Unit 1: Introduction

Introduction: Components of System Software: Text editors, Loaders, Assemblers, Macro processors, Compilers, Debuggers. Machine Structure, Machine language and Assembly Language. Assemblers: General design procedure, design of two pass assembler

Que1. What is Software? Explain the difference between System software and Application software.

Software

Software is a set of instructions that allows the user to perform a well-defined function or some specific task.

Software is responsible for guiding all computer-related devices and instructing them regarding what and how the task is to be performed or executed. However, the software is made up of binary language (composed of ones and zeros), and for a programmer writing the binary code would be a slow and boring task. Therefore, software programmers write the software program in various human-readable languages such as Java, Python, C#, etc. and later use the source code.

Types of Software

Software's are generally classified into two types, i.e., **System Software and Application Software**.

System Software	**Application Software**
System Software maintains the system resources and gives the path for application software to run.	Application software is built for specific tasks.
Low-level languages are used to write the system software.	While high-level languages are used to write the application software.
Without system software, the system stops and can't run. System software runs when the	While Without application software system always runs. Application software runs as per the user's request.

system is turned on and stops when the system is turned off.	
The Software that is designed to control, integrate and manage the individual hardware components and application software is known as system software.	A set of computer programs installed in the user's system and designed to perform a specific task is known as application software.
The system software has no interaction with users. It serves as an interface between hardware and the end user.	Application software connects an intermediary between the user and the computer.
A system software operates the system in the background until the shutdown of the computer.	Application software runs in the front end according to the user's request.
Example: System software is an operating system, etc.	Example: Application software is Microsoft word, excel, Photoshop, VLC player, etc.

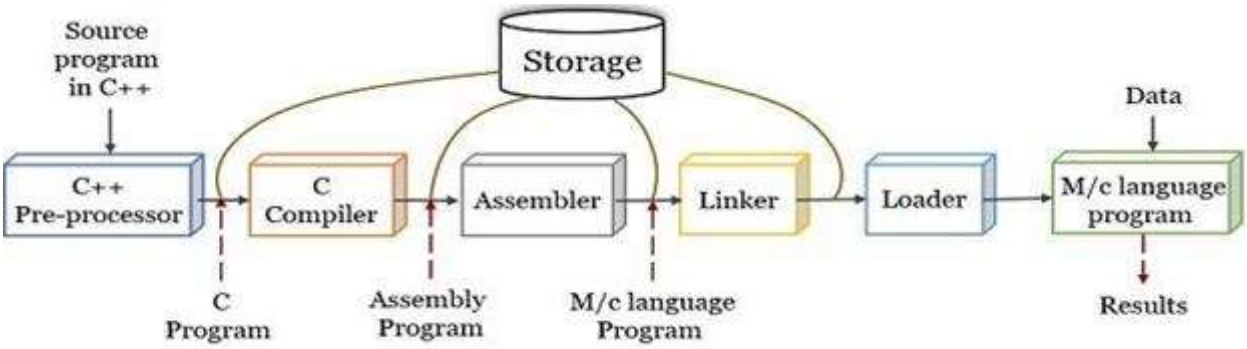
Que2. What is System software? Draw and describe the components of System software.

System software

System software is a computer program that helps the user to run computer hardware or software and manages the interaction between them. Basically, it is software that continuously runs in the computer background, maintaining the computer hardware and computer's basic functionalities, including the operating system, utility software, and interface. In simple terms, you can say that the system acts as a middle man that checks and facilitates the operations flowing between the user and the computer hardware.

System software is not only limited to the operating system. It also include the basic I/O system procedures, the boot program, assembler, computer device driver, etc. This software supports a high-speed platform to provide effective software for the other applications to work in smoothly manner. Therefore system software is a very essential part of our computer system. It is the first thing that gets loaded in the system's memory wherever you turn on your computer. System software is also known as "low-level software" because the end-users do not operate them.

Components of System software



1. **Text Editors:** These are fundamental tools for creating and modifying plain text files, including source code files written in various programming languages. They provide an interface for writing and editing program instructions.

2. **Loaders:** Loaders are responsible for taking machine language programs (executable files) and loading them into the computer's main memory, preparing them for execution by the Central Processing Unit (CPU). They resolve memory addresses and ensure the program is correctly placed in memory.

3. **Assemblers:** Assemblers translate programs written in assembly language into machine code. Assembly language is a low-level programming language that uses mnemonics to represent machine instructions, making it more human-readable than raw machine code.

4. **Macro Processors:** Macro processors are used to expand macro instructions within source code. A macro is a shorthand notation for a sequence of instructions, and the macro processor replaces these macros with their full definitions, simplifying code writing and improving readability.

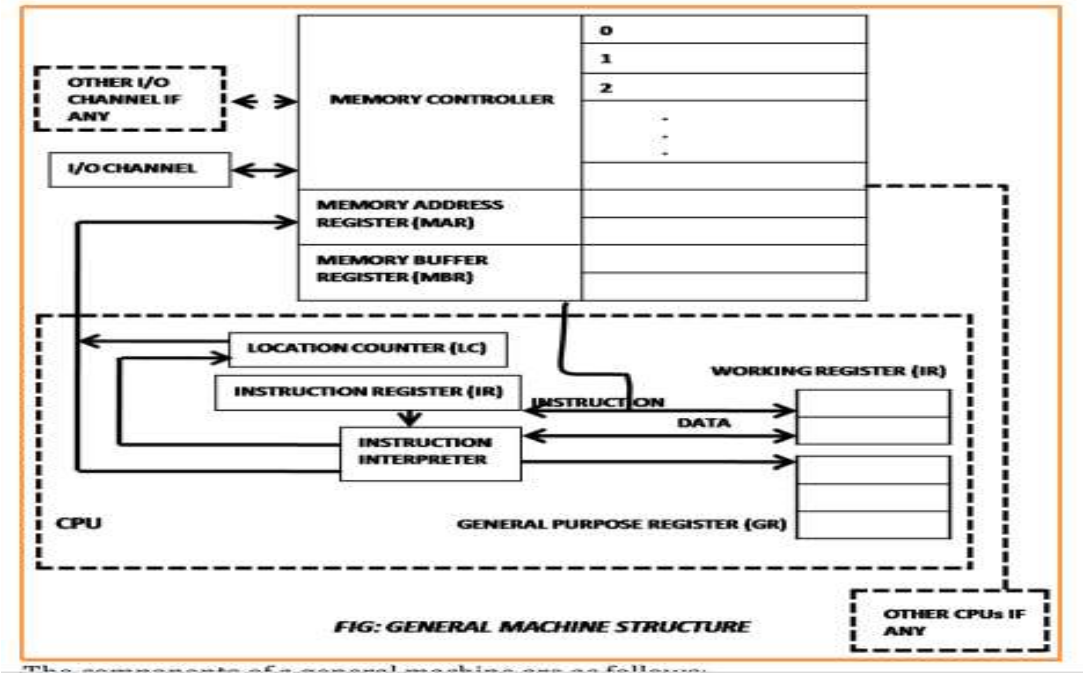
5. **Compilers:** Compilers translate programs written in high-level programming languages (like C++, Java, Python, etc.) into machine code or an intermediate representation that can then be

executed by the computer. They perform extensive analysis and optimization during this translation process.

6. **Debuggers:** Debuggers are essential tools for identifying and fixing errors (bugs) in computer programs. They allow programmers to step through code execution, inspect variable values, set breakpoints, and trace program flow to pinpoint the source of issues.

Que3. Explain in detail Machine Structure with a neat and clean diagram.

Machine Structure



The figure shows the general machine structure. All the conventional modern computers are based upon the concept of stored program computer, the model that was proposed by John von Neumann.

Instruction interpreter: A group of electronic circuits performs the intent of instruction of fetched from memory.

Location counter: LC otherwise called as program counter PC or instruction counter IC, is a hardware memory device which denotes the location of the current instruction being executed.

Instruction register: A copy of the content of the LC is stored in IR.

Working register: are the memory devices that serve as “scratch pad” for the instruction interpreter.

General register: are used by programmers as storage locations and for special functions.

Memory address register (MAR): contains the address of the memory location that is to read from or stored into.

Memory buffer register (MBR): contain a copy of the content of the memory location whose address is stored in MAR. The primary interface between the memory and the CPU is through memory buffer register.

Memory controller: is a hardware device whose work is to transfer the content of the MBR to the core memory location whose address is stored in MAR.

I/O channels: may be thought of as separate computers which interprets special instructions for inputting and outputting information from the memory.

Data and Instruction Flow:

- **Instruction Fetch:** The LC provides the address of the next instruction to the MAR, which then accesses memory via the Memory Controller. The instruction is fetched into the MBR and then transferred to the Instruction Register (IR).
- **Instruction Execution:** The Instruction Interpreter decodes the instruction in the IR. If the instruction requires data, the CPU accesses memory (using MAR and MBR) or uses data from General Purpose Registers (GR). Instructions and data can also flow between the CPU and I/O channels.
- **Data Processing:** Data is manipulated within the CPU, often involving the General Purpose Registers and Working Register (if distinct from IR), under the control of the Instruction Interpreter. Results can be stored back in memory or sent to I/O devices.

Que4. What is Assembler, Draw and explain? List and explain the types of statements of Assembler.

Assembler

An assembler is a type of computer program that takes in basic instructions and converts them into a pattern of bits that the computer's processor can use to perform basic operations. The assembler's job is to convert assembler or assembly language code into machine code that the computer can then read and execute.

In simple terms, an assembler is a piece of software that converts instructions written in assembly language to computer-readable machine code. It can also compile or convert low-level assembly language code into machine language code.

During the compilation process, the assembler assembles and converts assembly language source code into object code. Object code is written in a series of bits that the computer processor can then directly execute.

Types of Assembly Statements

1. Imperative statement

2. Declaration statement

3. Assembler directives

1. Imperative statement

- An imperative statement indicates an action to be performed during the execution of the assembled statement.
- Each imperative statement typically translates into one machine instruction.
- These are executable statements.
- Some example of imperative statement are given below.

MOVER BREG, X

2. Declaration Statements: syntax is as follows:

[Label] DS <constant>

[Label] DC '<value>'

- The DS (declare storage) statement reserves memory and associates names with them.
- Ex: A DS 1 ; reserves a memory area of 1 word, associating the name A to it

  G DS 200; reserves a block of 200 words & the name G is associated with the first word of the block (G+6 etc. to access the other words)
- The DC (declare constant) statement constructs memory words containing constants.
- Ex: ONE DC '1' ; associates name one with a memory word containing value 1

3. Assembler Directive

- Assembler directives instruct the assembler to perform certain action during the assembly program. For example

START

START <Constant>

This directive indicates that first word of machine should be placed in the memory word with address <constant>. Ex: START 500; First word of the target program is stored from memory location 500 onwards.

END

END <operand specification>

This directive indicates end of the source program. The operand indicates address of the instruction where the execution of program should begin. By default it is first instruction of the program. Execution control should transfer to label given in operand field.

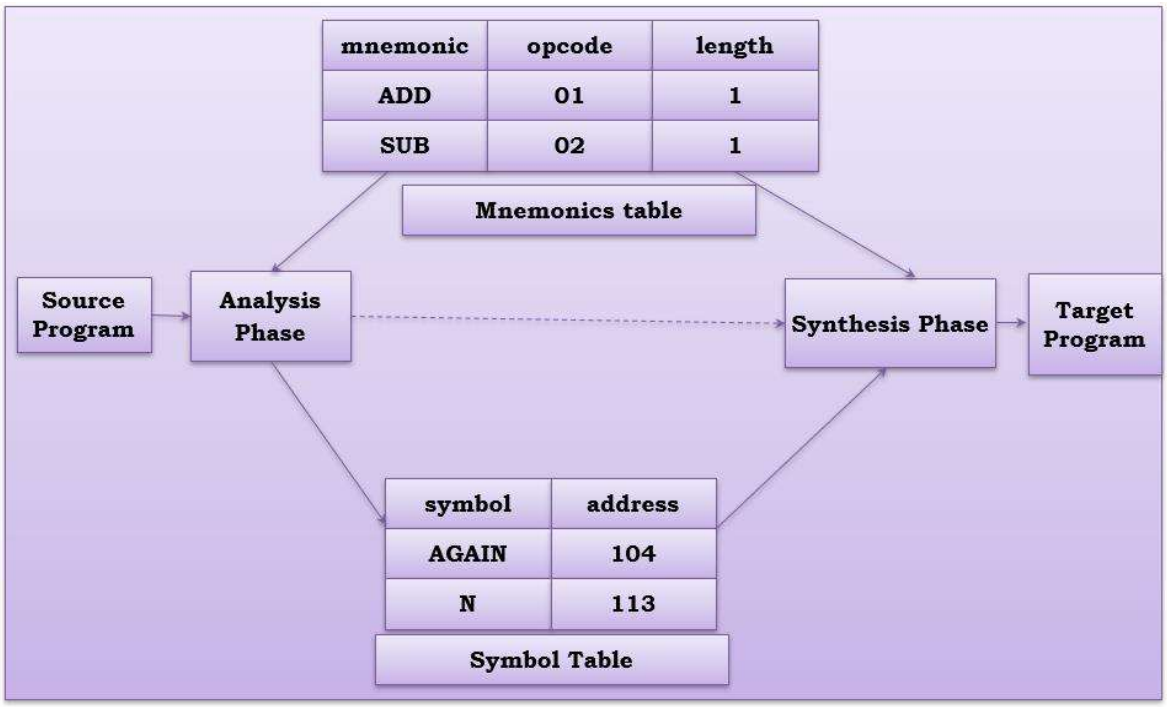
Que5. Describe in detail Analysis and Synthesis phase of Assembler? Draw design of Assembler.

Design of Assembler

Analysis Phase

- The primary function performed by the analysis phase is the building of the symbol table.
- For this purpose it must determine address of the symbolic name.
- It is possible to determine some address directly, however others must be inferred. And this function is called memory allocation.
- To implement memory allocation a data structure called location counter (LC) is used, it is initialized to the constant specified in the START statement.

The processing involved in maintaining the location counter is referred as LC processing.



