System Software and Operating System

By Chetan Bhojani

CS-16-System Software and OS

Language Processors	Introduction - Language Processing Activities Fundamentals of Language Processing Fundamentals of Language Specification	5
Scanning and Parsing	Scanning Parsing	8
Compilers and Interpreters	Aspects of Compilation Memory Allocation Compilation of Expressions	7
Linkers	Relocation and Linking Concepts Design of a Linker, Loader	5
Evolution of OS Functions	OS Functions Resource allocation and related functions User interface related functions Multiprogramming Systems Architectural support for M. P. User Services Real Time Operating Systems OS structure (UNIX)	15
Processes	Process Definition & Control Interacting Processes Inter process Messages	5
Scheduling .	Job Scheduling Process Scheduling Event Monitoring Process Scheduling Mechanisms	15
Deadlocks	Definitions and Handling Deadlocks	5
Memory Management	Contiguous Memory Allocation Noncontiguous Memory Allocation Virtual Memory Using Paging Pages. Page Blocks, Address Translation Demand Paging Using Segmentation	15
File Systems	Directory Structures File Protection, Implementing File Access. File Sharing	15
Distributed OS	Role of Distributed OS	5

(Note: No routines should be discussed, only theoretical aspects should be discussed) Reference Books:

1. System Programming & operating Systems-D. M. Dhamdhare (PHI)
2. OS Internals and Design Principles - William Stallings (PHI)
3. OS Design and Implementation - A. S. Tanenbaum (PHI)
4. The Unix Programming Environment - Keminghan & Pike (PHI)
5. YOUR UNIX THE ULTIMATE GUIDE - Sumitabha Das (TMH)
6. Linux Complete - (Sybex, BPB)
7. Linux in a NUTSHELL - Siever (0 ' REILLY)

System Software and
Operating System

BCA SEM 4 2
CS-16-System Software and OS 2
System Software 5
Application Software: 5
The Interaction between Users, Application Software, System Software & Computer Hardware: 6
System Development Software: 6
Language processors 8
Language Processing Activities 8
Problem Oriented and Procedure Oriented Languages: 8
Program Generation 9
FUNDAMENTALS OF LANGUAGE PROCESSING 11
Compilers 14
Assemblers & compilers 14
Overview of the compilation process: 14
Linker & Loader 19
LOADER: 20
Evolution of OS Functions 21
Functions of OS: 21
Resource Allocation & Related Functions: 21
User Interface Functions: 22
Evolution of OS Functions: 22
Types of Operating Systems: 24
Batch Processing Systems: 24
Multiprogramming System: 26
Functions of the Multiprogramming Supervisor: 26
Deadlocks 28
Deadlocks 28
HANDLING DEADLOCKS 28
DEADLOCK DETECTION AND RESOLUTION 30
Memory Management 31
Memory management: 31
CONTIGUOUS MEMORY ALLOCATION 32
NONCONTIGUOUS MEMORY ALLOCATION 34
Virtual Memory 35
Address translation 35
Process Scheduling 38
Process Termination: 38
Implementation of Interacting Processing 39
The following are the facilities for implementation of interacting process in programming languages and Operating Systems 39
Inter Process Communications 40
Process Scheduling 43
File Systems 45
File System 46
File protection 48
Implementing File access 49
File Sharing 51
Distributed Operating System 52
Definition and Examples 54

System Software

There are two broad categories of software:
System Software
Application Software

System Software is a set of programs that manage the resources of a compute system. System Software is a collection of system programs that perform a variety of functions.

File Editing
Resource Accounting
I/O Management
Storage, Memory Management access management.

System Software can be broadly classified into three types as:

System control programs controls the execution of programs, manage the storage & processing resources of the computer & perform other management & monitoring function. The most important of these programs is the operating system. Other examples are database management systems (DBMS) & communication monitors.

System support programs provide routine service functions to the other computer programs & computer users: E.g. Utilities, libraries, performance monitors & job accounting.

System development programs assists in the creation of application programs. E.g., language translators such as BASIC interpreter & application generators.

Application Software:

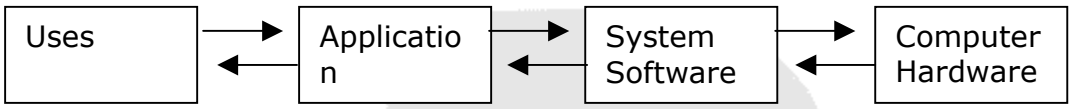
It performs specific tasks for the computer user. Application software is a program which program written for, or, by, a user to perform a particular job.

Languages already available for microcomputers include Clout, Q & A and Savvy ret rival (for use with Lotus 1-2-3).

The use of natural language touches on expert systems, computerized collections of the knowledge of many human experts in a given field, and artificial intelligence, independently smart computer systems – two topics that are receiving much attention and development and will continue to do so in the future.
1.Operating System Software
Storage Manager
Process Manager
File – System Manager
I/O Control System
Communication Manager
2. Standard System Software
Language Processor
Loaders
Software Tools

3. Application Software
Sort/Merge Package
Payroll/Accounting Package
DBMS
General-purpose application software such as electronic spreadsheet has a wide variety of applications. Specific – purpose application s/w such as payroll & sales analysis is used for the application for which it is designed

Application programmer writes these programs. Application programmer writes these programs.
Generally computer users interact with application software. Application and system software act as interface between users & computer hardware. An application & system software become more capable, people find computer easier to use.



The Interaction between Users, Application Software, System Software & Computer Hardware:

System Software controls the execution of the application software & provides other support functions such as data storage. E.g. when you use an electronic spreadsheet on the computer, MS-DOS, the computer's Operating System, handles the storage of the worksheet files on disk.

The language translators and the operating system are themselves programs. Their function is to get the users program, which is written, in a programming language to run-on the computer system.

All sucl. Programs, which help in the execution of user programs, are called system programs (SPs). The collection of such SPs is the "System Software" of a particular computer system.

Mast computer systems have support software, called Utility Programs, which perform routine tasks. These programs sort data, copy data from one storage medium to another, o/p data from a storage medium to printer & perform other tasks.

System Development Software:

System Development Software assists a programmer of user in developing & using an application program.

E.g. Language Translators
Linkage Editors
Application generators

Language Translators:

A language translator is a computer program that converts a program written in a procedural language such as BASIC into machine language that can be directly executed by the computer.

Computers can execute only machine language programs. Programs written in any other language must be translated into a machine language load module, which is suitable for loading directly into primary storage.

Subroutine or subprograms, which are stored on the system residence device to perform a specific standard function. E.g. if a program required the calculation of a square root,

Programmer would not write a special program. He would simply call a square root, subroutine to be used in the program.

Translators for a low-level programming language were assemblers

Language Processing Activities

Language Processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behaviour of a software and the manner in which these ideas are implemented in a computer system.
the interpreter is a language translator. This leads to many similarities between are
Translators and interpreters. From a practical viewpoint many differences also exist between translators and interpreters.
The absence of a target program implies the absence of an output interface the interpreter. Thus the language processing activities of an interpreter cannot be separated from its program execution activities. Hence we say that an interpreter 'executes' a program written in a PL.

Problem Oriented and Procedure Oriented Languages:

The three consequences of the semantic gap mentioned at the start of this section
are in fact the consequences of a specification gap. Software systems are poor in quality and require large amounts of time and effort to develop due to difficulties in bridging the specification gap. A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain.
Such PLs can only be used for specific applications; hence they are called *problem*-oriented *languages.* They have large execution gaps, however this is acceptable because the gap is bridged by the translator or interpreter and does not concern the software designer.

