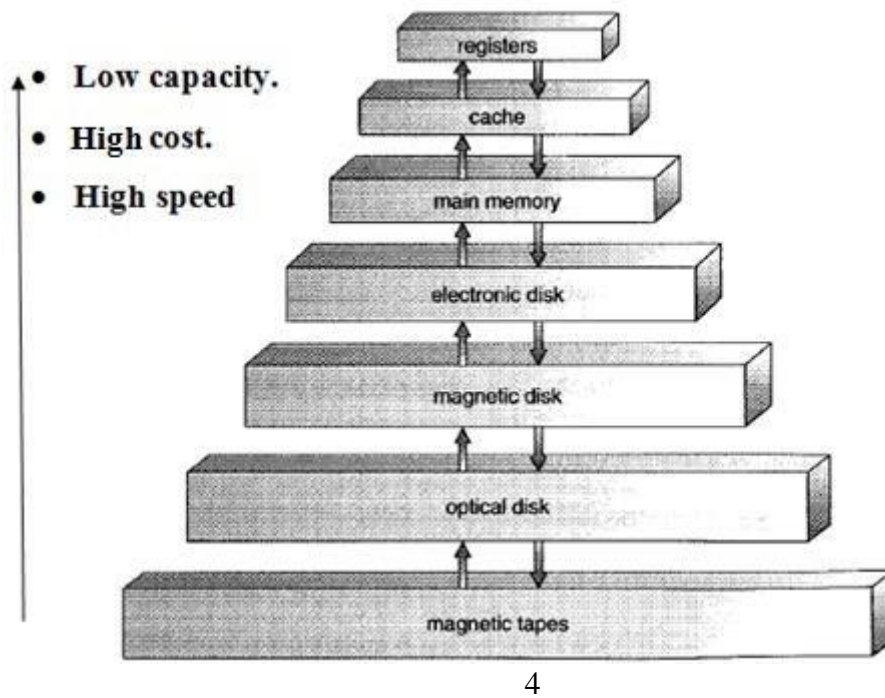
The **instruction-execution cycle** includes:

- 1) Fetches an instruction from memory and stores that instruction in the instruction register. And increment the PC register.
- 2) Decode the instruction and may cause operands to be fetched from memory and stored in some internal register.
- 3) Execute the instruction and store the result in memory.

The programs and data are not resided in main memory permanently for the following two reasons:

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

Thus, most computer systems provide secondary storage as an extension of main memory to hold large quantities of data permanently.



The wide variety of storage systems in a computer system can be organized in a hierarchy (figure 2).

The main differences among the various storage systems lie in speed, cost, size, and volatility. The higher levels are expensive, but they are fast.

Figure 3: Storage device hierarchy

I/O Structure

A computer system consists of **CPUs and multiple device controllers** that are connected through a common bus. **The device controller** is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a device driver for each device controller.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller examines the contents of these registers to determine what action to take. The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an **interrupt** that it has finished its operation. The device driver then returns control to the operating system. For other operations, the device driver returns status information.

For moving bulk data, direct memory access (DMA) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices.

Computer system structure

There are different categories for designing a computer system according to the number processors used.

1. Single-processor system: there is one CPU for executing instructions.
2. Multiprocessor system: It contains two or more processors that share bus, clock, physical memory and peripheral devices. The advantages of multiprocessors are:
 - a) Increase throughput.
 - b) Economy scale (less cost).
 - c) Increase reliability.
3. Clustered system: it consists of multiple computer systems connected by a local area network.

History of Operating system

Generation	Year	Electronic devices used	Types of OS and devices
First	1945 – 55	Vacuum tubes	Plug boards
Second	1955 – 1965	Transistors	Batch system
Third	1965 – 1980	Integrated Circuit (IC)	Multiprogramming
Fourth	Since 1980	Large scale integration	PC

Operating systems have been evolving through the years. Following table shows the history of OS.

Operating system Functions

OS performs many functions such as:

1. Implementing user interface.
2. Sharing HW among users.
3. Allowing users to share data among themselves.
4. Preventing users from interfering with one another.
5. Scheduling resource among users.

6. Facilitating I/O operations.
7. Recovering from errors.
8. Accounting for resource storage.
9. Facilitating parallel operations.
10. Organizing data for secure and rapid access.
11. Handling network communications.

Different kinds of Operating system

The main categories of modern OS may be classified into three groups which are distinguished by the nature of interaction that takes place between the computer and the user:

1. Batch system

In this type of OS, users submit jobs on regular schedule (e.g. daily, weekly, monthly) to a central place where the user of such system did not interact directly with computer system. To speed up the processing, jobs with similar needs were batched together and were run through the computer as a group. Thus, the programmer would leave the programs with the operator. The output from each job would send to the appropriate programmer. The major task of this type was to transfer control automatically from one job to the next.

Disadvantages of Batch System

1. Turnaround time can be large from user standpoint.
2. Difficult to debug program

2. Time-Sharing System

This type of OS provides on-line communication between the user and the system, the user gives his instructions directly and receives intermediate response, and therefore it called interactive system.

The time sharing system allows many user simultaneously share the computer system. The CPU is multiplexed rapidly among several programs, which are kept in memory and on disk. A program swapped in and out of memory to the disk.

Time sharing system reduces the CPU ideal time. The disadvantage is more complex.

3.Real time operating system

Real Time System is characterized by supplying immediate response. It guarantees that critical tasks complete on time. This type must have a pre- known maximum time limit for each of the functions to be performed on the computer. Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application.

The airline reservation system is an example of this type.

Performance development of OS

1.On-line and off-line operation

A special subroutine was written for each I/O device called a device controller. Some I/O devices has been equipped for either on-line operation (they are connected to the processor), or off-line operations (they are run by control unit).

2.Buffering

A buffer is an area of primary storage for holding data during I/O transfer. On input, the data are placed in the buffer by an I/O channel, when the transfer is complete the data may be accessed the processor. The buffing may be single or double.

3.Spooling (Simultaneously Peripheral Operation On-Line)

Spooling uses the disk as a very large buffer. Spooling is useful because device access data that different rates. The buffer provides a waiting station where data can rest while the slower device catches up.

Spooling allows overlapping between the computation of one job and I/O of another job

4.Multiprogramming

In multiprogramming several programs are kept in main memory at the same time, and the CPU is switching between them , thus the CPU always has a program to be execute. The OS begins to execute one program from memory, if this program need wait such as an I/O operation, the OS switches to another program. Multiprogramming increases CPU utilization. Multiprogramming system provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.

Advantages

- a) High CPU utilization.
- b) It appears that many programs are allotted CPU almost simultaneously.

Disadvantages

- a) CPU scheduling is requires.
- b) To accommodate many jobs in memory, memory management is required.

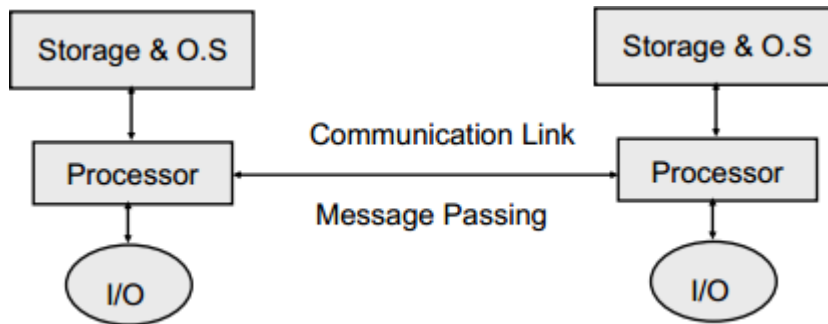
4.Parallel system

There are more than on processor in the system. These processors share the computer bus, clock, memory and I/O devices.

The advantage is to increase throughput (the number of programs completed in time unit).

6. Distributed system

Distribute the computation among several physical processors. It involves connecting 2 or more independent computer systems via communication link. So, each processor has its own O.S. and local memory; processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.



Advantages of distributed systems:

- Resources Sharing – You can share files and printers.
- Computation speed up – A job can be partitioned so that each processor can do a portion concurrently (load sharing).
- Reliability – If one processor failed the rest still can function with no problem.
- Communications – Such as electronic mail, ftp

Personal computer

Personal computers – computer system dedicated to a single user. PC operating systems were neither multi-user nor multi-tasking. The goal of PC operating systems were to maximize user convenience and responsiveness instead of maximizing CPU and I/O utilization.

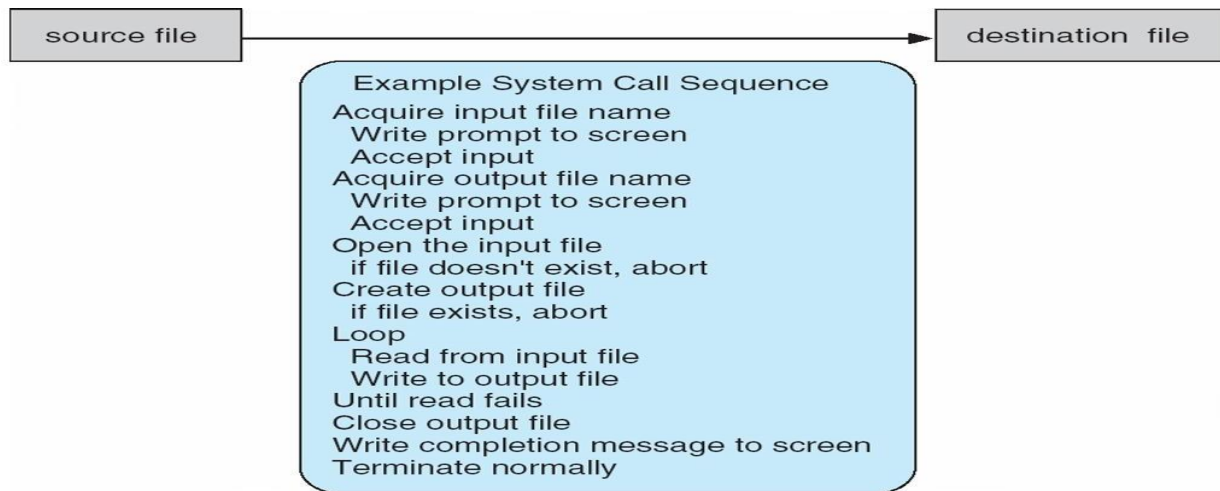
- Examples: Microsoft Windows and Apple Macintosh

System calls

System Calls

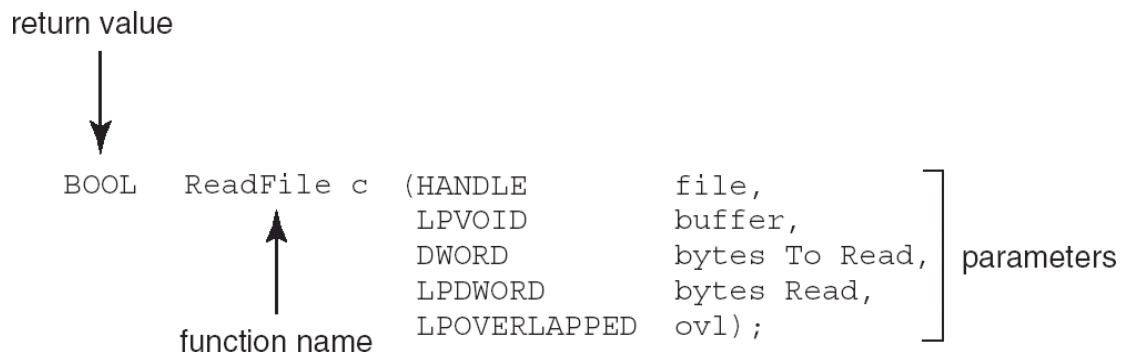
- Programming interface to the services
- provided by the OS Typically written in
- a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based 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)

Example of System Calls



Example of Standard API

Consider the ReadFile() function in the Win32 API—a function for reading from a file



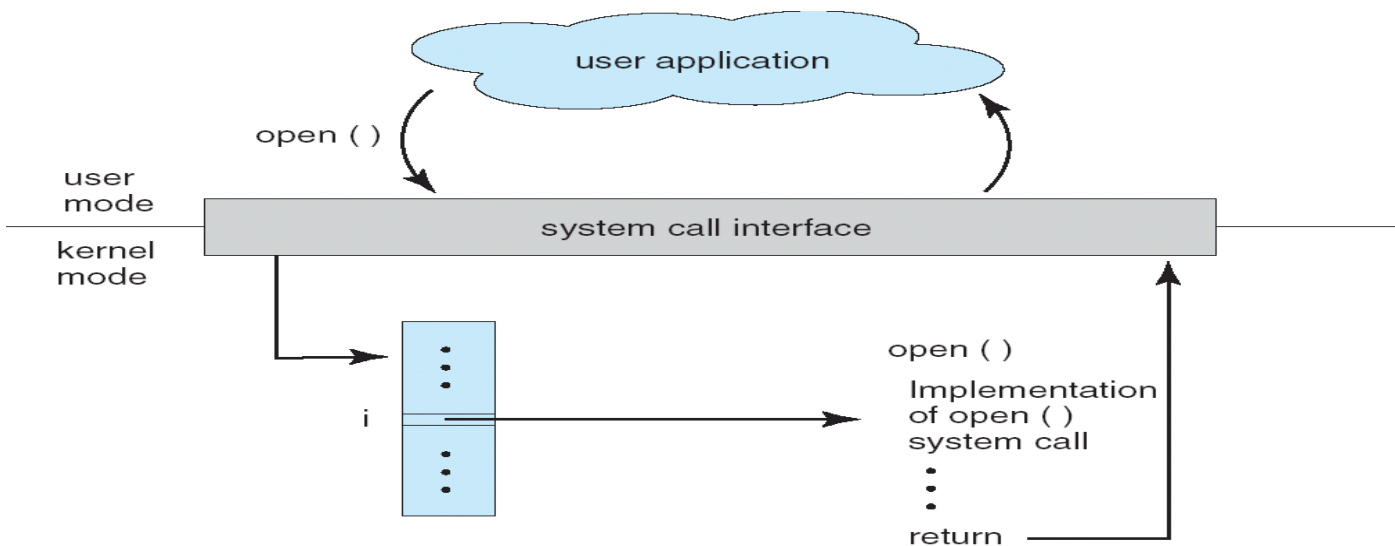
A description of the parameters

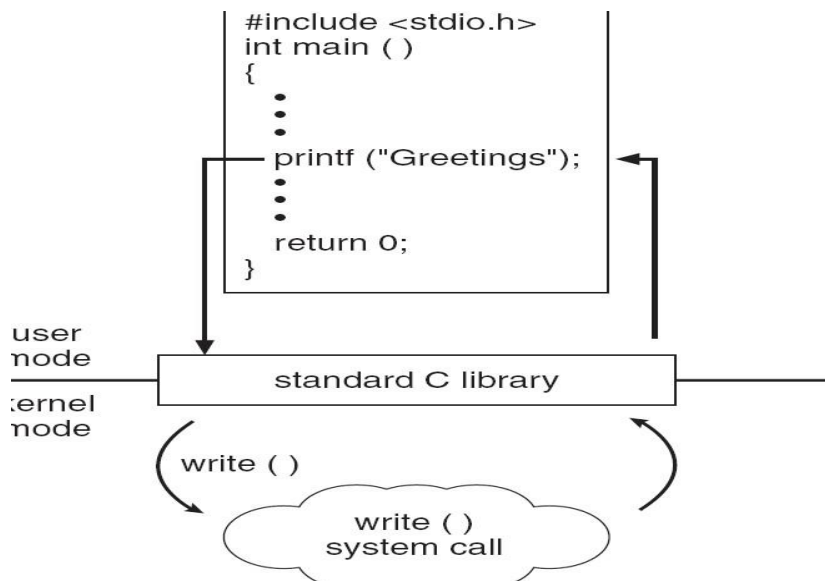
- passed to ReadFile()
- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

- Typically, a number associated with each system call
- System-call interface maintains a table indexed according to these Numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship

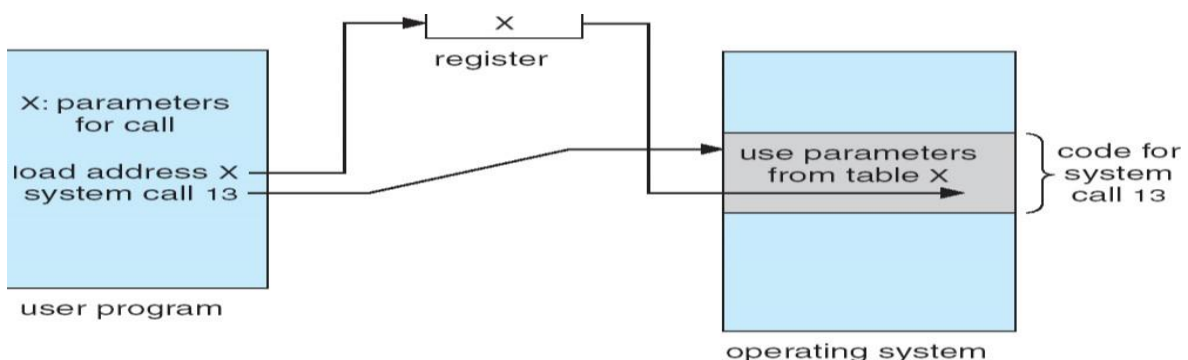




System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
- Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
- Simplest: pass the parameters in *registers*
 - In some cases, may be more parameters than registers
- Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
- Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



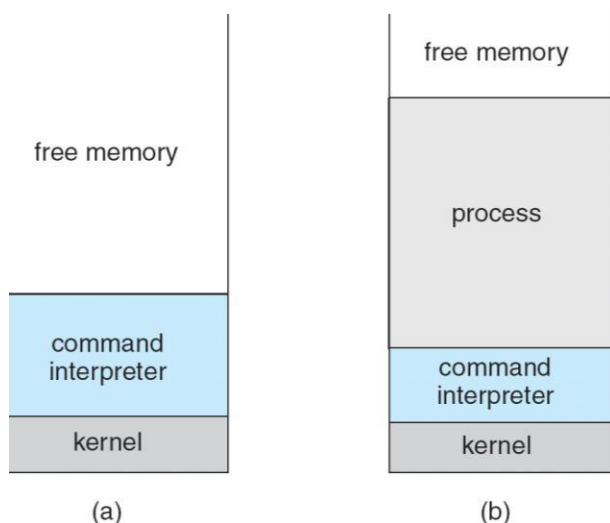
Types of System Calls

- ✓ **Process Control**
- ✓ **File Management**
- ✓ **Device Management**
- ✓ **Information maintenance**
- ✓ **Communications**
- ✓ **Protection**

Examples of Windows and Unix System Calls

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

MS-DOS execution



(a) At system startup

(b) running a program

Operating System Structure

MS-DOS System Structure

- ✓ MS-DOS – written to provide the most functionality in the least space.
- ✓ Not divided into modules.
- ✓ Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.

Unix System Structure

- **UNIX** – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.
- **Systems programs** – use kernel supported system calls to provide useful functions such as compilation and file manipulation.
- **The kernel** - Consists of everything below the system-call interface and above the physical hardware
- Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

Layered Approach

- ✓ The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- ✓ An OS layer is an implementation of an abstract object that is the encapsulation of data and operations that can manipulate those data. These operations (routines) can be invoked by higher-level layers. The layer itself can invoke operations on lower-level layers.
- ✓ Layered approach provides modularity. With modularity, layers are selected such that each layer uses functions (operations) and services of only lower-level layers.
- ✓ Each layer is implemented by using only those operations that are provided lower level layers.
- ✓ The major difficulty is appropriate definition of various layers.

Microkernel System Structure

- ✓ Moves as much from the kernel into “user” space.
- ✓ Communication takes place between user modules using message passing.