Analysis Phase – Implementing memory allocation

LC (location counter): It is always made to contain the address of the next memory word in the target program. It is initialized to the constant specified at the START statement.

- When a LABEL is encountered,
    - It enters the LABEL and the contents of LC in a new entry of the symbol table.

    LABEL – e.g. N, AGAIN, SUM etc

    - It then finds the number of memory words required by the assembly statement and updates the LC contents
    - To update the contents of the LC, analysis phase needs to know lengths of the different instructions
    - This information is available in the Mnemonics table and is extended with a field called length

Synthesis Phase

Consider the assembly statement, **MOVER BREG, ONE.** We must have following information to synthesize the machine instruction corresponding to this statement:

Address of name ONE

Machine operation code corresponding to mnemonics MOVER.

- The first item of information depends on the source program; hence it must be available by analysis phase.
- The second item of information does not depend on the source program; it depends on the assembly language.

Two data structures are used during synthesis phase:

Symbol table: Each entry in symbol table has two primary field- name and address. This table is built by analysis phase.

Mnemonics table: An entry in mnemonics table has two primary field- mnemonics and op-code.

Synthesis phase uses these tables to obtain

- The machine address with which a name is associated.
- The machine op code corresponding to a mnemonic.

Unit 2: Macro Processor, Linker and Loader

Macro Processor: Macro instructions, Features of macro facility, Design of two-pass, single pass and nested macro processor. Loaders: Loader schemes: Compile and go, General Loader Scheme, Absolute loaders, subroutine linkages, relocating loaders, direct linking loaders, overlay structure. Design of an absolute loader, Design of direct linking loader. Linkers: Relocation and linking concepts, Design of linker, self-relocating programs, Static and dynamic link libraries, use of call back functions. Case Study: Loading phases using Java.

Que1. Define Macro. What are the advantages of macro facility? Explain the formal structure of Macro with example.

Macro

A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program. Macros are used to make a sequence of computing instructions available to the programmer as a single program statement, making the programming task less tedious and less error-prone. Macro instructions called macro are single-line abbreviations for groups of instructions.

For every occurrence of this one-line macro instruction within a program, the instruction must be replaced by the entire block.

The **advantages** of using macro are as follows:

- Simplify and reduce the amount of repetitive coding.
- Reduce the possibility of errors caused by repetitive coding.
- Make an assembly program more readable.

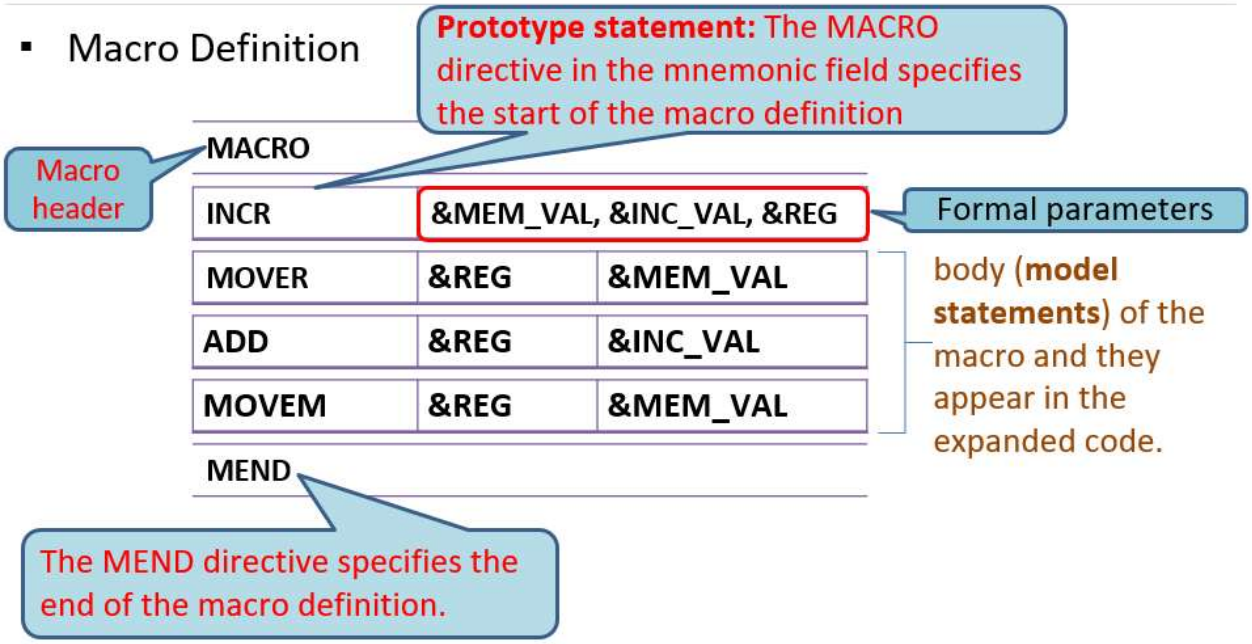
The formal structure of a macro includes the following features:

1. A Macro prototype statement: Specifies the name of the macro and name and type of formal parameters.

2. A Model statements: Specify the statements in the body of the macro from which assembly language statements are to be generated during expansion.

3. A Macro preprocessor statement: Specifies the statement used for performing auxiliary function during macro expansion

The syntax of a typical macro call can be of the following form:

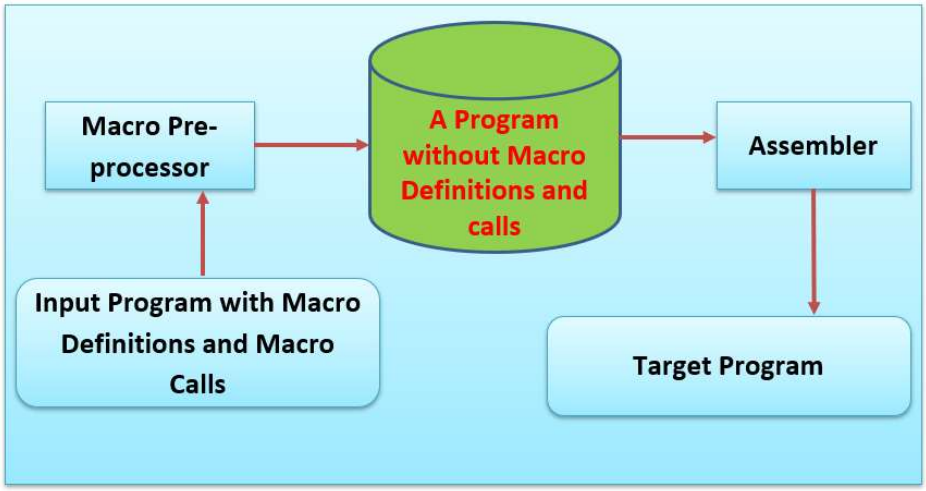


Que2. Draw and explain the Design of Macro pre-processor.

Design of Macro pre-processor

A macro preprocessor is a program that processes macro definitions and calls in a source program before its compiled or assembled. It essentially expands macro calls into their corresponding code sequences, making the code more concise and reusable. Key aspects of macro preprocessor design include handling macro definitions, expanding macro calls, and managing data structures to store macro information.

- Macro preprocessors are required for processing all programs that **contain** macro definitions and/or calls.
- Language translators such as assemblers and compilers **cannot** directly generate the target code from the programs containing definitions and calls for macros.



- Therefore, most language processing activities by assemblers and compilers preprocess these programs through **macro Pre-processors.**
- A macro preprocessor essentially accepts an assembly program with macro definitions and calls as its input.
- And processes it into an **equivalent expanded assembly program** with no macro definitions and calls.
- The macro preprocessor output program is **then passed over** to an assembler to generate the target object program.

Que3. What is Loader? List loader schemes and explain any one in detail with neat diagrammatical representation.

Que4. List and explain loader schemes.

Loaders

Loader is utility program which takes object code as input prepares it for execution and loads the executable code into the memory. Thus loader is actually responsible for initiating the execution process.

Functions of Loader: The loader is responsible for the activities such as allocation, linking, relocation and loading

1) It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.

2) It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.

3) There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.

4) Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.

The loader is responsible for the activities such as allocation, linking, relocation and loading

1. **Allocation:** allocating the space for program in the memory, by calculating the size of the program.
2. **Linking:** It resolves the symbolic references (code/data) between the object
3. **Relocation:** Address dependent locations in the program, such address constants must be adjusted according to allocated space
4. **Loading:** Physically places all the machine instructions and data into the memory

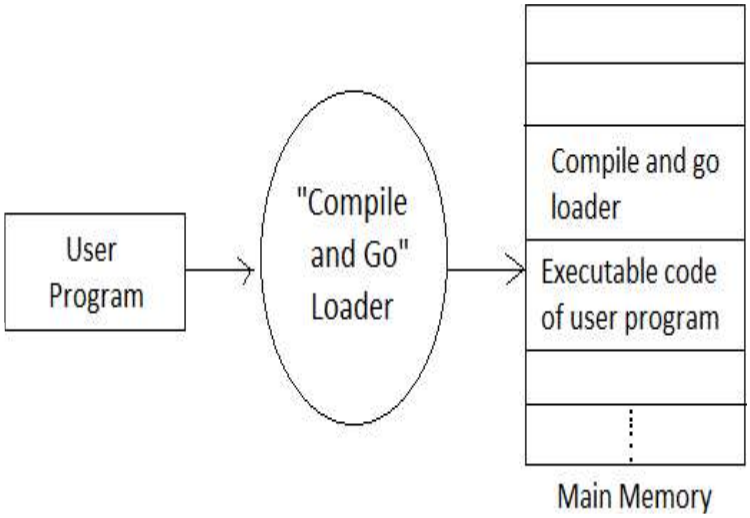
Different Loading Schemes

1. **“compile and go” loader**
2. **General Loader Scheme**
3. **Absolute Loader**
4. **Subroutine Linkage**
5. **Direct Linking Loaders**

1. Compile and go loader

In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one

part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter.



Advantages

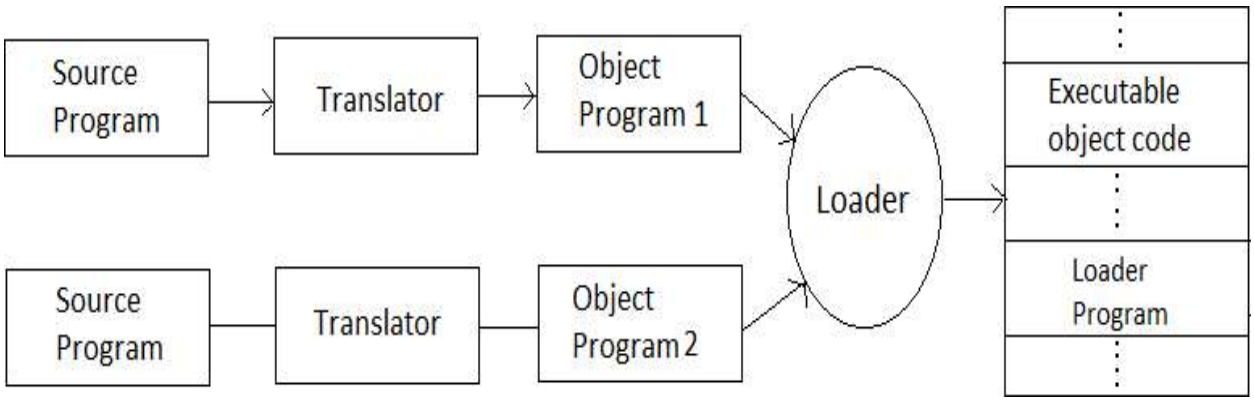
1. Easy to implement.
2. Program execution is sequential.

Disadvantages

1. Portion of memory is wasted because combination of assembler and loader activities, this combination program occupies large block of memory
2. There is no production of .obj file
3. It cannot handle multiple source programs or multiple programs written in different languages
4. The execution time will be more in this scheme as every time program is assembled and then executed

2. General Loader Scheme

In this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.



Advantages:

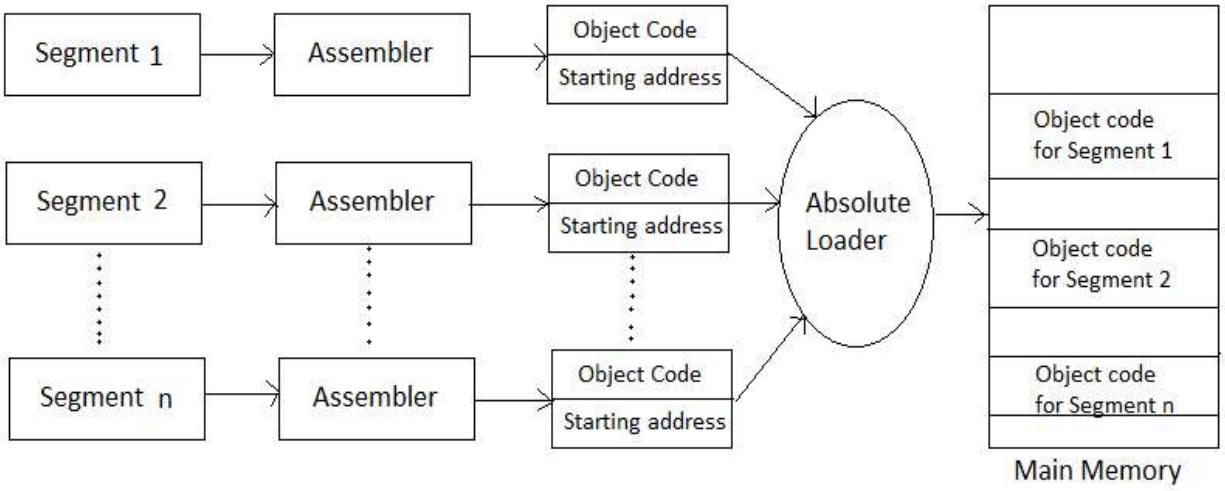
1. The program need not be retranslated each time while running it. Thus us because initially when source program gets executed and object program gets generated. If a program is not modified, then the loader can make use of this object program to convert it to executable form.
2. There is no wastage of memory because the assembler is not placed in the memory instead of it, loader occupies some portion of the memory. And the size of the loader is smaller than assembler so more memory is available to the user.
3. It is possible to write source program with multiple programs and multiple languages because the source program is first converted to an object program always and loader accepts these object modules to convert it to an executable format.