A *procedure*-oriented *language* provides general purpose facilities required in most application domains. Such a language is independent of specific application domains.

The fundamental language processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap. We name these activi-
ties as
1. Program generation activities
2. Program execution activities.

A program generation activity aims at automatic generation of a program. The source languages specification language of an application domain and the target language is typically a procedure oriented PL. A Program execution activity organizes the execution of a program written in a PL on computer system. Its source language could be a procedure-oriented language or a problem oriented language.

Program Generation

The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL. In effect, the program generator introduces a new domain between the application and PL domains we call this the *program generator domain.* The specification gap is now the gap between the application domain and the program generator domain. This gap is smaller than the gap between *the* application domain and the target PL domain.

Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.

The harder task of bridging the gap to the PL domain is performed by the generator.
This arrangement also reduces the testing effort. Proving the correctness of the pro-
gram generator amounts to proving the correctness of the transformation .
This would be performed while implementing the generator. To test an application generated by using the generator, it is necessary to only verify the correctness of the specification input to the program generator. This is a much simpler task than verifying correctness of the generated program. This task can be further simplified by providing a good diagnostic (i.e. error indication) capability in the program generator, which would detect inconsistencies in the specification.
It is more economical to develop a program generator than to develop a problem-oriented language. This is because a problem-oriented language suffers a very large execution gap between the PL domain and the execution domain whereas the program generator has a smaller semantic gap to the target PL domain, which is the domain of a standard procedure oriented language. The execution gap between the target PL domain and the execution domain is bridged by the compiler or interpreter for the PL.

Program Execution
Two popular models for program execution are translation and interpretation.
Program translation
The program translation model bridges the execution gap by translating a program written in a PL, called the *source program* (SP), into an equivalent program in the machine or assembly language of the computer system, called the *target program* (TP)
Characteristics of the program translation model are:

A program must be translated before it can be executed.
• The translated program may be saved in a file. The saved program may be executed repeatedly.
• A program must be retranslated following modifications.

Program interpretation

The interpreter reads the source program and stores it in its memory. During interpretation it takes a source statement, determines its meaning and performs actions which implement it. This includes computational and input-output actions.
The CPU uses a *program counter* (PC) to note the address of the next instruction to be executed. This instruction is subjected to the *instruction execution cycle* consisting of the following steps:
1. Fetch the instruction.
2. Decode the instruction to determine the operation to be performed, and also its operands.
3. Execute the instruction.
At the end of the cycle, the instruction address in PC is updated and the cycle is repeated for the next instruction. Program interpretation can proceed in an analogous manner. Thus, the PC can indicate which statement of the source program is to be interpreted next. This statement would be subjected to the *interpretation cycle,* which could consist of the following steps:
Fetch the statement.
Analyze the statement and determine its meaning, viz. the computation to be performed and its operands.
Execute the meaning of the statement.

From this analogy, we can identify the following characteristics of interpretation:
The source program is retained in the source form itself, i.e. no target program form exists,
A statement is analyzed during its interpretation.

Comparison

A fixed cost (the translation overhead) is incurred in the use of the program translation model. If the source program is modified, the translation cost must be incurred again irrespective of the size of the modification. However, execution of the target program is efficient since the target program is in the machine language. Use of the interpretation model does not incur the translation overheads. This is advantageous if a program is modified between executions, as in program testing and debugging.

FUNDAMENTALS OF LANGUAGE PROCESSING

Definition
Language Processing = Analysis of SP + Synthesis of TP.
Definition motivates a generic model of language processing activities.
We refer to the collection of language processor components engaged in analyzing a source program as the *analysis phase* of the language processor.
Components engaged in synthesizing a target program constitute the *synthesis phase.*
A specification of the source language forms the basis of source program analysis. The specification consists of three components:
1. *Lexical rules*, which govern the formation of valid lexical units in the source language.
2. *Syntax rules* which govern the formation of valid statements in the source language.
3. *Semantic rules* which associate meaning with valid statements of the language.

The analysis phase uses each component of the source language specification to determine relevant information concerning a statement in the source program. Thus, analysis of a source statement consists of lexical, syntax and semantic analysis.

The synthesis phase is concerned with the construction of target language statement(s) which have the same meaning as a source statement. Typically, this consist of two main activities:
- Creation of data structures in the target program
- Generation of target code.

We refer to these activities as *memory allocation* and *code generation,* respectively

Lexical Analysis (Scanning)

Lexical analysis identifies the lexical units in a source statement. It then classifies the units into different lexical classes e.g. id's, constants etc. and enters them into different tables. This classification may be based on the nature ofstring or on the specification of the source language. (For example, while an integer constant is a string of digits with an optional sign, a reserved id is an id whose name matches one of the reserved names mentioned in the language specification.) Lexical analysis builds a descriptor, called a *token,* for each lexical unit. A token contain two fields—*class code,* and *number in class, class code* identifies the class to which a lexical unit belongs, *number in class* is the entry number of the lexical unit in the relevant table.

Syntax Analysis (Parsing)

Syntax analysis processes the string of tokens built by lexical analysis to determine the statement class, e.g. assignment statement, if statement, etc. It then builds an IC which represents the structure of the statement. The IC is passed to semantic analysis to determine the meaning of the statement.

Semantic analysis

Semantic analysis of declaration statements differs from the semantic analysis of imperative statements. The former results in addition of information to the symbol table, e.g. type, length and dimensionality of variables. The latter identifies the sequence of actions necessary to implement the meaning of a source statement. In both cases the structure of a source statement guides the application of the semantic rules. When semantic analysis determines the meaning of a sub tree in the IC. It adds information a table or adds an action to the sequence.It then modifies the IC to enable further semantic analysis. The analysis ends when the tree has been completely processed.

FUNDAMENTALS OF LANGUAGE SPECIFICATION

A specification of the source language forms the basis of source program analysis. In this section, we shall discuss important lexical, syntactic and semantic features of a programming language.

Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. This section discusses key concepts and notions from formal language grammars. A language L can be considered to be a collection of valid sentences. Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as *a. formal language.* A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal symbols, alphabet and strings

The *alphabet* of L, denoted by the Greek symbol Z, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols in Z.

A symbol in the alphabet is known as a *terminal symbol* (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g.

$\Sigma \cong$ {a, b, ….z, 0,1....9}

Here the symbols {, ',' and} are part of the notation. We call them *met symbols* to differentiate them from terminal symbols. Throughout this discussion we assume that met symbols are distinct from the terminal symbols. If this is not the case, i.e. if a terminal symbol and a met symbol are identical, we enclose the terminal symbol in quotes to differentiate it from the metasymbol. For example, the set of punctuation symbols of English can be defined as

{:,;’,’-,...}

Where ',' denotes the terminal symbol 'comma'.

A *string* is a finite sequence of symbols. We will represent strings by Greek symbols-α β γ, etc. Thus α = axy is a string over Σ . The length of a string is the Number of symbols in it. Note that the absence of any symbol is also a string, the *null string* . The *concatenation* operation combines two strings into a single string.

Compilers

To evaluate an HLL program it should be converted into the Machine language. A compiler performs another very important function. This is in terms of the diagnostics.

I.e. error – detection capability.

The important tasks of a compiler are:

Translating the HLL program input to it.
Providing diagnostic messages whenever specifications of the HLL

Assemblers & compilers

Assembler is a translator for the lower level assembly language of computer, while compilers are translators for HLLs.

