|  |
| --- |
| **1)A)AIM**: Study of various System Programs like Assembler, Linker, Loader, Interpreter, Compiler, Macro, etc.  **System Programs:** System Programs are the softwares which directly interacts with the hardware and get the basic connectivity between the user programs and the system hardware.  System programs are basically supporting softwares that can interact with hardware and which are responsible for providing a basic interface between hardware and the user software.  Heirarchy of user applications, computer hardware, os and system programs.    1. Basic computer architecture.  **System Programs:**    2. Assemblers and Macros. **Assembler** An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an operation code ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities.[9] The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution – e.g., to generate common short sequences of instructions as inline, instead of called subroutines.  Types of asssemblers:  A macro assembler includes a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code.  A cross assembler (see also cross compiler) is an assembler that is run on a computer or operating system (the host system) of a different type from the system on which the resulting code is to run (the target system). Cross-assembling facilitates the development of programs for systems that do not have the resources to support software development, such as an embedded system. In such a case, the resulting object code must be transferred to the target system, either via read-only memory (ROM, EPROM, etc.) or a data link using an exact bit-by-bit copy of the object code or a text-based representation of that code, such as Motorola S-record or Intel HEX.  A high-level assembler is a program that provides language abstractions more often associated with high-level languages, such as advanced control structures (IF/THEN/ELSE, DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.  A microassembler is a program that helps prepare a microprogram, called firmware, to control the low level operation of a computer.  A meta-assembler is a term used in some circles for "a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language."[8]  Assembly time is the computational step where an assembler is run.  **2) Compiler:**  A compiler is a computer program that transforms computer code written in one programming language (the source language) into another programming language (the target language). Compilers are a type of translator that support digital devices, primarily computers. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.  A compiler executes four major steps:   * **Scanning**: The scanner reads one character at a time from the source code and keeps track of which character is present in which line. * **Lexical Analysis**: The compiler converts the sequence of characters that appear in the source code into a series of strings of characters (known as tokens), which are associated by a specific rule by a program called a lexical analyzer. A symbol table is used by the lexical analyzer to store the words in the source code that correspond to the token generated. * **Syntactic Analysis**: In this step, syntax analysis is performed, which involves preprocessing to determine whether the tokens created during lexical analysis are in proper order as per their usage. The correct order of a set of keywords, which can yield a desired result, is called syntax. The compiler has to check the source code to ensure syntactic accuracy. * **Semantic Analysis**: This step is comprised of several intermediate steps. First, the structure of tokens is checked, along with their order with respect to the grammar in a given language. The meaning of the token structure is interpreted by the parser and analyzer to finally generate an intermediate code, called object code. The object code includes instructions that represent the processor action for a corresponding token when encountered in the program. Finally, the entire code is parsed and interpreted to check if any optimizations are possible. Once optimizations can be performed, the appropriate modified tokens are inserted in the object code to generate the final object code, which is saved inside a file.   **3) Loader:**  In computer systems a loader is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.  In Unix, the loader is the handler for the system call execve().[1] The Unix loader's tasks include:  validation (permissions, memory requirements etc.);  copying the program image from the disk into main memory;  copying the command-line arguments on the stack;  initializing registers (e.g., the stack pointer);  jumping to the program entry point (\_start).  **4) Interpreter:**  an interpreter is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:  parse the source code and perform its behavior directly;  translate source code into some efficient intermediate representation and immediately execute this;  explicitly execute stored precompiled code[1] made by a compiler which is part of the interpreter system. **Applications**  * Interpreters are frequently used to execute command languages, and glue languages since each operator executed in command language is usually an invocation of a complex routine such as an editor or compiler. * Self-modifying code can easily be implemented in an interpreted language. This relates to the origins of interpretation in Lisp and artificial intelligence research. * Virtualization. Machine code intended for a hardware architecture can be run using a virtual machine. This is often used when the intended architecture is unavailable, or among other uses, for running multiple copies. * Sandboxing: While some types of sandboxes rely on operating system protections, an interpreter or virtual machine is often used. The actual hardware architecture and the originally intended hardware architecture may or may not be the same. This may seem pointless, except that sandboxes are not compelled to actually execute all the instructions the source code it is processing. In particular, it can refuse to execute code that violates any security constraints it is operating under. * Emulators for running computer software written for obsolete and unavailable hardware on more modern equipment.   **5) Linker:**    a linker or link editor is a computer utility program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another 'object' file.  A simpler version that writes its output directly to memory is called the loader, though loading is typically considered a separate process.  Many operating system environments allow dynamic linking, deferring the resolution of some undefined symbols until a program is run. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these. Loading the program will load these objects/libraries as well, and perform a final linking.  **6) Macros**: the compiler executes "preprocessor statements" at compilation time, and the output of this execution forms part of the code that is compiled. The ability to use a familiar procedural language as the macro language gives power much greater than that of text substitution macros, at the expense of a larger and slower compiler.  While *macro instructions* can be defined by a programmer for any set of native assembler program instructions, typically macros are associated with macro libraries delivered with the operating system allowing access to operating system functions such as   * peripheral access by access methods (including macros such as OPEN, CLOSE, READ and WRITE) * operating system functions such as ATTACH, WAIT and POST for subtask creation and synchronization. Typically such macros expand into executable code, e.g., for the EXIT macroinstruction, * a list of *define constant* instructions, e.g., for theDCB macro -- DTF (Define The File) forDOS -- or a combination of code and constants, with the details of the expansion depending on the parameters of the macro instruction (such as a reference to a file and a data area for a READ instruction); * the executable code often terminated in either a *branch and link register* instruction to call a routine, or asupervisor call instruction to call an operating system function directly. |

|  |
| --- |
| 1)b) To create your own library file in C/C++/Java.  Math library file :  import static java.lang.Integer.min;  public class MathLib {  static int factorial(int n){  if(n==0 || n==1)  return 1;  return n\*factorial(n-1);  }  static int random(int n1, int n2){  return (int) (min(n1, n2)+Math.random()/(n2-n1));  }  }  Main file using MathLib library.  import java.util.Scanner;  public class Python {  static Scanner scanner = new Scanner(System.in);  private static int input(String s){  System.out.print(s);  return scanner.nextInt();  }  public static void main(String[] args) {  int choice = input(  "-------------------------------------------------\n"+  "|Number | Function |\n" +  "|------------|-----------------------------------|\n" +  "| 1 | Get random number |\n" +  "| 2 | Get factorial of a number |\n" +  "-------------------------------------------------\n" +  "Enter your choice: "  );  if(choice == 1){  int n1 = input("Enter first number: ");  int n2 = input("Enter second number: ");  System.out.println(  String.format(  "random number between %d and %d is : %d",  n1,  n2,  MathLib.random(n1, n2)  )  );  }  else if(choice == 2){  int n3 = input("Enter the number to find factorial of: ");  int fact = MathLib.factorial(n3);  System.out.println(  String.format(  "factorial of %d is: %d",  n3,  fact  )  );  }  }  }  OUTPUT:      Conclusion:   1. Learnt what is system programs. 2. Studied types of system programs. 3. Made a library in java programming language that can be imported into another java file. 4. Created a choice based java program to demonstrate the use of MathLib functions created in another file. 5. Learnt the basic usages of system programs like assembler, compiler in converting higher language program to lower level machine language. |