Disadvantages

1. If the program is modified it has to be retranslated.
2. Some portion of the memory is occupied by the loader.

3. Absolute Loader

The absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at a specified location in the memory.



This type of loader is called absolute loader because no relocating information is needed, rather it is obtained from the programmer or assembler.

The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file then the task of loader becomes very simple that is to simply place the executable form of the machine instructions at the locations mentioned in the object file.

In this scheme, the programmer or assembler should have knowledge of memory management. The programmer should take care of two things:

- Specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next modules, it's then the programmer's duty to make necessary changes in the starting address of respective modules.
- While branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.

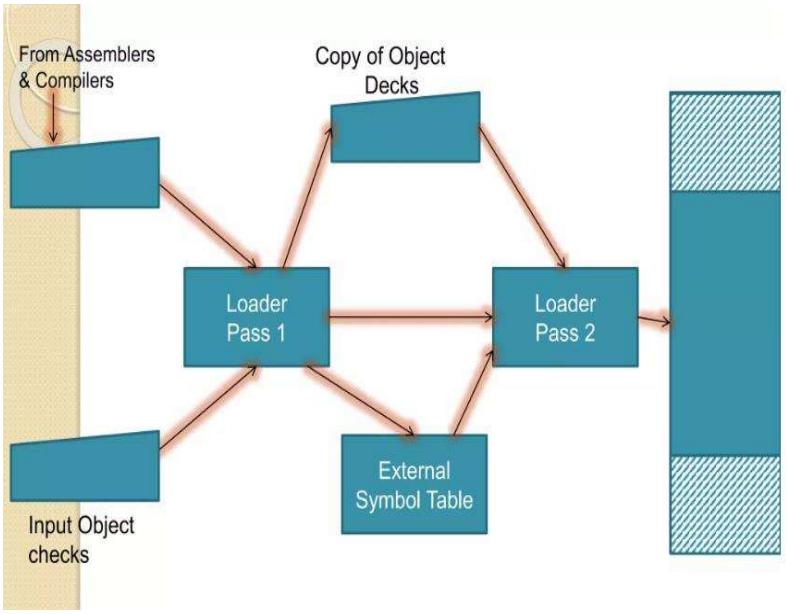
Que5. Draw and explain Direct linking loader concept.

The direct linking loader is the most common type of loader. The loader cannot have the direct access to the source code. The assembler should give the following information to the loader

1. The length of the object code segment

2. The list of all the symbols which are not defined in the current segment but can be used in the current segment.
3. The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table.



There are 4 types of cards available in the direct linking loader. They are

1. ESD-External symbol dictionary
2. TXT-card
3. RLD-Relocation and linking dictionary
4. END-card

1. ESD card: It contains information about all symbols that are defined in the program but reference somewhere, it contains:

- Reference number

- Symbol name
- Type Id
- Relative location
- Length

There are again ESD cards classified into 3 types of mnemonics. They are:

I. SD [Segment Definition]: It refers to the segment definition

II. LD; It refers to the local definition

III. ER: it refers to the external reference they are used in the [EXTRN] pseudo op code

2. TXT Card: It contains the actual information are text which are already translated.

3. RLD Card: This card contains information about location in the program whose contexts depends on the address at which the program is placed. In this we are used '+' and '–'sign, when we are using the '+' sign then no need of relocation, when we are using '-'sign relocation is necessary. The format of RLD contains:

- Reference number
- Symbol
- Flag
- Length
- Relative location

4. END Card: It indicates end of the object program.

Que6. What is Relocation in linker? Draw and explain the design of linker.

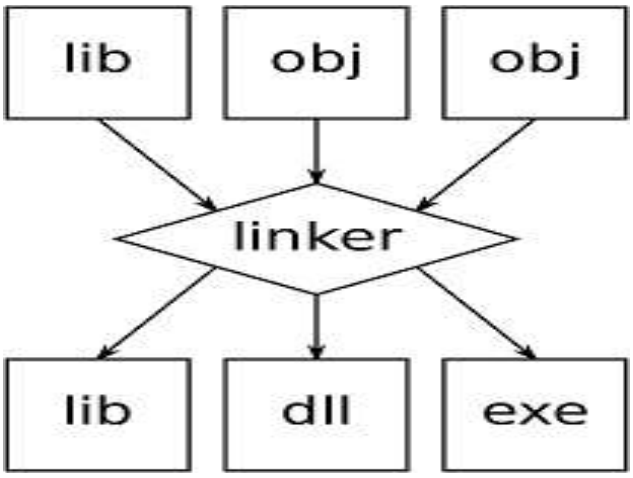
Relocation of Linking Concept

Relocation is the process of assigning load addresses for position-dependent code and data of a program and adjusting the code and data to reflect the assigned addresses. A linker usually

performs relocation in conjunction with symbol resolution, the process of searching files and libraries to replace symbolic references or names of libraries with actual usable addresses in memory before running a program. Relocation is typically done by the linker at link time, but it can also be done at load time by a relocating loader, or at run time by the running program itself.

Design of a Linker

A linker is a program in a system, also known as a link editor and binder, which combines object modules into a single object file. Generally, it is a program that performs the process of linking; it takes one or multiple object files, which are generated by compiler. And, then combines these files into an executable files. Modules are called for the different pieces of code, which are written in programming languages. Linking is a process that helps to gather and maintain a different piece of code into an executable file or single file. With the help of a linker, a specific module is also linked into the system library.



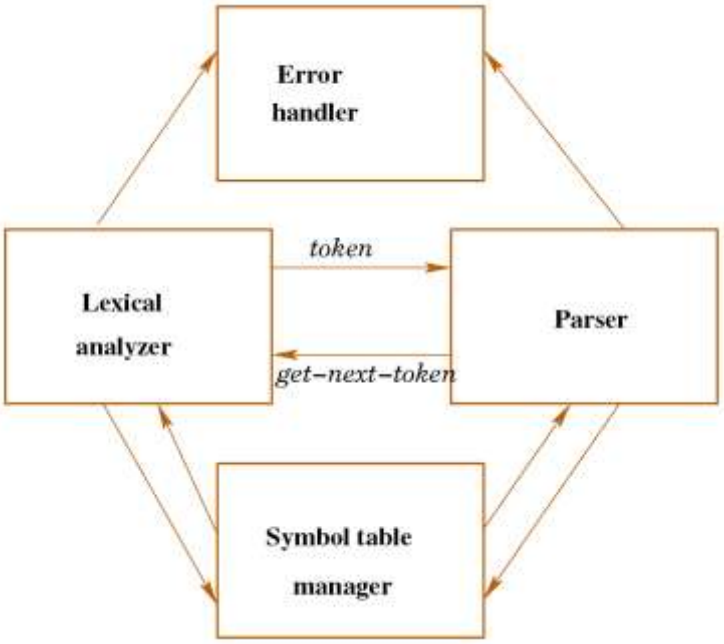
The primary function of the linker is to take objects from the assembler as input and create an executable file as output for the loader, as it helps to break down a large problem into a small module that simplifies the programming task. Usually, computer programs are made up of various modules in which all being a compiled computer programs and span separate object files.

Unit 3: Language Translator

Role of lexical analysis -parsing & token, patterns and Lexemes & Lexical Errors, regular definitions for the language constructs & strings, sequences, Comments & Transition diagram for recognition of tokens, reserved words and identifiers, examples Introduction to Compilers and Interpreters: General Model of Complier, Program interpretation, Comparison of compiler and Interpreter, Use of Interpreter and components of Interpreter. Case Study: Overview of LEX and YACC specification and features.

Que1. Explain the role of lexical analysis in compiler. Draw a neat and clean diagram.

Role of lexical analysis



The lexical analyzer is the first phase of compiler. A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner. A lexer often exists as a single function which is called by a parser or another function.

- Its main task is to read the input characters from the source Program and produces output a sequence of tokens that the parser uses for syntax analysis.
- To group them into lexemes

- Produce as output a sequence of tokens
- Group them into lexemes, produce as output a sequence of tokens
- Input for the syntactical analyzer, Interact with the symbol table, Insert identifier
- To strip out comments, whitespaces: blank, newline, tab, and other separators

Tokens, Patterns, and Lexemes

Tokens: Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

Patters: A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

Lexeme: A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Que2. Describe regular definitions for Regular expression, grammar, languages.

Regular Expressions

Regular expressions are symbolic notations used to define search patterns in strings. They describe regular languages and are commonly used in tasks such as validation, searching, and parsing.

A regular expression represents a regular language if it follows these rules:

1. ϵ (epsilon) is a regular expression representing the language $\{\epsilon\}$ (the empty string).
2. Any symbol 'a' from the input alphabet Σ is a regular expression, representing the language {a}.
3. Union (a + b) is a regular expression if a and b are regular expressions, representing the language {a, b}.
4. Concatenation (ab) is a regular expression if a and b are regular expressions.

Regular Grammar

A regular grammar is a formal grammar that generates regular languages. It consists of:

1. **Terminals:** Symbols that form strings (e.g., a, b).
2. **Non-terminals:** Variables used to derive strings (e.g., S, A).
3. **Production Rules:** Rules for transforming non-terminals into terminals or other non-terminals.
4. **Start Symbol:** The non-terminal from which derivations begin.

Regular Languages

Regular languages are the class of languages that can be represented using finite automata, regular expressions, or regular grammar. These languages have predictable patterns and are computationally efficient to recognize.

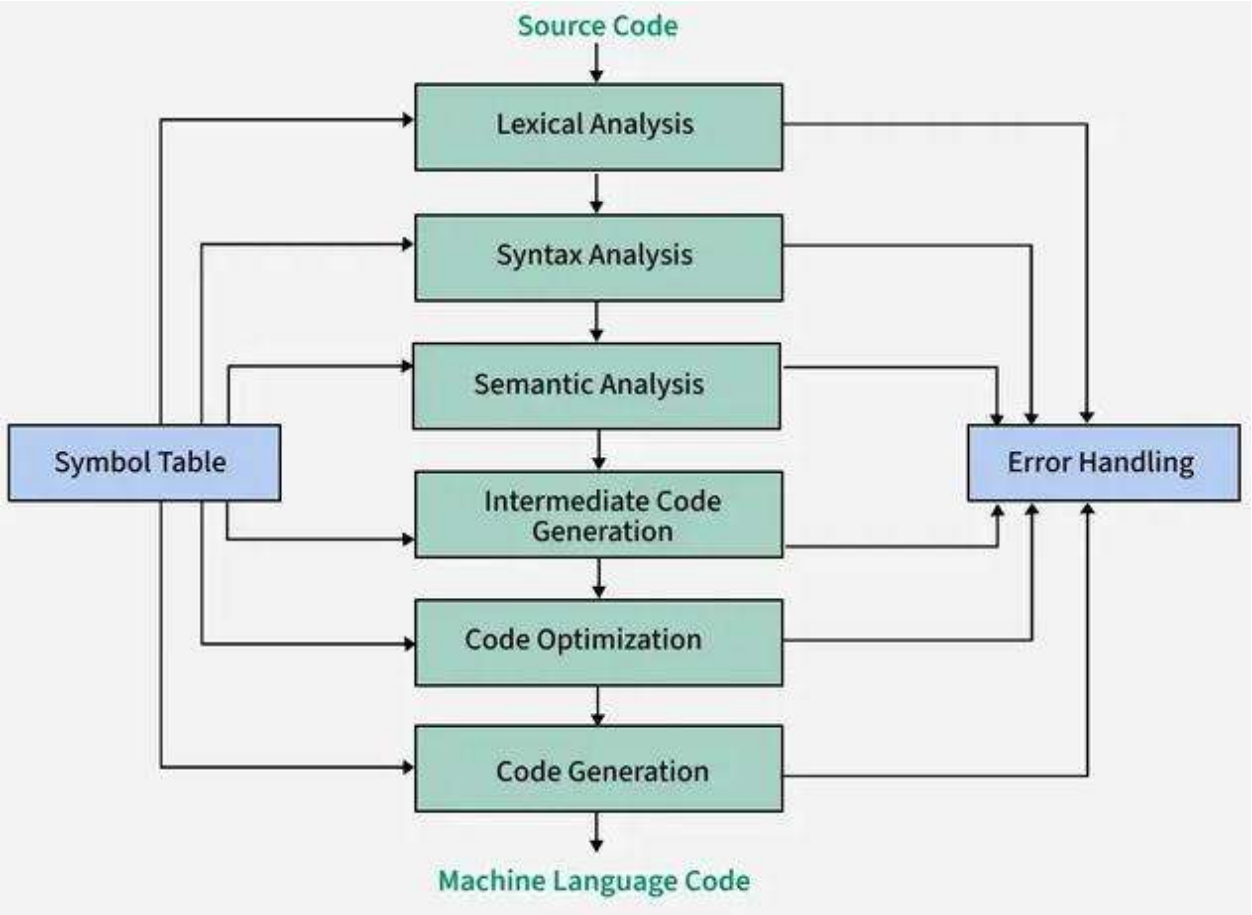
Que3. Explain the difference between compiler and Interpreter

Compiler	Interpreter
Compiler works on the complete program at once. It takes the entire program as input.	Interpreter program works line-by-line. It takes one statement at a time as input.
Compiler generates intermediate code, called the object code or machine code	Interpreter does not generate intermediate object code or machine code.
Compiler executes conditional control statements (like if-else and switch-case) and logical constructs faster than interpreter.	Interpreter execute conditional control statements at a much slower speed.
Compiled programs take more memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, interpreted programs are more memory efficient.
Compile once and run anytime. Compiled program does not need to be compiled every time. Errors are reported after the entire program is checked for syntactical and other errors.	Interpreted programs are interpreted line-by-line every time they are run. Error is reported as soon as the first error is encountered. Rest of the program will not be checked until the existing error is removed.
Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.

Examples of programming languages that use compilers: C, C++, COBOL.	Examples of programming languages that use interpreters: Python, Ruby, PHP, Perl.

Que4. What is compiler? Draw and explain the phases of compiler.