An assembly language is mostly peculated to a certain computer, while an HLL is generally machined independent & thus portable.

Overview of the compilation process:

The process of compilation is:

Analysis of Source Text	+	Synthesis of Target Text	=	Translation of Program

Source text analysis is based on the grimmer of the source of the source language.

The component sub – tasks of analysis phase are:
Syntax analysis, which determine the syntactic structure of the source statement.
Semantic analysis, which determines the meaning of a statement, once its grammatical structures become known.

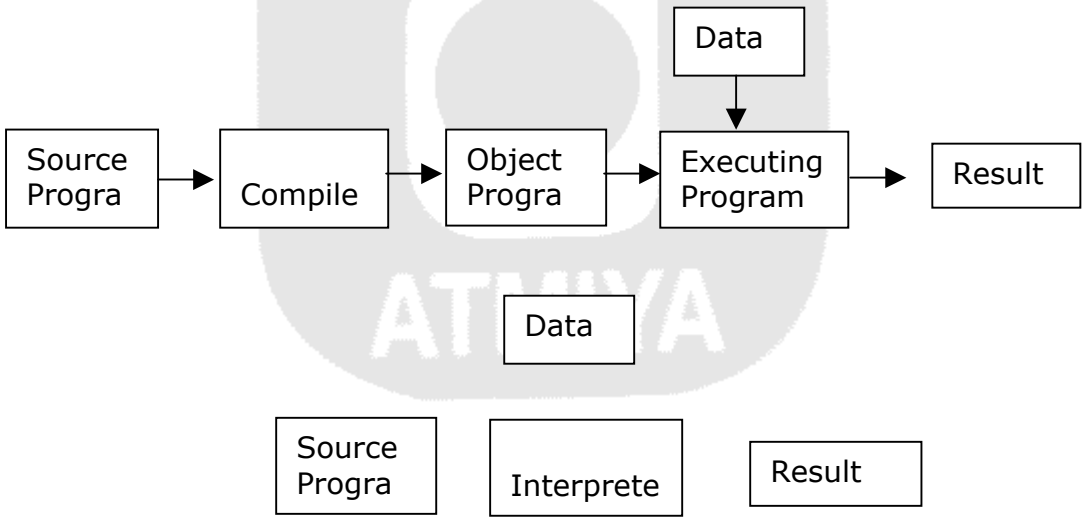
The analysis phase

The analysis phase of a compiler performs the following functions.
Lexical analysis
Syntax analysis
Semantic analysis

Syntax analysis determines the grammatical or syntactic structure or the input statement & represents it in an intermediate form from which semantic analysis can be performed.

A compiler must perform two major tasks:
The Analysis of a source program & the synthesis of its corresponding object program.

The analysis task deals with the decomposition of the source program into its basic parts using these basic parts the synthesis task builds their equivalent object program modules. A source program is a string of symbols each of which is generally a letter, a digit or a certain special constants, keywords & operators. It is therefore desirable for the compiler to identify these various types as classes.

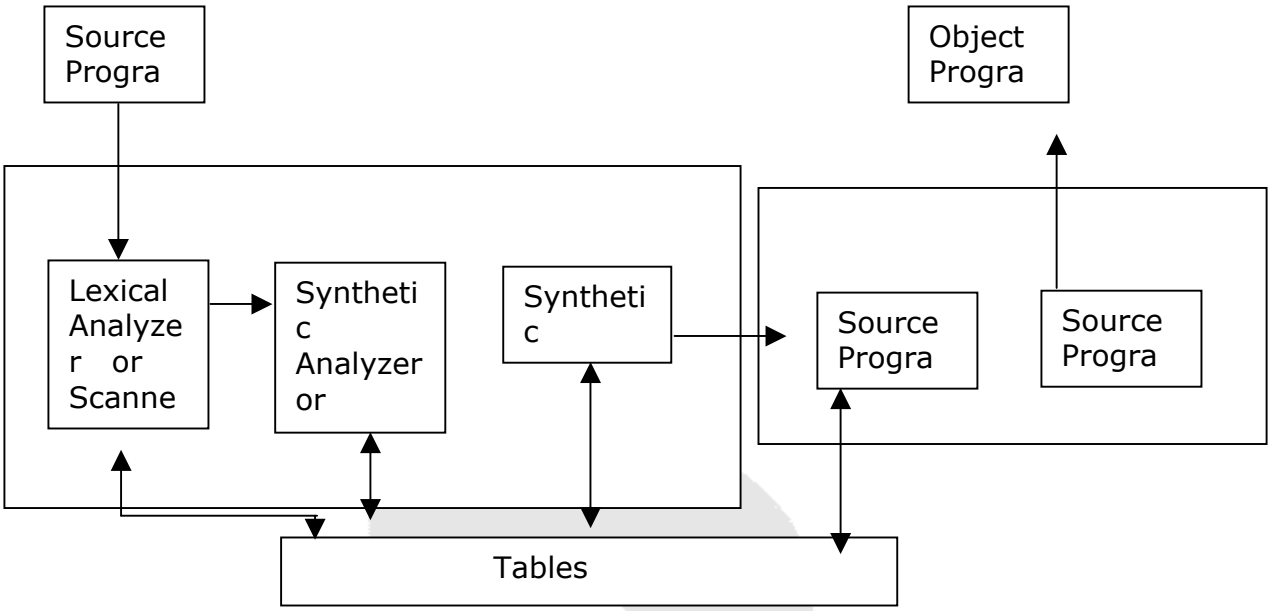

Data
Source Progra
Compile
Object Progra
Executing Program
Result
Data
Source Progra
Interprete
Result

The source program is input to a lexical analyzer or scanner whose purpose is to separate the incoming text into pieces or tokens such as constants, variable name, keywords & operators.

In essence, the lexical analyzer performs low- level syntax analysis performs low-level syntax analysis.

For efficiency reasons, each of tokens is given a unique internal representation number.

TEST: If A > B then X=Y;



The lexical analyzer supplies tokens to the syntax analyzer.

The syntax analyzer is much more complex then the lexical analyzer its function is to take the source program from the lexical analyzer & determines the manner in which it is to be decomposed into its constituent parts.

That is, the syntax analyzer determines the overall structure of the source program.

The semantic analyzer uses syntax analyzer.

The function of the semantic analyzer is to determine the meaning the meaning (or semantics) of the source program.

The semantic analyzer is passed on to the code generators.

At this point the intermediate form of the source language programs usually translated to either assembly language or machine language.

The output of the code generator is passed on to a code optimizer. It's purpose to produce more program.

Linker & Loader

A software processor, which performs some low level processing of the programs input to it, produces a ready to execute program form.

The basic loading function is that of locating a program in an appropriate area of the main store of a computer when it is to be executed.

A loader often performs the two other important functions.

The loader, which accepts the program form, produced by a translator & certain other program forms from a library to produce one ready – to – execute machine language program.

A unit of input to the loader is known as an object program or an object module.

The process of merging many object modules to from a single machine language program is known as linking.