❖ Benefits:

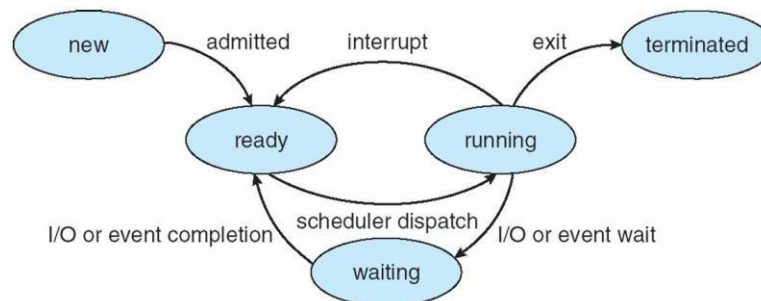
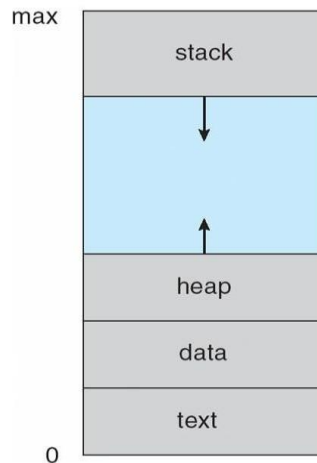
- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

UNIT-II

Processes and Threads

Processes-Process Concept:

- ✓ An operating system executes a variety of programs:
 - Batch system – —**jobs**”
 - Time-shared systems – —**user programs**” or —**tasks**”
- ✓ We will use the terms *job* and *process* almost interchangeably
- ✓ **Process** – is a program in execution (informal definition)
- ✓ Program is *passive* entity stored on disk (**executable file**), process is *active*
 - Program becomes process when executable file loaded into memory
- ✓ Execution of program started via GUI, command line entry of its name, etc
- ✓ One program can be several processes
 - Consider multiple users executing the same program
- ✓ In memory, a process consists of **multiple parts**:
 - **Program code**, also called **text section**
 - **Current activity** including
 - **program counter**
 - processor registers
 - **Stack** containing temporary data
 - Function parameters return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time



- As a process executes, it changes **state**
 - **new**: The process is being created
 - **ready**: The process is waiting to be assigned to a processor
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **terminated**: The process has finished execution

PROCESS CONTROL BLOCK (PCB)

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

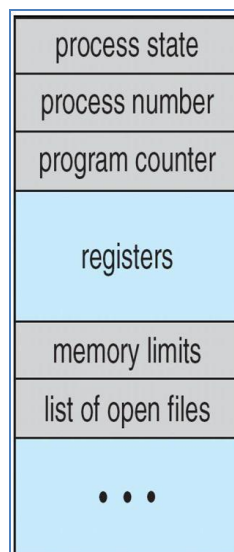


Figure 3.3 Process control block (PCB)

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters
- **Memory-management information:** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

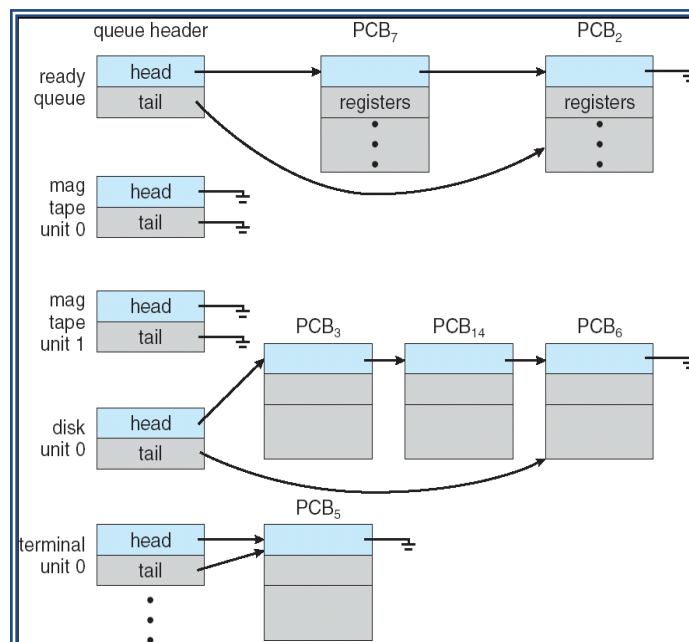
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Process Scheduling:

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

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

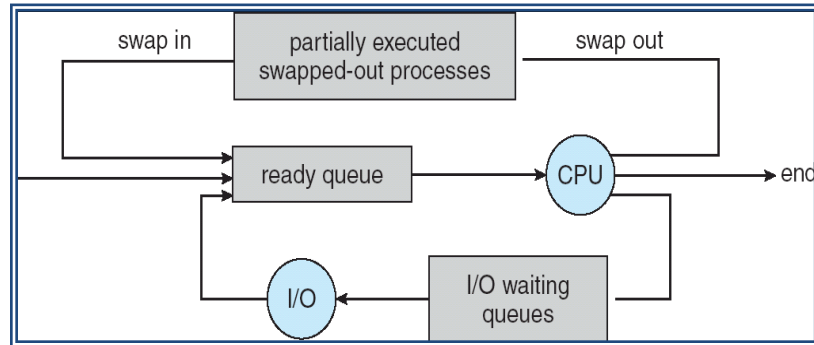


A common representation of process scheduling is a **queuing diagram**. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers

- **Long-term scheduler (or job scheduler)** – selects which processes should be brought into the ready queue
- **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU



- Short-term scheduler is invoked very frequently (milliseconds) → (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) → (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

Context Switch

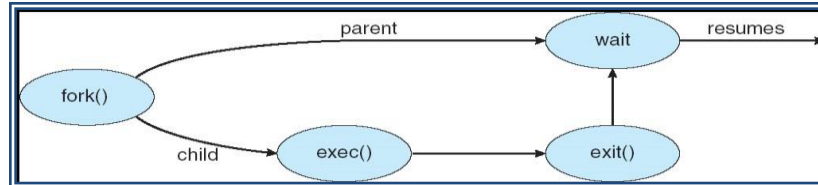
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Operations on Processes

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a **fork** to replace the process' memory space with a new program



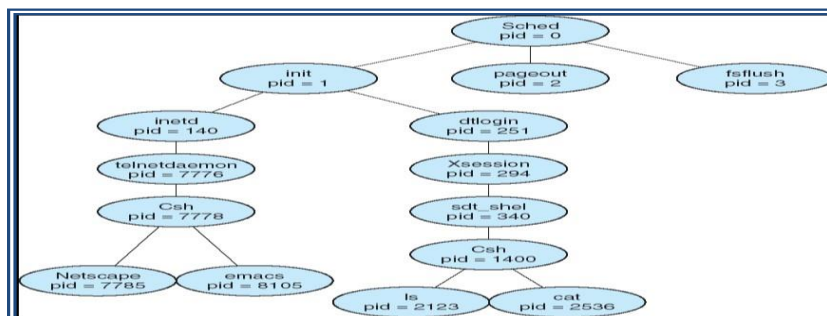
C Program Forking Separate Process

```

int main()
{
pid_t pid;
/* fork another process */
pid = fork();
if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
}
else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait (NULL);
    printf ("Child Complete");
    exit(0);
}
}

```

A tree of processes on a typical Solaris



Process Termination

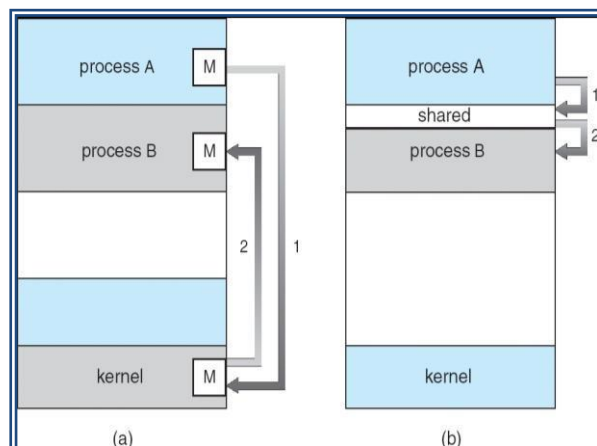
- Process executes last statement and asks the operating system to delete it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
 - Child has exceeded allocated resources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

Cooperating Processes

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

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(*message*) – message size fixed or variable
 - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)



Direct Communication

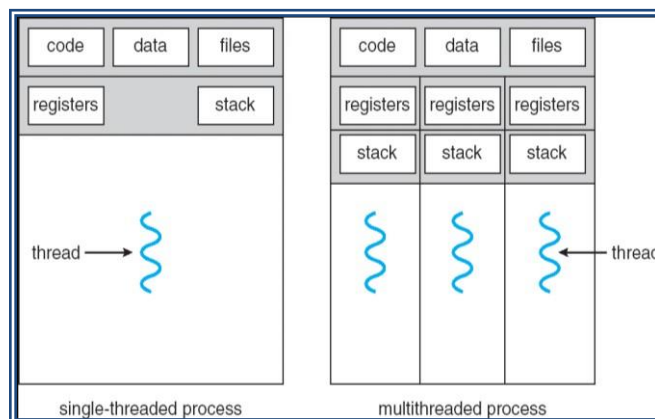
- Processes must name each other explicitly:
 - **send** ($P, message$) – send a message to process P
 - **receive**($Q, message$) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Threads- Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or *heavyweight*) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.



Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

2. **Resource sharing.** Processes can only share resources through techniques such as shared memory and message passing.
3. **Economy.** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems.

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however,

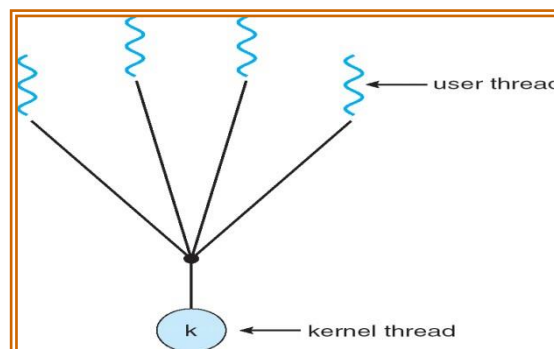
Concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core. Notice the distinction between *parallelism* and *concurrency* in this discussion. A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

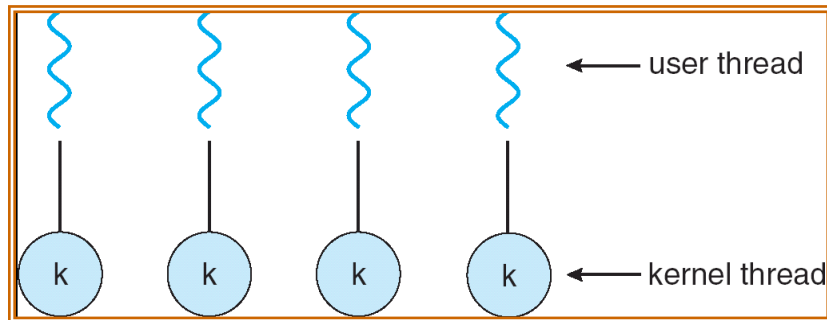
1. Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads



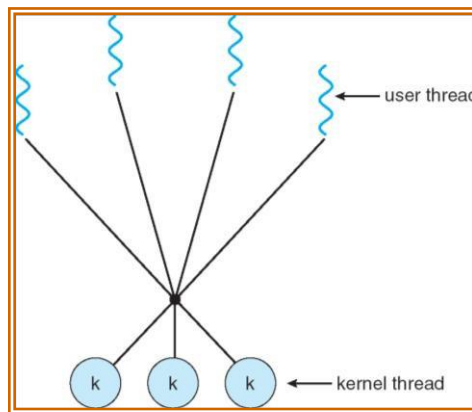
2. One-to-One

- Each user-level thread maps to kernel thread
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later



3. Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Windows 7

Windows implements the Windows API, which is the primary API for the family of Microsoft operating systems (Windows 98, NT, 2000, and XP, as well as Windows 7). Indeed, much of what is mentioned in this section applies to this entire family of operating systems. A Windows application runs as a separate process, and each process may contain one or more threads.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode

- A private storage area used by various run-time libraries and dynamic link libraries (DLLs).

The register set, stacks, and private storage area are known as the **context** of the thread. The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

Process Synchronization

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and increment it each time a new item is added to the buffer
- **Race condition:** The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, **concurrent processes** must be **synchronized**.

The Critical-Section Problem:

- There are n processes that are competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Requirements to be satisfied for a Solution to the Critical-Section Problem:

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

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

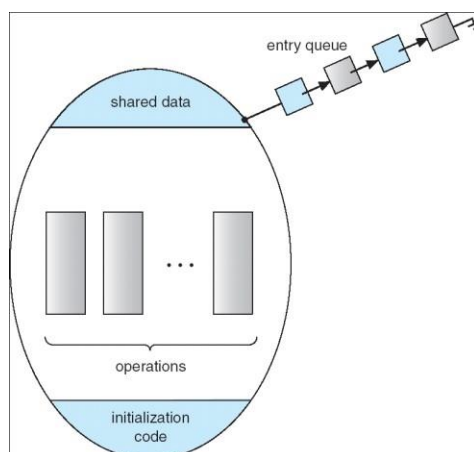
- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

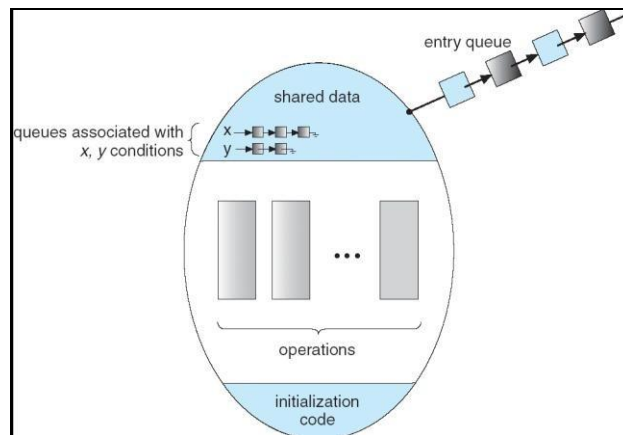
Mutex Locks

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time monitor monitor-name

```
{  
// shared variable declarations  
  
procedure body P1 (...) { .... }  
  
...  
  
procedure body Pn (...) {.....}  
  
{  
initialization code  
  
}  
  
}
```

- To allow a process to wait within the monitor, a condition variable must be declared as a condition x, y;
- Two operations on a condition variable:
 - x.wait ()—a process that invokes the operation is suspended.
 - x.signal ()—resumes one of the suspended processes(if any)





Solution to Dining Philosophers Problem

Monitor DP

```

{
enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];
void pickup (int i) {
state[i] = HUNGRY;
test(i);
if (state[i] != EATING) self [i].wait;
}
void putdown (int i) {
state[i] = THINKING;
// test left and right neighbors
test((i + 4) % 5);
test((i + 1) % 5);
}
void test (int i) {
if ( (state[(i + 4) % 5] != EATING) &&
(state[i] == HUNGRY) &&
(state[(i + 1) % 5] != EATING) ) {
state[i] = EATING ;
self[i].signal () ;
}
}
initialization_code() {
for (int i = 0; i < 5; i++)
state[i] = THINKING;
}
}

```

Semaphores

- It is a synchronization tool that is used to generalize the solution to the critical section problem in complex situations.
- A Semaphore s is an integer variable that can only be accessed via two indivisible (atomic) operations namely

```
wait (s)
{
1. wait or P operation ( to test )
2. signal or V operation ( to increment )
while(s <= 0);
s--;
}
signal (s)
{
s++;
}
```

Mutual Exclusion Implementation using semaphore

```
do
{
wait(mutex);
critical section

remainder section
} while (1);
signal(mutex);
```

Semaphore Implementation

- The semaphore discussed so far requires a busy waiting. That is if a process is in critical-section, the other process that tries to enter its critical-section must loop continuously in the entry code.
- To overcome the busy waiting problem, the definition of the semaphore operations wait and signal should be modified.
 - When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block operation places the process into a waiting queue associated with the semaphore.
 - A process that is blocked waiting on a semaphore should be restarted when some other process executes a signal operation. The blocked process should be restarted by a wakeup operation which put that process into ready queue.
- To implemented the semaphore, we define a semaphore as a record as:

```
typedef struct {
int value;
struct process *L;
} semaphore;
```

Deadlock & starvation:

Example: Consider a system of two processes , P0 & P1 each accessing two semaphores ,S & Q, set to the value 1.

P0	P1
Wait (S)	Wait (Q)
Wait (Q)	Wait (S)
.	.
.	.
.	.
Signal(S)	Signal(Q)
Signal(Q)	Signal(S)

- Suppose that P0 executes wait(S), then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal operations cannot be executed, P0 & P1 are deadlocked.
- Another problem related to deadlock is indefinite blocking or starvation, a situation where a process wait indefinitely within the semaphore. Indefinite blocking may occur if we add or remove processes from the list associated with a semaphore in LIFO order.

Types of Semaphores

- *Counting* semaphore – any positive integer value
- *Binary* semaphore – integer value can range only between 0 and 1

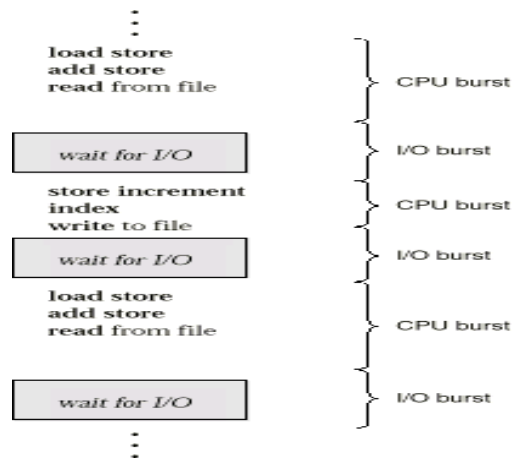
UNIT-III

CPU Scheduling

- CPU scheduling is the basis of multi programmed operating systems.
- The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization.
- Scheduling is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.

CPU-I/O Burst Cycle

- Process execution consists of a **cycle** of CPU execution and I/O wait.
- Processes alternate between these two states.
- Process execution begins with a **CPU burst**.
- That is followed by an **I/O burst**, then another CPU burst, then another I/O burst, and so on.
- Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.



CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

The selection process is carried out by the **short-term scheduler** (or CPU scheduler).

The ready queue is not necessarily a first-in, first-out (FIFO) queue. It may be a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

Preemptive Scheduling

- CPU scheduling decisions may take place under the following four circumstances:
 1. When a process switches from the running state to the waiting state
 2. When a process switches from the running state to the ready state
 3. When a process switches from the waiting state to the ready state
 4. When a process terminates
- Under 1 & 4 scheduling scheme is non preemptive.
- Otherwise the scheduling scheme is preemptive.

Non-preemptive Scheduling

- In non preemptive scheduling, once the CPU has been allocated a process, the process keeps the CPU until it releases the CPU either by termination or by switching to the waiting state.
- This scheduling method is used by the Microsoft windows environment.

Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

This function involves:

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program

Scheduling Criteria

- 1. CPU utilization:** The CPU should be kept as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- 2. Throughput:** It is the number of processes completed per time unit. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.
- 3. Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- 4. Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.
- 5. Response time:** It is the amount of time it takes to start responding, but not the time that it takes to output that response.

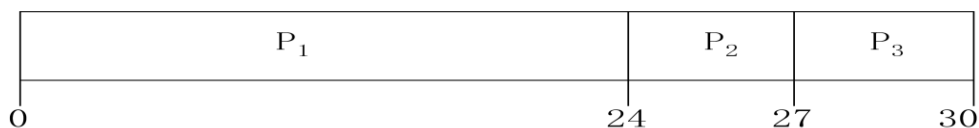
CPU Scheduling Algorithms

1. First-Come, First-Served Scheduling
2. Shortest Job First Scheduling
3. Priority Scheduling
4. Round Robin Scheduling

First-Come, First-Served (FCFS) Scheduling

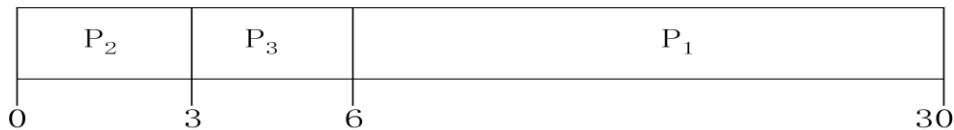
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

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



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Suppose that the processes arrive in the

order



The Gantt chart for the schedule is:

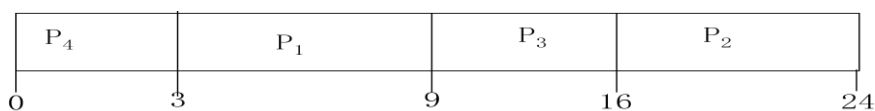
- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- *Convoy effect* short process behind long process

Shortest-Job-First (SJF) 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
 - The difficulty is knowing the length of the next CPU request

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

- SJF scheduling chart

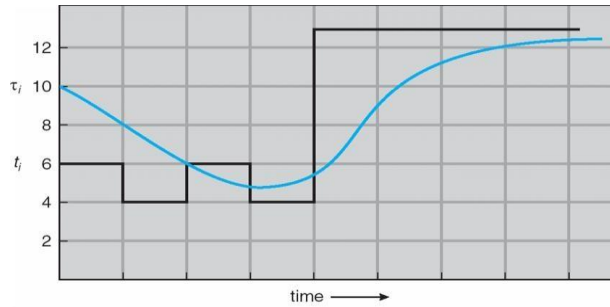


- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha \tau_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha \tau_n + (1 - \alpha) \alpha \tau_{n-1} + \dots \\ &+ (1 - \alpha)^j \alpha \tau_{n-j} + \dots \\ &+ (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

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)
 - Preemptive
 - 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 (RR)

- 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
 - q large \rightarrow FIFO
 - q small $\rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

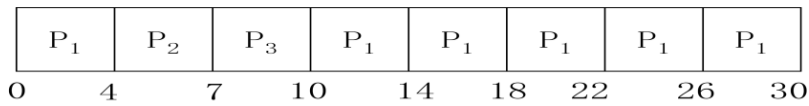
Process Burst Time

P_1 24

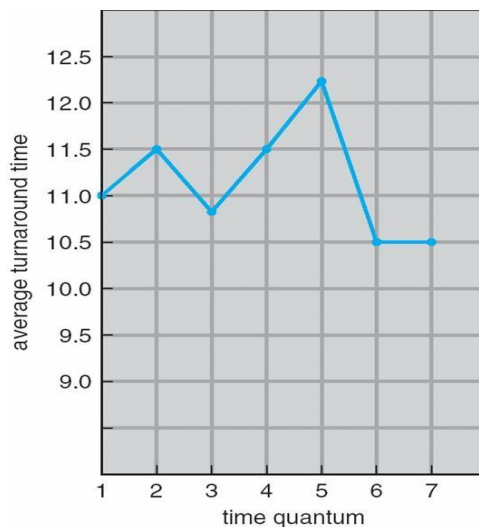
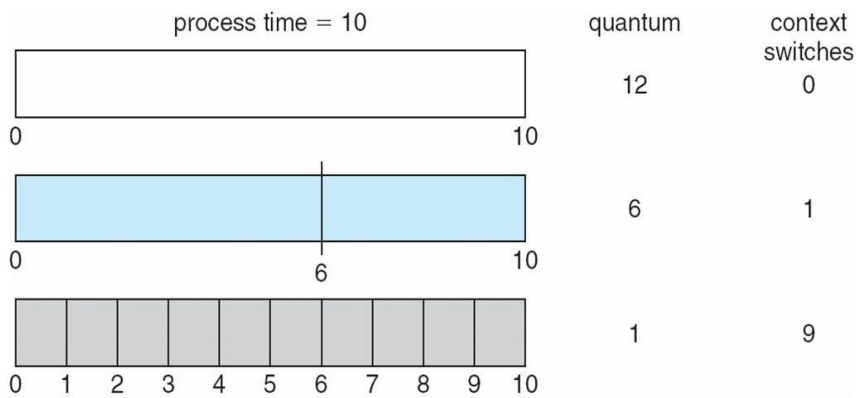
P_2 3

P_3 3

- The Gantt chart is:



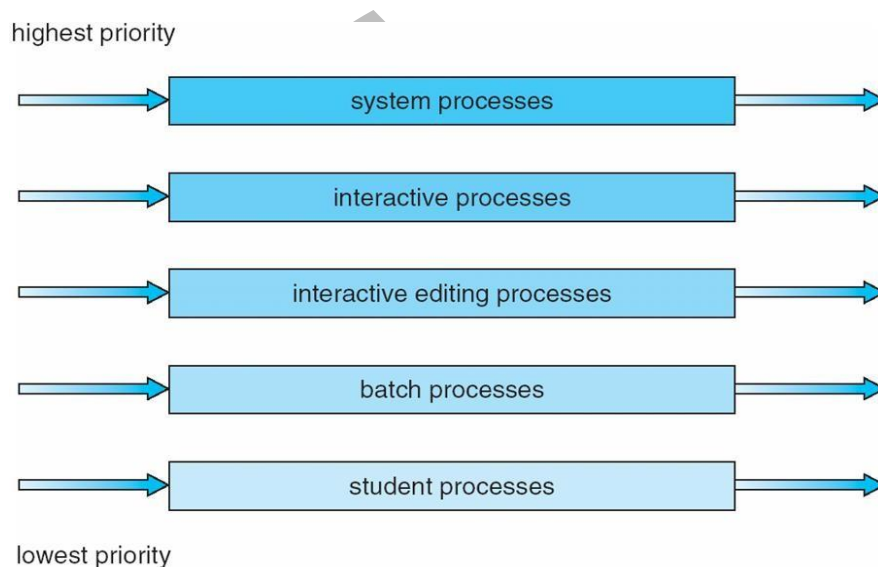
- Typically, higher average turnaround than SJF, but better *response*



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

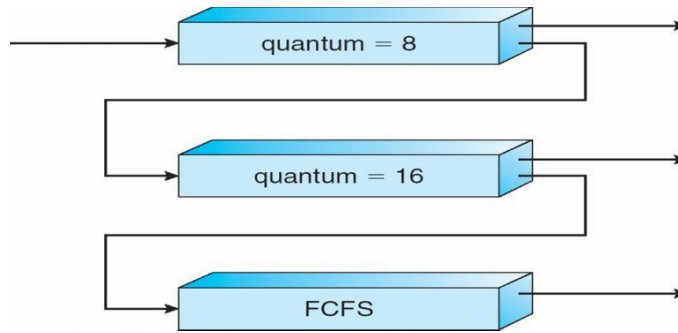
- Ready queue is partitioned into separate queues: foreground (interactive) background (batch)
- Each queue has its own scheduling algorithm
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - 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
 - 20% to background in FCFS



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process

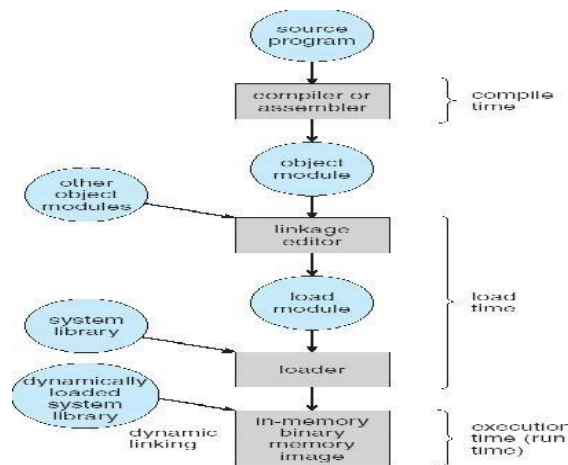
- method used to determine which queue a process will enter when that process needs service



- In general, to run a program, it must be brought into memory.
- Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run
- Address binding: Mapping of instructions and data from one address to another address in memory.