A compiler is a software tool that converts high-level programming code into machine code that a computer can understand and execute. It acts as a bridge between human-readable code and machine-level instructions, enabling efficient program execution. The process of compilation is divided into six phases:



1. Lexical Analysis

Lexical analysis is the first phase of a compiler, responsible for converting the raw source code into a sequence of tokens. A token is the smallest unit of meaningful data in a programming

language. Lexical analysis involves scanning the source code, recognizing patterns, and categorizing groups of characters into distinct tokens.

The lexical analyzer scans the source code character by character, grouping these characters into meaningful units (tokens) based on the language's syntax rules. These tokens can represent keywords, identifiers, constants, operators, or punctuation marks. By converting the source code into tokens, lexical analysis simplifies the process of understanding and processing the code in later stages of compilation.

2. Syntax Analysis

Syntax analysis, also known as parsing, is the second phase of a compiler where the structure of the source code is checked. This phase ensures that the code follows the correct grammatical rules of the programming language.

The role of syntax analysis is to verify that the sequence of tokens produced by the lexical analyzer is arranged in a valid way according to the language's syntax. It checks whether the code adheres to the language's rules, such as correct use of operators, keywords, and parentheses. If the source code is not structured correctly, the syntax analyzer will generate errors.

3. Semantic Analysis

Semantic analysis is the phase of the compiler that ensures the source code makes sense logically. It goes beyond the syntax of the code and checks whether the program has any semantic errors, such as type mismatches or undeclared variables. Semantic analysis checks the meaning of the program by validating that the operations performed in the code are logically correct. This phase ensures that the source code follows the rules of the programming language in terms of its logic and data usage.

4. Intermediate Code Generation

Intermediate code is a form of code that lies between the high-level source code and the final machine code. It is not specific to any particular machine, making it portable and easier to optimize. Intermediate code acts as a bridge, simplifying the process of converting source code into executable code. The use of intermediate code plays a crucial role in optimizing the program before it is turned into machine code.

5. Code Optimization

Code Optimization is the process of improving the intermediate or target code to make the program run faster, use less memory, or be more efficient, without altering its functionality. It involves techniques like removing unnecessary computations, reducing redundancy, and reorganizing code to achieve better performance.

6. Code Generation

Code Generation is the final phase of a compiler, where the intermediate representation of the source program (e.g., three-address code or abstract syntax tree) is translated into machine code or assembly code. This machine code is specific to the target platform and can be executed directly by the hardware.

The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

Symbol Table - It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

Error Handling in Phases of Compiler: Error Handling refers to the mechanism in each phase of the compiler to detect, report and recover from errors without terminating the entire compilation process.

Que5. What is need of Interpreter? List and explain components of Interpreter.

Need of Interpreter

The software by which the conversion of the high-level instructions is performed line-by-line to machine-level language, other than compiler and assembler, is known as **Interpreter**. The

interpreter in the compiler checks the source code line-by-line and if an error is found on any line, it stops the execution until the error is resolved.

The first and vital need of an interpreter is to translate source code from high-level language to machine language. However, we already had the compiler to serve the purpose, the compiler is a very powerful tool for developing programs in a high-level language. However, there are several demerits associated with the compiler. If the source code is huge in size, then it might take hours to compile the source code, which will significantly increase the compilation duration. Here, the Interpreter plays its role. The interpreter can cut this huge compilation duration. As they are designed to translate single instruction at a time and execute them immediately. So instead of waiting for the entire code, the interpreter translates a single line and executes it.

Components of Interpreter

- **Lexer/Tokenizer:** This component breaks down the source code into a sequence of tokens, which are the basic building blocks of the language (like keywords, identifiers, operators, etc.).
- **Parser:** The parser takes the token stream from the lexer and constructs an Abstract Syntax Tree (AST) (or parse tree) that represents the structure of the code. It ensures the code follows the syntax rules of the language.
- **Evaluator/Executor:** This component takes the AST or an intermediate representation of the code and executes the instructions, performing the necessary computations and operations.
- **Context:** In some interpreters, a context object stores global information needed for interpretation, such as variables and their values.

Que5. What is LEX in Compiler Design? Explain LEX file format in detail.

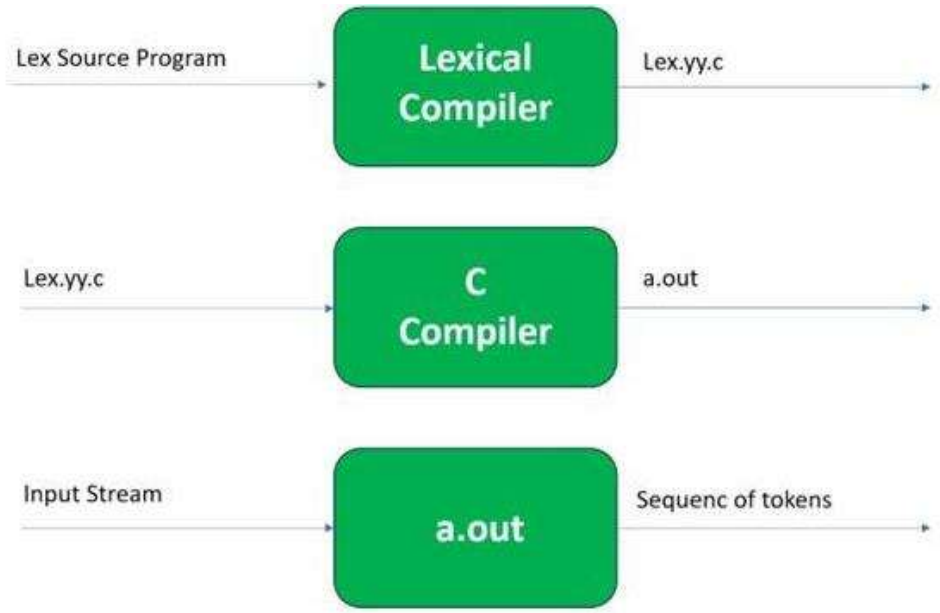
LEX in Compiler Design

Whenever a developer wants to make any software application they write the code in a high-level language. That code is not understood by the machine so it is converted into low-level machine-understandable code by the compiler. Lex is an important part of this compiler and is responsible for the classification of the generated tokens based on their purpose. In this article, we will understand what Lex in Compiler Design is but before understanding Lex we have to

understand what Lexical Analysis is. Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC (Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.

Function of Lex

1. In the first step the source code which is in the Lex language having the file name 'File.l' gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.
2. After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an 'a.out' file, and finally, the output file a.out will take the stream of character and generates tokens as output.



LEX File Format

A Lex program consists of three parts and is separated by % delimiters:-

Declarations

%%

Translation rules

%%

Auxiliary procedures

Declarations: The declarations include declarations of variables.

Transition rules: These rules consist of Pattern and Action.

Auxiliary procedures: The Auxilary section holds auxiliary functions used in the actions.

Que6. What is YACC in Compiler Design, Describe YACC file format?

YACC in Compiler Design

YACC is an LALR parser generator developed at the beginning of the 1970s by Stephen C. Johnson for the Unix operating system. It automatically generates the LALR(1) parsers from formal grammar specifications. YACC plays an important role in compiler and interpreter development since it provides a means to specify the grammar of a language and to produce parsers that either interpret or compile code written in that language.

Key Concepts and Features of YACC

- **Grammar Specification:** The input to YACC is a context-free grammar (usually in the Backus-Naur Form, BNF) that describes the syntax rules of the language it parses.
- **Parser Generation:** YACC translates the grammar into a C function that could perform an efficient parsing of input text according to such predefined rules.
- **LALR(1) Parsing:** This is a bottom-up parsing method that makes use of a single token lookahead in determining the next action of parsing.
- **Semantic Actions:** These are the grammar productions that are associated with an action; this enables the execution of code, usually in C, used in the construction of abstract syntax trees, the generation of intermediate representations, or error handling.
- **Attribute Grammars:** These grammars consist of non-terminal grammar symbols with attributes, which through semantic actions are used in the construction of parse trees or the output of code.
- **Integration with Lex:** It is often used along with Lex, a tool that generates lexical analyzers-scanners-which breaks input into tokens that are then processed by the YACC parser.

YACC File Format

Input File: YACC input file is divided into three parts.

/* definitions */

....

%%

/* rules */

....

%%

/* auxiliary routines */

....

Definition Part:

- The definition part includes information about the tokens used in the syntax. Yacc automatically assigns numbers for tokens, but it can be overridden by token.
- The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.

Rule Part:

- The rules part contains grammar definitions in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in the rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Unit 4: Operating Systems

Operating Systems: Introduction to different types of operating Real Time Operating Systems, System Components, OS services, System structure- Layered Approach. Process Management: Process Concept- Process states, Process control block, Threads, Process Scheduling: Types of process schedulers, Types of scheduling: Preemptive, Non preemptive. Scheduling algorithms: FCFS, SJF, RR and Priority, Deadlocks: Methods of handling deadlocks, Deadlock prevention, avoidance and detection, Recovery from deadlocks. Case Study: Process Management in multi-cores OS.

Que1. What is Operating System? List and explain in short different types of Operating Systems.

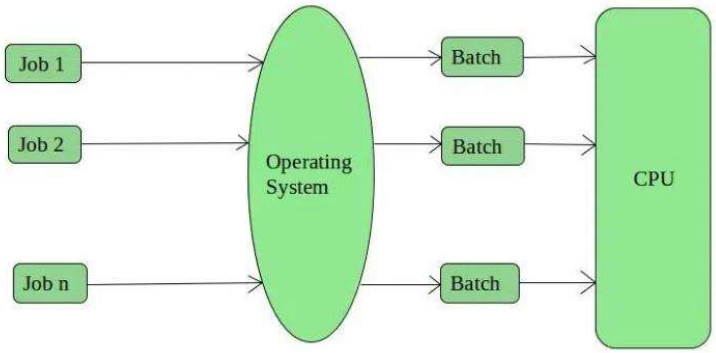
Operating System

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs conveniently and efficiently. An operating system is software that manages computer hardware and software. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

The operating system (OS) is a program that runs at all times on a computer. All other programs, including application programs, run on top of the operating system. It does assignment of resources like memory, processors and input / output devices to different processes that need the resources. The assignment of resources has to be fair and secure.

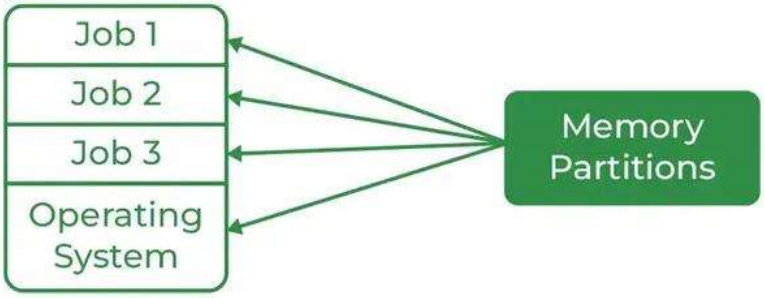
1. Batch Operating System

A Batch Operating System is designed to handle large groups of similar jobs efficiently. It does not interact with the computer directly but instead processes jobs that are grouped by an operator. These jobs are queued and executed one after the other, without user interaction during the process.



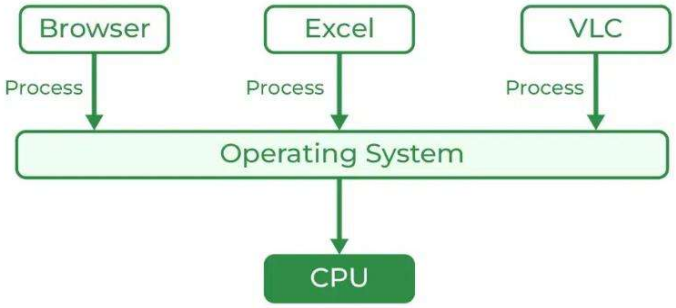
2. Multi-Programming Operating System

In a Multi-Programming Operating System, multiple programs run in memory at the same time. The CPU switches between programs, utilizing its resources more effectively and improving overall system performance.



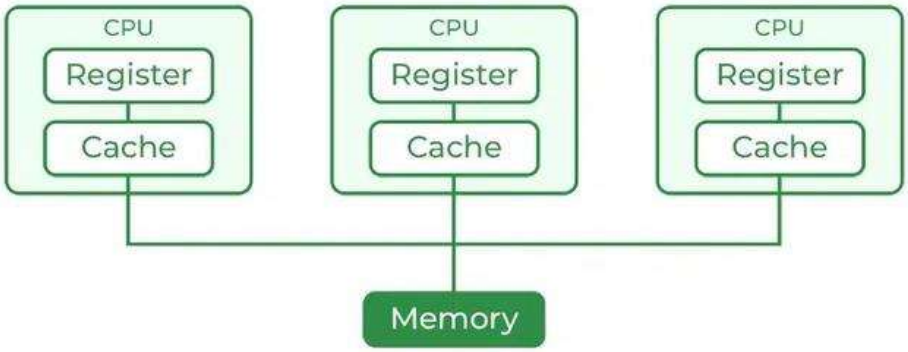
3. Multi-tasking/Time-sharing Operating systems

Multitasking OS is a type of Multiprogramming system with every process running in round robin manner. Each task is given some time to execute so that all the tasks work smoothly. Each user gets the time of the CPU as they use a single system. These systems are also known as Multitasking Systems. The task can be from a single user or different users. The time that each task gets to execute is called quantum. After this time interval is over, the OS switches over to the next task.



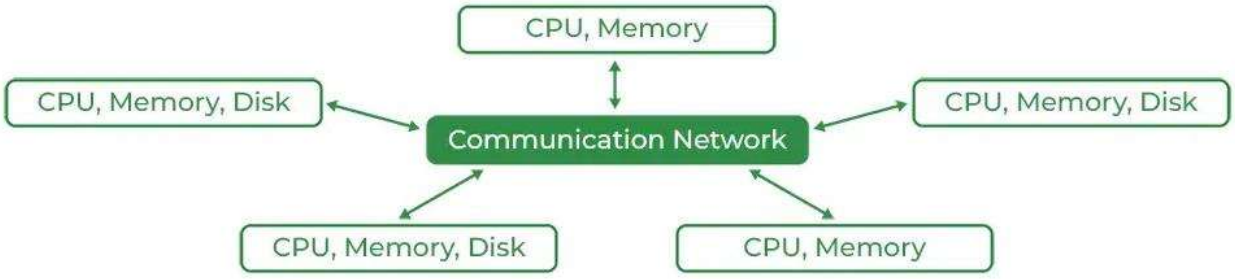
4. Multi-Processing Operating System

A Multi-Processing Operating System is a type of Operating System in which more than one CPU is used for the execution of resources. It betters the throughput of the System.