The function to be performed by:

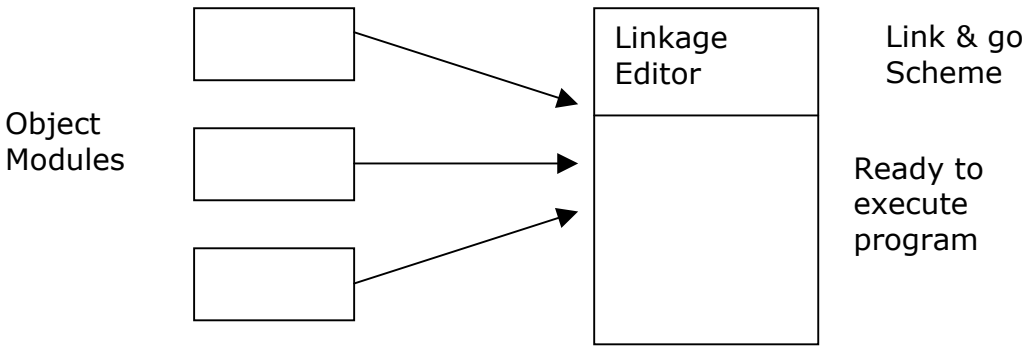
Assigning of loads the storage area to a program.

Loading of a program into the assigned area.

Relocations of a program to execute properly from its load-time storage area.

Linking of programs with one another.

Loader, linking loaders, linkage editors are used in software literature



LOADER:

The loader is program, which accepts the object program decks, prepares this program for execution by the computer and initializes the execution.

In particular the loader must perform four functions:

Allocate space in memory for the program (allocation).
Resolve symbolic references between objects decks (linking).
Adjust all address dependent locations, such as address constants, to correspond to the allocated space (relocation).
Physically place the machine instructions and data into memory (loading).

Operating System

Evolution of OS Functions

Functions of OS:

Operating System: "An operating system provides interface between the user & the hardware."

It can be basically classified into:

- Resource Allocation & Related Functions.
- User Interface Functions.

The Resource Allocation function implements resources sharing by the users of a computer system. Basically it performs binding of a set of resources with the requesting program-that is it associates resources with a program. The related functions implement protection of users sharing a set of resources against mutual interference.

Resource Allocation & Related Functions:

The resource allocation function allocates resources for use by a user's computation. Resources can be divided into two types:

1. System Provided Resources – like CPU, memory and IO devices

User created Resources – like files etc.
Resource allocation depends on whether a resource is a system resource or a user created resource.
There are two popular strategies for resource allocation**:**
Partitioning of resources
Allocation from a pool.

Using resource partition approach, OS decides priori what resources should be allocated to a user computation. This is known as static allocation as the allocation is made before the execution of the program starts.
Using pool allocation approach, OS maintains a common pool & allocates resources from this pool on a need basis. This is called dynamic allocation because it takes place during the execution of program. It can lead to better utilization of resources because the allocation is made when a program request a resource.
An OS can use a resource table as a central data structure for resource allocation. The table contains an entry for each resource unit in the system. The entry contains the name or address of the resource unit and its present system i.e whether it is free or allocated to some program. When a program raises a request for a resource ,the resource should be allocated to it if it is presently free.

In the partition resource allocation approach,the OS decides on the resources to be allocated to a program based on the number of the program in the system.

For Example, an OS may decide that a program can be allocated 1 MB of memory, 200 disk blocks and a monitor. Such a collection of resources is referred to as a partition. The resource table can have an entry for each resource partition. When a new program is to be started, an available partition is allocated to it.

User Interface Functions:

Its purpose is to provide the use of OS resources for processing a user's computational requirements. OS user interfaces use command languages. For this, the user uses Command to set up an appropriate computational structure to fulfill his computational requirements.

An OS can define a variety of computational structures. A sample list of computational structures is as follows:

1. A single program
2. A sequence of single program
3. A collection of programs

The single program consist the execution of a program on a given set of data. The user initiates execution of the program through a command. Two kinds of program can exist – Sequential and concurrent. A sequential program is the simplest computational structure. In concurrent program the OS has to be aware of the identities of the different parts, which can execute concurrently.

Evolution of OS Functions:

Operating System functions have evolved in response to the following considerations and issues.

1. Efficient utilization of computing resources
2. New features in computer Architecture
3. New user requirements.

Different operating systems address these issues in different manner, however most operating system contains components, which have similar functionalities. For example, all operating systems contain components for functions of memory management, process management and protection of users from one another. The techniques used to implement these functions may vary from one OS to another, but the fundamental concept is same.

Process:

A process is execution of a program or a part of a program.

Job:

A job is computational structure, which is a sequence of program.

ATMIYA

Types of Operating Systems:

1. Batch Processing system
2. Multiprogramming system
3. Time sharing system
4. Real time operating system
5. Distributed systems

Batch Processing Systems:

When Punch cards were used to record user jobs, processing of a job involved physical actions by the system operator e.g. loading a deck of cards into the card reader, pressing switches on the computer's console to initiate the job. These actions wasted a lot of CPU time. BP was introduced to avoid this wastage.

A batch is a sequence of user jobs. A computer operator forms a batch by arranging user jobs in a sequence and inserting special marker cards to indicate the start and end of the batch. After forming a batch, the operator submits it to the batch processing operating system. The primary function of the BP system is to implement the processing of the jobs in a batch.

Batch processing is implemented by locating a component of the BP system called the batch monitor or supervisor, permanently in one part of the computer's memory. The remaining memory is used to process a user job the current job in the batch. The batch monitor is responsible for implementing the various function of the batch processing system. It accepts a command from the system operator for initiating the processing of a batch and sets up the processing of the first job of the batch. At the end of the job, it performs job termination processing and initiates execution of the batch; it performs batch termination processing system and awaits initiation of the next batch by the operator.

The part of memory occupied by the batch monitor is called the system area and the part occupied by the user job is called the user area.

User Service:

A user evaluates the performance of an os on the basis of the service accorded to his or her job. The notion of *turn-around time* is used to quantity user service in a batch processing system.

Note: The turn around time of a user job is the time since its submission to the time its results become available to the user

Batch processing does not guarantee improvements in the turn around time of jobs.Batch processing does not aim at improving user services-it aims at improving CPU utilization.

Batch Monitor Functions:

The basic task of the batch monitor is to exercise effective control Over the BP environment. This task can be classified into the following three functions.
Scheduling
Memory Management

The batch monitor performs two functions before initiating the execution of a job. The third function is performed during the execution of a job.

In Batch Processing System, The CPU Of The Computer System Is The Server And The User Jobs Are The Service Requests. The Nature Of Batch Processing Dictates The Use Of The First Come First Serve(FCFS) Scheduling. The Batch Monitor Performs Scheduling By Always Selecting The Next Job In The Batch For Execution. Scheduling Does Not Influence The User Services In The BP System Because The Turn Around Time Of Each Job In A Batch Is Subject To Some Other Factors.

At any time during a BP system's operation, the memory is divided into the system area and the user area. The user area of the memory is sequentially shared by the jobs in the batch.

Multiprogramming System:

Early computer systems implemented IO operation as CPU instructions. It sent a signal to the card reader to read a card and waited for the operation to complete before initiating the next operation. However the speeds of operation of IO devices were much lower than the speed of the CPU. Programs took long to complete their execution. A new feature was introduced in the machine architecture when this weakness was realized. This feature permitted the CPU to delink itself from an IO operation so that it could execute instructions while an IO operation was in progress. Thus the CPU and the IO device could now operate concurrently.

If many user programs exist in the memory, the CPU can execute instructions of one program while the IO subsystem is busy with an IO operation for another program. The term multiprogramming is used to describe this arrangement. At any moment the program corresponding to the current job step of each job is in execution. The IO device and memory are allocated using the partitioned resource allocation approach. At any time, the CPU and IO subsystem are busy with programs belonging to different jobs. Thus they access different areas of memory. In principle the CPU and IO subsystem could operate on the same program. Each job in the memory could be current job of a batch of jobs. Thus one could have both batch processing and multiprogramming supervisor. Analogous to a BP supervisor, the MP supervisor also consists of a permanently resident part and a transient part.