Three different stages of binding:

1. Compile time: Must generate absolute code if memory location is known in prior.
2. Load time: Must generate relocatable code if memory location is not known at compile time
3. Execution time: Need hardware support for address maps (e.g., base and limit registers).

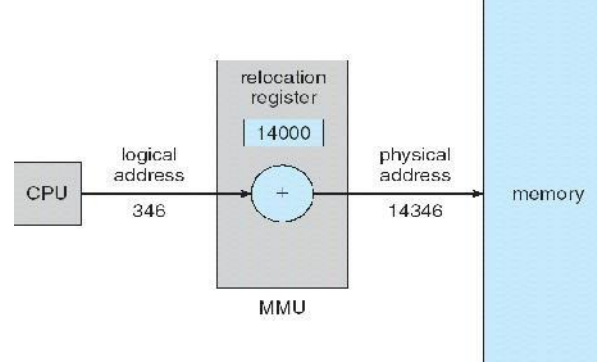


Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as “**virtual address**“
-
- **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 schemes||

Memory-Management Unit (MMU)

- It is a hardware device that maps virtual / Logical address to physical address
- In this scheme, the relocation register's value is added to Logical address generated by a user process.
-
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- Logical address range: 0 to max
- Physical address range : $R+0$ to $R+\text{max}$, where R —value in relocation register.



Dynamic Loading

- Through this, the routine is not loaded until it is called.
 - o Better memory-space utilization; unused routine is never loaded
 - o Useful when large amounts of code are needed to handle infrequently occurring cases
 - o No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time & is particularly useful for libraries
- Small piece of code called stub, used to locate the appropriate memory- resident library routine or function.
- 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
- Shared libraries: Programs linked before the new library was installed will continue using the older library.

Swapping

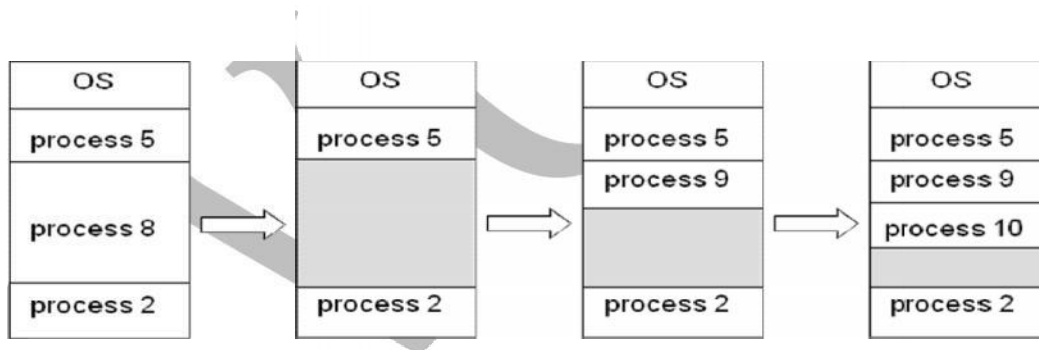
- A process can be swapped temporarily out of memory to a backing store (SWAP OUT) and then brought back into memory for continued execution (SWAP IN).
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users & it must provide direct access to these memory 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
- **Transfer time:** Major part of swap time is transfer time. Total transfer time is directly proportional to the amount of memory swapped.
- **Example:** Let us assume the user process is of size 1MB & the backing store is a standard hard disk with a transfer rate of 5MBPS.

Transfer time = $1000\text{KB} / 5000\text{KB per second}$
= $1/5 \text{ sec} = 200\text{ms}$

Contiguous Allocation

- Each process is contained in a single contiguous section of memory.
- There are two methods namely:
 - Fixed – Partition Method
 - Variable – Partition Method
- **Fixed – Partition Method :**
 - o Divide memory into fixed size partitions, where each partition has exactly one process.
 - o The drawback is memory space unused within a partition is wasted.(eg.when process size < partition size)
- **Variable-partition method:**
 - o Divide memory into variable size partitions, depending upon the size of the incoming process.
 - o When a process terminates, the partition becomes available for another process.
 - o As processes complete and leave they create holes in the main memory.
 - o **Hole** – block of available memory; holes of various size are scattered throughout memory.



Dynamic Storage- Allocation Problem:

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

Solution:

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

NOTE: First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

□ **Fragmentation:**

o **External Fragmentation** – This takes place when enough total memory space exists to satisfy a request, but it is not contiguous i.e, storage is fragmented into a large number of small holes scattered throughout the main memory.

o **Internal Fragmentation** – Allocated memory may be slightly larger than requested memory.

Example: hole = 184 bytes

Process size = 182 bytes.

We are left with a hole of 2 bytes.

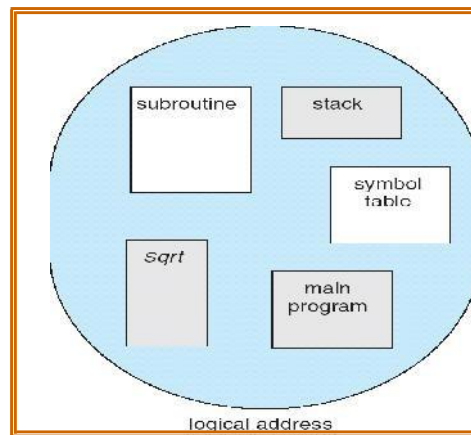
o **Solutions**

1. **Coalescing:** Merge the adjacent holes together.
2. **Compaction:** Move all processes towards one end of memory, hole towards other end of memory, producing one large hole of available memory. This scheme is expensive as it can be done if relocation is dynamic and done at execution time.

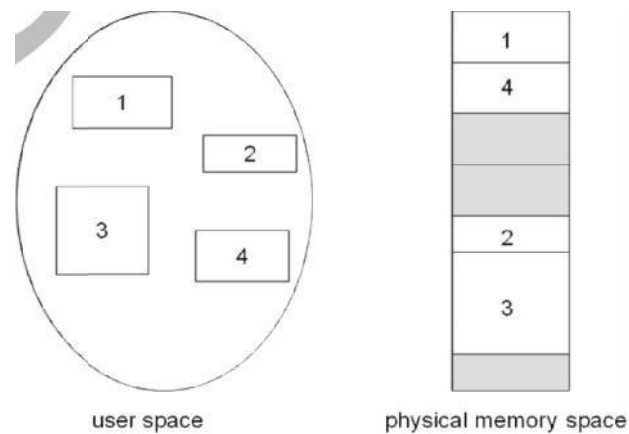
3. Permit the logical address space of a process to be **non-contiguous**. This is achieved through two memory management schemes namely **paging** and **segmentation**.

Segmentation

- o Memory-management scheme that supports user view of memory
- o A program is a collection of segments. A segment is a logical unit such as: Main program, Procedure, Function, Method, Object, Local variables, global variables, Common block, Stack, Symbol table, arrays

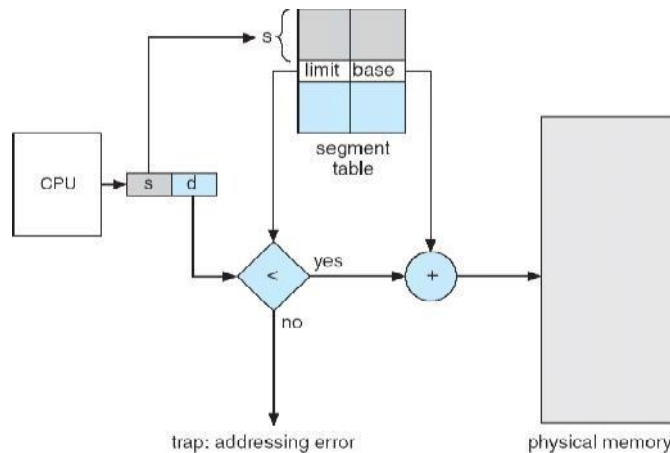


Logical View of Segmentation

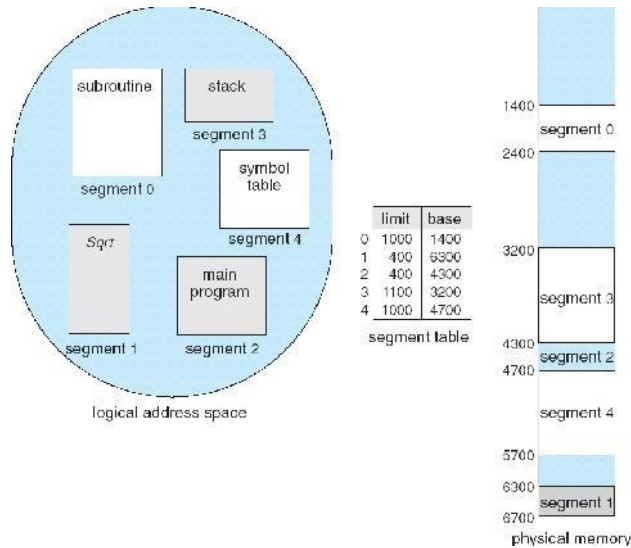


Segmentation Hardware

- o Logical address consists of a two tuple :
<Segment-number, offset>
- o **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - Base** – contains the starting physical address where the segments reside in memory
 - Limit** – specifies the length of the segment
- o **Segment-table base register (STBR)** points to the segment table's location in memory
- o **Segment-table length register (STLR)** indicates number of segments used by a program;
Segment number= s is legal, if $s < STLR$
- o **Relocation.**
 - dynamic
 - by segment table
- o **Sharing.**
 - shared segments
 - same segment number
- o **Allocation.**
 - first fit/best fit
 - external fragmentation
- o **Protection:** With each entry in segment table associate:
 - validation bit = 0 illegal segment
 - read/write/execute privileges
- o Protection bits associated with segments; code sharing occurs at segment level
- o Since segments vary in length, memory allocation is a dynamic storage- allocation problem
- o A segmentation example is shown in the following diagram



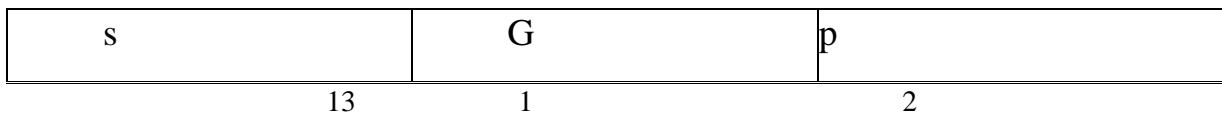
EXAMPLE



- o Another advantage of segmentation involves the sharing of code or data.
- o Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU.
- o Segments are shared when entries in the segment tables of two different processes point to the same physical location.

Segmentation with paging

- o The IBM OS/ 2.32 bit version is an operating system running on top of the Intel 386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.
 - o The local-address space of a process is divided into two partitions.
 - The first partition consists of up to 8 KB segments that are private to that process.
 - The second partition consists of up to 8KB segments that are shared among all the processes.
 - o Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**.
 - o Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.
- The logical address is a pair (selector, offset) where the selector is a 16-bit number:

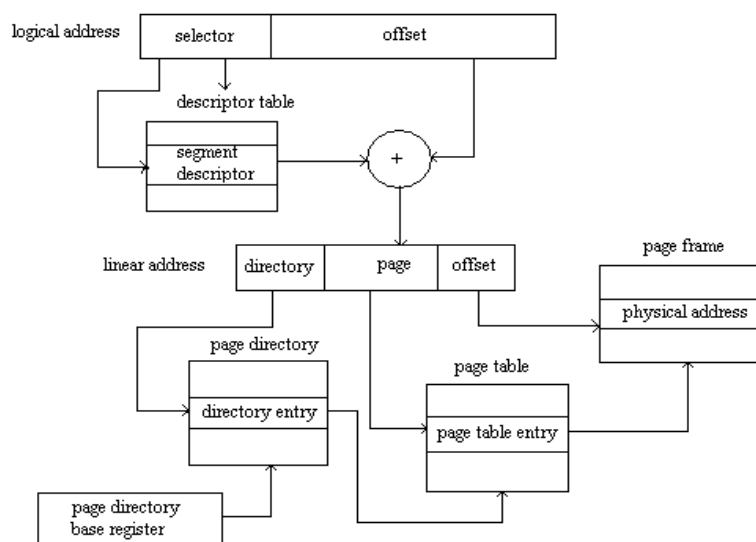
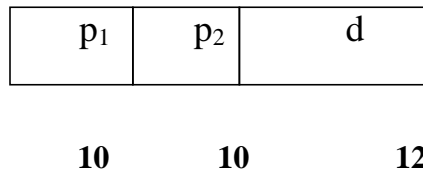


here s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

- o The base and limit information about the segment in question are used to generate a linear-

address.

- o First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.
- o The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows.



o To improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

o If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

Paging

- It is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

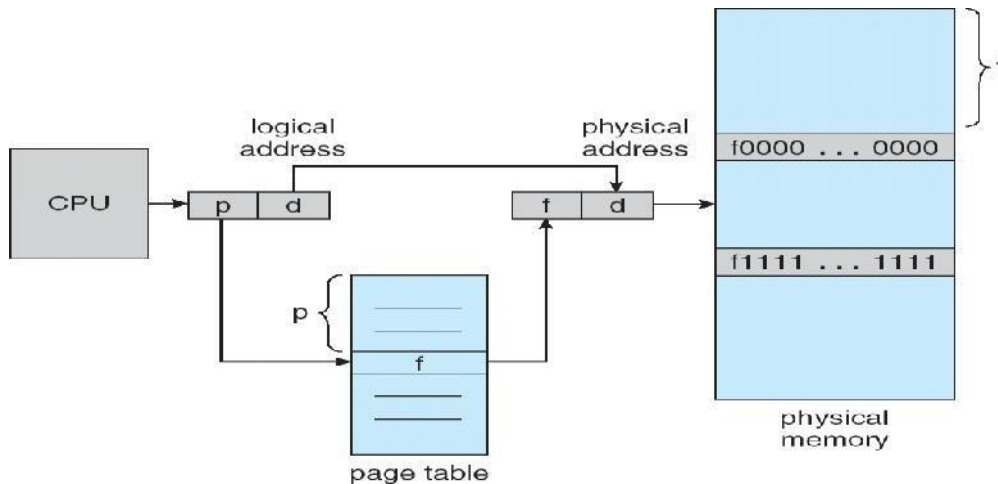
(i) Basic Method:

- o Divide logical memory into blocks of same size called “pages”.
- o Divide physical memory into fixed-sized blocks called “frames”
- o Page size is a power of 2, between 512 bytes and 16MB.

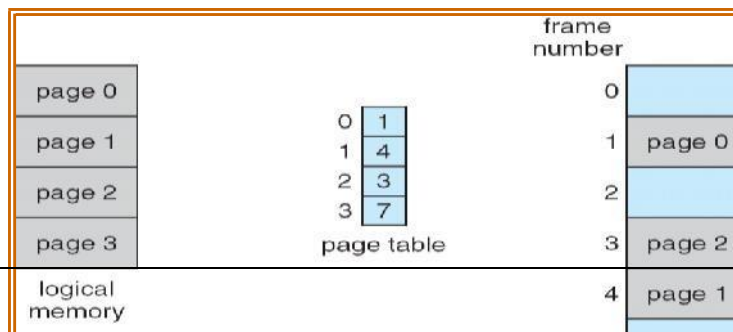
Address Translation Scheme

- o Address generated by CPU(logical address) is divided into:
 - **Page number (p)** – used as an index into a page table which contains base address of each page in physical memory
 - **Page offset (d)** –combined with base address to define the physical address i.e.,
$$\text{Physical address} = \text{base address} + \text{offset}$$

Paging Hardware



Paging model of logical and physical memory



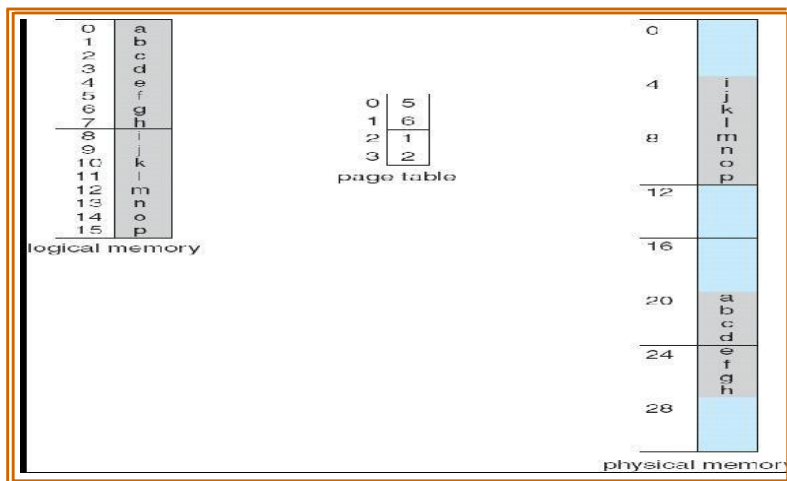
Paging example for a 32-byte memory with 4-byte pages

Page size = 4 bytes

Physical memory size = 32 bytes i.e (4 X 8 = 32 so, 8 pages)

Logical address 0 maps to physical address 20 i.e (5 X 4) + 0

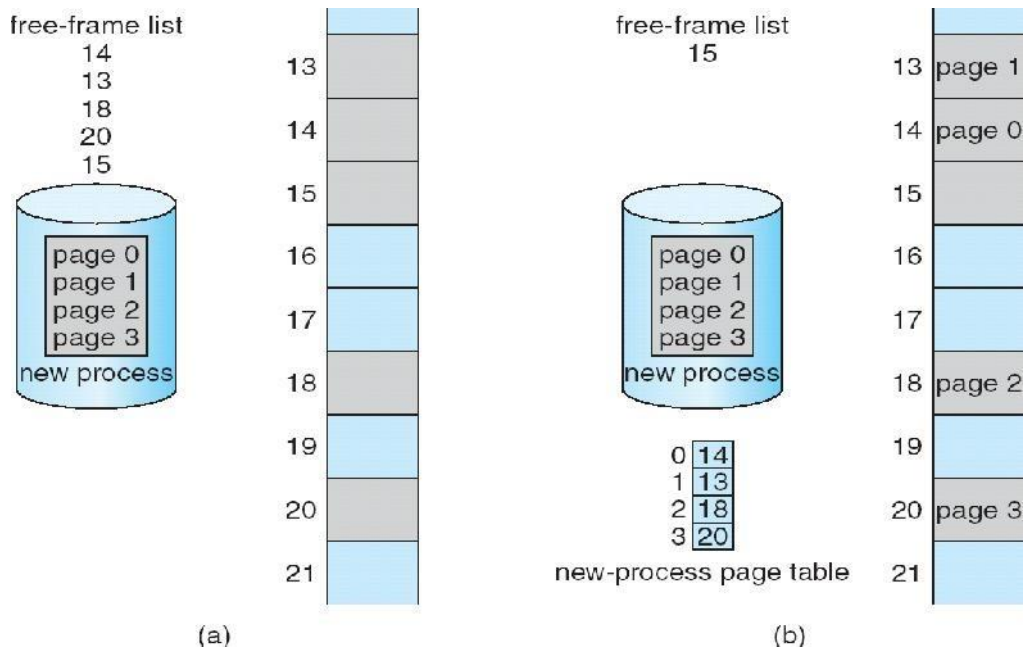
Where Frame no = 5, Page size = 4, Offset = 0



Allocation

- o When a process arrives into the system, its size (expressed in pages) is examined.
- o Each page of process needs one frame. Thus if the process requires n pages, at least n frames must be available in memory.

- o If n^{st} frames are available, they are allocated to this arriving process.
- o The 1st page of the process is loaded into one of the allocated frames & the frame number is put into the page table.
- o Repeat the above step for the next pages & so on.



(a) Before Allocation

(b) After Allocation

Frame table: It is used to determine which frames are allocated, which frames are available, how many total frames are there, and so on. (ie) It contains all the information about the frames in the physical memory.

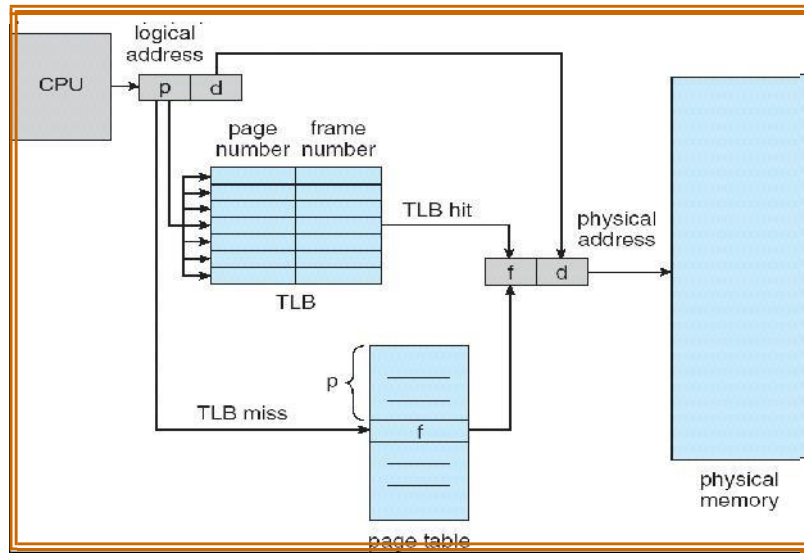
(ii) Hardware implementation of Page Table

- o This can be done in several ways :
 1. Using PTBR
 2. TLB
- o The simplest case is **Page-table base register (PTBR)**, is an index to point the page table.

o TLB (Translation Look-aside Buffer)

- It is a fast lookup hardware cache.
- It contains the recently or frequently used page table entries.
- It has two parts: Key (tag) & Value.
- More expensive.

Paging Hardware with TLB



- When a logical address is generated by CPU, its page number is presented to TLB.
 - **TLB hit:** If the page number is found, its frame number is immediately available & is used to access memory
 - **TLB miss:** If the page number is not in the TLB, a memory reference to the page table must be made.
 - **Hit ratio:** Percentage of times that a particular page is found in the TLB.
 - For example hit ratio is 80% means that the desired page number in the TLB is 80% of the time.

○ **Effective Access Time:**

- Assume hit ratio is 80%.
- If it takes 20ns to search TLB & 100ns to access memory, then the memory access takes 120ns(TLB hit)
- If we fail to find page no. in TLB (20ns), then we must 1st access memory for page table (100ns) & then access the desired byte in memory (100ns).

$$\text{Therefore Total} = 20 + 100 + 100$$

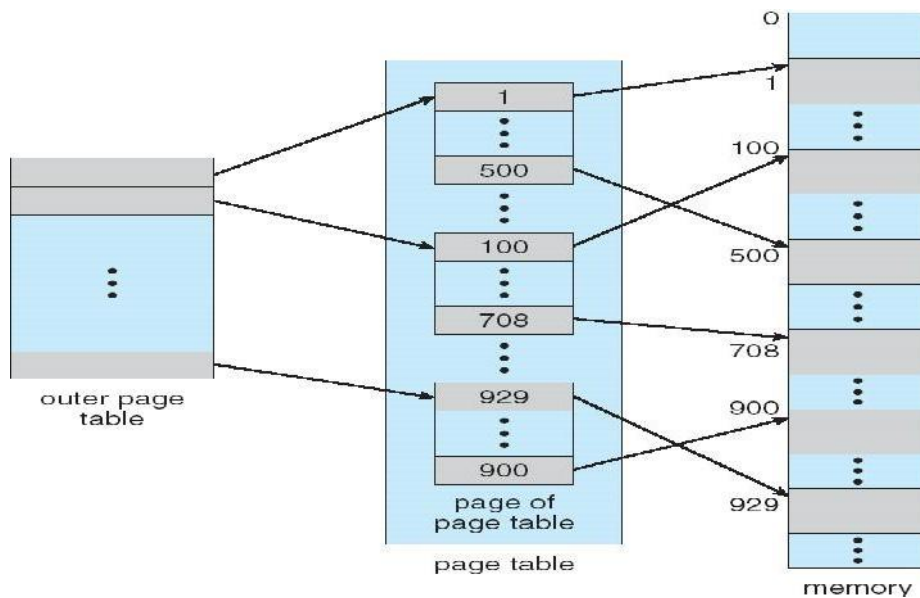
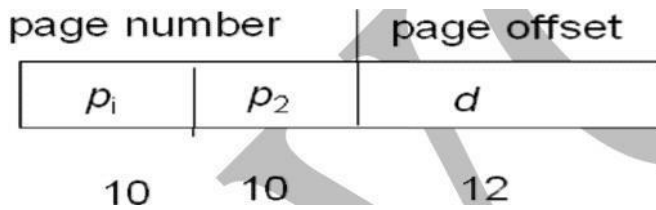
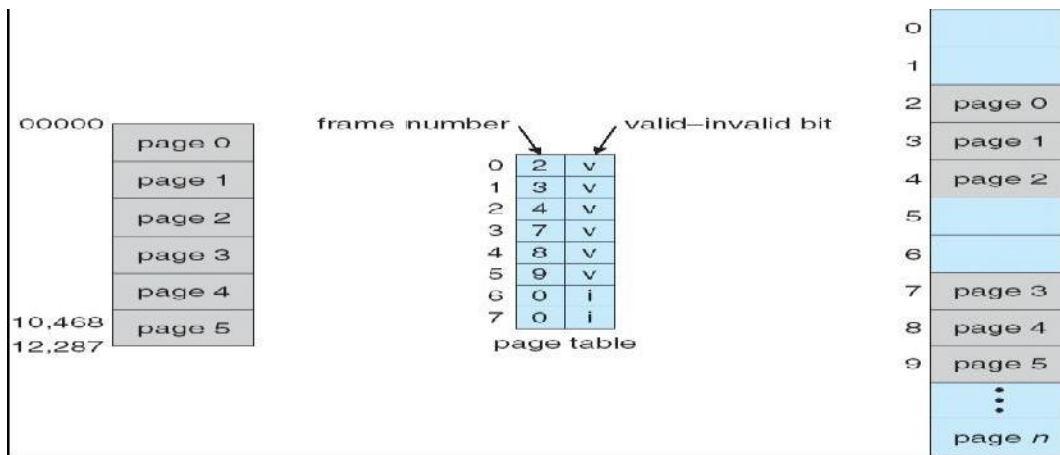
$$= 220 \text{ ns(TLB miss).}$$

$$\text{Then Effective Access Time (EAT)} = 0.80 \times (120 + 0.20 \times 220).$$

$$= 140 \text{ ns.}$$

(iii) Memory Protection

- Memory protection implemented by associating protection bit with each frame
- Valid-invalid bit attached to each entry in the page table:
 - **“valid (v)”** indicates that the associated page is in the process' logical address space, and is thus a legal page
 - **“invalid (i)”** indicates that the page is not in the process' logical address spaces



Virtual Memory

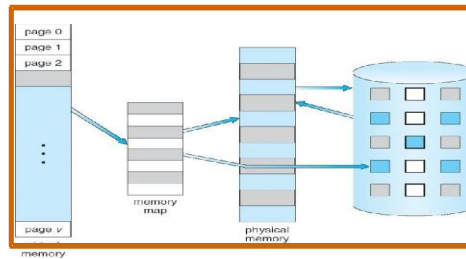
o It is a technique that allows the execution of processes that may not be completely in main memory.

o **Advantages:**

- Allows the program that can be larger than the physical memory.
- Separation of user logical memory from physical memory

- Allows processes to easily share files & address space.
 - Allows for more efficient process creation.
- o Virtual memory can be implemented using
- Demand paging
 - Demand segmentation

Virtual Memory That is Larger than Physical Memory



Demand Paging

- o It is similar to a paging system with swapping.
- o Demand Paging - Bring a page into memory only when it is needed
- o To execute a process, swap that entire process into memory. Rather than swapping the entire process into memory however, we use -Lazy Swapper
- o **Lazy Swapper** - Never swaps a page into memory unless that page will be needed.
- o **Advantages**
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users

Transfer of a paged memory to contiguous disk space

Basic Concepts:

- o Instead of swapping in the whole processes, the pager brings only those necessary pages into memory.
- Thus,
1. It avoids reading into memory pages that will not be used anyway.

- 2. Reduce the swap time.
- 3. Reduce the amount of physical memory needed.
- o To differentiate between those pages that are in memory & those that are on the disk we use the **Valid-Invalid bit**
- o A valid – invalid bit is associated with each page table entry.
- o Valid associated page is in memory.
- o In-Valid
 - invalid page
 - valid page but is currently on the disk

Page table when some pages are not in main memory

1. Determine whether the reference is a valid or invalid memory access
2. a) If the reference is invalid then terminate the process.
 - b) If the reference is valid then the page has not been yet brought into main memory.
3. Find a free frame.
4. Read the desired page into the newly allocated frame.
5. Reset the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted .

Pure demand paging

- o Never bring a page into memory until it is required.
- o We could start a process with no pages in memory.
- o When the OS sets the instruction pointer to the 1st instruction of the process,

Performance of demand paging

- o Let p be the probability of a page fault $0 \leq p \leq 1$
- o Effective Access Time (EAT)

$$\text{EAT} = (1 - p) \times \text{ma} + p \times \text{page fault time.}$$

- Where ma \square memory access, p \square Probability of page fault ($0 \leq p \leq 1$)
- o The memory access time denoted ma is in the range 10 to 200 ns.
 - o If there are no page faults then $\text{EAT} = \text{ma}$.
 - o To compute effective access time, we must know how much time is needed to service a page fault.
 - o A page fault causes the following sequence to occur:
 1. Trap to the OS
 2. Save the user registers and process state.
 3. Determine that the interrupt was a page fault.
 4. Check whether the reference was legal and find the location of page on disk.
 5. Read the page from disk to free frame.
 - a. Wait in a queue until read request is serviced.
 - b. Wait for seek time and latency time.
 - c. Transfer the page from disk to free frame.
 6. While waiting ,allocate CPU to some other user.
 7. Interrupt from disk.
 8. Save registers and process state for other users.
 9. Determine that the interrupt was from disk.
 7. Reset the page table to indicate that the page is now in memory.
 8. Wait for CPU to be allocated to this process again.
 9. Restart the instruction that was interrupted .

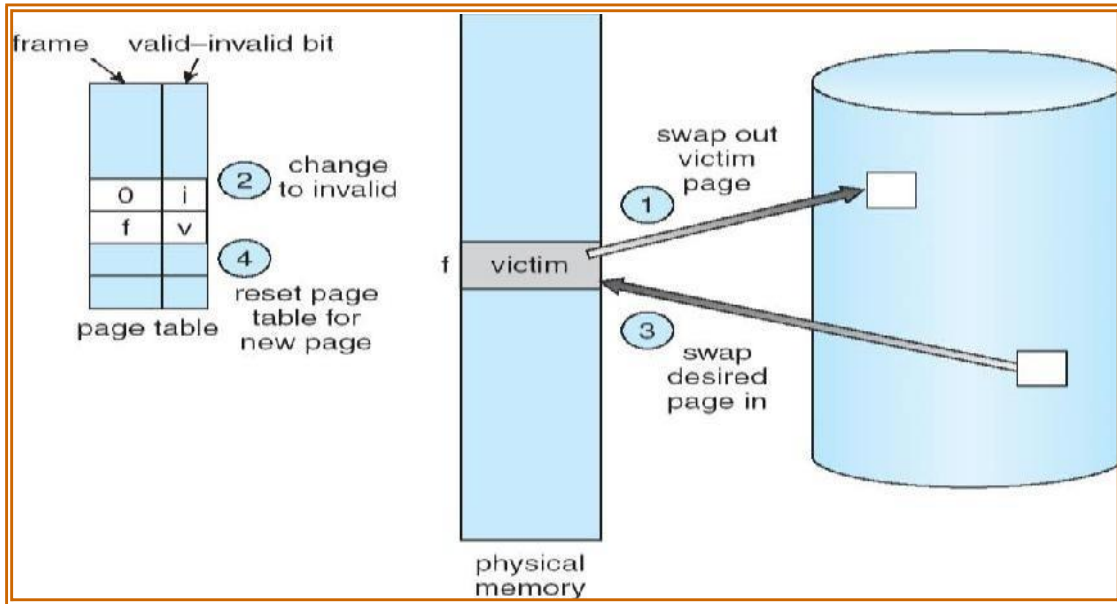
Page Replacement

- o If no frames are free, we could find one that is not currently being used & free it.
- o We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory.
- o Then we can use that freed frame to hold the page for which the process faulted.

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
 - If there is a free frame , then use it.
 - If there is no free frame, use a page replacement algorithm to select a **victim** frame
 - Write the victim page to the disk, change the page & frame tables accordingly.

3. Read the desired page into the (new) free frame. Update the page and frame tables.
4. Restart the process



Page Replacement Algorithms

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

(a) FIFO page replacement algorithm

o **Replace the oldest page.**

o This algorithm associates with each page, the time when that page was brought in.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No. of available frames = 3 (3 pages can be in memory at a time per process)

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

No. of page faults = 15

- o FIFO page replacement algorithm's performance is not always good.
- o To illustrate this, consider the following example:

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

- o If No.of available frames = 3 then the no.of page faults=9
- o If No.of available frames =4 then the no.of page faults=10
- o Here the no. of page faults increases when the no.of frames increases .This is called as **Belady's Anomaly**.

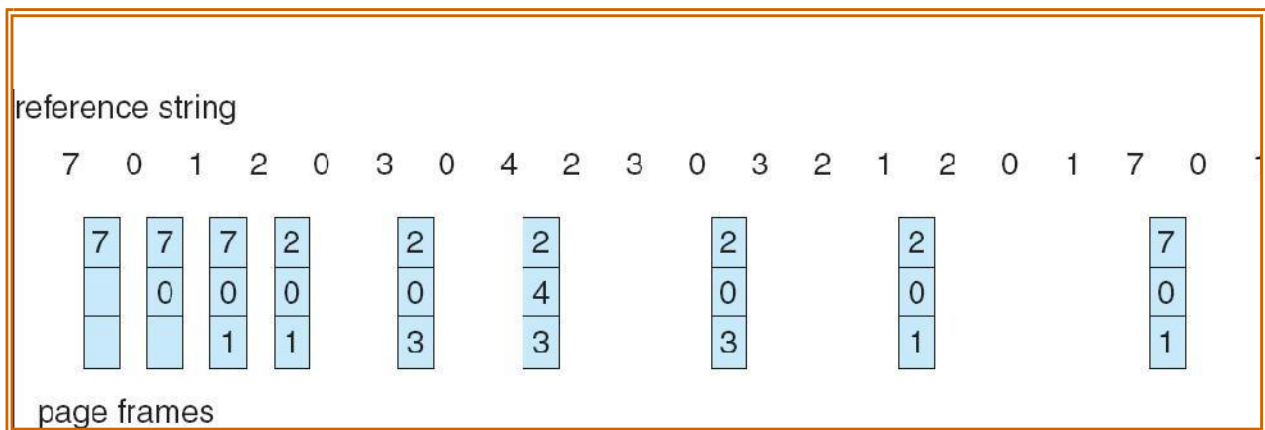
(b) Optimal page replacement algorithm

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

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No .of page faults=9

- o It is difficult to implement as it requires future knowledge of the reference string.

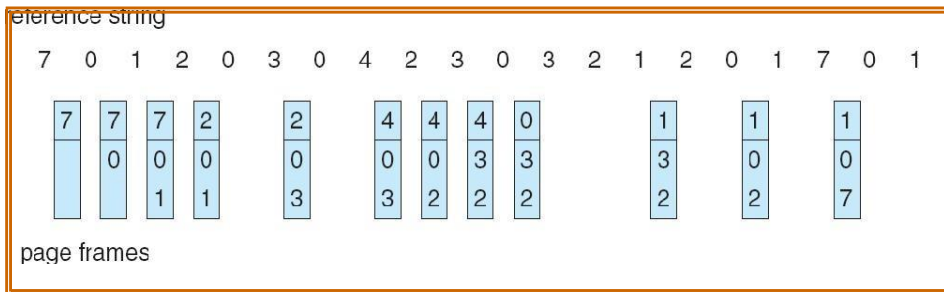
(c) LRU(Least Recently Used) page replacement algorithm

o **Replace the page that has not been used for the longest period of time.**

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No. of page faults = 12

o LRU page replacement can be implemented using

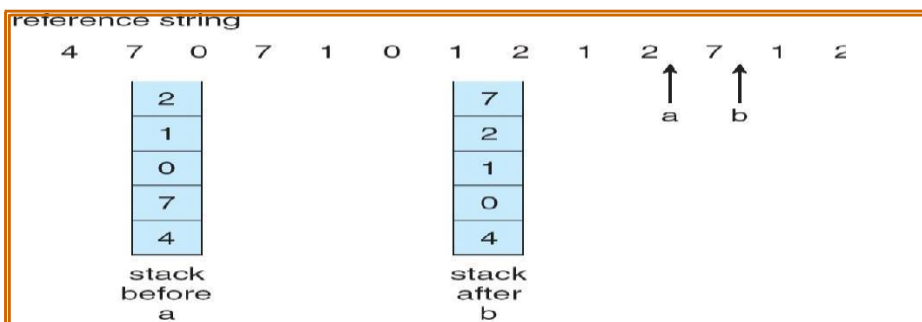
1. Counters

- Every page table entry has a time-of-use field and a clock or counter is associated with the CPU.
- The counter or clock is incremented for every memory reference.
- Each time a page is referenced, copy the counter into the time-of-use field.
- When a page needs to be replaced, replace the page with the smallest counter value.

2. Stack

- Keep a stack of page numbers
- Whenever a page is referenced, remove the page from the stack and put it on top of the stack.
- When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

Use of A Stack to Record The Most Recent Page Reference



(d) LRU Approximation Page Replacement

- o Reference bit
 - With each page associate a reference bit, initially set to 0
 - When page is referenced, the bit is set to 1

When a page needs to be replaced, replace the page whose reference bit is 0

- o The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- o Keep an 8-bit byte for each page in a table in memory.
- o At regular intervals , a timer interrupt transfers control to OS.
- o The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

Example:

oIf reference bit is 00000000 then the page has not been used for 8 time periods.

oIf reference bit is 11111111 then the page has been used atleast once each time period.

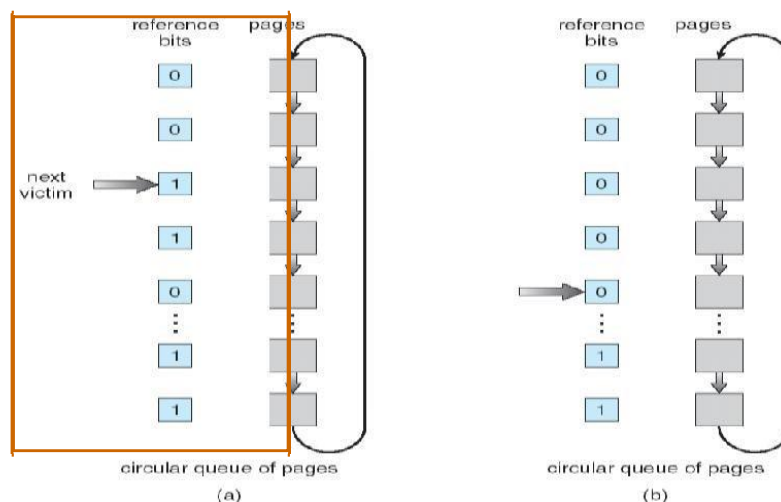
oIf the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

oBasic algorithm is FIFO

oWhen a page has been selected , check its reference bit.

- If 0 proceed to replace the page
- If 1 give the page a second chance and move on to the next FIFO page.
- When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.
- Hence a second chance page will not be replaced until all other pages are replaced.
-



(iii) Enhanced Second Chance Algorithm o Consider both reference bit and modify bit o There are four possible classes

1. (0,0) – neither recently used nor modified □ Best page to replace
2. (0,1) – not recently used but modified □ page has to be written out before replacement.
3. (1,0) - recently used but not modified □ page may be used again
4. (1,1) – recently used and modified □ page may be used again and page has to be written to disk

(e) Counting-Based Page Replacement

- o Keep a counter of the number of references that have been made to each page
 1. **Least Frequently Used (LFU)Algorithm:** replaces page with smallest count
 2. **Most Frequently Used (MFU)Algorithm:** replaces page with largest count
 - It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page Buffering Algorithm

- o These are used along with page replacement algorithms to improve their performance

Technique 1:

- o A pool of free frames is kept.
- o When a page fault occurs, choose a victim frame as before.
- o Read the desired page into a free frame from the pool
- o The victim frame is written onto the disk and then returned to the pool of free frames.

Technique 2:

- o Maintain a list of modified pages.
- o Whenever the paging device is idles, a modified is selected and written to disk and its modify bit is reset.

Technique 3:

- o A pool of free frames is kept.
- o Remember which page was in each frame.
- o If frame contents are not modified then the old page can be reused directly from the free frame pool when needed

Allocation of Frames

- o There are two major allocation schemes
 - Equal Allocation
 - Proportional Allocation

o Equal allocation

- If there are n processes and m frames then allocate m/n frames to each process.
- Example:** If there are 5 processes and 100 frames, give each process 20 frames.

o Proportional allocation

- Allocate according to the size of process
Let s_i be the size of process i.

Let m be the total no. of frames

Then $S = \sum s_i$

$a_i = s_i / S * m$

where a_i is the no.of frames allocated to process i.

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
 - System has 2 disk drives.
 - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
 - semaphores A and B , initialized to 1

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

System Model

- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.