5. Distributed Operating System

Distributed operating systems are a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, at a great pace. Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU. Systems. These systems' processors differ in size and function. The major benefit of working with these types of operating systems is that it is always possible that one user can access the files or software which are not present on his system but on some other system connected within this network, i.e., remote access is enabled within the devices connected to that network.



Que2. What is process? What is process management? Draw and explain process state diagram.

Process

A process is a program in execution and it is more than a program code called as text section and this concept works under all the operating system because all the task perform by the operating

system needs a process to perform the task. The process executes when it changes the state. The state of a process is defined by the current activity of the process.

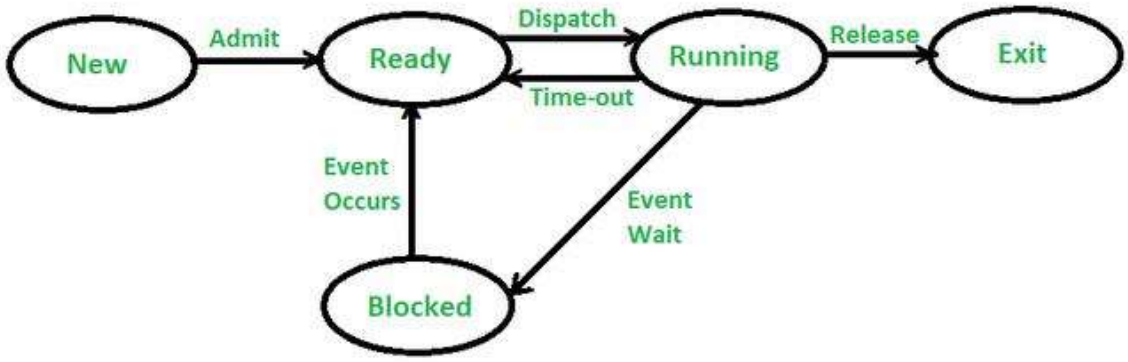
Process Management

Process Management for a single tasking or batch processing system is easy as only one process is active at a time. With multiple processes (multiprogramming or multitasking) being active, the process management becomes complex as a CPU needs to be efficiently utilized by multiple processes. Multiple active processes can may share resources like memory and may communicate with each other. This further makes things complex as an Operating System has to do process synchronization.

Process State Diagram.

The Five-State Model

The **five-state process lifecycle** is an expanded version of the two-state model. The two-state model works well when all processes in the **not running** state are ready to run. However, in some operating systems, a process may not be able to run because it is waiting for something, like input or data from an external device. To handle this situation better, the **not running state** is divided into two separate states:



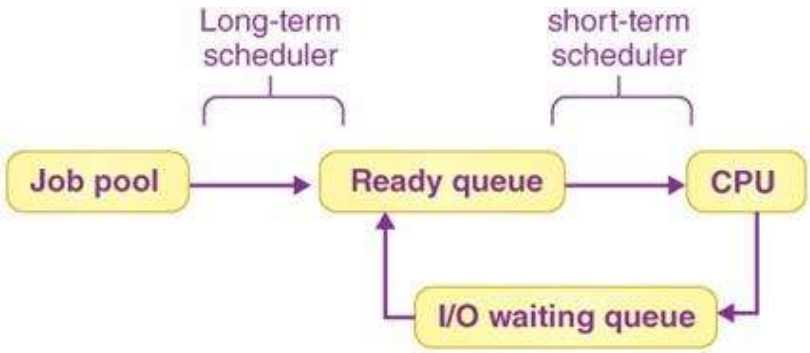
- **New:** This state represents a newly created process that hasn't started running yet. It has not been loaded into the main memory, but its process control block (PCB) has been created, which holds important information about the process.
- **Ready:** A process in this state is ready to run as soon as the CPU becomes available. It is waiting for the operating system to give it a chance to execute.

- **Running**: This state means the process is currently being executed by the CPU. Since we're assuming there is only one CPU, at any time, only one process can be in this state.
- **Blocked/Waiting:** This state means the process cannot continue executing right now. It is waiting for some event to happen, like the completion of an input/output operation (for example, reading data from a disk).
- **Exit/Terminate:** A process in this state has finished its execution or has been stopped by the user for some reason. At this point, it is released by the operating system and removed from memory.

Que3. Describe the types of process schedulers with neat and clean diagram.

Types of process schedulers

Process schedulers are divided into three categories.



1. Long-Term Scheduler or Job Scheduler

The job scheduler is another name for Long-Term scheduler. It selects processes from the pool (or the secondary memory) and then maintains them in the primary memory's ready queue. The Multiprogramming degree is mostly controlled by the Long-Term Scheduler. The goal of the Long-Term scheduler is to select the best mix of IO and CPU bound processes from the pool of jobs.

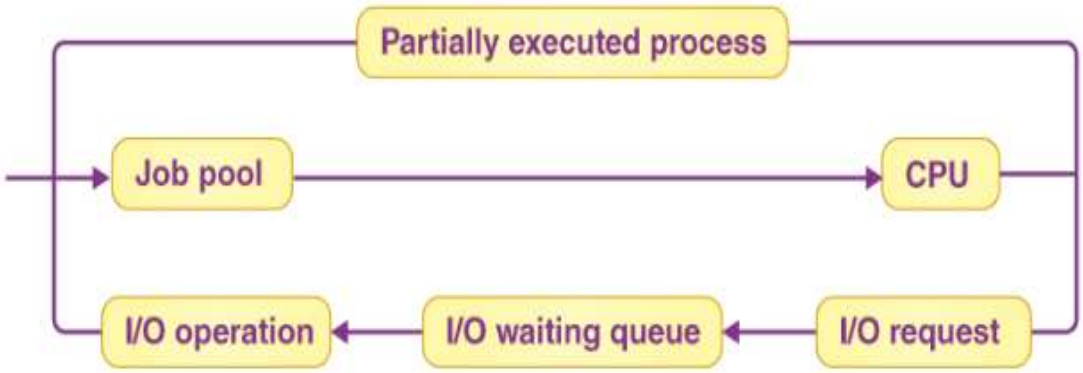
If the job scheduler selects more IO bound processes, all of the jobs may become stuck, the CPU will be idle for the majority of the time, and multiprogramming will be reduced as a result. Hence, the Long-Term scheduler's job is crucial and could have a Long-Term impact on the system.

2. Short-Term Scheduler or CPU Scheduler

CPU scheduler is another name for Short-Term scheduler. It chooses one job from the ready queue and then sends it to the CPU for processing.

To determine which work will be dispatched for execution, a scheduling method is utilised. The Short-Term scheduler's task can be essential in the sense that if it chooses a job with a long CPU burst time, all subsequent jobs will have to wait in a ready queue for a long period. This is known as hunger, and it can occur if the Short-Term scheduler makes a mistake when selecting the work.

3. Medium-Term Scheduler



The switched-out processes are handled by the Medium-Term scheduler. If the running state processes require some IO time to complete, the state must be changed from running to waiting.

This is accomplished using a Medium-Term scheduler. It stops the process from executing in order to make space for other processes. Swapped out processes are examples of this, and the operation is known as swapping. The Medium-Term scheduler here is in charge of stopping and starting processes.

Que4. List and explain the types of CPU scheduling algorithms.

CPU scheduling algorithms

1. First Come First Served (FCFS) scheduling

2. Shortest Job First (SJF) scheduling

3. Round Robin (RR) scheduling

4. Priority scheduling

1. First Come First Served (FCFS) scheduling

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

2. Shortest Job First (SJF) scheduling

- This is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

3. Round Robin (RR) scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

4. Priority scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.

- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Que5. What is Deadlock? What are the Methods for Handling Deadlock in an Operating System?

Deadlock

A deadlock occurs when a process or a collection of processes is stalled while waiting for a resource held by another waiting process. It is an un-favorable state of affairs. If a system encounters a deadlock, there are a few methods that can be used to handle it.

Methods for Handling Deadlock

1. Deadlock Prevention

Deadlock happens whenever four conditions occur at the same time. If one of the four conditions can be interrupted, we can prevent the system from becoming stuck in a deadlock. This mechanism exists for a straightforward reason. To put it another way, we only need to fail in one of the requirements. It's critical to avoid a given deadlock before it happens. As a result, the system verifies each and every transaction before executing it to ensure that it does not cause a deadlock. If a transaction has even the slightest chance of causing a deadlock, it is then not allowed to run at all. Read more on deadlock prevention here.

2. Deadlock Detection

The resource scheduler can detect deadlock because it maintains track of all the resources allotted to different processes. When a deadlock is found, it can be resolved using the methods provided.

- All of the processes implicated in the deadlock are shut down. This strategy is ineffective since all of the processes' progress is lost.
- Some processes' resources can be diverted to others till the deadlock condition is addressed.

We can check for deadlocks in the system on a regular basis in deadlock detection and recovery. But if there is a deadlock in our system, we use several techniques to break the deadlock. Deadlock

detection is accomplished by using an algorithm that tracks cyclic waiting and kills one or more processes in order to break the deadlock.

3. Deadlock Avoidance

It is another method for dealing with a deadlock. The OS verifies/examines the system state in this manner, which means it checks whether the system is in a safe or hazardous condition at each step. This procedure is repeated until the system is in safe mode. In the event that the system becomes dangerous, the operating system will take a step backwards.

Que6. What is Deadlock Prevention and Avoidance? List and explain Necessary Conditions for Deadlock

Deadlock prevention and avoidance are strategies used in computer systems to ensure that different processes can run smoothly without getting stuck waiting for each other forever. Think of it like a traffic system where cars (processes) must move through intersections (resources) without getting into a gridlock.

Necessary Conditions for Deadlock

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Deadlock Prevention

We can prevent a Deadlock by eliminating any of the above four conditions.

1. Eliminate Mutual Exclusion

It is not possible to violate mutual exclusion because some resources, such as the tape drive, are inherently non-shareable. For other resources, like printers, we can use a technique called Spooling (Simultaneous Peripheral Operations Online).

In spooling, when multiple processes request the printer, their jobs (instructions of the processes that require printer access) are added to the queue in the spooler directory. The printer is allocated

to jobs on a First-Come, First-Served (FCFS) basis. In this way, a process does not have to wait for the printer and can continue its work after adding its job to the queue.

2. Eliminate Hold and Wait

Hold and wait is a condition in which a process holds one resource while simultaneously waiting for another resource that is being held by a different process. The process cannot continue until it gets all the required resources.

There are two ways to eliminate hold and wait:

- **By eliminating wait**: The process specifies the resources it requires in advance so that it does not have to wait for allocation after execution starts. For Example, Process1 declares in advance that it requires both Resource1 and Resource2.
- **By eliminating hold**: The process has to release all resources it is currently holding before making a new request. For Example: Process1 must release Resource2 and Resource3 before requesting Resource1.

3. Eliminate No Preemption

Preemption is temporarily interrupting an executing task and later resuming it. Two ways to eliminate No Preemption:

- **Processes must release resources voluntarily**: A process should only give up resources it holds when it completes its task or no longer needs them.
- **Avoid partial allocation**: Allocate all required resources to a process at once before it begins execution. If not all resources are available, the process must wait.

4. Eliminate Circular Wait

To eliminate circular wait for deadlock prevention, we can use **order on resource allocation**.

- Assign a unique number to each resource.
- Processes can only request resources in an increasing order of their numbers.

This prevents circular chains of processes waiting for resources, as no process can request a resource lower than what it already holds.

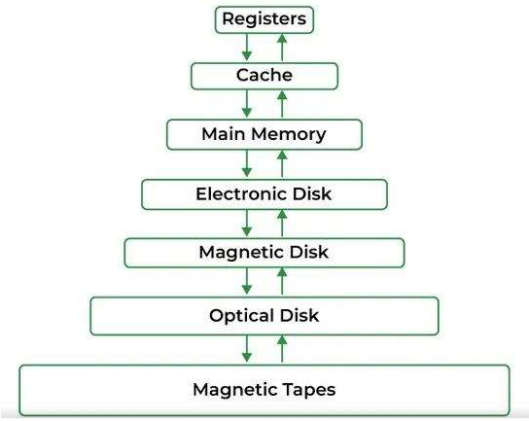
Unit 5: Memory Management

Memory management: Review of Programming Model of Intel 80386, Contiguous and non-contiguous, Swapping, Paging, Segmentation, Segmentation with Paging. Virtual Memory: Background, Demand paging, Page replacement scheme- FIFO, LRU, Optimal, Thrashing. Case Study: Memory Management in multi-cores OS

Que1. What is Memory Management, explain with diagram? Why Memory Management is required? What is Logical and Physical Address Space in memory?

Memory Management

Memory is a hardware component that stores data, instructions and information temporarily or permanently for processing. It consists of an array of bytes or words, each with a unique address. Memory holds both input data and program instructions needed for the CPU to execute tasks. Memory works closely with the CPU to provide quick access to data being used.



Memory management is a critical aspect of operating systems that ensures efficient use of the computer's memory resources. It controls how memory is allocated and de-allocated to processes, which is key to both performance and stability.

Need of Memory Management

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.

- To maintain data integrity while executing of process.