The multiprogramming arrangement ensures concurrent operation of the CPU and the IO subsystem without requiring a program to use the special buffering techniques. It simply ensures that the CPU is allocated to a program only when it is not performing an IO operation.

Functions of the Multiprogramming Supervisor:

Scheduling
Memory Management
IO management

The MP supervisor uses simple techniques to implement its functions. Function like scheduling implies sharing of the CPU between the jobs existing in the MP system. This function is performed after servicing every interrupt using a simple priority based scheme described in the next section. The allocation of memory and IO devices is performed by static partitioning of resources. Thus a part of memory and some IO devices are allocated to each job. It is necessary to protect the data and IO operations of one program from interference by another program. This is achieved by using memory protection hardware and putting CPU in non-privileged mode while executing a user program. Any effort by a user program to access memory locations situated outside its memory area now leads to an interrupt. The interrupting processing routines for these interrupts simply terminates the program causing the interrupt.

Scheduling:

The goal of multiprogramming is to exploit the concurrency of operation between the CPU and IO subsystem to achieve high levels of system utilization. A useful characterization of system utilization is offered by *throughput* of a system .

Throughput: The throughput of a system is the number of programs processed by it per unit time.

$$\text{Throughput} = \frac{\text{Number of programs completed}}{\text{Total time taken}}$$

To optimize the throughput, a MP system uses the following concepts:

A proper mix of programs:

For good throughput it is important to keep both the CPU and IO subsystems busy.

A CPU bound program is a program involving a lot of computation and very little IO. It uses the CPU for a long time.

An IO bound program is a program involving very little computation and a lot of IO.

2.Preemptive and priority based scheduling:

Scheduling is priority based that is the CPU is always allocated to the highest priority programs. The Scheduling is preemptive the is a low priority program executing on the CPU is preempted if a higher priority program wishes to use the CPU.

3.Degree of multiprogramming:

Degree of multiprogramming is the number of programs existing simultaneously in the system's memory.

Deadlocks

Processes compete for physical and logical resources in the system (e.g. disks or files). Deadlocks affect the progress of processes by causing indefinite delays in resource allocation. Such delays have serious consequences for the response times of processes, idling and wastage of resources allocated to processes, and the performance of the system. Hence an OS must use resource allocation policies, which ensure an absence of deadlocks. This chapter characterizes the deadlock problem and describes the policies an OS can employ to ensure an absence of deadlocks.

DEFINITIONS

We define three events concerning resource allocation:

1. *Resource request:* A user process requests a resource prior to its use. This is done through an OS call. The OS analyses the request and determines whether the requested resource can be allocated to the process immediately. If not. The process remains blocked on the request till the resource is allocated.
2. *Resource allocation:* The OS allocates are source to a requesting process. The resource status information is updated and the state of the process is changed to *ready.* The process now becomes the *holder* of the resource.
3. *Resource release:* After completing resource usage, a user process releases the resource through an OS call. If another process is blocked on the resource, OS allocates the resource to it. If several processes are blocked on the resource, the

OS uses some tie-breaking rule, e.g. FCFS allocation or allocation according to process priority, to perform the allocation.

Deadlock: A deadlock involving a set of processes D is a situation in which

1.Every process pi in D is blocked on some event ei

Event ei can only be caused by some process (es) in D.

If the event awaited by each process in D is the granting of some resource, it results in a resource deadlock. A communication deadlock occurs when the awaited events pertain to the receipt of interprocess messages, and synchronization deadlock when the awaited events concern the exchange of signals between processes. An

OS is primarily concerned with resource deadlocks because allocation of resources is an OS responsibility. The other two forms of deadlock are seldom tackled by an OS.

HANDLING DEADLOCKS

Two fundamental approaches used for handling deadlocks are:

1. Detection and resolution of deadlocks
2. Avoidance of deadlocks.

In the former approach, the OS detects deadlock situations as and when they arise. It then performs some actions aimed at ensuring progress for some of the deadlocked processes. These actions constitute deadlock resolution. The latter approach focuses on avoiding the occurrence of deadlocks. This approach involves checking each resource request to ensure that it does not lead to a

deadlock. The detection and resolution approach does not perform any such checks. The choice of the deadlock handling approach would depend on the relative costs of the approach, and its consequences for user processes.

DEADLOCK DETECTION AND RESOLUTION

The deadlock characterization developed in the previous section is not very useful in practice for two reasons. First, it involves the overheads of building and maintaining an RRAG. Second, it restricts each resource request to a single resource unit of one or more resource classes. Due to these limitations, deadlock detection cannot be implemented merely as the determination of a graph property. For a practical implementation, the definition can be interpreted as follows: A set of *blocked* processes
D is deadlocked if there does not exist any sequence of resource allocations and resource releases in the system whereby each process in *D* can complete. The OS must determine this fact through exhaustive analysis.
Deadlock analysis is performed by simulating the completion of a *running* process. In the simulation it is assumed that a *running* process completes without making additional resource requests. On completion, the process releases all resources allocated to it. These resources are allocated to a blocked process only if the process can enter the *running* state. The simulation terminates in one of two situations—either all *blocked* processes become *running* and complete, or some set *B of blocked* processes cannot be allocated their requested resources. In the former case no deadlock exists in the system at the time when deadlock analysis is performed, while in the latter case processes in *B* are deadlocked.

Deadlock Resolution

Given a set of deadlocked processes *D, deadlock resolution* implies breaking the deadlock to ensure progress for some processes *{pi}* £ *D.* This can be achieved by satisfying the resource request of a process *pi* in one of two ways:
1. Terminate some processes *{pj}* e *D* to free the resources required by *pi.* (We call each *pj* a *victim* of deadlock resolution.)
2. Add a new unit of the resource requested by *pi.*
Note that deadlock resolution only ensures some progress for *pi.* It does not guarantee that a *pi* would run to completion. That would depend on the behaviour of processes after resolution.

Memory Management

Memory management:

The memory of computer system consist of primary memory, the secondary memories like disks and tapes and buffer or cache memories. Of these, management of a secondary memory is under the control of the file system. Cache memories are captive to the hardware subsystems and are not accessible to a user program. Hence the term memory management is used for the management of the primary memory.
Memory management includes the memory allocation, efficient utilization of memory and protection of memory allocated to a program or a process from interference by other programs.
Two kind of memory management are performed during the operations of an OS. These involves allocation and deallocation of memory
To programs or processes
Within a program or process

Allocation of memory is done by the kernel.

CONTIGUOUS MEMORY ALLOCATION

This approach uses the classical program model where each *program is allocated* a single contiguous area in memory. The memory allocation decision is made *statically,* i.e. before the execution of a program begins. Variations in this approach concern the scope of the decision—whether the allocation is made for a job as a unit or separately for each job step, i.e. each program, in the job.

1. *Protection* of programs from one another
2. *Static relocation* of a program to execute from the allocated memory area
3. *Memory fragmentation* and measures to overcome it.

Memory Protection

Memory protection is used to avoid interference between programs existing in memory. Implementation of memory protection requires support from a machine's architecture in the form of memory bound registers or memory protection keys associated with different areas of memory.

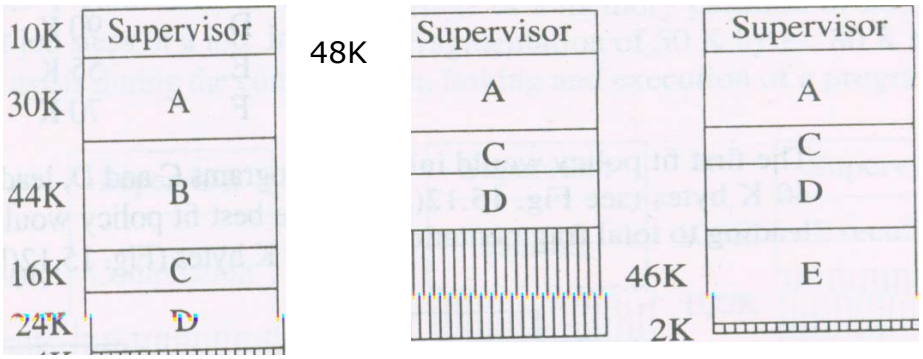
Memory Fragmentation

Memory fragmentation is defined as the presence of unusable memory areas in a computer system. *External fragmentation* arises when free memory areas existing in a system are too small to be allocated to programs. *Internal fragmentation* arises when memory allocated to a program is not fully utilized by it. In this section we discuss techniques used to overcome the problem of memory fragmentation.

Memory compaction

To eliminate external fragmentation, OS can *periodically perform memory* compaction. This produces a single unused area in memory. This area may be large enough to accommodate one or more new programs.

Example: Figure illustrates memory compaction. Initially, programs A, B, C and D are in execution. Compaction is performed when program B terminates. This produces a single unused memory area of 48 K bytes at the high end of memory. Program E with a size of 46 K bytes can now be initiated in this memory area. This leaves a single unused area of 2 Kbytes



(a) (b) (c)

Memory compaction
Compaction involves movement of programs in the memory during their execution. This is *dynamic relocation of* a program. This can be achieved quite simply in computer systems using a relocation register. Consider program C of Fig. 15.11 (a).
Assuming its linked origin to be '0000', relocation register should contain the address 84 K. During memory compaction (see Fig. 15.11(b)) it is moved to the memory area with start address 40 K. To implement this dynamic relocation, relocation register would be loaded with the address 40 K whenever program C is scheduled.
This is achieved by storing the value 40 Kin a relevant field of program C's PCB.

Garbage collection
Garbage collection is aimed at selective reuse of memory fragments. When a program terminates and releases its allocation of, say, *z* memory units, the OS searches through the list of programs awaiting memory allocation and selects a pi

NONCONTIGUOUS MEMORY ALLOCATION

The classical contiguous memory allocation model requires a program to be allocated a single contiguous area of memory. This causes memory fragmentation as we saw in the previous section. An effective solution of the fragmentation problem therefore lies in permitting a program to occupy noncontiguous areas of memory.

Virtual Memory

A computer system is said to use *virtual memory* (VM) if a memory address use (by an instruction is likely to be different from the effective address of the memory location accessed during its execution. In other words, a virtual memory system one that makes noncontiguous memory allocation feasible. The memory handler in a VM system is called the *VM handler.*

While loading a program for execution, it is viewed as a sequential arrangement of abstract part called *program components.* Each component is allocated a single contiguous are of memory. However, areas allocated to different components can be noncontiguous with one another. An address used in an instruction of a program, say, an operand address, is considered to consist of two parts—the id of the program component containing the address, and the id of the word within the component. Each address is represented by the pair *(comp„wordi).* In an address (10, 349), *compi = 10* an *wordi* = 349. It is thus the address of the 349'*h* word in the program component

Address translation

Let us see the basics of address translation with the help of an address (10, 349 Consider program component 10 to be allocated the memory area starting at addre;
27000. Then,
Effective memory address corresponding to the address (10, 349)
= (Start address of component 10) + 349
= 27000+349 = 27349 (15.

Address translation can be performed in a table-based manner where the table contains start addresses of various components of a program. In an address *(compj, word;), comp;* is used to index this table to obtain its start address in memory.

ATMIYA

ATMIYA

CHAPTER: 4 Process Management

Process Management

Process Scheduling

Process scheduling consist of the following sub-functions:
Scheduling: Selects the process to be executed next on the CPU
Dispatching: Sets up execution of the selected process on the CPU
Context Save: Saves the status of a running process when its execution is to be suspended

The Scheduling function uses information from the PCB's and selects a process based on the scheduling policy. Dispatching involves setting up the execution environment of the selected process, and loading information from the PSR and registers field of the PCB into the CPU. The context save function performs housekeeping whenever a process releases the CPU. It involves saving the PSR and registers in the appropriate fields of PCB and changing process state.

Event
↓
Context Save
↓
Scheduling
↓
Dispatching

Occurrence of an event invokes the context save function The kernel now processes the event which has occurred. This processing may lead to change of state for some process. The scheduling function is now invoked to select a process for execution on the CPU. The dispatching function arranges execution of the selected function on the CPU.

Process Termination:

When a process terminates the kernel has to release all resources allocated to that process.
If the termination process was created by another user process, some additional actions are necessary to implement process synchronization.

Implementation of Interacting Processing

Interacting Processes:

Execution of an application may lead to creation of many processes

Control Synchronization: In control synchronization, interacting processes co-ordinate their execution with respect to one another. Synchronization can occur at any point in the lifetime of a process, including at the start or end of its lifetime.

An application may consist of a set of processes sharing some data. Such arrangements are common in systems supporting database query and update services. This is done with the help of data access synchronization.

Concurrent processes may need to interact in ways other than control and data access synchronization. Such needs are satisfied using inter process messages.

Process p_i, sends a message msg_i to process p_j by executing the OS call *send* .The OS copies the message into a buffer area, and awaits execution of the receive call by process pj. When pj executes the *receive* call ,the OS copies msg_i out of the buffer and into the data area with the address. If no message has been sent by the time p_j executes a receive call, the OS blocks p_j pending arrival of a message for it. If many messages have been sent to pj, the OS queues up them in FIFO order when p_j executes receive calls.

Inter process communication requirements can be satisfied using shared variables.

The following are the facilities for implementation of interacting process in programming languages and Operating Systems.

1.Fork-Join Primitives
2.Parbegin-Parend control Structure
3.Unix Processes
4.OSF threads

Fork –Join

Fork and Join are primitives in higher level programming languages. The Syntax of these is as follows:

Fork <lable>
Join <var>

Where <lable> is a lable associated with some program statement and <var> is a variable. A statement fork lab1 causes creation of a new process, which starts executing the statement with the lable lab1. This process is concurrent with the process executed the statement fork lab1. A join statement synchronizes the birth of a process with the termination of one or more processes.

Fork-join provides a functionally complete facility for control synchronization. However its use is error prone.