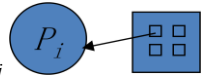
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.

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

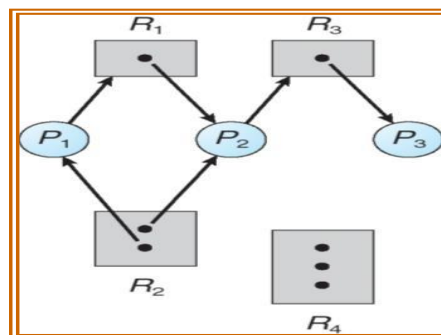
Resource-Allocation Graph

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

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 
- P_i is holding an instance of R_j 

Example of a Resource Allocation Graph



Basic Facts

- If graph contains no cycles \rightarrow no deadlock.
- If graph contains a cycle \rightarrow
 - if only one instance per resource type, then deadlock.
 - if several instances per resource type, possibility of deadlock.

Deadlock Prevention

- Mutual Exclusion – not required for sharable resources; must hold for non-sharable resources.
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
 - Low resource utilization; starvation possible.
- **No Preemption** –
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
 - Preempted resources are added to the list of resources for which the process is waiting.
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

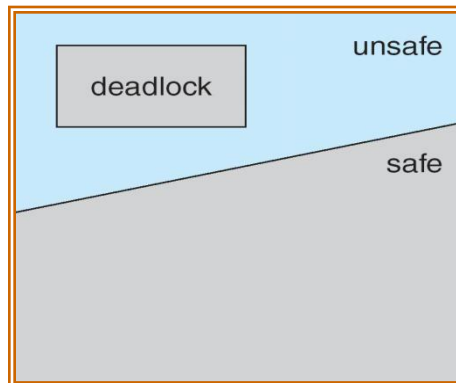
Deadlock Avoidance

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

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

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$.
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

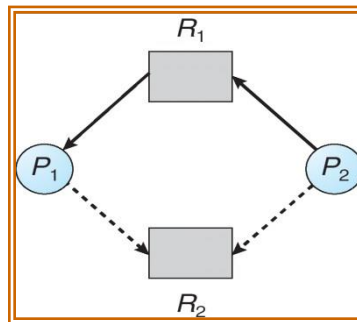


Avoidance algorithms

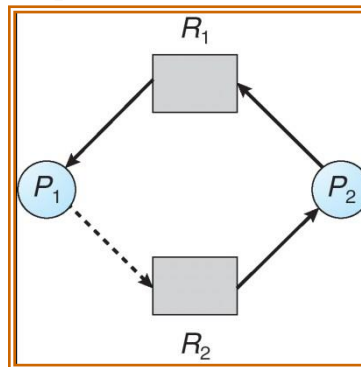
- Single instance of a resource type. Use a resource-allocation graph
- Multiple instances of a resource type. Use the banker's algorithm

Resource-Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4 ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances).

- Snapshot at time T_0 :

	<i>Allocation</i>			<i>Max</i>			<i>Available</i>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

- The content of the matrix *Need* is defined to be $Max - Allocation$.

	<i>Need</i>		
	<i>A</i>	<i>B</i>	<i>C</i>
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

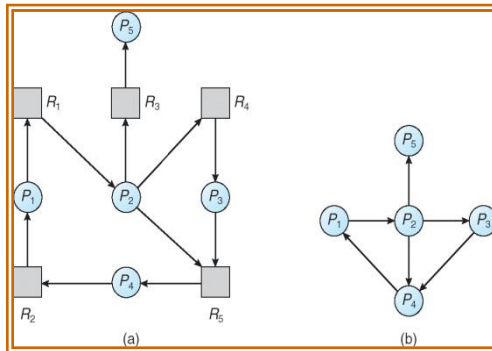
- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Deadlock Detection

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

Single Instance of Each Resource Type

- Maintain *wait-for* graph
 - Nodes are processes.
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j .
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.



Several Instances of a Resource Type

- *Available*: A vector of length m indicates the number of available resources of each type.
- *Allocation*: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- *Request*: An $n \times m$ matrix indicates the current request of each process. If $Request[i_j] = k$, then process P_i is requesting k more instances of resource type R_j .

Detection Algorithm

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

(a) $Work = Available$

(b) For $i = 1, 2, \dots, n$, if $Allocation_i \leq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.

2. Find an index i such that both:

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

(b) $Request_i \leq Work$

If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2.

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

deadlocked. Example of Detection

Algorithm

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0 :

	<i>Allocation</i>			<i>Request Available</i>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	0
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all i .
- P_2 requests an additional instance of type C.

	<i>Request A</i>		
	<i>B</i>	<i>C</i>	
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes; requests.
 - Deadlock exists, consisting of processes P_1, P_2, P_3 , and P_4 .

Recovery from Deadlock: 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?
 - Priority of the process.
 - How long process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - Is process interactive or batch?
- 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.

Multiprocessor:

- A set of processors connected by a communications network

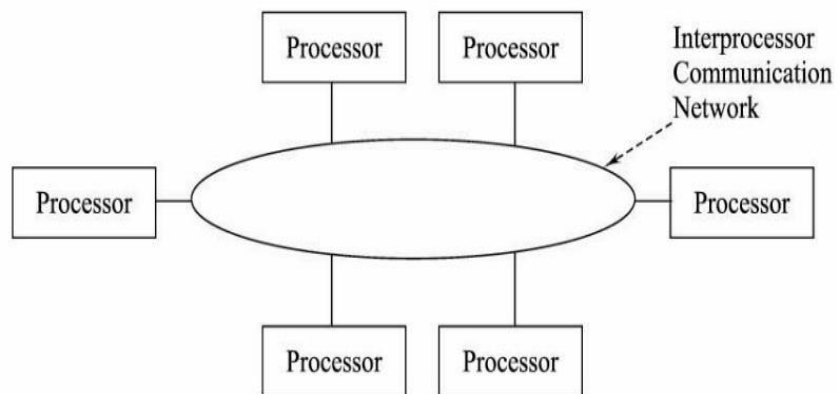


Fig. 5.1 Basic multiprocessor architecture

- A multiprocessor system is an interconnection of two or more CPU's with memory and input-output equipment.

- Multiprocessors system are classified as multiple instruction stream, multiple data stream systems(MIMD).
- There exists a distinction between multiprocessor and multicomputers that though both support concurrent operations.
- In multicomputers several autonomous computers are connected through a network and they may or may not communicate but in a multiprocessor system there is a single OS Control that provides interaction between processors and all the components of the system to cooperate in the solution of the problem.
- VLSI circuit technology has reduced the cost of the computers to such a low Level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

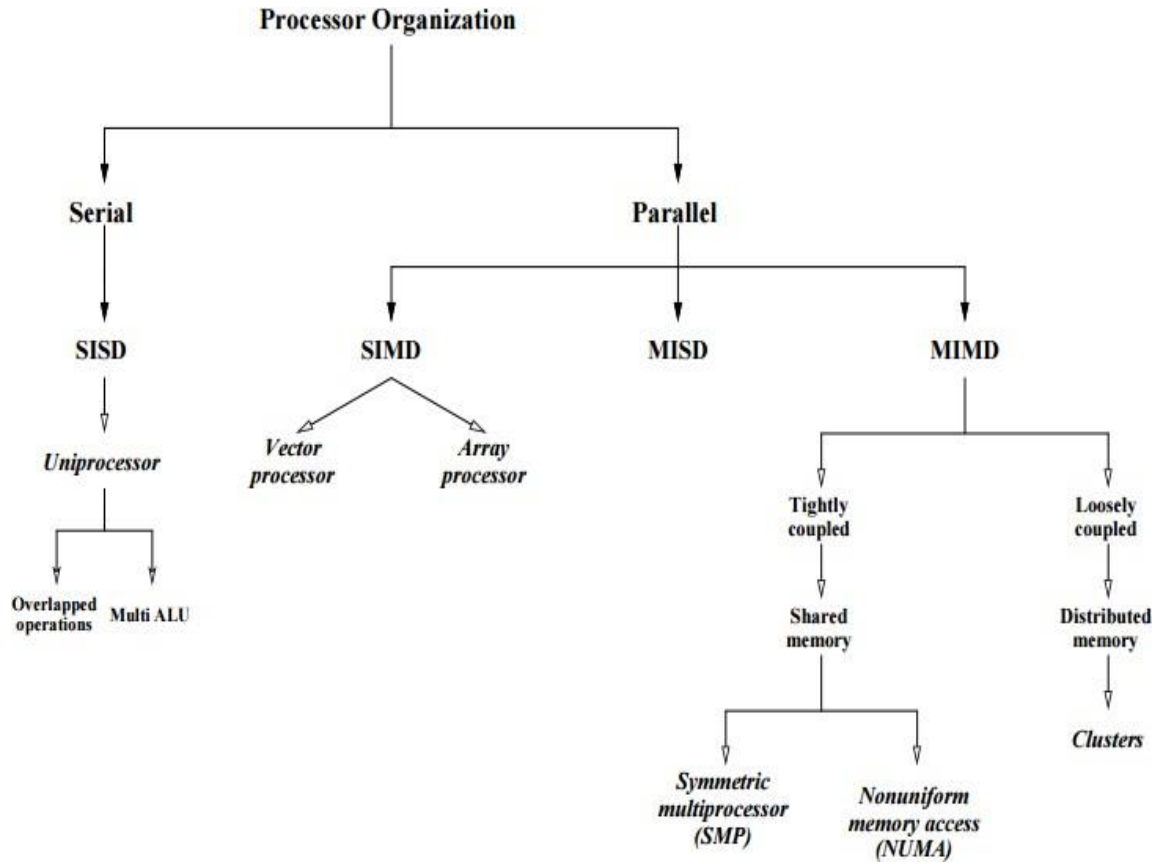


Fig. Taxonomy of mono- mulitporcessor organizations

Characteristics of Multiprocessors:

Benefits of Multiprocessing:

1. Multiprocessing increases the reliability of the system so that a failure or error in one part has limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled one.

2. Improved System performance. System derives high performance from the fact that computations can proceed in parallel in one of the two ways:

- a) Multiple independent jobs can be made to operate in parallel.
- b) A single job can be partitioned into multiple parallel tasks. This can

be achieved in two ways:

- The user explicitly declares that the tasks of the program be executed in parallel

- The compiler provided with multiprocessor s/w that can automatically detect parallelism in program. Actually it checks for Data dependency

Coupling of Processors

Tightly Coupled System/Shared Memory:

- Tasks and/or processors communicate in a highly synchronized fashion
- Communicates through a common global shared memory
- Shared memory system. This doesn't preclude each processor from having its own local memory(cache memory)

Loosely Coupled System/Distributed Memory

- Tasks or processors do not communicate in a synchronized fashion.
- Communicates by message passing packets consisting of an address, the data content, and some error detection code.
- Overhead for data exchange is high
- Distributed memory system

Loosely coupled systems are more efficient when the interaction between tasks is minimal, whereas tightly coupled system can tolerate a higher degree of interaction between tasks.

Shared (Global) Memory

- A Global Memory Space accessible by all processors
- Processors may also have some local memory

Distributed (Local, Message-Passing) Memory

- All memory units are associated with processors
- To retrieve information from another processor's memory a message must be sent there

Uniform Memory

- All processors take the same time to reach all memory locations Non-

uniform (NUMA) Memory

- Memory access is not uniform

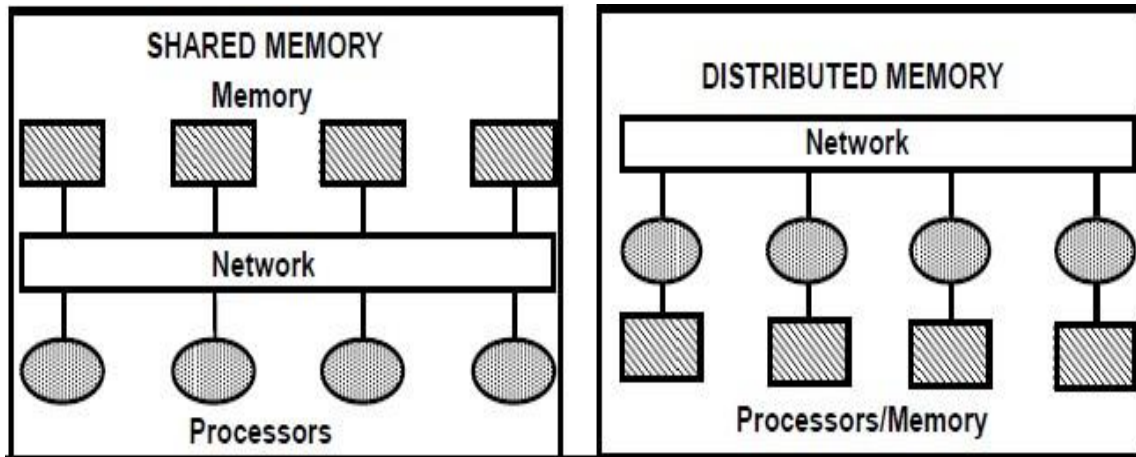


Fig. Shared and distributed memory Shared

memory multiprocessor:

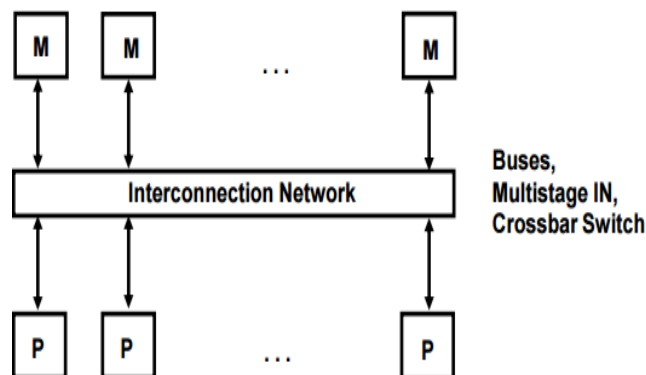


Fig Shared memory multiprocessor

Characteristics

- All processors have equally direct access to one large memory address space

Limitations

- Memory access latency; Hot spot problem

5.2 Interconnection Structures:

The interconnection between the components of a multiprocessor System can have different physical configurations depending on the number of transfer paths that are available between the processors and memory in a shared memory system and among the processing elements in a loosely coupled system.

Some of the schemes are as:

- Time-Shared Common Bus
- Multiport Memory
- Crossbar Switch
- Multistage Switching Network
- Hypercube System

a. Time shared common Bus

- All processors (and memory) are connected to a common bus or busses
- Memory access is fairly uniform, but not very scalable
- A collection of signal lines that carry module-to-module communication
- Data highways connecting several digital system elements
- Operations of Bus

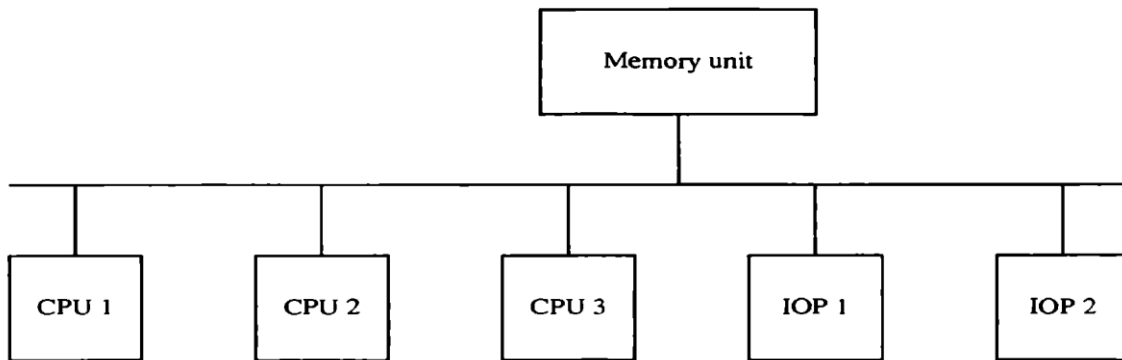


Fig. 5.5 Time shared common bus organization

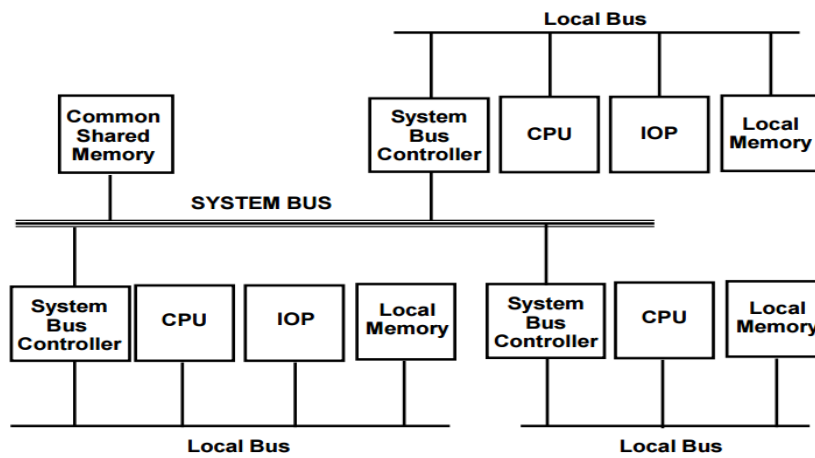


Fig. system bus structure for multiprocessor

In the above figure we have number of local buses to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combinations of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus the I/O devices attached to it may be made available to all processors

Disadvantage.:

- Only one processor can communicate with the memory or another processor at any given time.
- As a consequence, the total overall transfer rate within the system is limited by the speed of the single path

b. Multiport Memory:

Multiport Memory Module

- Each port serves a CPU

Memory Module Control Logic

- Each memory module has control logic
- Resolve memory module conflicts Fixed priority among CPUs

Advantages

- The high transfer rate can be achieved because of the multiple paths.

Disadvantages:

- It requires expensive memory control logic and a large number of cables and connections

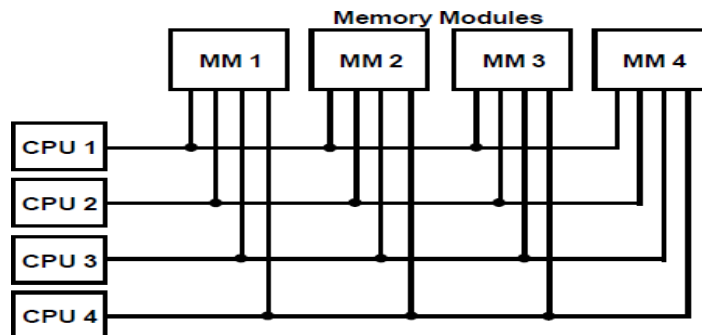


Fig. Multiport memory

c. Crossbar switch:

- Each switch point has control logic to set up the transfer path between a processor and a memory.
- It also resolves the multiple requests for access to the same memory on the predetermined priority basis.
- Though this organization supports simultaneous transfers from all memory modules because there is a separate path associated with each Module.
- The H/w required to implement the switch can become quite large and complex

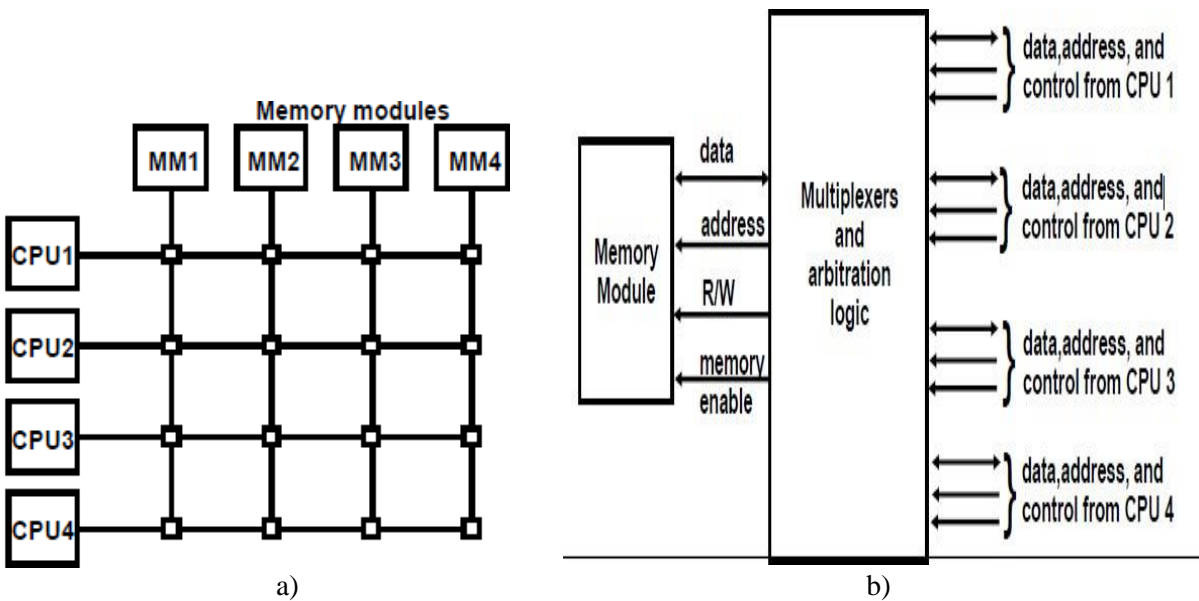


Fig. a) cross bar switch

b) Block diagram of cross bar switch

Advantage:

- Supports simultaneous transfers from all memory modules

Disadvantage:

- The hardware required to implement the switch can become quite large and complex.

d. Multistage Switching Network:

- The basic component of a multi stage switching network is a two-input, two- output interchange switch.

Interstage Switch

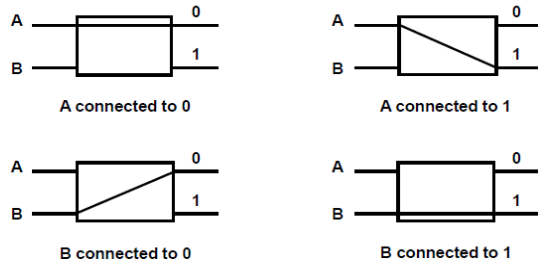


Fig. 5.9 operation of 2X2 interconnection switch

Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations.

- To see how this is done, consider the binary tree shown in Fig. below.
- Certain request patterns cannot be satisfied simultaneously. i.e., if P1 000~011, then P2 100~111

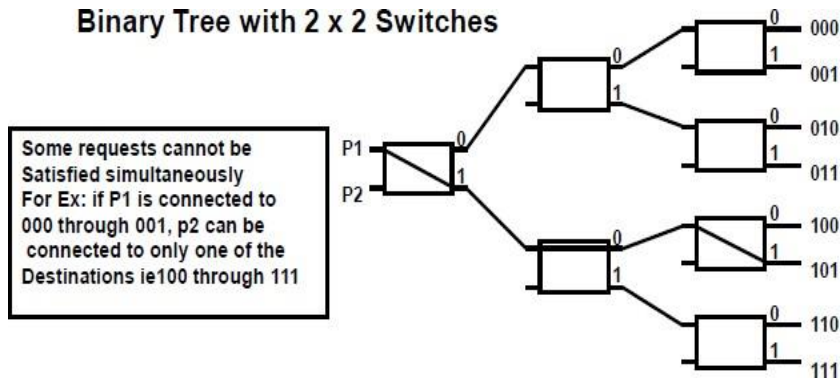


Fig 5.10 Binary tree with 2x2 switches

8x8 Omega Switching Network

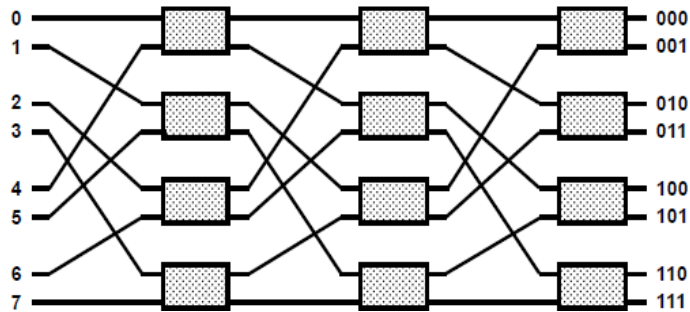


Fig. 8X8 Omega switching network

- Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001
- In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module.
- Set up the path-> transfer the address into memory->transfer the data
- In a loosely coupled multiprocessor system, both the source and destination are Processing elements.

e. Hypercube System:

The hypercube or binary n -cube multiprocessor structure is a loosely coupled system composed of $N=2^n$ processors interconnected in an n -dimensional binary cube.

- Each processor forms a node of the cube, in effect it contains not only a CPU but also local memory and I/O interface.
- Each processor address differs from that of each of its n neighbors by exactly one bit position.
- Fig. below shows the hypercube structure for $n=1, 2,$ and 3 .
- Routing messages through an n -cube structure may take from one to n links from a source node to a destination node.
- A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
- The message is then sent along any one of the axes that the resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ.
- A representative of the hypercube architecture is the Intel iPSC computer complex.
- It consists of $128(n=7)$ microcomputers, each node consists of a CPU, a floating point processor, local memory, and serial communication interface units.

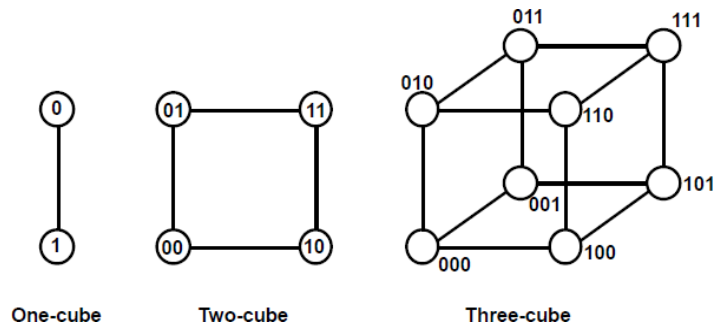
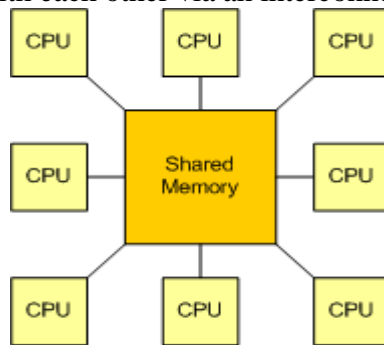


Fig. Hypercube structures for n=1,2,3

Multicomputer:

A multicomputer system is a computer system with multiple processors that are connected together to solve a problem. Each processor has its own memory and it is accessible by that particular processor and those processors can communicate with each other via an interconnection network.



As the multicomputer is capable of messages passing between the processors, it is possible to divide the task between the processors to complete the task. Hence, a multicomputer can be used for distributed computing. It is cost effective and easier to build a multicomputer than a multiprocessor.

BASIS FOR COMPARISON	MULTIPROCESSOR	MULTICOMPUTER
Basic	Multiple processors in a single computer.	Interlinked multiple autonomous computers.
Memory attached to the processing elements	Single shared	Multiple distributed
Communication between processing elements	Mandatory	Not necessary
Type of network	Dynamic network	Static network
Example	Sequent symmetry S-81	Message passing multicomputer

UNIT-V

I/O Systems

I/O Hardware

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. A device communicates with a computer system by sending signals over a cable or even through the air.

Port: The device communicates with the machine via a connection point (or port), for example, a serial port.

Bus: If one or more devices use a common set of wires, the connection is called a bus.

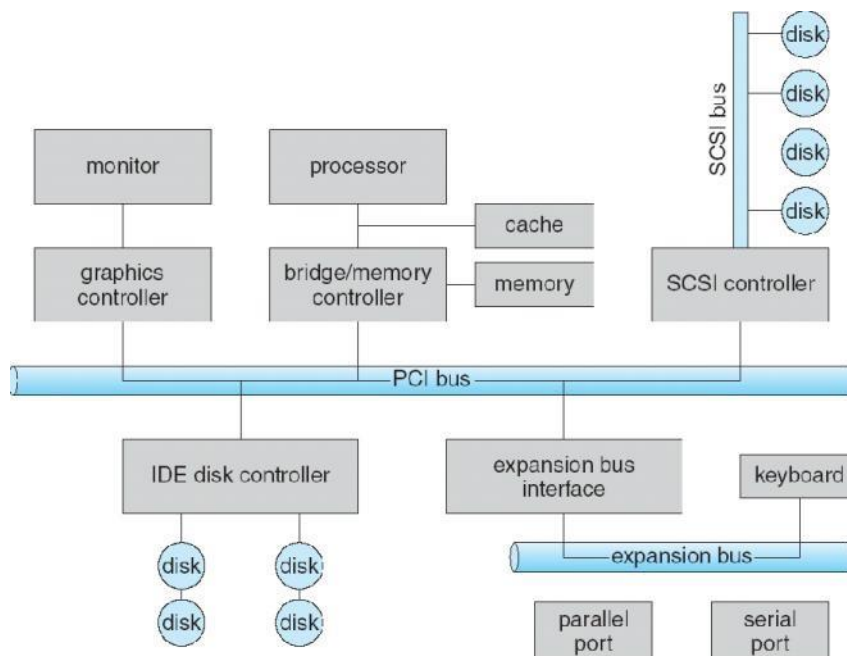
Daisy chain: Device _A_ has a cable that plugs into device _B_, and device _B_ has a cable that plugs into device _C_, and device _C_ plugs into a port on the computer, this arrangement is called a daisy chain. A daisy chain usually operates as a bus.

PC bus structure:

A PCI bus that connects the processor-memory subsystem to the fast devices, and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

A **controller or host adapter** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI

protocol is complex, the SCSI bus controller is often implemented as a separate circuit board. It typically contains a processor, microcode, and some private memory. Some devices have their own built-in controllers.



- How can the processor give commands and data to a controller to accomplish an I/O transfer?
 - Direct I/O instructions
 - Memory-mapped I/O

Direct I/O instructions

Use special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register

Memory-mapped I/O

The device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

An I/O port typically consists of four registers: status, control, data-in, and data-out registers.

Status register	Read by the host to indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error.
Control register	Written by the host to start a command or to change the mode of a device.
data-in register	Read by the host to get input
data-out register	Written by the host to send output

Polling

Interaction between the host and a controller

□ The controller sets the busy bit when it is busy working, and clears the busy bit when it is ready to accept the next command.

The host sets the command ready bit when a command is available for the controller to execute

Coordination between the host & the controller is done by handshaking as follows:

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.
5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte, and does the I/O to the device.
6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

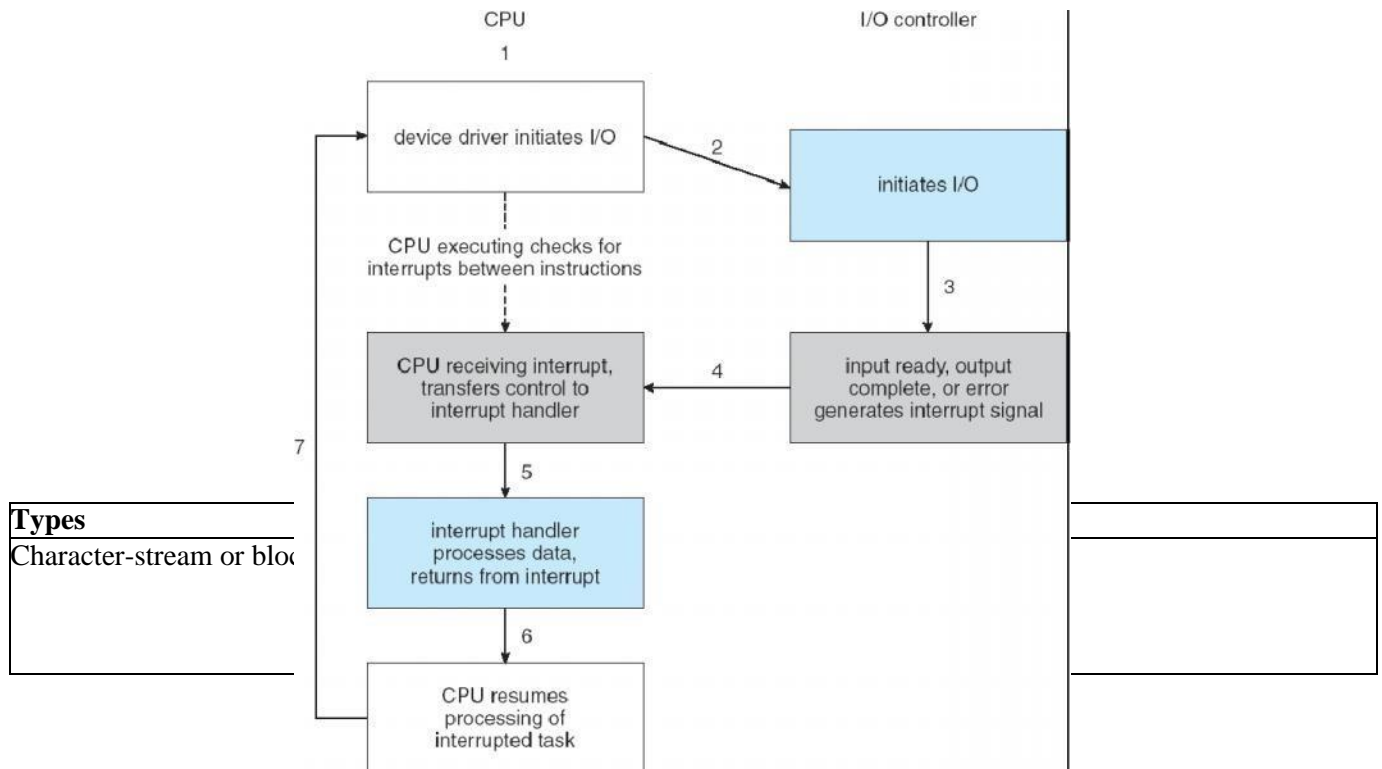
In step 1, the host is —**busy-waiting or polling**! It is in a loop, reading the status register over and over until the busy bit becomes clear.

Interrupts

The CPU hardware has a wire called the —interrupt-request line!.

The basic interrupt mechanism works as follows;

1. Device controller raises an interrupt by asserting a signal on the interrupt request line.
2. The CPU catches the interrupt and dispatches to the interrupt handler and
3. The handler clears the interrupt by servicing the device. Two interrupt request lines:
 1. **Nonmaskable interrupt**: which is reserved for events such as unrecoverable memory errors?
 2. **Maskable interrupt**: Used by device controllers to request service



Types
Character-stream or block

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device -driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
 1. Character-stream or block
 2. Sequential or random-access
 3. Sharable or dedicated
 4. Speed of operation
 5. read-write, read only, or write only

Block and Character Devices

Block-device: The block-device interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device should understand the commands such as read () & write (), and if it is a random access device, it has a seek() command to specify which block to transfer next.

Sequential or random-access	A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.	Modem, CD-ROM
Sharable or dedicated	A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.	Tape, Keyboard
Speed of operation	Latency, seek time, transfer rate, delay between operations	
read-write, read only, or write only	Some devices perform both input and output, but others support only one data direction.	CD-ROM, Graphics controller, Disk

Applications normally access such a device through a file-system interface. The OS itself, and special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**.

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory.

Character Devices: A keyboard is an example of a device that is accessed through a character stream interface. The basic system calls in this interface enable an application to get() or put() one character.

On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services.

(+) This style of access is convenient for input devices where it produce input "spontaneously".

(+) This access style is also good for output devices such as printers or audio boards, which naturally fit the concept of a linear stream of bytes.

Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the read() - write() - seek() interface used for disks.

□ Windows NT provides one interface to the network interface card, and a second interface to the network protocols.

□ In UNIX, we find half -duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues and sockets.

Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

1. Give the current time
2. Give the elapsed time
3. Set a timer to trigger operation X at time T

These functions are used by the operating system & also by time sensitive applications. Programmable interval timer: The hardware to measure elapsed time and to trigger operations is called a programmable interval timer. It can be set to wait a certain amount of time and then to generate an interrupt. To generate periodic interrupts, it can be set to do this operation once or to repeat.

Blocking and Non-blocking I/O (or) synchronous & asynchronous:

Blocking I/O: When an application issues a blocking system call;

- The execution of the application is suspended.
- The application is moved from the operating system's run queue to a wait queue.
- After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call.

Non-blocking I/O: Some user-level processes need non-blocking I/O.

Examples: 1. User interface that receives keyboard and mouse input while processing and displaying data on the screen.

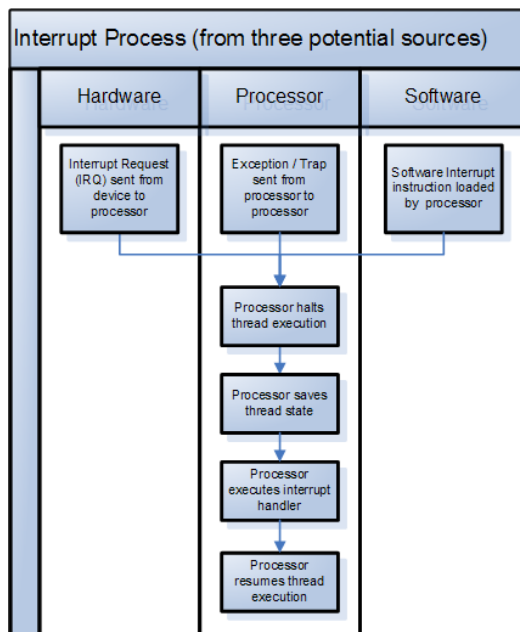
2. Video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.

Kernel I/O Subsystem

Kernels provide many services related to I/O.

- One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations.
- Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

Principles of I/O Software



Goals of the I/O Software

- A key concept in the design of I/O software is known as **device independence**. It means that I/O devices should be accessible to programs without specifying the device in advance.
- **Uniform Naming**, simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file-system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device.
- **Error Handling:** If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

- **Synchronous (blocking) and Asynchronous (interrupt-driven) transfers:** Most physical I/O is asynchronous, however, some very high-performance applications need to control all the details of the I/O, so some operating systems make asynchronous I/O available to them.
- **Buffering:** Often data that come off a device cannot be stored directly in their final destination.
- **Sharable and Dedicated devices:** Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished. Then another user can have the printer. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

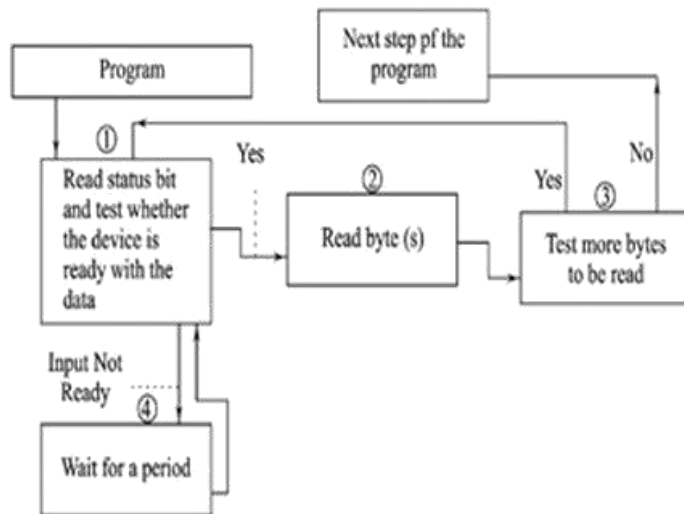
Programmed I/O

This is one of the three fundamentally different ways that I/O can be performed. The programmed I/O was the most simple type of I/O technique for the exchanges of data or any types of communication between the processor and the external devices. With programmed I/O, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time. The overall operation of the programmed I/O can be summaries as follow:

- The processor is executing a program and encounters an instruction relating to I/O operation.
- The processor then executes that instruction by issuing a command to the appropriate I/O module.
- The I/O module will perform the requested action based on the I/O command issued by the processor (READ/WRITE) and set the appropriate bits in the I/O status register.
- The processor will periodically check the status of the I/O module until it find that the operation is complete.

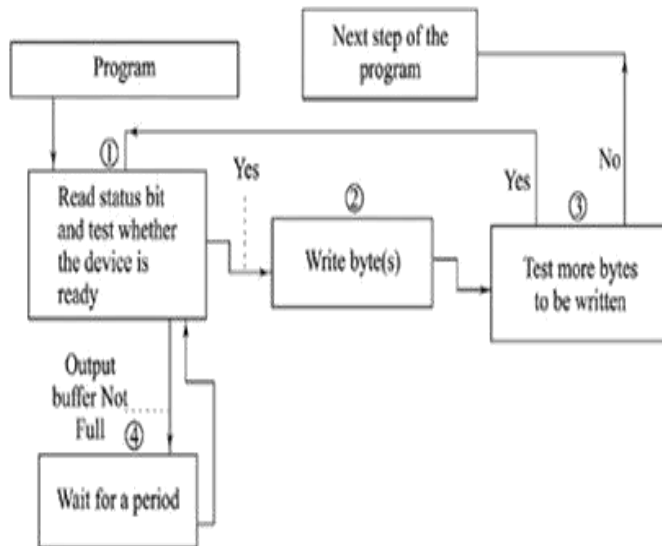
Programmed I/O Mode: Input Data Transfer

- Each input is read after first testing whether the device is ready with the input (a state reflected by a bit in a status register).
- The program waits for the ready status by repeatedly testing the status bit and till all targeted bytes are read from the input device.
- The program is in busy (non-waiting) state only after the device gets ready else in wait state.



Programmed I/O Mode: Output Data Transfer

- Each output written after first testing whether the device is ready to accept the byte at its output register or output buffer is empty.
- The program waits for the ready status by repeatedly testing the status bit(s) and till all the targeted bytes are written to the device.
- The program in busy (non-waiting) state only after the device gets ready else wait state.



Programmed I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:

- **Control**
- **Test**

- Read
- Write

```

copy_from_user(buffer, p, count);          /* p is the kernel bufer */
for (i = 0; i < count; i++) {             /* loop on every character */
    while (*printer_status_reg != READY); /* loop until ready */
    *printer_data_register = p[i];        /* output one character */
}
return_to_user();

```

Advantages of Programmed I/O

- Simple to implement
- Very little hardware support

Disadvantages of Programmed I/O

- Busy waiting
- Ties up CPU for long period with no useful work

Interrupt-Driven I/O

Interrupt driven I/O is an alternative scheme dealing with I/O. Interrupt I/O is a way of controlling input/output activity whereby a peripheral or terminal that needs to make or receive a data transfer sends a signal. This will cause a program interrupt to be set. At a time appropriate to the priority level of the I/O interrupt. Relative to the total interrupt system, the processors enter an interrupt service routine.

Interrupt I/O Inputs

For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports or memory mapping.

Interrupt I/O Outputs

For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

Operations in Interrupt I/O

- CPU issues read command.
- I/O module gets data from peripheral whilst CPU does other work.

- I/O module interrupts CPU.
- CPU requests data.
- I/O module transfers data.

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
```

(a)

```
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

(b)

Advantages of Interrupt-Driven I/O

- Its faster than Programmed I/O.
- Efficient too.

Disadvantages of Interrupt-Driven I/O

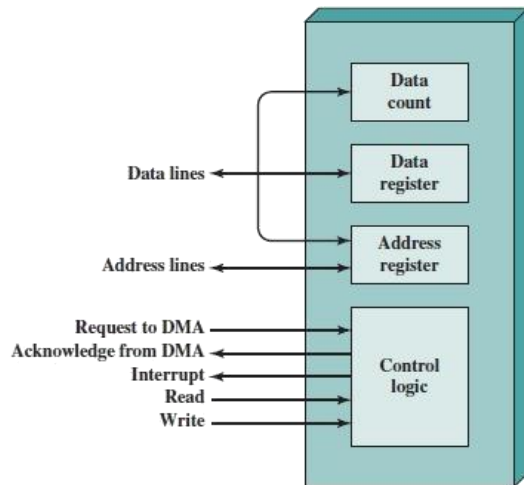
- It can be tricky to write if using a low level language.
- It can be tough to get various pieces to work well together.

I/O Using DMA

Direct Memory Access is a technique for transferring data within main memory and external device without passing it through the CPU. DMA is a way to improve processor activity and I/O transfer rate by taking-over the job of transferring data from processor, and letting the processor to do other tasks. This technique overcomes the drawbacks of other two I/O techniques which are the time consuming process when issuing a command for data transfer and tie-up the processor in data transfer while the data processing is neglected. It is more efficient to use DMA method when large volume of data has to be transferred. For DMA to be implemented, processor has to share its' system bus with the DMA module. Therefore, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily.

Operations of Direct Memory Access

- Read or Write Command
- Control Lines Communication
- Data Lines Communication



```
copy_from_user(buffer, p, count);
set_up_DMA_controller();
scheduler();
```

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Advantages of DMA

- Speed: no waiting due to much shorter execution path and no rotation delay.

File System Storage-File Concepts

File Concept

A file is a named collection of related information that is recorded on secondary storage.

□ From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.

Examples of files:

□ A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker.

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

File Attributes

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

File Operations

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file

Access Methods

1. Sequential Access

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

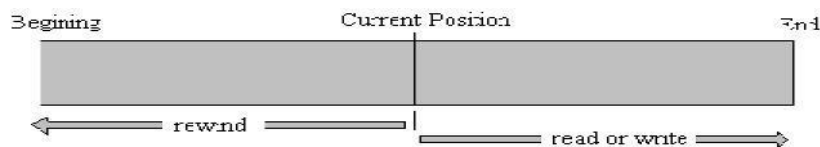


Fig 4.10 Sequential-access file

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file).

Such a file can be reset to the beginning and, on some systems, a program may be able to skip forward or back ward n records, for some integer n -perhaps only for $n=1$. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random – access ones.

2. Direct Access

Another method is direct access (or relative access). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct- access methods is based on a disk model of a file, since disks allow random access to any file block.

For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file. Direct – access files are of great use for immediate access to large amounts of information. Database is often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information

Directory and Disk Structure

There are five directory structures. They are

1. Single-level directory
2. Two-level directory
3. Tree-Structured directory
4. Acyclic Graph directory
5. General Graph directory

1. Single – Level Directory

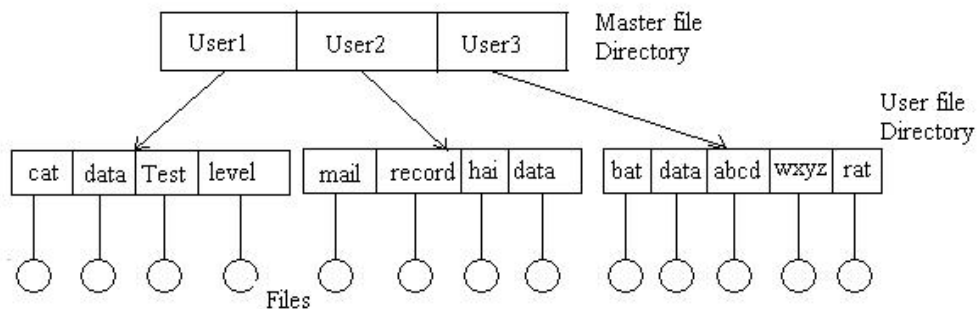
- The simplest directory structure is the single- level directory.
- All files are contained in the same directory.
- **Disadvantage:**
 - When the number of files increases or when the system has more than one user, since all files are in the same directory, they must have unique names.

2. Two – Level Directory

- In the two level directory structures, each user has her own user file directory (UFD).
- When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
- When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name.
- Although the two – level directory structure solves the name-collision problem

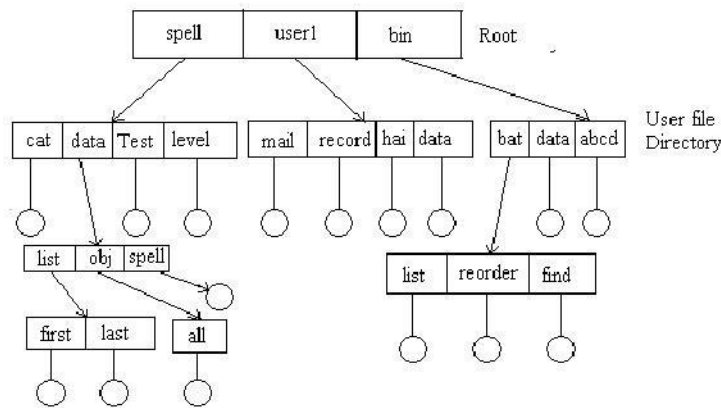
Disadvantage:

- Users cannot create their own sub-directories.



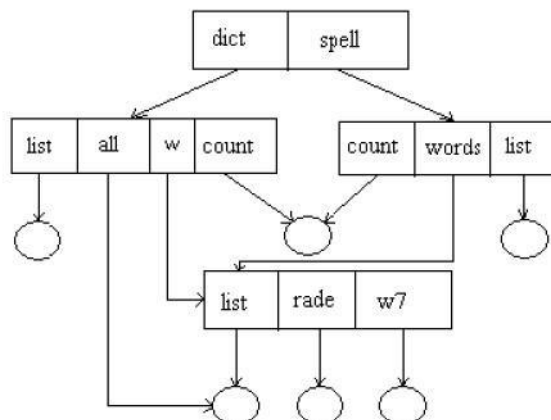
3. Tree – Structured Directory

- A tree is the most common directory structure.
- The tree has a root directory. Every file in the system has a unique path name.
- A **path name** is the path from the root, through all the subdirectories to a specified file.
- A directory (or sub directory) contains a set of files or sub directories.
- A directory is simply another file. But it is treated in a special way.
- All directories have the same internal format.
- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
- Special system calls are used to create and delete directories.
- Path names can be of two types: absolute path names or relative path names.
- An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A relative path name defines a path from the current directory .



4. Acyclic Graph Directory.

- An acyclic graph is a graph with no cycles.
- To implement shared files and subdirectories this directory structure is used.
- An acyclic – graph directory structure is more flexible than is a simple tree structure, but it is also more complex. In a system where sharing is implemented by symbolic link, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed.
- Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.



Sharing and

Protection File

Sharing

1. Multiple Users:

□ When an operating system accommodates multiple users, the issues of file sharing, file naming and file protection become preeminent.

□ The system either can allow user to access the file of other users by default, or it may require that a user specifically grant access to the files.

□ These are the issues of access control and protection.

□ To implementing sharing and protection, the system must maintain more file and directory attributes than a on a single-user system.

□ The owner is the user who may change attributes, grand access, and has the most control over the file or directory.

□ The group attribute of a file is used to define a subset of users who may share access to the file.

□ Most systems implement owner attributes by managing a list of user names and associated user identifiers (user Ids).

□ When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of user's processes and threads. When they need to be user readable, they are translated, back to the user name via the user name list. Likewise, group functionality can be implemented as a system wide list of group names and group identifiers.

□ Every user can be in one or more groups, depending upon operating system design decisions. The user's group Ids is also included in every associated process and thread.

2. Remote File System:

□ Networks allowed communications between remote computers.

□ Networking allows the sharing or resource spread within a campus or even around the world.

□ User manually transfer files between machines via programs like **ftp**.

□ A **distributed file system** (DFS) in which remote directories is visible from the local machine.

□ The **World Wide Web**: A browser is needed to gain access to the remote file and separate operations (essentially a wrapper for ftp) are used to transfer files.

a) The client-server Model:

□ Remote file systems allow a computer to a mount one or more file systems from one or more remote machines.

□ A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client –server facility.

□ Client identification is more difficult. Clients can be specified by their network name or other identifier, such as IP address, but these can be spoofed (or imitate). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access.

b) Distributed Information systems:

□ Distributed information systems, also known as distributed naming service, have been devised to provide a unified access to the information needed for remote computing.

□ Domain name system (DNS) provides host -name-to-network address translations for their entire Internet (including the World Wide Web).

□ Before DNS was invented and became widespread, files containing the same information were sent via e-mail of ftp between all networked hosts.

c) Failure Modes:

□ **Redundant arrays of inexpensive disks (RAID)** can prevent the loss of a disk from resulting in the loss of data.

□ Remote file system has more failure modes. By nature of the complexity of networking system and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.

d) Consistency Semantics:

□ It is characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously.

□ These semantics should specify when modifications of data by one user are observable by other users.

□ The semantics are typically implemented as code with the file system.

□ A series of file accesses (that is reads and writes) attempted by a user to the same file is always enclosed between the open and close operations.

□ The series of access between the open and close operations is a **file session**.

(i) UNIX Semantics:

The UNIX file system uses the following consistency semantics:

1. Writes to an open file by a user are visible immediately to other users that have this file open at the same time.
2. One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users.

(ii) Session Semantics:

The Andrew file system (AFS) uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.
2. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect this change.

(iii) Immutable –shared File Semantics:

□ Once a file is declared as shared by its creator, it cannot be modified.

□ An immutable file has two key properties:

□ Its name may not be reused and its contents may not be altered.

File Protection

(i) Need for file protection.

□ When information is kept in a computer system, we want to keep it safe from **physical damage** (reliability) and **improper access** (protection).

□ Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

□ File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

□ Protection can be provided in many ways. For a small single -user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

(ii) Types of Access

- Complete protection is provided by prohibiting access.
- Free access is provided with no protection.
- Both approaches are too extreme for general use.
- What is needed is **controlled access**.
- Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:
 1. **Read:** Read from the file.
 2. **Write:** Write or rewrite the file.
 3. **Execute:** Load the file into memory and execute it.
 4. **Append:** Write new information at the end of the file.
 5. **Delete:** Delete the file and free its space for possible reuse.
 6. **List:** List the name and attributes of the file.

(iii) Access Control

- Associate with each file and directory an access -control list (ACL) specifying the user name and the types of access allowed for each user.
- When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs and the user job is denied access to the file.
- This technique has two undesirable consequences:
 - Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
 - The directory entry, previously of fixed size, now needs to be of variable size, resulting in more complicated space management.
- To condense the length of the access control list, many systems recognize three classifications of users in connection with each file:
 - **Owner:** The user who created the file is the owner.
 - **Group:** A set of users who are sharing the file and need similar access is a group, or work group.
 - **Universe:** All other users in the system constitute the universe.

File System Implementation- File System Structure

- **Disk** provide the bulk of secondary storage on which a file system is maintained.

□ Characteristics of a disk:

1. They can be rewritten in place, it is possible to read a block from the disk, to modify the block and to write it back into the same place.
 2. They can access directly any given block of information to the disk.
- To produce an efficient and convenient access to the disk, the operating system imposes one or more file system to allow the data to be stored, located and retrieved easily.
 - The file system itself is generally composed of many different levels. Each level in the design uses the features of lower level to create new features for use by higher levels.

Layered File System

- The **I/O control** consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system .

□ The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive -1, cylinder 73, track 2, sector 10)

Directory Implementation

1. Linear List

□ The simplest method of implementing a directory is to use a linear list of file names with pointer to the data blocks.

□ A linear list of directory entries requires a linear search to find a particular entry.

□ This method is simple to program but time-consuming to execute. To create a new file, we must first search the but time – consuming to execute.

□ The real disadvantage of a linear list of directory entries is the linear search to find a file.

2. Hash Table

□ In this method, a linear list stores the directory entries, but a hash data structure is also used.

□ The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.

□ Therefore, it can greatly decrease the directory search time.

□ Insertion and deleting are also fairly straight forward, although some provision must be made for collisions – situation where two file names hash to the same location.

□ The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

Allocation Methods

□ The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.

□ There are three major methods of allocating disk space:

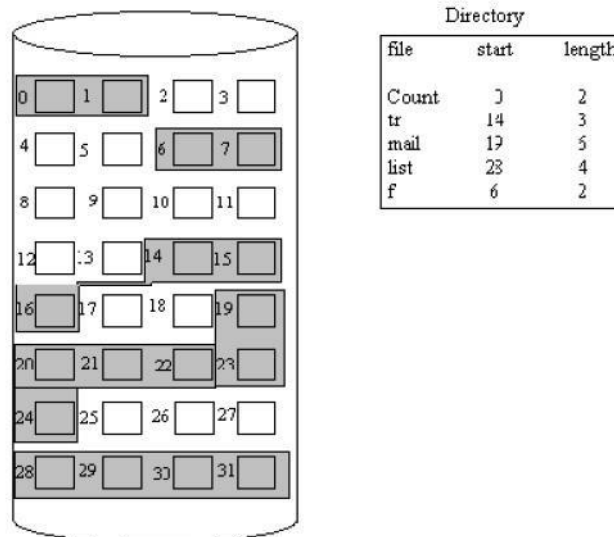
1. Contiguous Allocation

2. Linked Allocation

3. Indexed Allocation

1. Contiguous Allocation

□ The contiguous – allocation method requires each file to occupy a set of contiguous blocks on the disk.



Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b , then it occupies blocks $b, b+1, b+2, \dots, b+n-1$.

- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Disadvantages:

1. Finding space for a new file.

- The contiguous disk space allocation problem suffers from the problem of external fragmentation. As files are allocated and deleted, the free disk space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data.

2. Determining how much space is needed for a file.

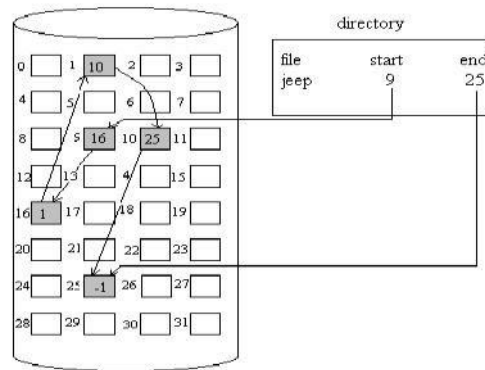
- When the file is created, the total amount of space it will need must be found and allocated. How does the creator know the size of the file to be created?
 - If we allocate too little space to a file, we may find that file cannot be extended. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions may be repeated as long as space exists, although it can be time-consuming. However, in this case, the user never needs to be informed explicitly about what is happening; the system continues despite the problem, although more and more slowly.
 - Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient.
 - A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time the file, therefore has a large amount of internal fragmentation.

To overcome these disadvantages:

- Use a modified contiguous allocation scheme, in which a contiguous chunk of space called as an **extent** is allocated initially and then, when that amount is not large enough another chunk of contiguous space an extent is added to the initial allocation.
- Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes are allocated and deallocated.

2. Linked Allocation

- Linked allocation solves all problems of contiguous allocation.
- With linked allocation, each file is a linked list of disk blocks, the disk blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25.
 - Each block contains a pointer to the next block. These pointers are not made available to the user.
 - There is no external fragmentation with linked allocation, and any free block on the free space list can be used to satisfy a request.
 - The size of a file does not need to be declared when that file is created. A file can continue to grow as long as free blocks are available consequently, it is never necessary to compact disk space.



Disadvantages:

1. Used effectively only for sequential access files.

□ To find the *i*th block of a file, we must start at the beginning of that file, and follow the pointers until we get to the *i*th block. Each access to a pointer requires a disk read, and sometimes a disk seek consequently, it is inefficient to support a direct-access capability for linked allocation files.

2. Space required for the pointers

□ If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

□ Solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate the clusters rather than blocks. For instance, the file system may define a cluster as 4 blocks, and operate on the disk in only cluster units.

3. Reliability

□ Since the files are linked together by pointers scattered all over the disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file. Partial solutions are to use doubly linked lists or to store the file names in a relative block number in each block; however, these schemes require even more overhead for each file.

File Allocation Table(FAT)

□ An important variation on the linked allocation method is the use of a file allocation table(FAT).

□ This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems.

□ A section of disk at beginning of each partition is set aside to contain the table.

□ The table has entry for each disk block, and is indexed by block number.

□ The FAT is much as is a linked list.

□ The directory entry contains the block number the first block of the file.

□ The table entry indexed by that block number contains the block number of the next block in the file.

□ This chain continues until the last block which has a special end-of-file value as the table entry.

□ Unused blocks are indicated by a 0 table value.

□ Allocating a new block file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block.

□ The 0 is replaced with the end-of-file value, an illustrative example is the FAT structure for a file consisting of disk blocks 217,618, and 339.

3. Indexed Allocation

□ Linked allocation solves the external-fragmentation and size-declaration problems of

contiguous allocation.

□ Linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order.

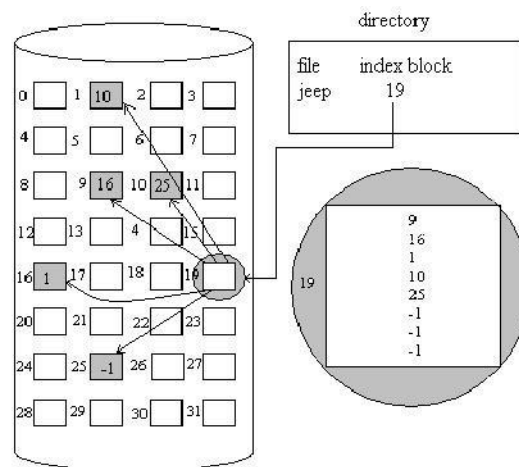
□ Indexed allocation solves this problem by bringing all the pointers together into one location: the **index block**.

□ Each file has its own index block, which is an array of disk – block addresses.

□ The *i*th entry in the index block points to the *i*th block of the file.

□ The directory contains the address of the index block .

□ To read the *i*th block, we use the pointer in the *i*th index – block entry to find and read the desired block this scheme is similar to the paging scheme .



□ When the file is created, all pointers in the pointers in the index block are set to nil. when the *i*th block is first written, a block is obtained from the free space manager, and its address is put in the *i*th index – block entry.

□ Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Disadvantages

1.Pointer

Overhead

□ Indexed allocation does suffer from wasted space. The pointer over head of the index block is generally greater than the pointer over head of linked allocation.

2. Size of Index block

If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue:

□ **Linked Scheme:** An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks.

□ **Multilevel index:** A variant of the linked representation is to use a first level index block to point to a set of second – level index blocks.

□ **Combined scheme:**

o Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode.

o The first 12 of these pointers point to direct blocks; that is for small (no more than 12 blocks) files do not need a separate index block

o The next pointer is the address of a single indirect block.

- The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data.
- o Then there is a double indirect block pointer, which contains the address of a block that contain pointers to the actual data blocks. The last pointer would contain pointers to the actual data blocks.
- o The last pointer would contain the address of a triple indirect block.

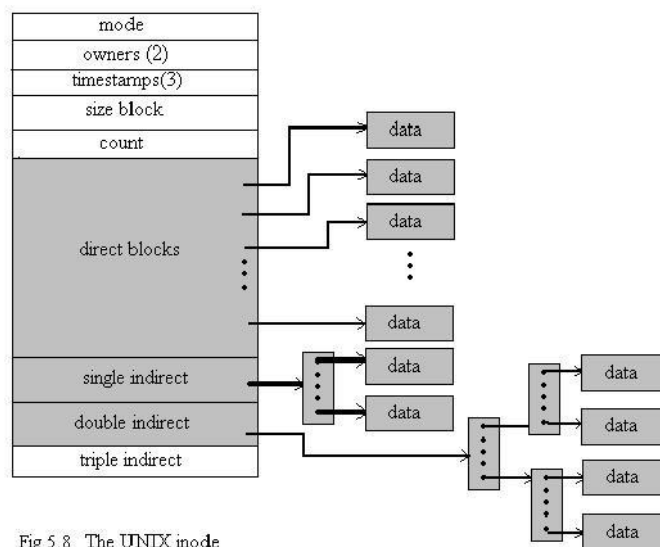


Fig 5.8 The UNIX inode

Free-space Management

- Since disk space is limited, we need to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a free -space list.
- The free-space list records all free disk blocks – those not allocated to some file or directory.
- To create a file, we search the free -space list for the required amount of space, and allocate that space to the new file.
- This space is then removed from the free-space list.
- When a file is deleted, its disk space is added to the free-space list.

1. Bit Vector

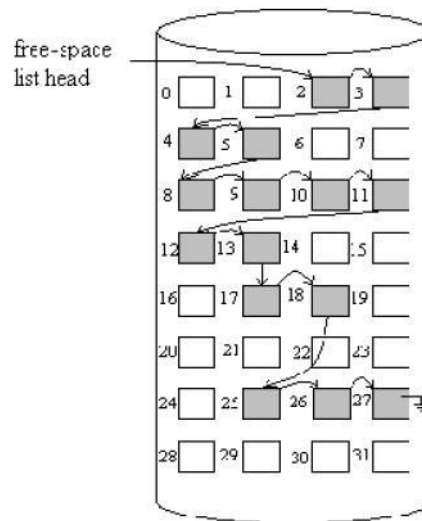
- The free -space list is implemented as a bit map or bit vector.
- Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where block 2,3,4,5,8,9,10,11,12,13,17,18,25,26 and 27 are free, and the rest of the block are allocated. The free space bit map would be

001111001111110001100000011100000 ...

- The main **advantage** of this approach is its relatively simplicity and efficiency in finding the first free block, or n consecutive free blocks on the disk.

2. Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on.
- In our example, we would keep a pointer to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.
- However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.
- The FAT method incorporates free-block accounting data structure. No separate method is needed.



3. Grouping

- A modification of the free-list approach is to store the addresses of n free blocks in the first free block.
- The first $n - 1$ of these blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on.
- The importance of this implementation is that the addresses of a large number of free blocks can be found quickly.

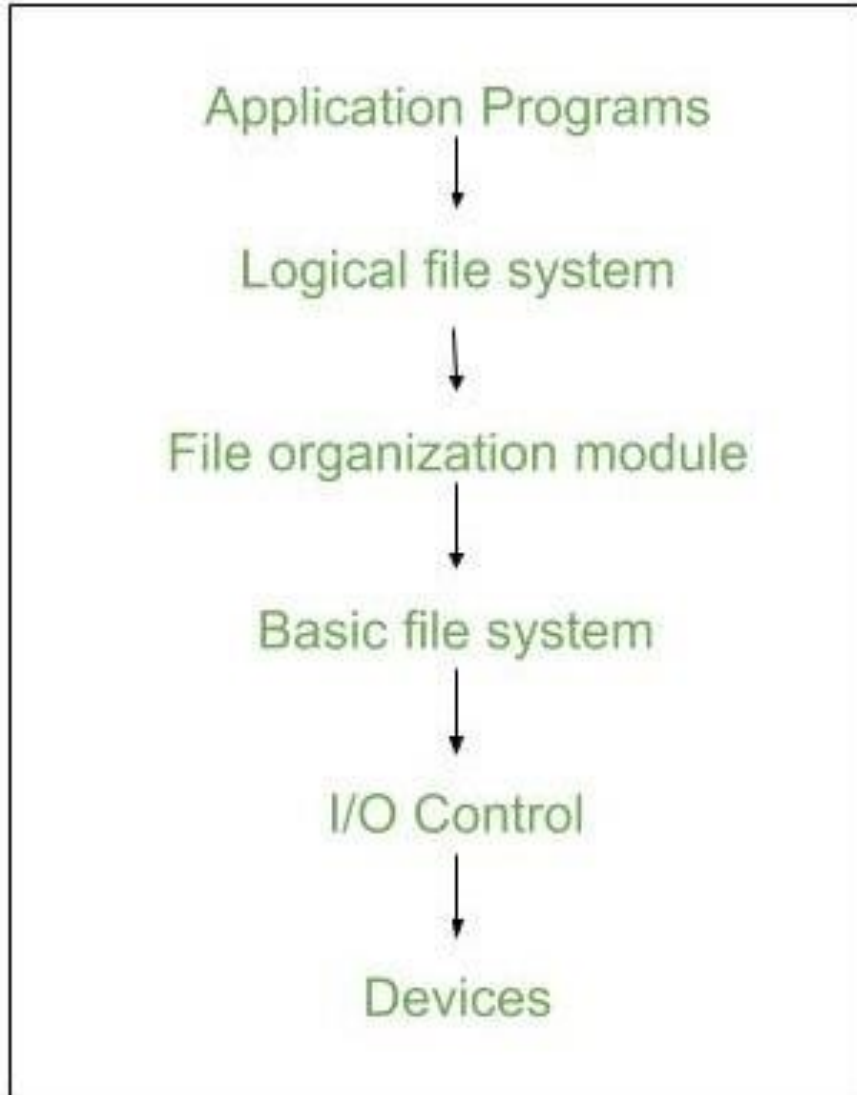
4. Counting

- We can keep the address of the first free block and the number n of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.
- Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

File System Implementation in Operating System

A file is a collection of related information. The file system resides on secondary storage and provides efficient and convenient access to the disk by allowing data to be stored, located, and retrieved.

File system organized in many layers :



- **I/O Control level –**

Device drivers acts as interface between devices and Os, they help to transfer data between disk and main memory. It takes block number a input and as output it gives low level hardware specific instruction.

/li>

- **Basic file system –**

It Issues general commands to device driver to read and write physical blocks on disk.It manages the memory buffers and caches. A block in buffer can hold the contents of the disk block and cache stores frequently used file system metadata.

- **File organization Module –**

It has information about files, location of files and their logical and physical blocks. Physical blocks do not match with logical numbers of logical block numbered from 0 to N. It also has a free space which tracks unallocated blocks.

- **Logical file system –**

It manages metadata information about a file i.e includes all details about a file except the actual contents of file. It also maintains via file control blocks. File control block (FCB) has information about a file – owner, size, permissions, location of file contents.

Advantages :

1. Duplication of code is minimized.
2. Each file system can have its own logical file system.

Disadvantages :

If we access many files at same time then it results in low performance.

We can **implement** file system by using two types data structures :

1. On-disk Structures –

Generally they contain information about total number of disk blocks, free disk blocks, location of them and etc.

Below given are different on-disk structures :

1. **Boot Control Block –**

It is usually the first block of volume and it contains information needed to boot an operating system. In UNIX it is called boot block and in NTFS it is called as partition boot sector.

2. **Volume Control Block –**

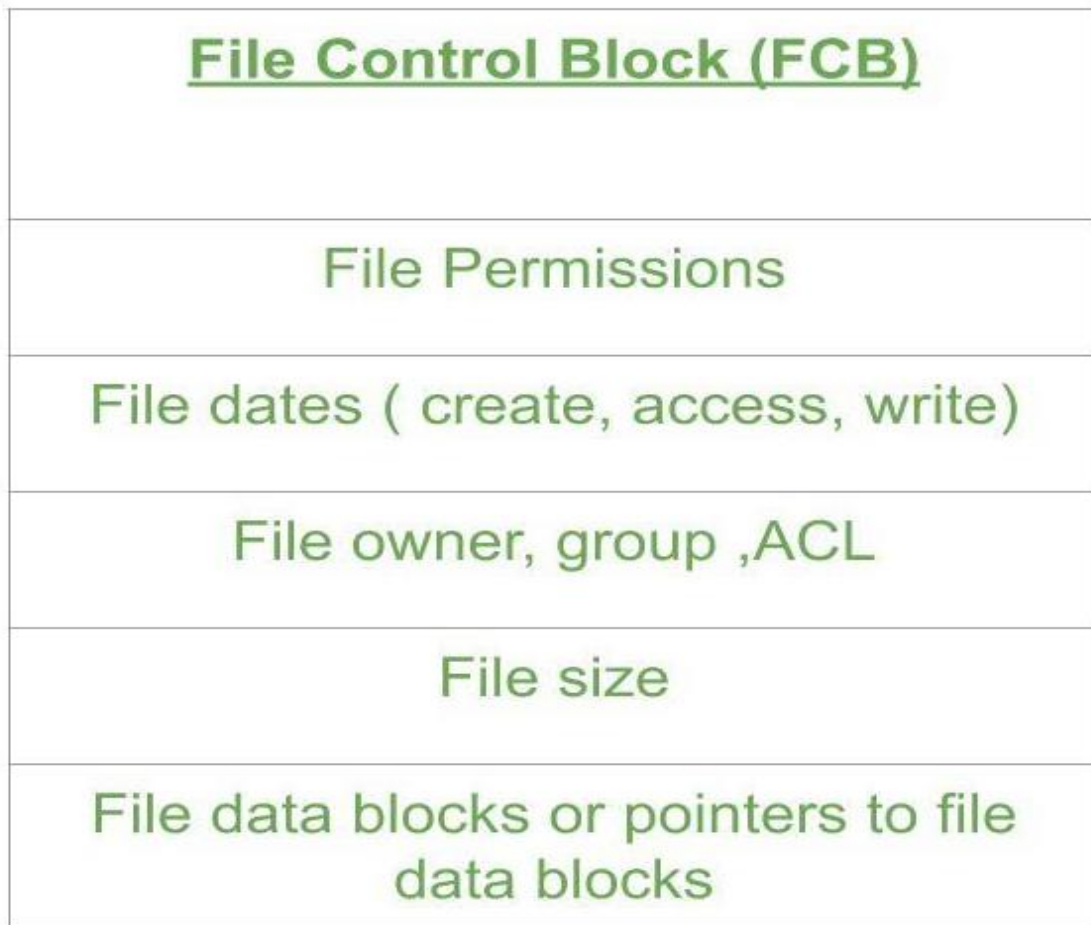
It has information about a particular partition ex:- free block count, block size and block pointers etc. In UNIX it is called super block and in NTFS it is stored in master file table.

3. **Directory Structure –**

They store file names and associated inode numbers. In UNIX, includes file names and associated file names and in NTFS, it is stored in master file table.

4. **Per-File FCB –**

It contains details about files and it has a unique identifier number to allow association with directory entry. In NTFS it is stored in master file table.



2. In-Memory Structure :

They are maintained in main-memory and these are helpful for file system management for caching. Several in-memory structures given below :

5. **Mount Table** –
It contains information about each mounted volume.
6. **Directory-Structure cache** –
This cache holds the directory information of recently accessed directories.
7. **System wide open-file table** –
It contains the copy of FCB of each open file.
8. **Per-process open-file table** –
It contains information opened by that particular process and it maps with appropriate system wide open-file.

Directory Implementation :

9. Linear List –

It maintains a linear list of filenames with pointers to the data blocks. It is time-consuming also. To create a new file, we must first search the directory to be sure that no existing file has the same name then we add a file at end of the directory. To delete a file, we search the directory for the named file and release the space. To reuse the directory entry either we can mark the entry as unused or we can attach it to a list of free directories.

10. Hash Table –

The hash table takes a value computed from the file name and returns a pointer to the file. It decreases the directory search time. The insertion and deletion process of files is easy. The major difficulty is hash tables are its generally fixed size and hash tables are dependent on hash function on that size.

File System Management and Optimization

Name	Size	Type:	File Folder
99998.txt	1 KB	Location:	C:\
99999.txt	1 KB	Size:	488 KB (500,059 bytes)
100000.txt	1 KB	Size on disk:	390 MB (409,608,192 bytes)
mkfile.bat	1 KB	Contains:	100,002 Files, 0 Folders
source.txt	1 KB		

Disk-Space Management

Since all the files are normally stored on disk one of the main concerns of file system is management of disk space.

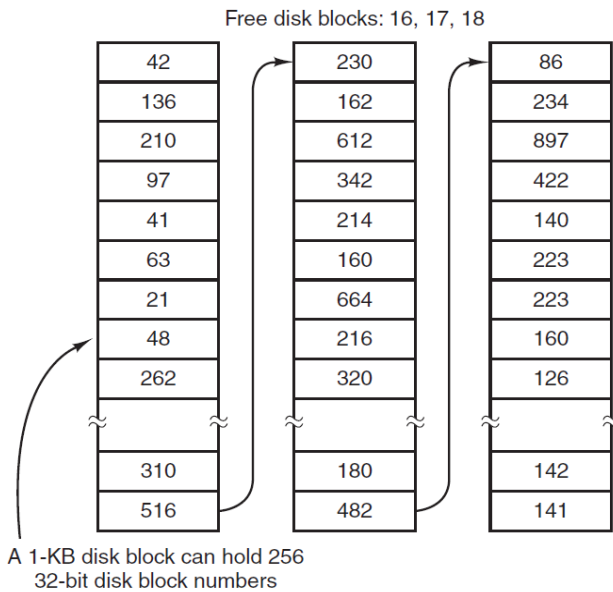
Block Size

The main question that arises while storing files in a fixed-size blocks is the size of the block. If the block is too large space gets wasted and if the block is too small time gets waste. So, to choose a correct block size some information about the file-size distribution is required. Performance and space-utilization are always in conflict.

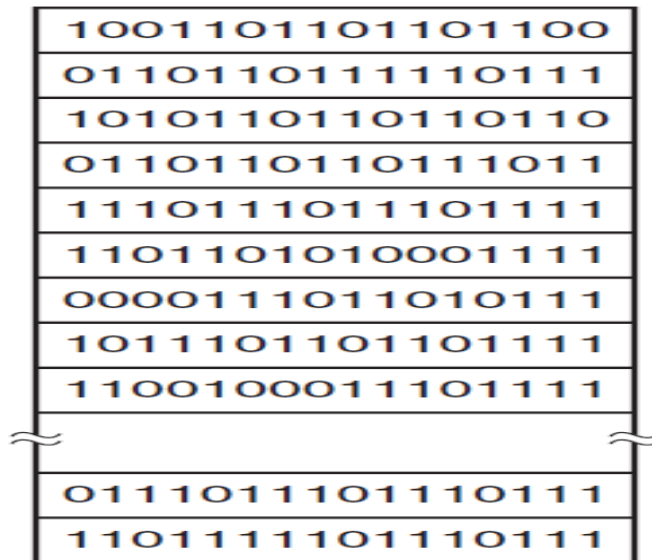
Keeping track of free blocks

After a block size has been finalized the next issue that needs to be catered is how to keep track of the free blocks. In order to keep track there are two methods that are widely used:

- Using a linked list: Using a linked list of disk blocks with each block holding as many free disk block numbers as will fit.



- Bitmap: A disk with n blocks has a bitmap with n bits. Free blocks are represented using 1's and allocated blocks as 0's as seen below in the figure.



Disk quotas

Multuser operating systems often provide a mechanism for enforcing disk quotas. A system administrator assigns each user a maximum allotment of files and blocks and the operating system makes sure that the users do not exceed their quotas. Quotas are kept track of on a per-user basis in a quota table.

File-system Backups

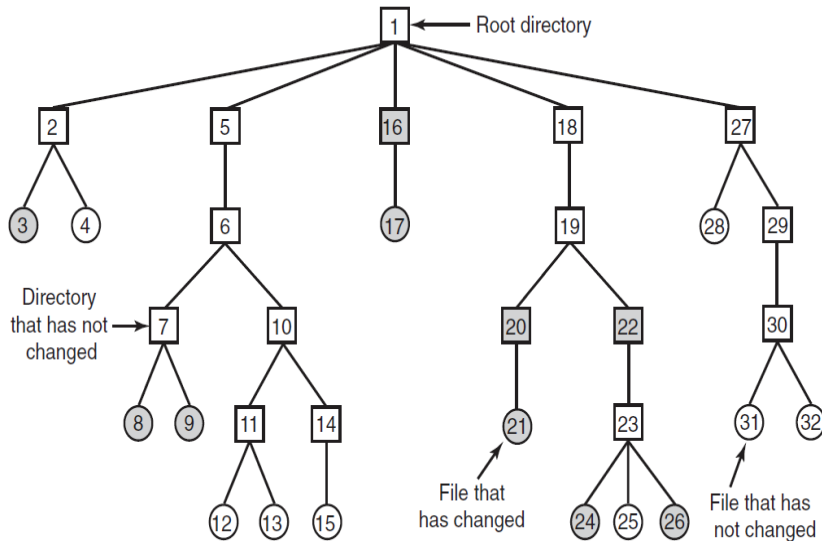
If a computer's file system is irrevocably lost, whether due to hardware or software restoring all the information will be difficult, time consuming and in many cases impossible. So it is advised to always have file-system backups.

Backing up files is time consuming and as well occupies large amount of space, so doing it efficiently and conveniently is important. Below are few points to be considered before creating backups for files.

- Is it required to backup the entire file system or only a part of it.
- Backing up files that haven't been changed from previous backup leads to **incremental dumps**. So it's better to take a backup of only those files which have changed from the time of previous backup. But recovery gets complicated in such cases.
- Since there is immense amount of data, it is generally desired to compress the data before taking a backup for the same.
- It is difficult to perform a backup on an active file-system since the backup may be inconsistent.
- Making backups introduces many security issues

There are two ways for dumping a disk to the backup disk:

- **Physical dump:** In this way dump starts at block 0 of the disk, writes all the disk blocks onto the output disk in order and stops after copying the last one.
Advantages: Simplicity and great speed.
Disadvantages: inability to skip selected directories, make incremental dumps, and restore individual files upon request
- **Logical dump:** In this way the dump starts at one or more specified directories and recursively dump all files and directories found that have been changed since some given base date. This is the most commonly used way.



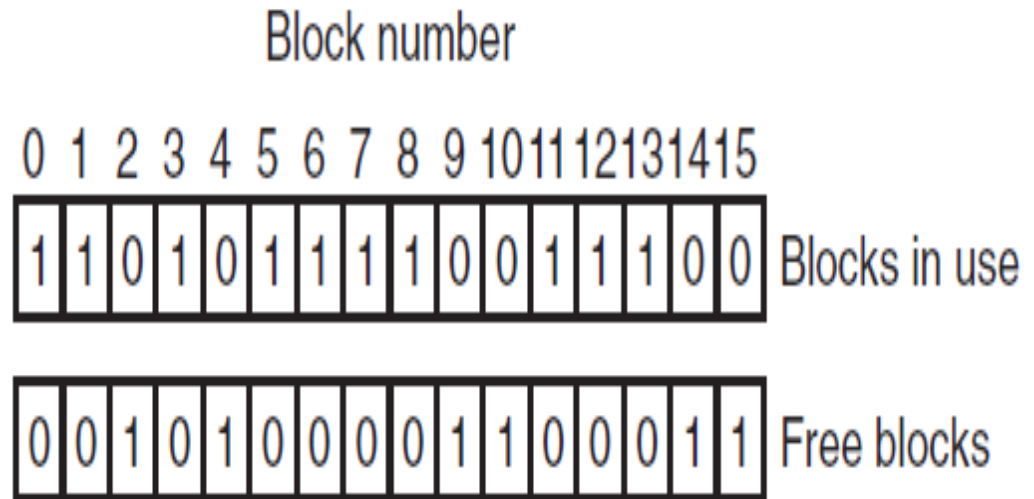
The above figure depicts a popular algorithm used in many UNIX systems wherein squares depict directories and circles depict files. This algorithm dumps all the files and directories that have been modified and also the ones on the path to a modified file or directory. The dump algorithm maintains a bitmap indexed by i-node number with several bits per i-node. Bits will be set and cleared in this map as the algorithm proceeds. Although logical dumping is straightforward, there are few issues associated with it.

- Since the free block list is not a file, it is not dumped and hence it must be reconstructed from scratch after all the dumps have been restored
- If a file is linked to two or more directories, it is important that the file is restored only one time and that all the directories that are supposed to point to it do so
- UNIX files may contain holes
- Special files, named pipes and all other files that are not real should never be dumped.

File-system Consistency

To deal with inconsistent file systems, most computers have a utility program that checks file-system consistency. For example, UNIX has fsck and Windows has sfc. This utility can be run whenever the system is booted. The utility programs perform two kinds of consistency checks.

- **Blocks:** To check block consistency the program builds two tables, each one containing a counter for each block, initially set to 0. If the file system is consistent, each block will have a 1 either in the first table or in the second table as you can see in the figure below.



In case if both the tables have 0 in it that may be because the block is missing and hence will be reported as a missing block. The two other situations are if a block is seen more than once in free list and same data block is present in two or more files.

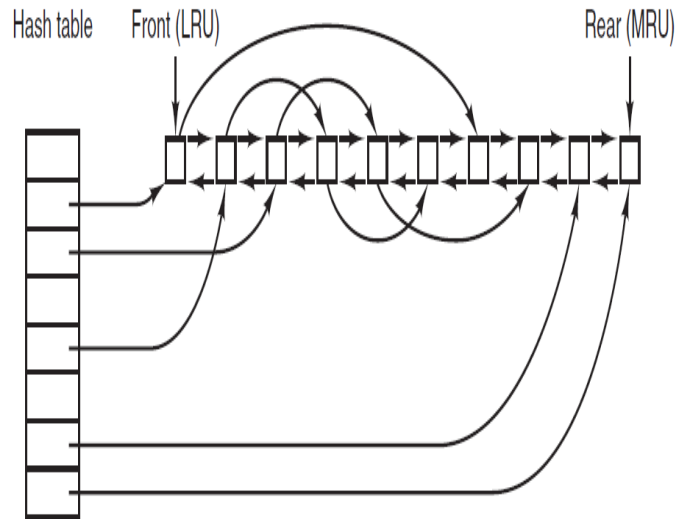
- In addition to checking to see that each block is properly accounted for, the file-system checker also checks the directory system. It too uses a table of counters but per file-size rather than per block. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree

File-system Performance

Since the access to disk is much slower than access to memory, many file systems have been designed with various optimizations to improve performance as described below.

Caching

The most common technique used to reduce disk access time is the block cache or buffer cache. Cache can be defined as a collection of items of the same type stored in a hidden or inaccessible place. The most common algorithm for cache works in such a way that if a disk access is initiated, the cache is checked first to see if the disk block is present. If yes then the read request can be satisfied without a disk access else the disk block is copied to cache first and then the read request is processed.



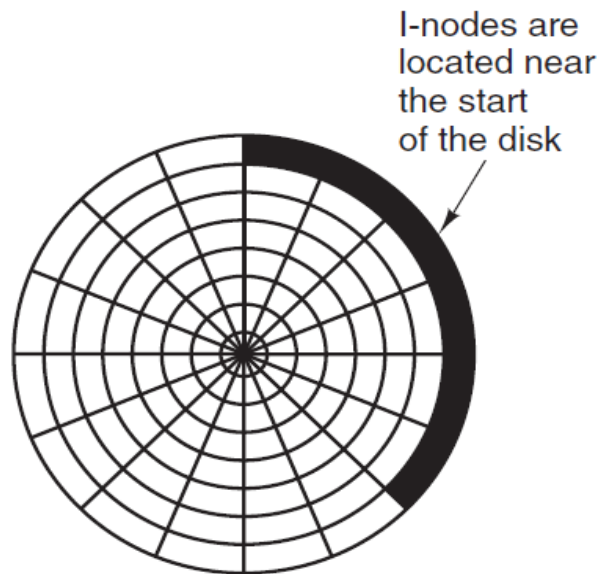
The above figure depicts how to quickly determine if a block is present in a cache or not. For doing so a hash table can be implemented and look up the result in a hash table.

Block Read Ahead

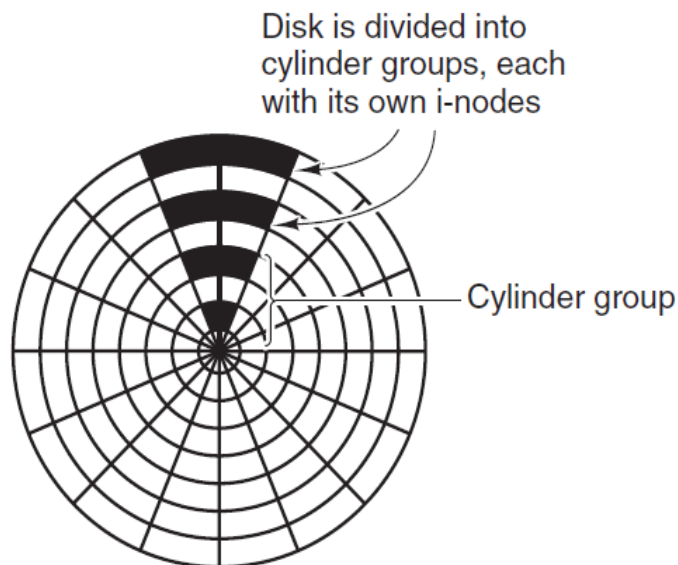
Another technique to improve file-system performance is to try to get blocks into the cache before they are needed to increase the hit rate. This works only when files are read sequentially. When a file system is asked for block 'k' in the file it does that and then also checks before hand if 'k+1' is available if not it schedules a read for the block k+1 thinking that it might be of use later.

Reducing disk arm motion

Another way to increase file-system performance is by reducing the disk-arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder.



In the above figure all the i-nodes are near the start of the disk, so the average distance between an inode and its blocks will be half the number of cylinders, requiring long seeks. But to increase the performance the placement of i-nodes can be modified as below.



Defragmenting Disks

Due to continuous creation and removal of files the disks get badly fragmented with files and holes all over the place. As a consequence, when a new file is created, the blocks used for it may be spread all over the disk, giving poor performance. The performance can be restored by moving files around to make them contiguous and to put all (or at least most) of the free space in one or more large contiguous regions on the disk.