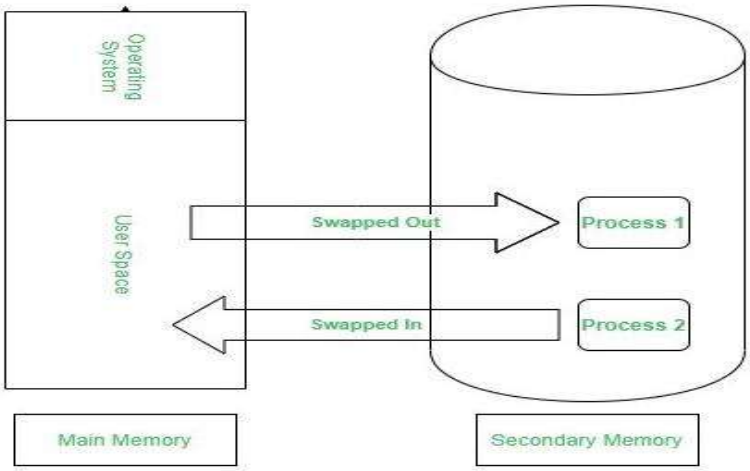
Logical and Physical Address Space in memory

- **Logical Address Space:** An address generated by the CPU is known as a "Logical Address". It is also known as a Virtual address. Logical address space can be defined as the size of the process. A logical address can be changed.
- **Physical Address Space:** It refers to the set of actual addresses used by the memory hardware. A physical address, also called a real address, is generated by the Memory Management Unit (MMU) through run-time mapping of virtual addresses. Unlike virtual addresses, physical addresses remain constant.

Que2. Describe the concept of Swapping in the Operating System with neat and clean diagram? Explain advantages and disadvantages of swapping.

Swapping

Swapping in an operating system is a process that moves data or programs between the computer's main memory (RAM) and a secondary storage (usually a hard disk or SSD). This helps manage the limited space in RAM and allows the system to run more programs than it could otherwise handle simultaneously. It's important to remember that swapping is only used when data isn't available in RAM. Although the swapping process degrades system performance, it allows larger and multiple processes to run concurrently. Because of this, swapping is also known as memory compaction.



The CPU scheduler determines which processes are swapped in and which are swapped out. Consider a multiprogramming environment that employs a priority-based scheduling algorithm. When a high-priority process enters the input queue, a low-priority process is swapped out so the high-priority process can be loaded and executed. When this process terminates, the low-priority process is swapped back into memory to continue its execution. The below figure shows the swapping process in the operating system:

Swapping has been subdivided into two concepts: swap-in and swap-out.

- Swap-out is a technique for moving a process from RAM to the hard disc.
- Swap-in is a method of transferring a program from a hard disc to main memory, or RAM.

Process of Swapping

- When the RAM is full and a new program needs to run, the operating system selects a program or data that is currently in RAM but not actively being used.
- The selected data is moved to secondary storage, freeing up space in RAM for the new program
- When the swapped-out program is needed again, it can be swapped back into RAM, replacing another inactive program or data if necessary.

Advantages

- Swapping minimizes the waiting time for processes to be executed by using the swap space as an extension of RAM, allowing the CPU to keep working efficiently without long delays due to memory limitations.
- Swapping allows the operating system to free up space in the main memory (RAM) by moving inactive or less critical data to secondary storage (like a hard drive or SSD). This ensures that the available RAM is used for the most active processes and applications, which need it the most for optimal performance.
- Using only single main memory, multiple process can be run by CPU using swap partition.
- It allows larger programs or applications to run on systems with limited physical memory by swapping less critical data to secondary storage and loading the necessary parts into RAM.

- By swapping out inactive processes, the operating system can prevent the system from becoming overloaded, ensuring that the most important and active processes have access to enough memory for smooth execution.

Disadvantages

- Risk of data loss during swapping arises because of the dependency on secondary storage for temporary data retention. If the system loses power before this data is safely written back into RAM or saved properly, it can result in the loss of important data, files, or system states.
- The number of page faults increases as the system frequently swaps pages in and out of memory, which directly impacts performance because fetching data from the disk (or swap space) is much slower than accessing it from RAM.
- If the system Swaps-in and out too often, the performance of system can severely decline as CPU will spend more time swapping then executing processes.

Que3. What is Paging in Operating Systems? Draw and explain hardware implementation of Paging.

Paging

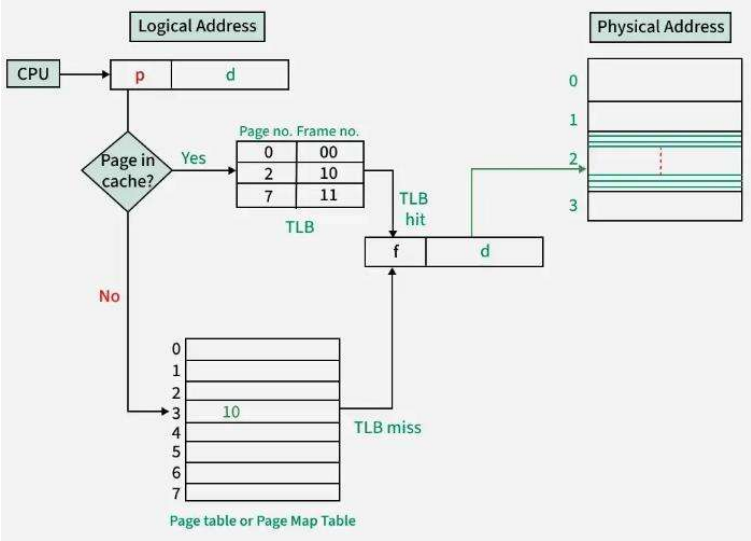
Paging is a memory management technique that addresses common challenges in allocating and managing memory efficiently. When a process requests memory, the operating system allocates one or more page frames to the process and maps the process's logical pages to the physical page frames. When a program runs, its pages are loaded into any available frames in the physical memory.

Each program has a page table, which the operating system uses to keep track of where each page is stored in physical memory. When a program accesses data, the system uses this table to convert the program's address into a physical memory address.

Hardware implementation of Paging

The hardware implementation of the page table can be done by using dedicated registers. But the usage of the register for the page table is satisfactory only if the page table is small. If the page

table contains a large number of entries then we can use TLB (translation Look-aside buffer), a special, small, fast look-up hardware cache.



- The TLB is associative, high-speed memory.
- Each entry in TLB consists of two parts: a tag and a value.
- When this memory is used, then an item is compared with all tags simultaneously. If the item is found, then the corresponding value is returned.

Advantages of Paging

- **Eliminates External Fragmentation:** Paging divides memory into fixed-size blocks (pages and frames), so processes can be loaded wherever there is free space in memory. This prevents wasted space due to fragmentation.
- **Efficient Memory Utilization:** Since pages can be placed in non-contiguous memory locations, even small free spaces can be utilized, leading to better memory allocation.
- **Supports Virtual Memory:** Paging enables the implementation of virtual memory, allowing processes to use more memory than physically available by swapping pages between RAM and secondary storage.
- **Ease of Swapping:** Individual pages can be moved between physical memory and disk (swap space) without affecting the entire process, making swapping faster and more efficient.
- **Improved Security and Isolation:** Each process works within its own set of pages, preventing one process from accessing another's memory space.

Disadvantages of Paging

- **Internal Fragmentation:** If the size of a process is not a perfect multiple of the page size, the unused space in the last page results in internal fragmentation.
- **Increased Overhead:** Maintaining the Page Table requires additional memory and processing. For large processes, the page table can grow significantly, consuming valuable memory resources.
- **Page Table Lookup Time:** Accessing memory requires translating logical addresses to physical addresses using the page table. This additional step increases memory access time, although Translation Look-aside Buffers (TLBs) can help reduce the impact.
- **I/O Overhead During Page Faults:** When a required page is not in physical memory (page fault), it needs to be fetched from secondary storage, causing delays and increased I/O operations.
- **Complexity in Implementation:** Paging requires sophisticated hardware and software support, including the Memory Management Unit (MMU) and algorithms for page replacement, which add complexity to the system.

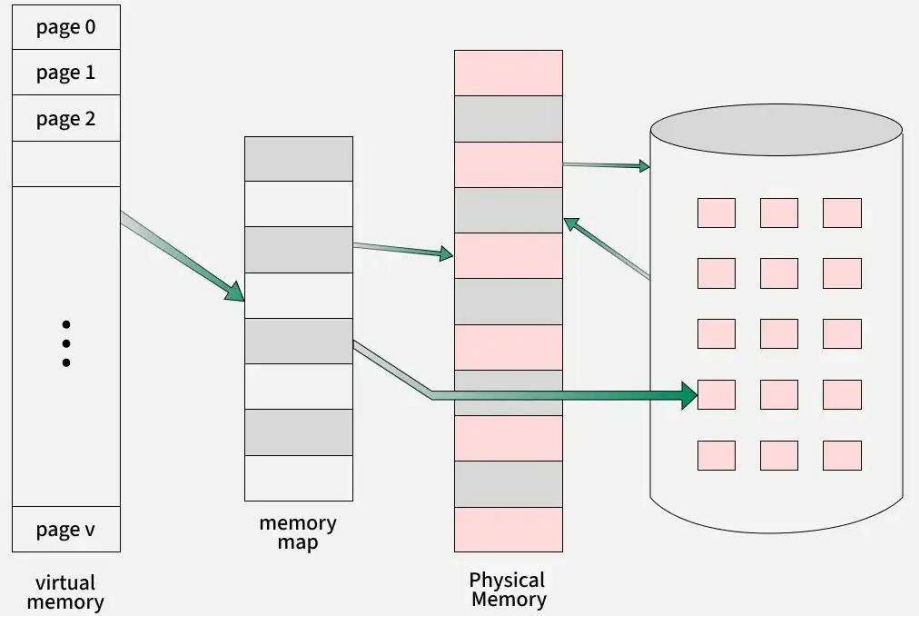
Que4. What is Virtual Memory? How Virtual Memory Works.

Virtual Memory

Virtual memory is a memory management technique used by operating systems to give the appearance of a large, continuous block of memory to applications, even if the physical memory (RAM) is limited. It allows larger applications to run on systems with less RAM.

Objectives of Virtual Memory

- To support multiprogramming, it allows more than one program to run at the same time.
- A program doesn't need to be fully loaded in memory to run. Only the needed parts are loaded.
- Programs can be bigger than the physical memory available in the system.
- Virtual memory creates the illusion of a large memory, even if the actual memory (RAM) is small.
- It uses both RAM and disk storage to manage memory, loading only parts of programs into RAM as needed.
- This allows the system to run more programs at once and manage memory more efficiently.



How Virtual Memory Works

- Virtual memory uses both hardware and software to manage memory.
- When a program runs, it uses virtual addresses (not real memory locations).
- The computer system converts these virtual addresses into physical addresses (actual locations in RAM) while the program runs.
- **Adjust the Page File Size:** All contemporary operating systems including Windows contain the auto-configuration option for the size of the empirical page file. But depending on the size of the RAM, they are set automatically.
- **Place the Page File on a Fast Drive:** Regarding systems having multiple drives involved, the page file needs to be placed on a different drive than the OS and that shall in turn improve its performance.
- Monitor and Optimize Usage

Que5. What is Demand Paging in Operating System? What is Page Fault in demand paging?

Demand Paging

Demand paging is a memory management scheme used in operating systems to improve memory usage and system performance. Let's understand demand paging with real life example Imagine

you are reading a very thick book, but you don't want to carry the entire book around because it's too heavy. Instead, you decide to only bring the pages you need as you read through the book. When you finish with one page, you can put it away and grab the next page you need.

Demand paging is a technique used in virtual memory systems where pages enter main memory only when requested or needed by the CPU. In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start. A page fault occurred when the program needed to access a page that is not currently in memory.

The operating system then loads the required pages from the disk into memory and updates the page tables accordingly. This process is transparent to the running program and it continues to run as if the page had always been in memory.

Pure Demand Paging

Pure demand paging is a specific implementation of demand paging. The operating system only loads pages into memory when the program needs them. In on-demand paging only, no pages are initially loaded into memory when the program starts, and all pages are initially marked as being on disk.

Operating systems that use pure demand paging as a memory management strategy do so without preloading any pages into physical memory prior to the commencement of a task. Demand paging loads a process's whole address space into memory one step at a time, bringing just the parts of the process that are actively being used into memory from disc as needed.

Page Fault

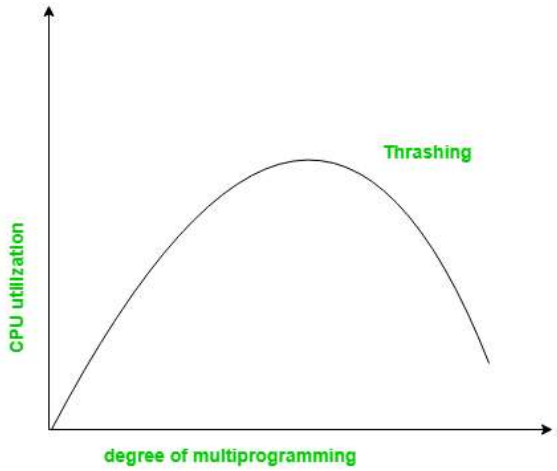
The term "page miss" or "page fault" refers to a situation where a referenced page is not found in the main memory.

When a program tries to access a page, or fixed-size block of memory, that isn't currently loaded in physical memory (RAM), an exception known as a page fault happens. Before enabling the program to access a page that is required, the operating system must bring it into memory from secondary storage (such a hard drive) in order to handle a page fault.

Que6. What is Thrashing? Describe Techniques to handle Thrashing

Thrashing

Thrashing is the term used to describe a state in which excessive paging activity takes place in computer systems, especially in operating systems that use virtual memory, severely impairing system performance. Thrashing occurs when a system's high memory demand and low physical memory capacity cause it to spend a large amount of time rotating pages between main memory (RAM) and secondary storage, which is typically a hard disc. It is caused due to insufficient physical memory, overloading and poor memory management. The operating system may use a variety of techniques to lessen thrashing, including lowering the number of running processes, adjusting paging parameters, and improving memory allocation algorithms. Increasing the system's physical memory (RAM) capacity can also lessen thrashing by lowering the frequency of page swaps between RAM and the disc.