Inter Process Communications

Parbegin-Parend

The Parbegin-Parend control structure has the following syntax:

```
Parbegin
        <List of statements>
Parend
```

Execution of the control structure results in concurrent execution of each statement in <list of statements>. Execution of the control structure spawns n processes. Execution of the statement following parend is initiated only after all the processes spawned by Parbegin-Parend have terminated

Processes in Unix

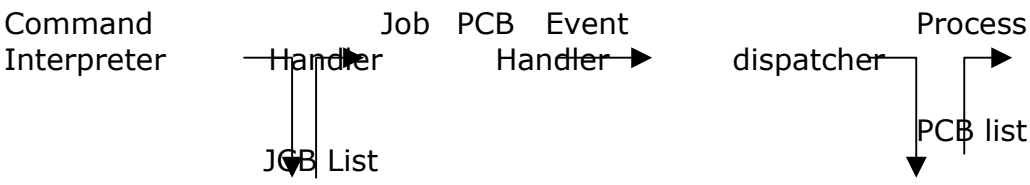
Unix permits a user process to create child processes and to synchronize its activity with respect to its child processes. The following features are provided for this purpose:

1. Creation of a child process
2. Termination of a process
3. Waiting for termination of a child process

Process Creation:

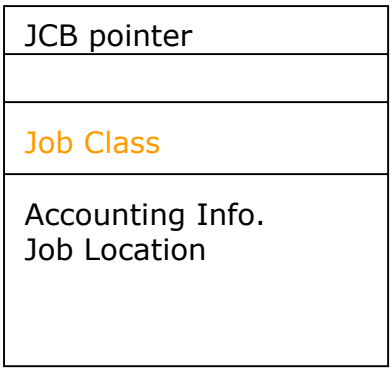
A process creates a child process through the system call fork. Fork creates a child process and sets up its execution environment. It then allocates an entry in the PCB for the newly created process and marks its state as *ready* .The child process shares the address space and file pointer of the parent process. hence data and files can be directly shared. A child process can create its own child process, thus leading to the creation of process tree. The system keeps track of the parent-child relationship.

Job Scheduling:



Modules in the left half of the figure such as command interpreter, job scheduler and job handler are the job scheduling components while modules in the right half are process-scheduling components. The command interpreter reads

command initiating a job interprets it to obtain all information about the job and records this information into job control block (JCB). It next enters the JCB into the job-scheduling list. The next figure shows the structure of JCB


JCB pointer
Job Class
Accounting Info.
Job Location

The job location field contains names of files containing the text of the jobs.

The job scheduler analyses the information in the JCB existing in the scheduling list and selects a job for execution. It passes the JCB of the selected job to the job handler. The Job handler locates the text of the job and performs allocation of resources like memory, IO Devices etc. After resource allocation, a PCB is handed over to the event handler, which enters it into the process scheduling list.

Process Scheduling

Process Scheduling involves the following tasks:
Creating new processes
Monitoring process states
Selecting a process for execution (process scheduling)
Allocating the CPU to the selected process (process dispatching)
Deallocating the cpu from a process and determining the new state of the process
Termination of the process
Supporting communication and synchronization requirements of process
Interfacing with other OS modules, which affect the process state.]

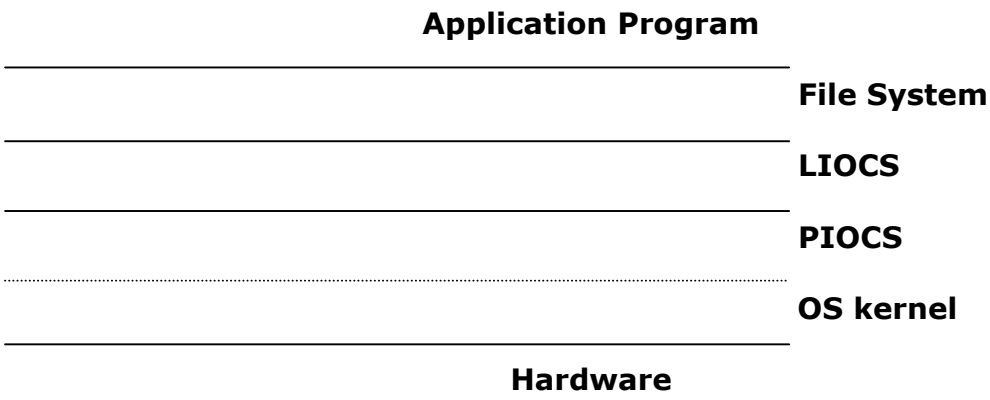
Function 3 is primary function in scheduling, Function 1,2,5 involves the housekeeping actions in support of process scheduling e.g. saving process state and contents of cpu registers in PCB
The process scheduling components consist of three smaller components-event handlers, process
Scheduler and process dispatcher. Process scheduler examines the information in PCB to select a process for execution and hands over its PCB to dispatcher. The dispatcher loads the PSR and the register contents from the PCB to the CPU to initiate the execution of the process Event handler process each interrupt occurring in the system.

ATMIYA

IO management

IO management component consist of mechanism and policies. IO mechanisms are concerned with the implementations of IO operations in the system. IO policies are concerned with performance issues or user oriented issues like sharing and protection of files.
The IO management modules can be structured in the following layers:

1. Physical input output control system (PIOCS)
2. Logical input output control system (LIOCS)
3. File system (FS)



The PIOCS layer is closest to the hardware of the computer system, while the FS layer is closest to an application program. Each layer provides an interface to the higher level through which its facilities can be accessed.

Mechanism and policies implemented by IO modules:
1.Physical IOCS layer
(a) Mechanism
IO initiation
IO Operation status
IO completion processing
IO error recovery
(b) Policies
Optimization of IO device performance
2. Logical IOCS layer
Mechanism
Label Processing
File open and close
File read and write
(b) Policies
Optimization of file access performance
File System layer
Mechanism
Directory maintenance mechanism
(b) Policies
Sharing and protection of files

The file system layer provides the following facilities to application program:

1.File naming freedom
2.File sharing
3.Protection against illegal file accesses
4. Reliable storage of files

File System

The FS layer hides all details of IO organization and LIOCS module interfaces from the application program. It is thus independent of the hardware and software of a computer system.
The fundamental difference between LIOCS and FS is:

The LIOCS views a file as an entity which is created or manipulated by application program, whereas FS views a file as an entity owned by a user, can be shared by a set of authorized users. and has to be reliably stored over an extended period of time.
Let two users **A** and **B** create file named virani. If A wishes to open his file virani using the LIOCS interface, LIOCS will simply open his file using LIOCS interface, LIOCS will simply open the first file virani it can find in the *VTOC* of a disk. This could also be a B's file. The only way to avoid this problem is for A and B to somehow ensure unique names for their files.
FS provides complete file naming freedom to users. It achieves this by using directories to differentiate between the files created by different users.
The presence of directories enables FS to support the sharing and protection. Sharing is simply a matter of permitting a user to access the files existing in some other user's directory.
FS provides reliability of file storage by adopting specific techniques to guard against corruption and loss of data due to malfunction.

VTOC –**Volume table of contents**

Directory Structure

A directory contains the information about files. Thus it is the central to the functioning of FS. To implement the mapping function, FS use a separate directory called user directory (UD),for files owned by each user. Assume that we have two User directories for the users A and B. When a program executed by user A performs the open call

Open (*virani*,..)
FS searches for the file virani in the UD of user **A**. if the statement was executed by some program executed by **B**; FS would have searched B's UD for *virani*. This arrangement ensures that the correct file would be accessed even if many files with identical names exist in the same system. FS uses the master directory to store the information concerning different UDs.

Syntax may be provided to refer to another user's file. For example an application program executed by user C may perform the open call to open A's directory.

Open (A→virani…)

FS can implement this by using A's directory, rather than C's directory, to search and locate the file *virani.* The directory entry of file should now contain the necessary protection the necessary protection information such that FS can determine the validity of such an open call.

File name	location info	protection Info	flags

Directory Hierarchies

The directory is group of files owned by a user. The grouping can be extended to permit grouping of files according to some user-defined criteria.
For example, a user engaged in many distinct activities may desire to group the files related to each activities separately. The FS can provide a tree structure of directories for this purpose.

Current and home directories:

At any moment of time, a user is said to be in a specific directory. This is called the current directory of the user. Any filename specified by the user is searched in this directory. While registering a user with the OS, a directory hierarchy is specified as his home directory. The OS puts a user into his home directory at login time.
A user may change his current directory to any other directory through a change directory command.
Example:
Change directory command of Unix

cd ..	change to the parent directory
cd vsc	directory vsc becomes the current directory

Access paths

A user referred to his own files by simply providing the file name.

Access path: " An Access path is a sequence of one or more access components separated by '/', each access component being a reference through a directory.

File protection

file sharing is achieved by permitting a user to traverse the FS directory hierarchy. Protection is implemented by using the *protection info field* in directory entry. The protection information can be stored in the form of an *access control list* , each element of the list being an access control pair

(<user_name>, <list of access_privileges>)

When a program executed by some user X tries to perform an operation <opn> on the file vsc, FS checks to see if this open is contained in the list of access privileges of X. The program is aborted if it is not in the list.
However , it is infeasible to use an access control pair for every user if a system contains a large number of users. To reduce the size of protection information, the users can be classified In some convenient manner an access control pair can be used for each user class rather than for each individual user.

The Unix operating system limits the size of the access control list of a file by dividing all users in the system into the following three classes:
Class 1: Owner of the file vsc

Class 2: Users in the same group as the owner of the file vsc
Class 3 : All other users in the system

The directory entry of vsc contains the user id's of its owner, and access privileges assigned to each user class.

Implementing File access

Data Structures

FCB data structures can are used in implementing a file access. Another key data structure is *active files table* (AFT). This table contains the description of all files, which are open at any moment of time.
AFT can be structured in two ways- it may contain pointers to the FCBs of all open files, or it may contain FCBs themselves.
To perform an operation on a file, its FCB has to be located. This involves a search in the AFT. To avoid repeated searches, FS notes the FCB address for a file, or simply the offset of its FCB in the AFT, when the file is opened . This is called the internal id of a file. . It is used to perform all operations on the file.

FS actions at open:

When a user program executes the call

open (<filename>..);

where <file name> is an access path for a file, FS determines two items of information:

1. Pointer to FCB of the directory containing the file(directory FCB pointer)
2. Internal id of the file

The internal id is passed back to the application program for use during file processing. The directory FCB pointer is used to update the directory while closing a newely created file. FS uses the following procedure to determine these items of information:

1. If the access path is absolute, locate the FCB of the FS root directory. Else locate the FCB of the current directory. Set a pointer called directory FCB pointer at this FCB.
2. (a) Search for the next component of the access path in the directory represented by directory FCB pointer. Give an error if the access path is not valid.
   (b) Create an FCB in a new entry of the AFT for the file described by the access path component.
   (c) Set a pointer called file FCB pointer at this FCb

(d) If more components exist in the access path, set directory FCB pointer= file FCB pointer and repeat the step 2
(e) Set internal of the file to the offset of file FCB pointer in AFT.

For an existing file, initialize its FCB using the information in the directory entry of the file.
Return the internal id of the file <filename> to the application program.

FS actions at a file operation

After opening a file a program performs some read or write operations on <filename>
This is translated into a call

<open>(internal_id, byte_id…)

where internal id is the internal_id of <filename> returned by the open call.

FS takes the following actions to process this call:

1. Locate the FCB of the file in the AFT using internal_id
2. Search the access control list of the file for the pair. Give an error if <open> does not exist in the list of access privileges for the user.
3. Determine the correct LIOC module for performing the operation <open>.

FS action at close

When application program executes the following statement:

close(internal_id,…);

FS performs the following actions:
1. If the file has been newly created or updated,
(a) If a newly created file, use directory FCB pointer to locate the FCB of the directory in which the file is to exist. Create an entry for the file in this directory.
(b) If an updated file, update the directory entry of the file to reflect the change in the size etc.
2. The FCB of the file and the FCB's of the directories containing it are erased from the AFT.

File Sharing

File protection controls the kinds of accesses users are permitted to make to a file. While file sharing determines the manner in which authorized users of a file may share a file and the manner in which the result of their file manipulations are visible to one another. Different file sharing modes have different implications for users and for the FS

Some popular file sharing modes are:

1. Sequential Sharing
2. Concurrent Sharing

We have a file vsc, which is to be shared by the application programs p1 and p2. In the sequential sharing, file accesses by p1 and p2 are spaced out over time that is only one program can access the file vsc at any point of time. Now a lock file is added to each directory entry. Setting and resetting of the lock at file open and close ensures that only one program can use the file at any time.
When programs share a file concurrently, it is essential to avoid mutual interference between them. Since an FCB contains the address of the next record to be processed, it is essential to create an FCB for each program. This can be achieved by following the procedure at every open of the file vsc. When vsc is shared in the immutable mode, none of the file sharing programs can modify it. This mode has the advantage that sharing programs are independent of one another- the order in which records of vsc are processed by p1 is immaterial to the execution of p2.
Two important issues arise in the case of mutable files:

1. Visibility of the files updates to other programs.
2. Interference between sharing programs

In single image immutable files, changes made by one program are immediately visible to the other programs.
Interference between p1 and p2 may arise if the set of records processed by them overlap e.g. both the program update the same record

In multiple image mutable files, many programs can concurrently update vsc. Each updating program creates a new version of vsc, which is distinct from other version created by concurrent program.

Distributed Operating System

Advantages:

Distributed Operating System provide the following advantages:

- Resource Sharing
- Reliability
- Computation Speed
- Communication
- Incremental growth

Resource Sharing:

It has been the traditional motivation for the distributed systems. The earliest form of a distributed system was a computer network, which enabled the use of specialized hardware and software resources by geographical distant users. Resource sharing continues to be an important aspect of distributed systems. Sharing of resources is useful in the local area networks (LAN). Thus low cost computers and workstations in an office can share some expensive resources like laser printer.

Reliability:

A distributed environment can be used to enhance the reliability of a system. There are two aspects for this- availability of a user resource and reliability of data. Availability of resource can be ensured through redundancy. For example availability of a disk can be increased by having two or more disks located at different sites in the system. Now if one disk is unavailable due to a disk failure a program can access other disk. Availability of a data resource e .g. a file can be ensured by keeping copies of the files at various sites in the system. The second aspect is data reliability, which implies guarding against corruption and loss of data. Basic Data reliability can be implemented by fault tolerance techniques like disk mirroring.

Computation speed up:

It implies to obtaining better response times or turn around time by distributing compution between different computers in a distributed system.

Communication:

Communication between users at different locations can be done using a distributed system. There are two aspects to communication: first users have unique id's. Their use in communication provides the security mechanism of the distributed system. Thus no separate authentication is needed. Second, use of a distributed system also implies continued availability of communication when users travel between different sites.

Incremental Growth:

Distributed systems are capable of incremental growth i.e. the capabilities of a system can be enhanced at a price proportional to the nature and size of

enhancement. A major advantage of this feature is that, enhancement need not to be planned in advance.

Definition and Examples

Definition: *"A distributed System is a system consisting of two or more nodes, each node being a computer system with its own memory, some communication hardware and a capability to perform some control functions of an OS."*

The node may have its own OS, or may perform certain control functions of a system wide OS. A node may perform all file system operations in a local file system, or may not perform any file system operations at all.

Examples:

ARPA-net was the first well-known computer network. It was set up by Defense Advanced Research Project Agency of USA. This network was an interconnection of different heterogeneous computer systems. A major emphasis of the Arpanet was on file sharing and transmission of messages over the network. Every Intermediate message processor (IMP) maintained routing tables, which contained information concerning alternative routes to a given destination. These tables were consulted while dispatching a message.