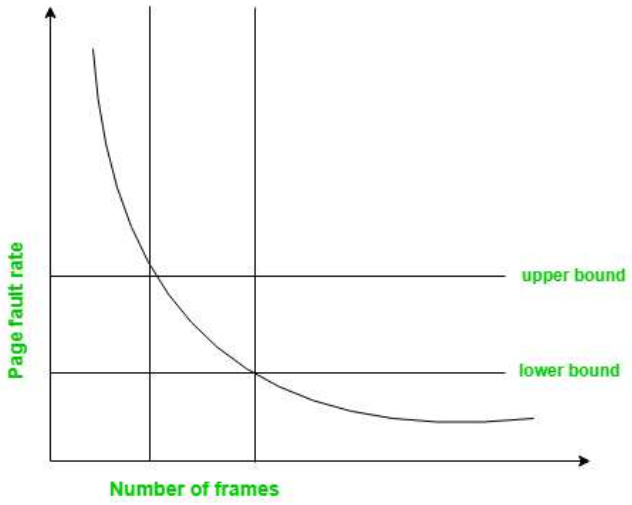
Techniques to handle Thrashing

1. **Working Set Model -** This model is based on the above-stated concept of the Locality Model.

   The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

   According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

2. **Page Fault Frequency -** A more direct approach to handling thrashing is the one that uses the Page-Fault Frequency concept. The problem associated with Thrashing is the high page fault rate and thus, the concept here is to control the page fault rate. If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames. Upper and lower limits can be established on the desired page fault rate as shown in the diagram.



If the page fault rate falls below the lower limit, frames can be removed from the process. Similarly, if the page fault rate exceeds the upper limit, more frames can be allocated to the process.

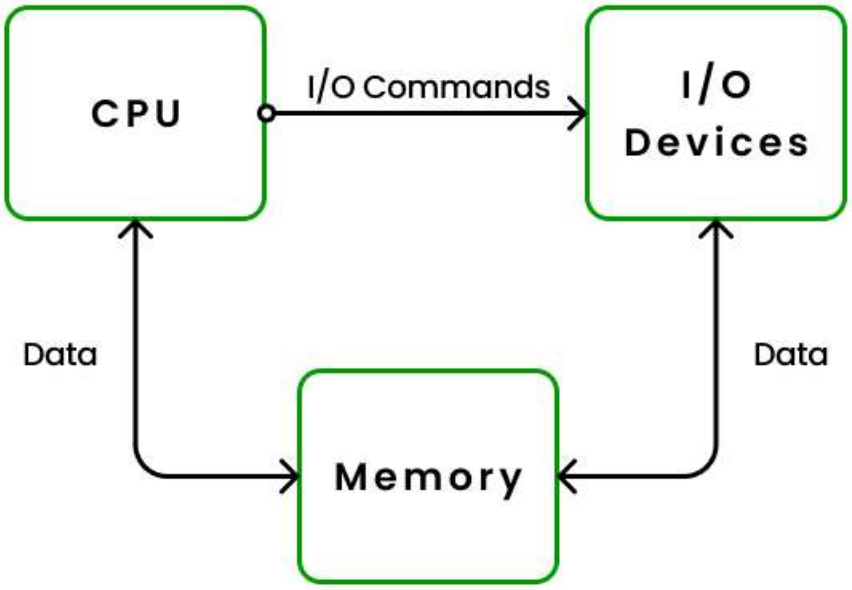
Unit 6: I/O Management

I/O Management: I/O Devices, Organization of I/O function, I/O Buffering, Disk Scheduling- Disk Scheduling policies like FIFO, LIFO, STTF, SCAN, C-SCAN. File Management: Concept, Access methods, Directory Structure, Protection, File System implementation, Directory Implementation, Allocation methods, Free Space management. Case Study: I/O and File Management in multi-cores OS Case Study: Light weight and heavy weight OS: Linux, Tizen

Que1. Write short note on Communication to I/O Devices in Operating System. Why OS and I/O devices communicate with each another?

Communication to I/O Devices in Operating System

The foundation of efficient computing rests on robust interaction between users and an operating system through **Input/output (I/O)** devices in today's world. The smooth functioning necessitates dependable exchange of data among keyboard/screen/mouse/printers/network adapters facilitated by such intermediary agents.



Why OS and I/O devices communicate with each another?

Operating systems and input/output devices must interact in order to ensure the smooth operation of a computer system and user accessibility for a variety of reasons, each with a distinct purpose.

Input and Output

I/O devices let users provide the operating system input and receive output from it. Users can issue commands, engage with programs, and traverse the system via input devices such as keyboards and mice. The user gets information, results, or visual feedback through output devices like monitors and printers. Users are able to successfully employ the features of a computer system through seamless interaction facilitated by communication between the operating system and peripheral devices.

Device Control and Configuration

The operating system must communicate with I/O devices in order to regulate and control their behavior. This includes actions such as configuring device settings, initializing devices during system startup, assigning system resources such as interrupts or memory, and managing power states. The operating system can guarantee that the devices are correctly set up and working inside the system environment thanks to this communication devices.

Data Transfer and Storage

Modern operating systems would struggle to move or store any kind of digital data without I/O components like hard drives or solid-state drives (SSDs). These crucial parts help with the process by establishing efficient lines of communication between your computer's CPU and motherboard AND storing relevant data such as files, programs, systems, and data, enabling you to have seamless access due to their speedy reading and writing capabilities. When working on or saving any type of digital information imaginable, lag-free performance is ensured by this cooperation between the OS and I/O device.

Peripheral Device Support

Support for numerous peripherals and external hardware elements is provided via I/O devices. Data exchange through local networks or the internet is made possible, for instance, by network adapters, which provide the operating system access to other computers or network devices. Similar to that, USB ports enable the connection of extra devices including storage devices, printers, scanners, and cameras. The operating system may make use of these peripherals' capabilities and offer a smooth user experience by communicating with them.

System Monitoring and Control

Efficient monitoring and management of diverse system operations require an operating system to interface with specific I/O devices. To maintain the optimal health of a functioning computer ecosystem while making necessary modifications, sensors within the hardware or external mechanisms collect data on various environmental parameters like temperature and voltage. Moreover, effective scheduling of tasks and smooth time management is facilitated by intercommunication with complementary components such as a real-time clock or a standard clock.

Que2. What is I/O buffering, explain with uses? List and Describe types of I/O Buffering Techniques.

I/O buffering

I/O buffering is a technique used in computer systems to improve the efficiency of input and output (I/O) operations. It involves the temporary storage of data in a buffer, which is a reserved area of memory, to reduce the number of I/O operations and manage the flow of data between fast and slow devices or processes

Uses of I/O Buffering

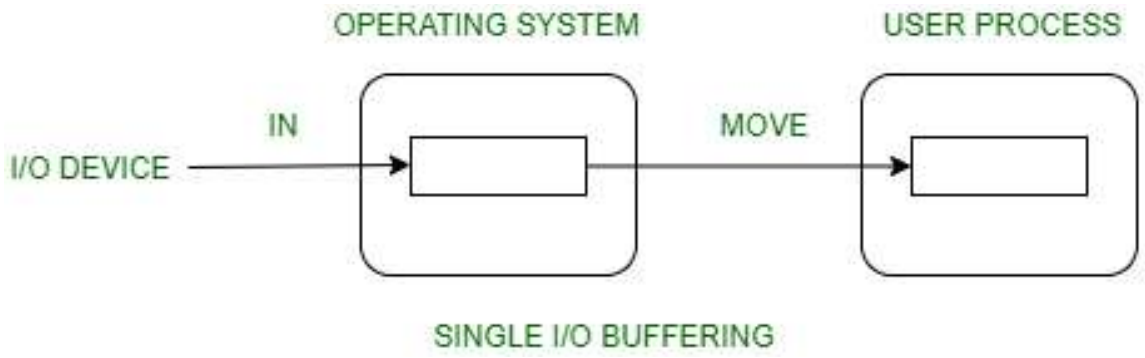
- Buffering is done to deal effectively with a speed mismatch between the producer and consumer of the data stream.
- A buffer is produced in the main memory to heap up the bytes received from the modem.
- After receiving the data in the buffer, the data gets transferred to disk from the buffer in a single operation.
- This process of data transfer is not instantaneous, therefore the modem needs another buffer to store additional incoming data.
- When the first buffer is filled, then it is requested to transfer the data to disk.
- The modem then starts filling the additional incoming data in the second buffer while the data in the first buffer gets transferred to the disk.
- When both buffers complete their tasks, then the modem switches back to the first buffer while the data from the second buffer gets transferred to the disk.

Types of I/O Buffering Techniques

1. Single Buffer
2. Double Buffer
3. Circular Buffer

1. Single Buffer

Using one buffer to store data temporarily. A buffer is provided by the operating system to the system portion of the main memory.



SINGLE I/O BUFFERING

Block Oriented Device

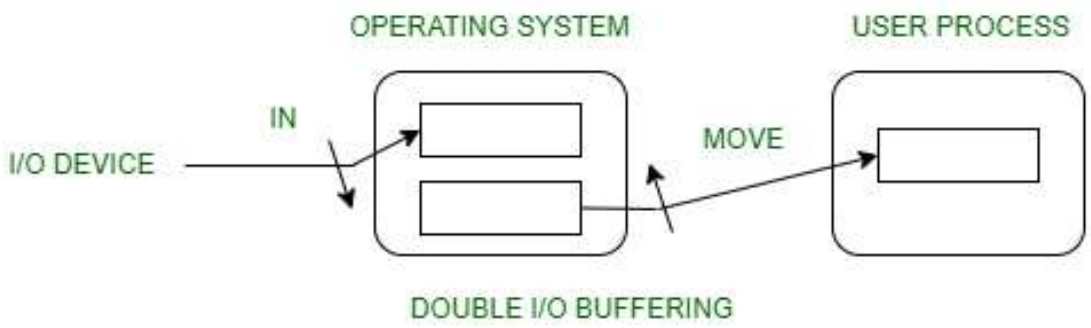
- The system buffer takes the input.
- After taking the input, the block gets transferred to the user space by the process and then the process requests for another block.
- Two blocks work simultaneously, when one block of data is processed by the user process, the next block is being read in.
- OS can swap the processes.
- OS can record the data of the system buffer to user processes.

Stream Oriented Device

- Line- at a time operation is used for scroll-made terminals. The user inputs one line at a time, with a carriage return signaling at the end of a line.
- Byte-at-a-time operation is used on forms mode, terminals when each keystroke is significant.

2. Double Buffer

In this technique the operating system uses two buffers to allow continuous data transfer between two process and two devices.



Block Oriented

- There are two buffers in the system.
- One buffer is used by the driver or controller to store data while waiting for it to be taken by higher level of the hierarchy.
- Other buffer is used to store data from the lower level module.
- Double buffering is also known as buffer swapping.
- A major disadvantage of double buffering is that the complexity of the process get increased.
- If the process performs rapid bursts of I/O, then using double buffering may be deficient.

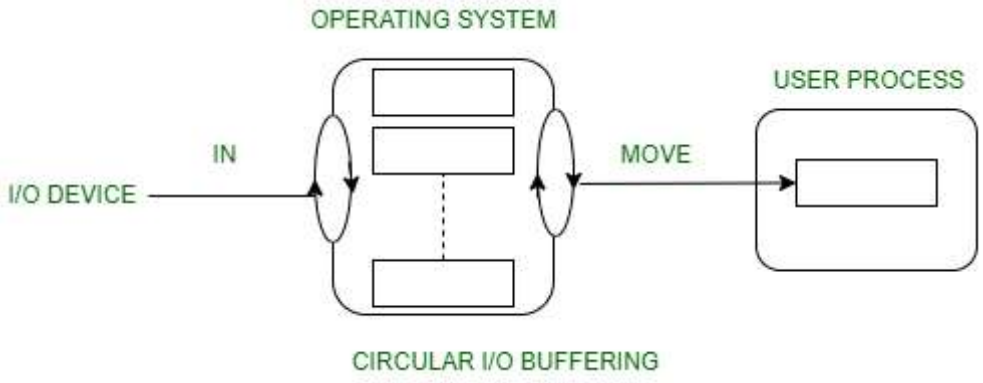
Stream Oriented

- Line- at a time I/O, the user process need not be suspended for input or output, unless process runs ahead of the double buffer.
- Byte- at a time operations, double buffer offers no advantage over a single buffer of twice the length.

3. Circular Buffer

In this technique the operating system uses a circular buffer to manage continuous data streams efficiently.

- When more than two buffers are used, the collection of buffers is itself referred to as a circular buffer.
- In this, the data do not directly passed from the producer to the consumer because the data would change due to overwriting of buffers before they had been consumed.
- The producer can only fill up to buffer i-1 while data in buffer i is waiting to be consumed.



Que3. What is File Management? Describe the different types of File Access Methods in Operating System.

File Management

File management, in the context of an operating system, involves creating, storing, organizing, and manipulating data files. These files can be of various types, such as text documents, images, videos, and more. File management provides a structured approach to handle data, ensuring that it is easily accessible and secure.

File Access Methods

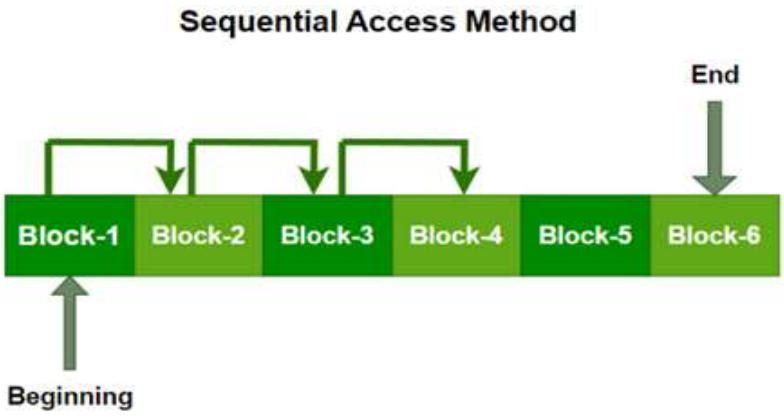
File access methods in an operating system are the techniques and processes used to read from and write to files stored on a computer's storage devices. There are several ways to access this information in the file. Some systems provide only one access method for files. These methods determine how data is organized, retrieved, and modified within a file system.

There are three ways to access a file in a computer system:

- Sequential-Access
- Direct Access
- Index sequential Method

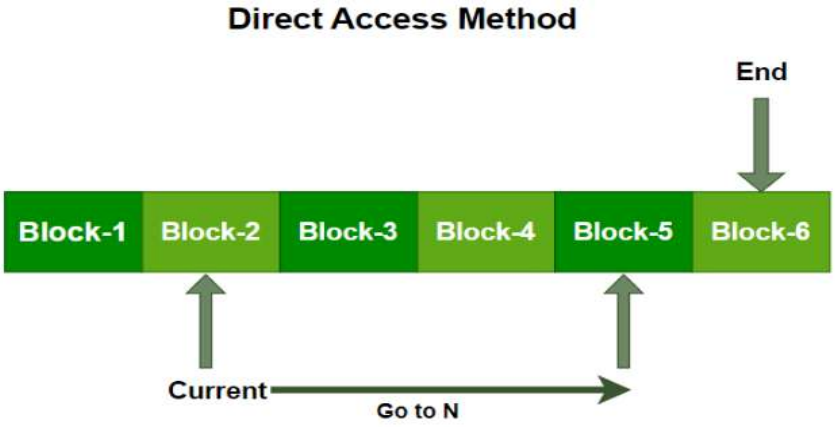
1. Sequential Access

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, the editor and compiler usually access the file in this fashion. Read and write make up the bulk of the operation on a file. A read operation -*read next*- reads the next position of the file and automatically advances a file pointer, which keeps track of the I/O location. Similarly, for the -write *next*- append to the end of the file and advance to the newly written material.



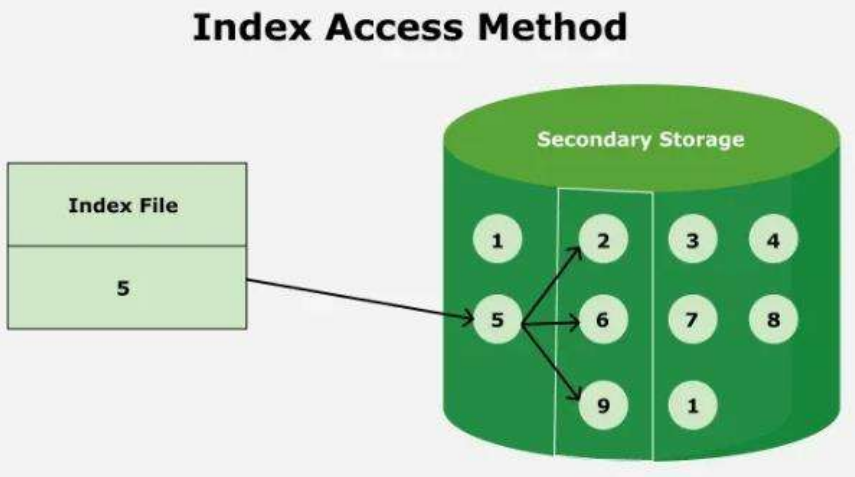
2. Direct Access Method

The **Direct Access Method** (also known as the **Relative Access Method**) allows rapid reading and writing of fixed-length records in any order. Based on the disk model, which supports random access, the file is viewed as a sequence of numbered blocks or records. This means data can be read or written to any block in no specific order. The block numbers provided by the user are relative, starting from 0, 1, and so on. There are no restrictions on the order of accessing records in direct access files.



3. Index Sequential method

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



Que4. Draw and explain the File System structure in detail.

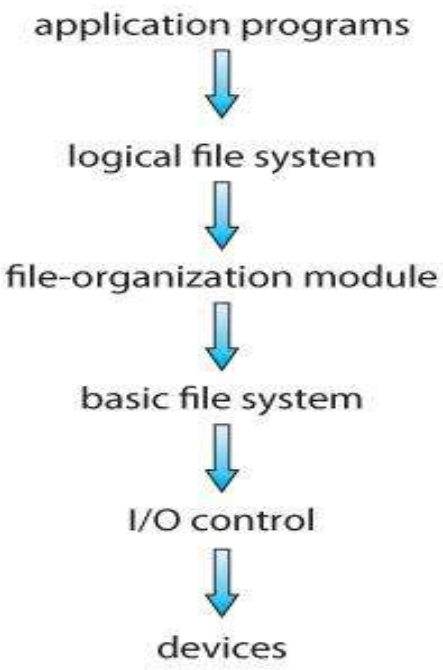
Hard disks have two important properties that make them suitable for secondary storage of files in file systems:

(1) Blocks of data can be rewritten in place.

(2) They are direct access, allowing any block of data to be accessed with only minor movements of the disk heads and rotational latency.

Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger. File systems organize storage on disk drives, and can be viewed as a layered design:

At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.



I/O Control consists of device drivers, special software programs which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card on a system has a different set of addresses. The basic file system level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block.

The file organization module knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.

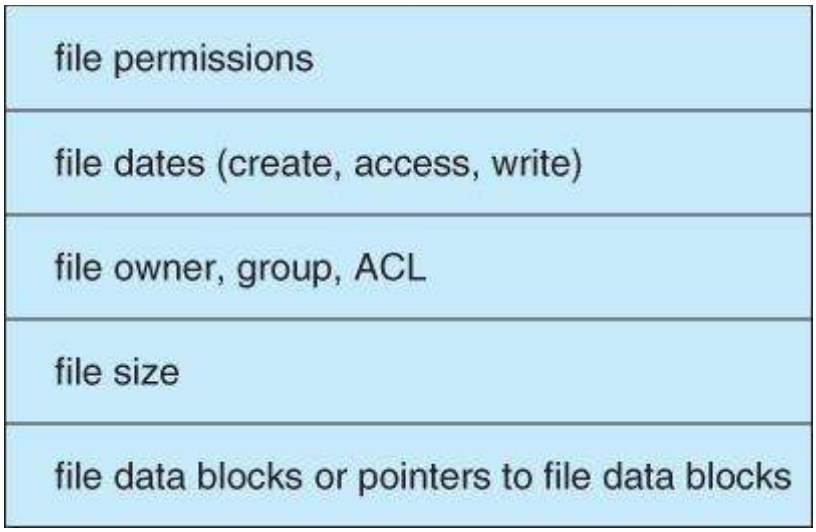
The logical file system deals with all of the Meta data associated with a file i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to file control blocks, FCBs, which contain all of the Meta data as well as block number information for finding the data on the disk.

The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be file system specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System,

FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3.

Que5. Describe the concept of File System implementation of Operating System with diagram?

File systems store several important data structures on the disk:

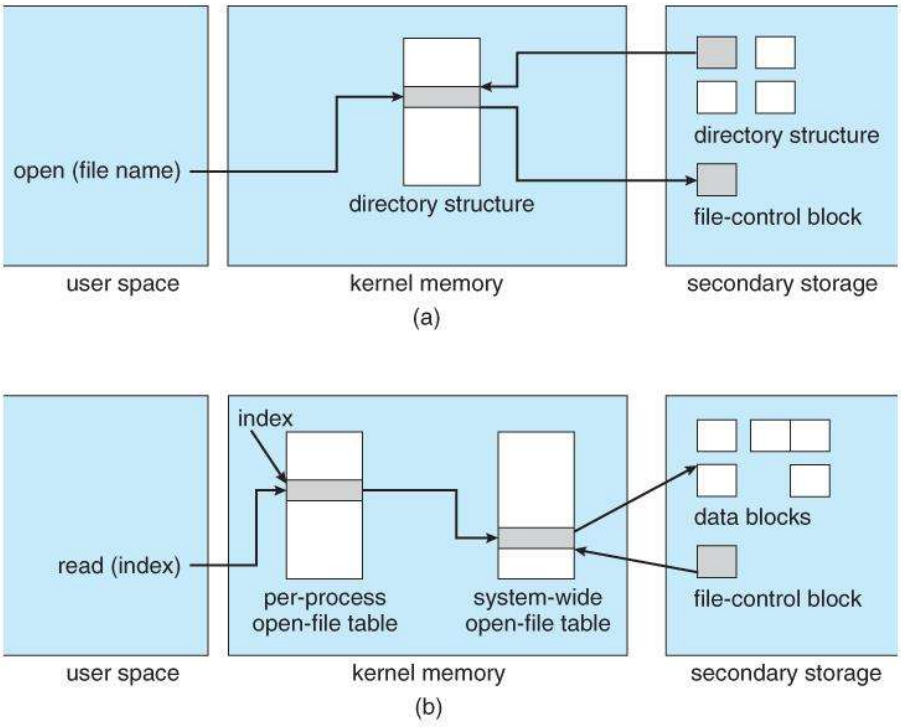


- **A boot-control block**, (per volume) a.k.a. the boot block in UNIX or the partition boot sector in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
- A volume control block, (per volume) a.k.a. the master file table in UNIX or the superblock in Windows, which contains information such as the partition table, number of blocks on each file system, and pointers to free blocks and free FCB blocks.
- A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses i-node numbers, and NTFS uses a master file table.
- **The File Control Block**, FCB, (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in i-nodes, and NTFS in the master file table as a relational database structure.

There are also several key data structures stored in memory:

- An in-memory mount table.
- An in-memory directory cache of recently accessed directory information.
- A system-wide open file table, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
- A per-process open file table, containing a pointer to the system open file table as well as some other information.

Following figure illustrates some of the interactions of file system components when files are created and/or used:



- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
- When a file is accessed during a program, the open ( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open ( ) system call. UNIX refers to this index as a file descriptor, and Windows refers to it as a file handle.

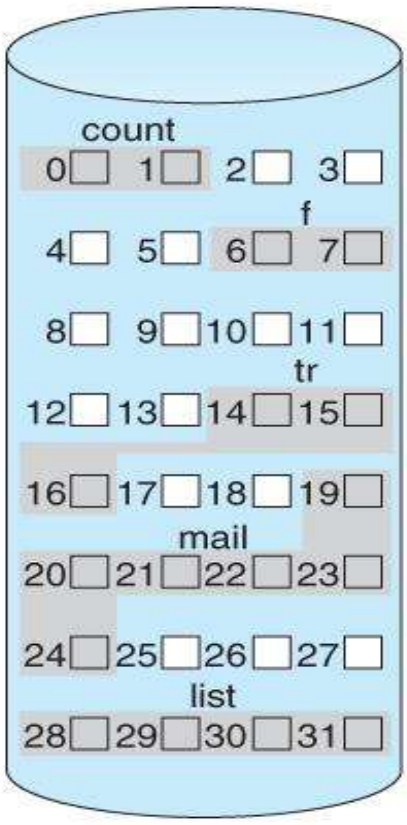
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

Que6. Write short note on types File allocation methods with diagram.

File allocation methods

There are three major methods of storing files on disks: contiguous, linked, and indexed.

1. Contiguous Allocation



directory

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

Contiguous Allocation requires that all blocks of a file be kept together contiguously. Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory. The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.

Problems can arise when files grow, or if the exact size of a file is unknown at creation time:

- **Over-estimation** of the file's final size increases external fragmentation and wastes disk space.
- **Under-estimation** may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
- If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.

A variation is to allocate file space in large contiguous chunks, called extents. When a file outgrows its original extent, then an additional one is allocated. The high-performance files system Verities uses extents to optimize performance.

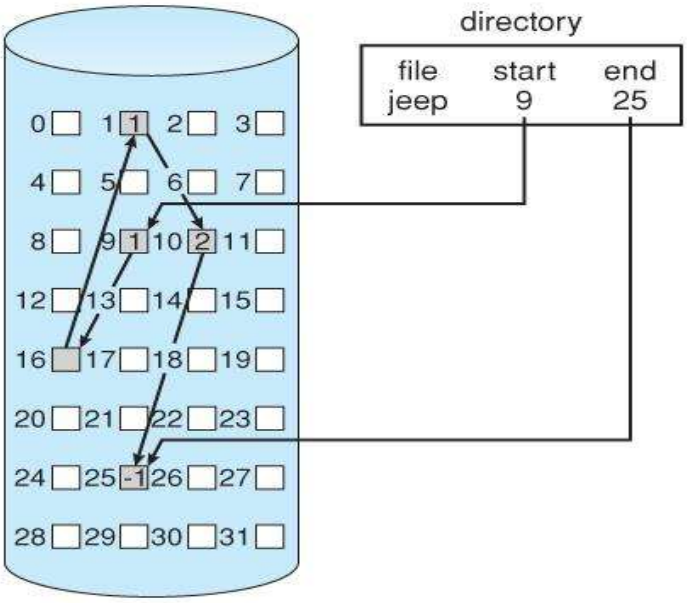
2. Linked Allocation

Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.

Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.

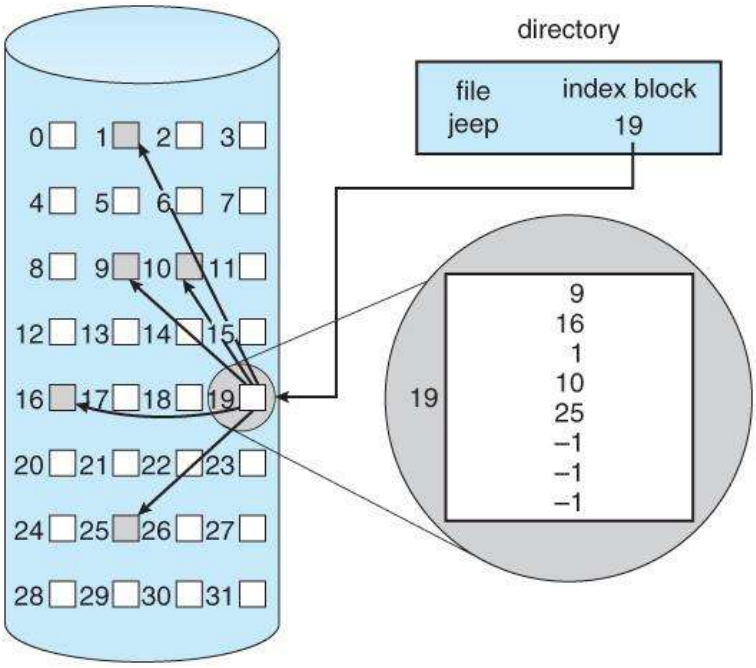
Allocating clusters of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.



3. Indexed Allocation

Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.



Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

- **Linked Scheme -** An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
- **Multi-Level Index -** The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
- **Combined Scheme -** This is the scheme used in UNIX i-nodes, in which the first 12 or so data block pointers are stored directly in i-node, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed.