MCA 304C: SYSTEM PROGRAMMING

UNIT I

Background introduction, system software and machine architecture, SIC, RISC, and CISC architecture. Assembler: basic assembler functions, machine dependent and independent assembler features, assembler design options, and implementation examples.

UNIT II

Loading and linkers basic loader junction, machine dependent and independent loader features, loader design options and implementation examples. Macro processors, basic macro processor functions machines – independent macro processor features, macro processor design options, implementation examples.

UNIT III

Compilers: basic compiler functions, machine dependent and independent compiler features, compiler design options and implementation examples. Other system software: text editors and interactive debugging systems

UNIT-IV

Introduction to Device Drivers, Design issues-Types of Drivers, Character driver-1 and Design issues, Character Driver-2- A/D converter and its design issues, Block driver-1 and its design issues- RAM DISK driver-Anatomy-Prologue of drivers and programming Considerations.

UNIT-V

Introduction to Linux**-** Linux Architecture- X-windows- Linux administration tools - Commands to use Linux OS- Executing Linux Shell scripts – Shell Programming concepts-Shell scripts.

Text Books:

1. Leland .Beck, System Software: An Introduction to system programming:3/e, Pearson Educations Asia,2003.

   George pajari Writing Unix Drivers, Addison – Wesley, 1991.
2. Richard Petersen, Linux complete Reference, McGraw Hill Education (India) Private Limited; 6 edition (21 November 2007

Reference Books:

1.Dhamdhere , System programming and operation Systems Book 2/E, Tata Mc Graw , Hill, 199

2. A.V.Aho , Ravi Sethi and J D Ullman , “compilers, Techniques and Tools”, Addison Wesley, 1986.Jhon J. Donovan, System Programming Tata McGraw Hill 2005.

Lecture Notes

UNIT-1

System program:

System software is **a type of computer program that is designed to run a computer's hardware and application programs**. If we think of the computer system as a layered model, the system software is the interface between the hardware and user applications. The operating system is the best-known example of system software.

Important features of system software:

Computer manufacturers usually develop the system software as an integral part of the computer. The primary responsibility of this software is to create an interface between the computer hardware they manufacture and the end user.

1. **Hard to manipulate.** It often requires the use of a programming language, which is more difficult to use than a more intuitive user interface (UI).
2. **Written in a low-level computer language.** System software must be written in a computer language the central processing unit (CPU) and other computer hardware can read.
3. **Close to the system.** It connects directly to the hardware that enables the computer to run.
4. **Versatile.** System software must communicate with both the specialized hardware it runs on and the higher-level application software that is usually hardware-agnostic and often has no direct connection to the hardware it runs on. System software also must support other programs that depend on it as they evolve and change.

Types of system software:

The five **types of systems software**, are all designed to control and coordinate the procedures and functions of computer hardware. They actually enable functional interaction between hardware, software and the user.

Systems software carries out middleman tasks to ensure communication between other software and hardware to allow harmonious coexistence with the user.

Systems software can be categorized under the following:

- **Operating system:** Harnesses communication between hardware, system programs, and other applications.
- **Device driver:** Enables device communication with the OS and other programs.
- **Firmware:** Enables device control and identification.
- **Translator:** Translates high-level languages to low-level machine codes.
- **Utility:** Ensures optimum functionality of devices and applications.

SIC architecture:

SIC/XE stands for **Simplified Instructional Computer Extra Equipment or Extra Expensive**. This computer is an advance version of SIC. Both SIC and SIC/XE are closely related to each other that’s why they are Upward Compatible.
SIC/XE machine architecture:
1. Memory:
Memory consists of 8 bit-bytes and the memory size is 1 megabytes (2^{20} bytes). Standard SIC memory size is very small. This change in the memory size leads to change in the instruction formats as well as addressing modes. 3 consecutive bytes form a word (24 bits) in SIC/XE architecture.
All address are byte addresses and words are addressed by the location of their lowest numbered byte.

2. Registers:
It contain 9 registers (5 SIC registers + 4 additional registers). Four additional registers are:

Mnemonics	**Use of Register**
B	Base register
S	General working register
T	General working register
F	Floating-point accumulator

3. Data Formats:

- Integers are represented by Binary numbers.
- Characters are represented using ASCII codes.
- Floating points are represented using 48-bits.

4. Instruction formats:

- In SIC/XE architecture there are 4 types of formats available
- The Bit(e) is used to distinguish between Formats 3 and Formats 4,
  `e=0 means Format 3 and e=1 means Format 4`

Assembler:

"Assembler Design Options One-Pass and Multi-Pass Assemblers."— Presentation transcript:

Assembler Design Options One-Pass and Multi-Pass Assemblers

One-Pass Assemblers One-pass assemblers are used when –it is necessary or desirable to avoid a second pass over the source program –the external storage for the intermediate file between two passes is slow or is inconvenient to use Main problem: forward references to both data and instructions One simple way to eliminate this problem: require that all areas be defined before they are referenced. –It is possible, although inconvenient, to do so for data items. –Forward jump to instruction items cannot be easily eliminated.

Sample Program for a One-Pass Assembler

Load-and-Go Assembler Load-and-go assembler generates their object code in memory for immediate execution. No object program is written out, no loader is needed. It is useful in a system oriented toward program development and testing such that the efficiency of the assembly process is an important consideration.

How to Handle Forward References Load-and-go assembler –Omits the operand address if the symbol has not yet been defined –Enters this undefined symbol into SYMTAB and indicates that it is undefined –Adds the address of this operand address to a list of forward references associated with the SYMTAB entry –Scans the reference list and inserts the address when the definition for the symbol is encountered. –Reports the error if there are still SYMTAB entries indicated undefined symbols at the end of the program –Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error.

Machine dependent and independent Assembler features:

Machine Dependent Assembler Features l Instruction Format and Addressing Mode » PC-relative or Base-relative addressing: op m » Indirect addressing: op @m » Immediate addressing: op #c » Extended format: +op m » Index addressing: op m, x » register-to-register instructions » larger memory -> multi-programming.

Machine-dependent software is software that runs only on a specific computer. Applications that run on multiple computer architectures are called machine-independent, or cross-platform.

Short questions:

1. What do you mean by System Software and its importance?
2. Write about machine dependent and independent assembler features?
3. Write about assembler design options?
4. Write the architecture of SIC and RISC?

Long questions:

1. Write about the implementation Examples?
2. Write about the machine architecture?
3. Discuss the basic assembler functions?

4. Write the difference between the architectures SIC and RISC?

Unit-2

Linkers and Loaders

A loader loads the programs into the main memory from the storage device. The OS transparently calls the loader when needed.

A linker links and combines objects generated by a compiler into a single executable. A linker is also responsible to link and combine all modules of a program if written separately.loader is a program used by an operating system to load programs from a secondary to main memory so as to be executed.

Usually large applications are written into small modules and are then compiled into object codes. A linker is a program that combines these object modules to form an executable. Loaders are used mainly for **loading materials into trucks, laying pipe, clearing rubble, and digging**. A loader is not the most efficient machine for digging as it cannot dig very deep below the level of its wheels, like a backhoe or an excavator can.

Loaders are used mainly for **loading materials into trucks, laying pipe, clearing rubble, and digging**. A loader is not the most efficient machine for digging as it cannot dig very deep below the level of its wheels, like a backhoe or an excavator can.

Linker is a program in a system which **helps to link object modules of a program into a single object file**. It performs the process of linking. Linkers are also called as link editors. Linking is a process of collecting and maintaining piece of code and data into a single file.

Macro Processor design Options

1.**Macro Processor Design Options Recursive Macro** Expansion General- Purpose Macro Processors Macro Processing within Language Translators

2. **Recursive Macro Expansion**
3. **Recursive Macro Expansion** • RDCHAR: • read one character from a specified device into register A • should be defined beforehand (i.e., before RDBUFF)
4. **Implementation of Recursive Macro Expansion** • Previous macro processor design cannot handle such kind of recursive macro invocation and expansion, e.g., RDBUFF BUFFER, LENGTH, F1 • Reasons: • The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten. • The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, that is, the macro process would forget that it had been in the middle of expanding an "outer" macro. • A similar problem would occur with PROCESSLINE since this procedure too would be called recursively. • Solutions: • Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained. • Use a stack to take care of pushing and popping local variables and return addresses. • Another problem: can a macro invoke itself recursively?
5. **General-Purpose Macro Processors** • Goal: • macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages • Pros • Programmers do not need to learn many macro languages. • Although its development costs are somewhat greater than those for a language-specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost. • Cons • Large number of details must be dealt with in a real programming language • Situations in which normal macro parameter substitution should not occur, e.g., comments. • Facilities for grouping together terms, expressions, or

statements • Tokens, e.g., identifiers, constants, operators, keywords • Syntax

Loader design options:

1. **Loader Design Options Linkage Editors Dynamic Linking Bootstrap** Loaders
2. **Linkage Editors** • Difference between a linkage editor and a linking loader: • Linking loader • performs all linking and relocation operations, including automatic library search, and loads the linked program into memory for execution. • Linkage editor • produces a linked version of the program, which is normally written to a file or library for later execution. • A simple relocating loader (one pass) can be used to load the program into memory for execution. • The linkage editor performs relocation of all control sections relative to the start of the linked program. • The only object code modification necessary is the addition of an actual load address to relative values within the program.
3. **Linkage Editors** • Difference between a linkage editor and a linking loader: • Linking loader • Suitable when a program is reassembled for nearly every execution •In a program development and testing environment • When a program is used so infrequently that it is not worthwhile to store the assembled and linked version. • Linkage editor • Suitable when a program is to be executed many times without being reassembled because resolution of external references and library searching are only performed once.
4. **Linking Loader vs. Linkage Editor**
5. **Additional Functions of Linkage Editors** • Replacement of subroutines in the linked program • For example: INCLUDE PLANNER (PROGLIB) DELETE PROJECT INCLUDE PROJECT (NEWLIB) REPLACE PLANNER (PROGLIB) • Construction of a package for subroutines generally used together • There are a large number of cross-references between these subroutines due to their closely related functions. • For example: INCLUDE

READR (FTNLIB) INCLUDE WRITER (FTNLIB): SAVEFTNIO (SUBLIB) • Specification of external references not to be resolved by automatic library search • Can avoid multiple storage of common libraries in programs. • Need a linking loader to combine the common libraries at execution time.

6. **Address Binding** • Address Binding: • Symbolic Address (label) ⟍ Machine Address • Address Binding: • Assembling Time: 8051 • Load Time: 8086 • Run Time: Dynamic Linking Library • Address Binding • Complexity, Flexibility
7. **Linking Time** • Linkage editors: before load time • Linking loaders: at load time • Dynamic linking: after load time • A scheme that postpones the linking function until execution time. • A subroutine is loaded and linked to the test of the program when it is first called. • Other names: dynamic loading, load on call
8. **Dynamic Linking Application** • Allows several executing programs to share one copy of a subroutine or library. • Allows the implementation of the shared object and its methods to be determined at execution time in an object-oriented system • Provides the ability to load the routines only when (and if) they are needed. • This can result in substantial savings of load time and memory space. • For example, error handling routines.
9. **Implementation of Dynamic Linking** • Subroutines to be dynamically loaded must be called via an operating system service request, e.g., load-and-call, instead of executing a JSUB instruction. • The service request will be handled by a part of the OS, called the dynamic loader, which is kept in memory during execution of the program. • The parameter of this request is the symbolic name of the routine to be called.
10. **Example of Dynamic Linking** Issue a load-and-call request Load the routine from the specified library
11. **Jump back to the dynamic loader** Jump to the loaded routine Jump back to the user program Second call to this subroutine may not require load operation.

Independent Macro processor features:

Keyword Parameters

- All macro instructions used so far = positional parameters
- Parameters and arguments are associated according to their positions in the macro prototype and macro invocation statement
- If argument is to be omitted, a null argument (,,) used to maintain the proper order in macro invocation
- Not suitable if a macro has a large number of parameters, and only a few of these are given values in a typical invocation

-EX: GENER, DIRECT, 3

-3rd and 9th parameters used (out of 10 possible ones)

Generation of Unique Labels

Concatenation of Macro Parameters

- Allow parameters to be concatenated with other character strings
- Concatenation operator -> marks the end of parameter.
- -> deleted after macro expansion
- Ex: LDA X&ID->1
- &ID is concatenated after string X and before string 1
- LDA XA1 (A=&ID)
- LDA XB1 (B=&ID)

- Directive:

-WHILE-ENDW

-Lines generated repeatedly as long as condition is true

-Done while macro is being expanded

-Conditions to be tested involve macro-time variables and arguments

- Arguments in macro invocations can be used to substitute parameters in macro body

- Additional capability to modify sequence of statements in macro body for conditional macro expansion

-Add to power and flexibility of macro language

Cont.

- Directives:

-IF-ElSE-ENDIF statements

-SET: Assign variables to set symbols (macro-time variables)

- Set Symbols:

-Store working values during macro expansion

-Any symbol that begins with & and is not a macro parameter

-Initialized to 0; change value with SET

-EX: &EORCK SET 1

Macro-time Conditional Statement

Machine-Independent Macro Processor Features

Positional Parameters

- Different form of parameter specification
- Each argument value is written with a keyword that names the corresponding parameter

-Arguments may appear in any order

-Easier to read statements

-Less error-prone

-EX: GENER TYPE=DIRECT, CHANNEL=3

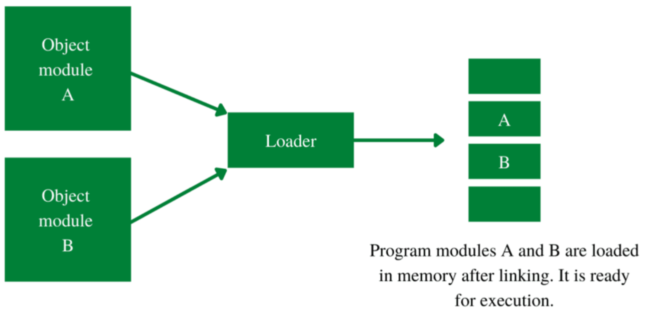
- Generally not possible to use labels in body of macro instructions

Basic loader junctions:

Assemblers and compilers are used to convert source code to object code. The loader will accept that object code, make it ready for execution, and helps to execute. Loader performs its task via four functions, these are as follows:

1. **Allocation:** It allocates memory for the program in the main memory.
2. **Linking:** It combines two or more separate object programs or modules and supplies necessary information.
3. **Relocation:** It modifies the object program so that it can be loaded at an address different from the location.
4. **Loading:** It brings the object program into the main memory for execution.

A general loading scheme is shown below:


Object module A
Object module B
Loader
A
B
Program modules A and B are loaded in memory after linking. It is ready for execution.

General loading scheme

Allocation:

In order to allocate memory to the program, the loader allocates the memory on the basis of the size of the program, this is known as **allocation**. The loader gives the space in memory where the object program will be loaded for execution.

Linking:

The linker resolves the symbolic reference code or data between the object modules by allocating all of the user subroutine and library subroutine addresses. This process is known as **linking**. In any language, a program written has a function, it can be user-defined or can be a library function. For example, in C language we have a print f () function. When the program control goes to the line where the print f () is written, then the linker comes into the picture and it links that line to the module where the actual implementation of the print f () function is written.

Short questions

1. What are linker and loaders?
2. Write the loader design options?
3. What are the basic macro processor functions?
4. Write the difference between linker and loader?

Long questions

5. Write about the machine dependent and independent loader functions?
6. Discuss the loader design implementation examples?
7 .What are the macro processor features?
8. Discuss about the loading and linkers basic loader junction?

UNIT-3

Compiler

In computing, a **compiler** is a computer program that translates computer code written in one programming language (the *source* language) into another language (the *target* language). 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.[1][2]:p1[3]

There are many different types of compilers which produce output in different useful forms. A *cross-compiler* produces code for a different CPU or operating system than the one on which the cross-compiler itself runs. A *bootstrap compiler* is often a temporary compiler, used for compiling a more permanent or better optimised compiler for a language.

Related software include, a program that translates from a low-level language to a higher level one is a *de compiler* ; a program that translates between high-level languages, usually called a *source-to-source compiler* . A language *rewriter* is usually a program that translates the form of expressions without a change of language.

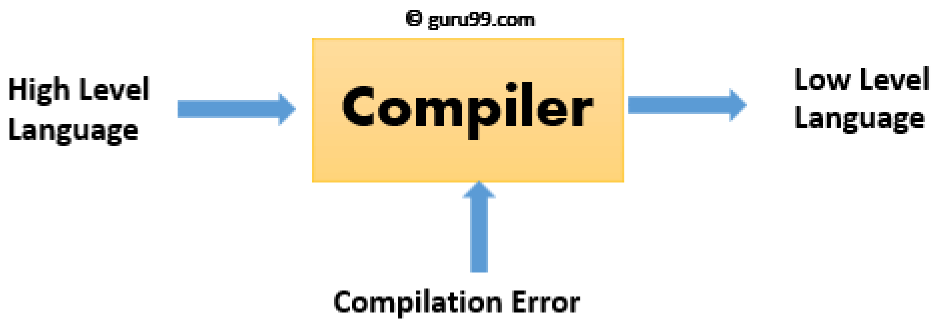
A *compiler-compiler* is a compiler that produces a compiler (or part of one), often in a generic and reusable way so as to be able to produce many differing compilers. A compiler is likely to perform some or all of the following operations, often called phases: lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and code generation. Compilers generally implement these phases as modular components, promoting efficient design and correctness of transformations of source input to target output. Program faults caused by incorrect compiler behaviour can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness.[4]

Compilers are not the only language processor used to transform source programs. An interpreter is computer software that transforms and then executes the indicated operations. The translation process influences the design of computer languages, which leads to a preference of compilation or interpretation. In theory, a programming language can have both a compiler and an interpreter. In practice, programming languages tend to be associated with just one (a compiler or an interpreter).

Compiler Functions:

A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient, which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. The compiler process goes through lexical, syntax, and semantic analysis at the front end and code generation and optimization at the back-end.



Features of Compilers

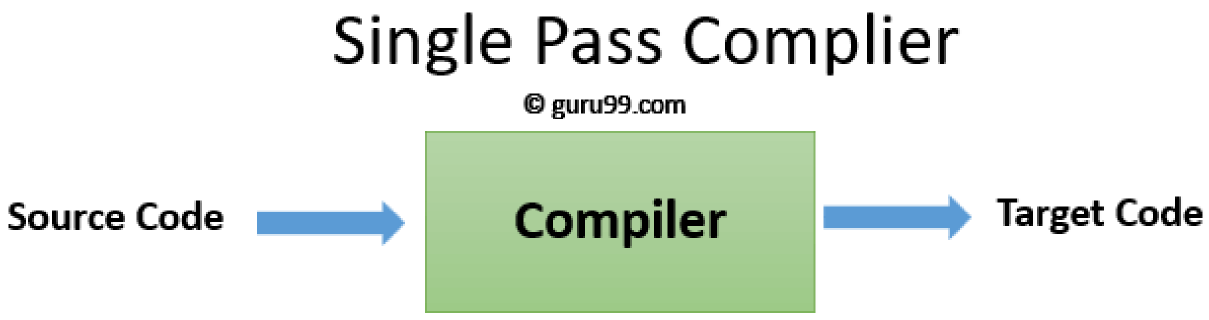
- Correctness
- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs
- Good error reporting/handling
- Code debugging help

Types of Compiler

Following are the different types of Compiler:

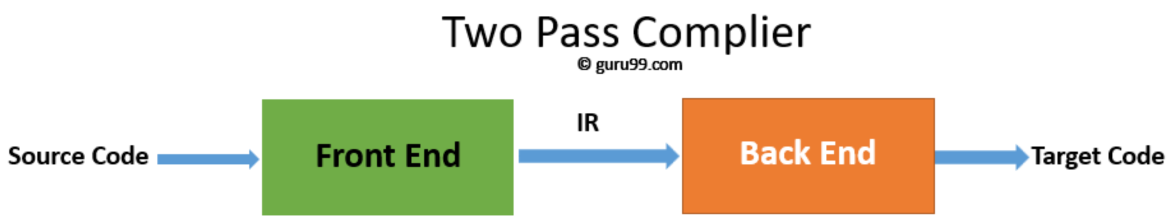
- Single Pass Compilers
- Two Pass Compilers
- Multi pass Compilers

Single Pass Compiler



In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

Two Pass Compiler



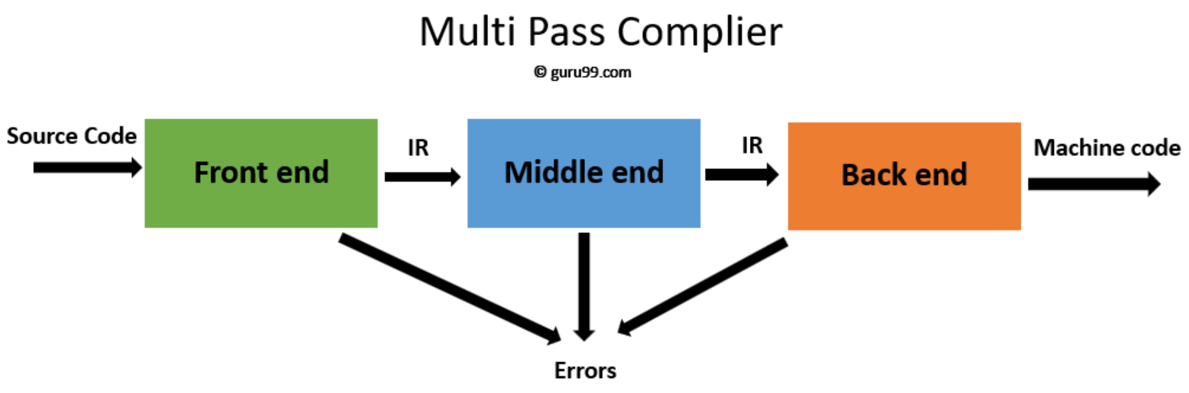
Two Pass Compiler

Two pass Compiler is divided into two sections, viz.

1. **Front end:** It maps legal code into Intermediate Representation (IR).
2. **Back end:** It maps IR onto the target machine

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

Multi pass Compilers



Interactive Debugging System:

Debuggers and interactive debugging

Outline

- First, we will discuss the concept of debugging beyond print statements.
- Then, we will discuss the basic use of traditional debuggers.
- Finally, will discuss advanced techniques for different languages.

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? - Brian Kernighan, 1973

Debugging:

- Debugging is a general process of making code work correctly, which can involve many different things.
- **Print debugging**: add print statements to figure out where you Are and the value of variables.

- This talk will focus on interactive debugging tools: tools that allow you to *interact* with code as it's running in order to find bugs more *efficiently*.

If you have programmed, you have debugged. Any program will have bugs the first time. Debugging is an art itself, and one talk can't possibly teach you how to be an expert debugger. This talk is mainly about some tools which will make debugging easier. These tools let you see into your programs in greater detail, which helps you to debug better.

Print debugging:

1. Run code, see something wrong
2. Add in print statements to show values of variables
3. Re-run code
4. Repeat from (2) *many, many times*

This tends to be the first way of debugging. It works, but is usually slow since you have to re-run the program each time it goes. Still, it is a good first step for small and fast programs.

Interactive development

- You have an interactive shell open
- You immediately see errors, and can
    - Print variables
    - Re-run lines until it works.

This is a good system, but the things you type are temporary. They can't be used as part of other functions or anything. This makes it harder to use for big programs.

Debuggers

- Allow you to see inside of running processes

- Combines print debugging and interactivity
    - A real program runs
    - Debugger is connected to the process
    - You can run line-by-line or wait for a crash
    - You can print variables, type lines, etc. to figure out the problem

A debugger is a tool that can inspect other processes and view internal state. It is the equivalent of a medical imaging device for programs. By being able to see directly inside of running processes, your debugging efficiency can increase greatly. Imagine a doctor trying to diagnose hard problems without advanced imaging.

Basic compiler functions:

BASIC COMPILER FUNCTIONS: A compiler accepts a program written in a high level language as input and produces its machine language equivalent as output. For the purpose of compiler construction, a high level programming language is described in terms of a grammar. This grammar specifies the formal description of the syntax or legal statements in the language

Example: Assignment statement in Pascal is defined as :< variable > :=< Expression >The compiler has to match statement written by the programmer to the structure defined by the grammars and generates appropriate object code for each statement. The compilation process is so complex that it is not reasonable to implement it in one single step. It is partitioned into a series of sub-process called phases. A phase is a logically cohesive operation that takes as input one representation of the source program and produces an output of another representation. The basic phases are - Lexical Analysis, Syntax Analysis, and Code Generation. **Lexical Analysis:** It is the first phase. It is also called scanner. It separates characters of the source language into groups that logically belong together. These groups are called tokens. The usual tokens are: Key word: such as DO or IF Identifiers: such as x. Operator symbols: such as <, =, or, +, and Punctuation symbols: such as parentheses.

Syntax Analyzer: It groups tokens together into syntactic structure. For example, the three tokens representing A + B might be grouped into a syntactic structure called as expression. Expressions might further be combined to form statements. Often the syntactic structures can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of token that logically belong together. Fig. 1 shows the syntax tree for READ statement in PASCAL.

Text Editors:

A text editor is a computer program that lets a user enter, change, store, and usually print text (characters and numbers, each encoded by the computer and its input and output devices, arranged to have meaning to users or to other programs). Typically, a text editor provides an "empty" display screen (or "scrollable page") with a fixed-line length and visible line numbers. You can then fill the lines in with text, line by line. A special command line lets you move to a new page, scroll forward or backward, make global changes in the document, save the document, and perform other actions. After saving a document, you can then print it or display it. Before printing or displaying it, you may be able to format it for some specific output device or class of output device. Text editors can be used to enter program language source statements or to create documents such as technical manuals.

A popular text editor in IBM's large or mainframe computers is called XEDIT. In UNIX systems, the two most commonly used text editors are EMACS and. In personal computer systems, word processor s are more common than text editors. However, there are variations of mainframe and UNIX text editors that are provided for use on personal computers. An example is KEDIT, which is basically XEDIT for Windows.

Short questions

1. Write about compiler?

2. Explain basic compiler functions?

3. What is debugging?

4. What are text editors?

Long questions

5. Write about diff types of debugging system?

6. Explain text editors and its features?

7. Write the difference between the machine dependent and independent compiler features?

8. Discuss the compiler design options and implementation examples?

UNIT-4

Device drivers :

Device Driver in computing refers to a special kind of software program or a specific type of software application that controls a specific hardware device that enables different hardware devices to communicate with the computer's Operating System. A device driver communicates with the computer hardware by computer subsystem or computer bus connected to the hardware.
Device Drivers are essential for a computer system to work properly because without a device driver the particular hardware fails to work accordingly, which means it fails in doing the function/action it was created to do. Most use the term **Driver,** but some may say **Hardware Driver**, which also refers to the **Device Driver.**

Types of Device Drivers:

For almost every device associated with the computer system there exist a Device Driver for the particular hardware. But it can be broadly classified into two types i.e.

1. **Kernel-mode Device Driver –**
   This Kernel-mode device driver includes some generic hardware that loads with the operating system as part of the OS these are BIOS, motherboard, processor, and some other hardware that are part of kernel software. These include the minimum system requirement device drivers for each operating system.

2. **User-mode Device Driver –**
   Other than the devices which are brought by the kernel for working the system the user also brings some devices for use during the using of a system that devices need device drivers to function those drivers fall under User mode device driver. For example, the user needs any plug-and-play action that comes under this.

Virtual Device Driver:

There are also virtual device drivers (V x D), which manage the virtual device. Sometimes we use the same hardware virtually at that time virtual driver controls/manages the data flow from the different applications used by different users to the same hardware.

It is essential for a computer to have the required device drivers for all its parts to keep the system running efficiently. Many device drivers are provided by manufacturers from the beginning and also we can later include any required device driver for our system

Design issues:

The distributed information system is defined as "a number of interdependent computers linked by a network for sharing information among them". A distributed information system consists of multiple autonomous computers that communicate or exchange information through a computer network. **Design issues of distributed system –**

1. **Heterogeneity**: Heterogeneity is applied to the network, computer hardware, operating system and implementation of different developers. A key component of the heterogeneous distributed system client-server environment is middleware. Middleware is a set of services that enables application and end-user to interacts with each other across a heterogeneous distributed system.
2. **Openness**: The openness of the distributed system is determined primarily by the degree to which new resource-sharing services can be made available to the users. Open systems are characterized by the fact that their key interfaces are published. It is based on a uniform communication mechanism and published interface for access to shared resources. It can be constructed from heterogeneous hardware and software.

3. **Scalability**: Scalability of the system should remain efficient even with a significant increase in the number of users and resources connected.
4. **Security**: Security of information system has three components   and availability. Encryption protects shared resources, keeps sensitive information secrets when transmitted.
5. **Failure Handling**: When some faults occur in hardware and the software program, it may produce incorrect results or they may stop before they have completed the intended computation so corrective measures should   implemented to handle this case. Failure handling is difficult in distributed systems because the failure is partial i.e., some components fail while others continue to function
6. **Concurrency**: There is a possibility that several clients will attempt to access a shared resource at the same time. Multiple users make requests on the same resources, i.e., read, write, and update. Each resource must be safe in a concurrent environment. Any object that represents a shared resource in a distributed system must ensure that it operates correctly in a concurrent environment.

7. **Transparency**: Transparency ensures that the distributes system should be perceived as a single entity by the users or the application programmers rather than the collection of autonomous systems, which is cooperating. The user should be unaware of where the services are located and the transferring from a local machine to a remote one should be transparent.

Ram and disk Drivers:

RAM disks are a portion of memory allocated to be used as a partition. The RAM disk driver takes some of that memory and uses it as a storage device. This storage device can then be formatted, mounted, have files saved to it, etc. Every bit of RAM is important for the wellbeing of a system. The bigger a RAM disk is, the less memory will be available for the system.

Formatting

A RAM disk is reformatted after each start up. Exceptions to this rule are RAM

`ram: 1:(", NULL`) for the second and so on.

Sample Configuration

The following sample shows how to configure a RAM disk instance. To add the driver, () which is called at the file system initialization. The size of the storage is defined as the number of sectors reserved for the drive. Each sector in this sample consists of 512 bytes but other sector sizes are also supported. The minimum value for the number of sectors when using a file system is 7. A FAT file system needs a minimum sector size of 512 bytes.

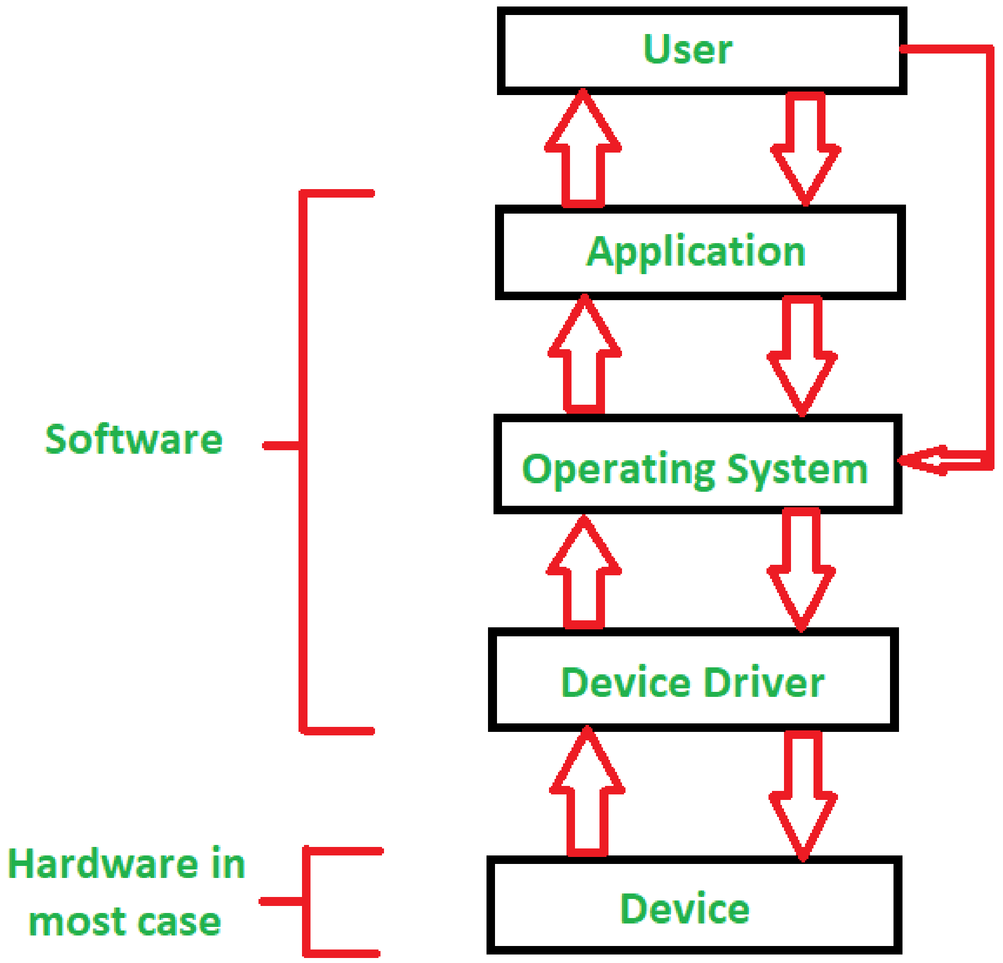
Driver in computing refers to a special kind of software program or a **Device** specific type of software application that controls a specific hardware device that enables different hardware devices to communicate with the computer's Operating System. A device driver communicates with the computer hardware by computer subsystem or computer bus connected to the hardware.

Device Drivers are essential for a computer system to work properly because without a device driver the particular hardware fails to work accordingly, which means it fails in doing the function/action it was created to do. Most use the term **Driver,** but some may say **Hardware Driver**, which also refers to the **Device Driver.**

Working of Device Driver:

Device Drivers depend upon the Operating System's instruction to access the device and perform any particular action. After the action, they also show their reactions by delivering output or status/message from the hardware device to the Operating system. For example, a printer driver tells the printer in which format to print after getting instruction from OS, similarly, A sound card driver is there due to which 1's and 0's data of the MP3 file is converted to audio signals and you enjoy the music. Card reader, controller, modem, network card, sound card, printer, video card, USB devices, RAM, Speakers,need Device Drivers to operate.

The following figure illustrates the interaction between the user, OS, Device driver, and the devices:



Types of Device Driver:

For almost every device associated with the computer system there exist a Device Driver for the particular hardware. But it can be broadly classified into two types i.e.,

1. **Kernel-mode Device Driver –**
   This Kernel-mode device driver includes some generic hardware that loads with the operating system as part of the OS these are BIOS, motherboard, processor, and some other hardware that are part of kernel software. These include the minimum system requirement device drivers for each operating system.
2. **User-mode Device Driver –**
   Other than the devices which are brought by the kernel for working the system the user also brings some devices for use during the using of a system that devices need device drivers

to function those drivers fall under User mode device driver. For example, the user needs any plug-and-play action that comes under this.

Functions and features of device drivers:

Function and Features of Device Driver:
Using interrupts driven device drivers to transfer data to or from hardware devices works well when the amount of data is reasonably low. Device drivers have to be careful when using memory. As they are part of the Linux kernel they cannot use virtual memory. Each time a device driver runs, maybe as an interrupt is received or as a bottom half or task queue handler is scheduled, the current process may change. The device driver cannot rely on a particular process running even if it is doing work on its behalf.

Like the rest of the kernel, device drivers use data structures to keep track of the device that it is controlling. These data structures can be statically allocated; part of the device driver's code, but that would be wasteful as it makes the kernel larger than it need be. Most device drivers allocate kernel, non-paged, memory to hold their data.

Short answers

1. Explain types of drivers?

2. Write about device drivers?

3. Write the functions and features of device drivers?

4. Explain the working of the device drivers?

Long questions

5. Write about Ram, Disk drivers?

6. Write about the architecture of device drivers and its design issues?

7. Write about machine dependent and independent drivers?

8. Design interactive debugging system?

UNIT-5

Linux:

Linux® is an open source system programming. An system programming is the software that directly manages a system's hardware and resources, like CPU, memory, and storage. The SP sits between applications and hardware and makes the connections between all of your software and the physical resources that do the work.

Think about an SP like a car engine. An engine can run on its own, but it becomes a functional car when it's connected with a transmission, axles, and wheels. Without the engine running properly, the rest of the car won't work.

Linux was designed to be similar to UNIX, but has evolved to run on a wide variety of hardware from phones to supercomputers. Every Linux-based SP involves the Linux kernel—which manages hardware resources—and a set of software packages that make up the rest of the operating system.

The SP includes some common core components, like the GNU tools, among others. These tools give the user a way to manage the resources provided by the kernel, install additional software, configure performance and security settings, and more. All of these tools bundled together make up the functional operating system. Because Linux is an open source SP, combinations of software can vary between Linux distributions.

Linux administration tools:

A Linux system administrator's job includes OS installation, upgrade, and monitoring system performance by constantly validating its essential software and functions. Even though every person might have their favourites, it's necessary to have a set of tried and tested tools that enable you to manage your Linux systems as a sysadmin.

Whether you are an experienced sysadmin or have just started exploring Linux, the following tools will offer you practical solutions without incurring a steep learning curve.

1. Zenmap:

Zenmap is the GUI version of the official Nmap security scanner with multi-OS (Windows, Linux, BSD, msacOS) compatibility. Zenmap is free to download tool that allows a new user to understand Nmap easily while offering a host of advanced features for expert users. Its open-source, no-fuss architecture makes the tool a favorite amongst the majority of Linux system administrators.

Zenmap comes in handy when you need to identify issues related to the system network. Although it’s not something that you will need regularly, Zenmap can save your day whenever you need to perform network scanning and troubleshooting.

Not only that, but you can also use this tool for penetration testing and port scanning, which is an added advantage. It's possible to store profiles in Zen map for future scanning needs.

3. Webmin

It’s a versatile system administration tool with an easy-to-work web-based interface for Linux and other Unix-type servers. A sysadmin can configure and modify various internals of a system. These include disk quotas, users, configuration files or services, control Apache web server, BIND DNS server, and different databases such as PHP MySQL.

There is a wide range of third-party extensions you can add to increase the functionality of this tool in case it lacks certain requisite features.

3. Cockpit

Cockpit tends to be every sysadmin’s favorite tool for regular server administration tasks, given its user-friendly features. Beginners who are still wetting their feet with Linux concepts would tend to like Cockpit’s simple, lightweight, yet compelling functionalities, each of which are delivered through an easy-to-use web GUI.

Cockpit assists in essential tasks such as starting or stopping services, journal inspection and storage, multi-server setup, and configuration management. Although this tool is best suited for Red Hat OS, nevertheless, users can run it on several other Linux server distros like Arch Linux, Fedora, CentOS, Ubuntu, and more.

4. Graphical Ping (gping)

Graphical Ping or g ping is an excellent server admin tool that can generate a detailed visual graph by pinging more than one host simultaneously over a predefined period. With numerous handy options, g ping proves to be an essential piece of application for both beginners and advanced network administrators.

5. Shorewall
Firewall setup and configuration in Linux servers can be daunting, especially for novices with less experience with the **tables** utility. Fortunately, Shore wall comes to the rescue with its user-centric solutions.

To achieve a high-level Net filter configuration, users can add an abstraction layer with this tool. Utilizing TC, IP, tables, and tables-restore commands, Shore wall can read configuration files and subsequently configure Net filter inside the Linux kernel.

Additionally, it can divide interfaces into multiple zones and assign a different access level to each zone. Since the tool enables administrators to operate on several systems connected to the interface, they can quickly deploy individual policies for different zones.

Shell programming

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

Command Line Shell

Shell can be accessed by user using a command line interface. A special program called Terminal in linux, mac, OS or Command Prompt in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute. The result is then displayed on the terminal to the user. A terminal in Ubuntu 16.4 system looks like this –

In above screenshot “**ls**” command with “**-l**” option is executed.
It will list all the files in current working directory in long listing format.
Working with command line shell is bit difficult for the beginners because it’s hard to memorize so many commands. It is very powerful, it allows user to store commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called batch files in Windows and **Shell** Scripts in Linux, mac, OS systems.

Graphical shell

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.

Programming concepts

A computer program is a sequence of instructions written using a Computer Programming Language to perform a specified task by the computer.

- A computer program is also called a computer software, which can range from two lines to millions of lines of instructions.

- Computer program instructions are also called program source code and computer programming is also called program coding.

- A computer without a computer program is just a dump box; it is programs that make computers active. The two important terms that we have used in the above definition are:

- Sequence of instruction

- Computer Programming Language A program is a set of instruction written in a language (such as BASIC) understandable by the computer to perform a particular function on the computer. A well written program could be parcelled well to form an application package customized for solving specific type of problem on the computer system. A computer programmer is computer scientist (a professional) skilled in using constructs of programming languages to develop executable and acceptable computer programs.

Linux operating system

An operating system can be described as an interface among the computer hardware and the user of any computer. It is a group of software that handles the resources of the computer hardware and facilitates basic services for computer programs.

An operating system is an essential component of system software within a computer system. The primary aim of an operating system is to provide a platform where a user can run any program conveniently or efficiently.

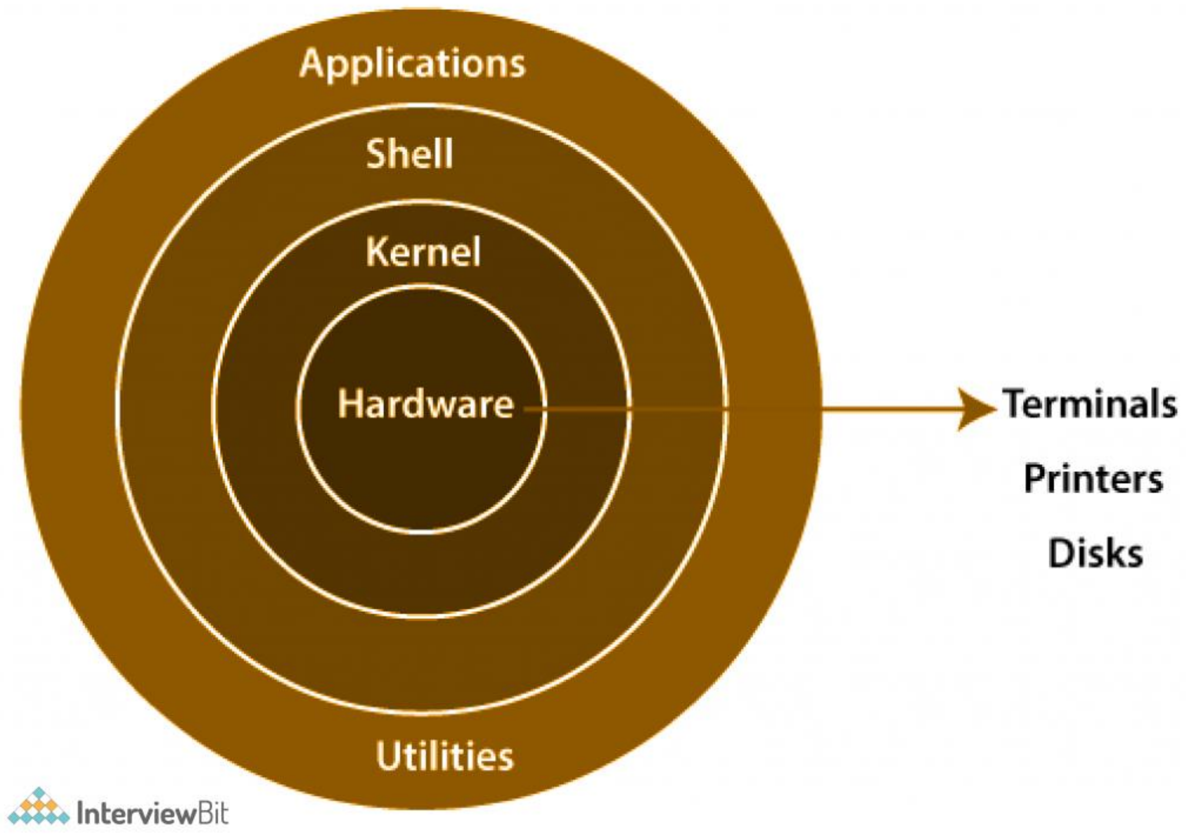
On the other hand, Linux OS is one of the famous versions of the UNIX OS. It is developed to provide a low-cost or free OS for several personal computer system users. Remarkably, it is a complete OS Including an **X Window System, Emacs editor, IP/TCP, GUI** (graphical user interface), etc.

Architecture of Linux system

Linux Architecture

A computer's operating system interface to the hardware is referred to as a software application. A number of software applications are run on operating systems to manage hardware resources on a computer.

The diagram illustrates the structure of the Linux system, according to the layers concept.



The Linux architecture is largely composed of elements such as the Kernel, System Library, Hardware layer, System, and Shell functions.

Kernel: The kernel is one of the fundamental parts of an operating system. It is responsible for each of the primary duties of the Linux OS. Each of the major procedures of Linux is coordinated with hardware directly. The kernel is in charge of creating an appropriate abstraction for concealing trivial hardware or application strategies. The following kernel varieties are mentioned:

1. Monolithic Kernel
2. Micro kernels
3. Exo kernels
4. Hybrid kernels

System Libraries: A set of library functions may be specified as these functions. These functions are implemented by the operating system and do not require code access rights on the kernel modules.

System Utility Programs: A system utility program performs specific and individual jobs.

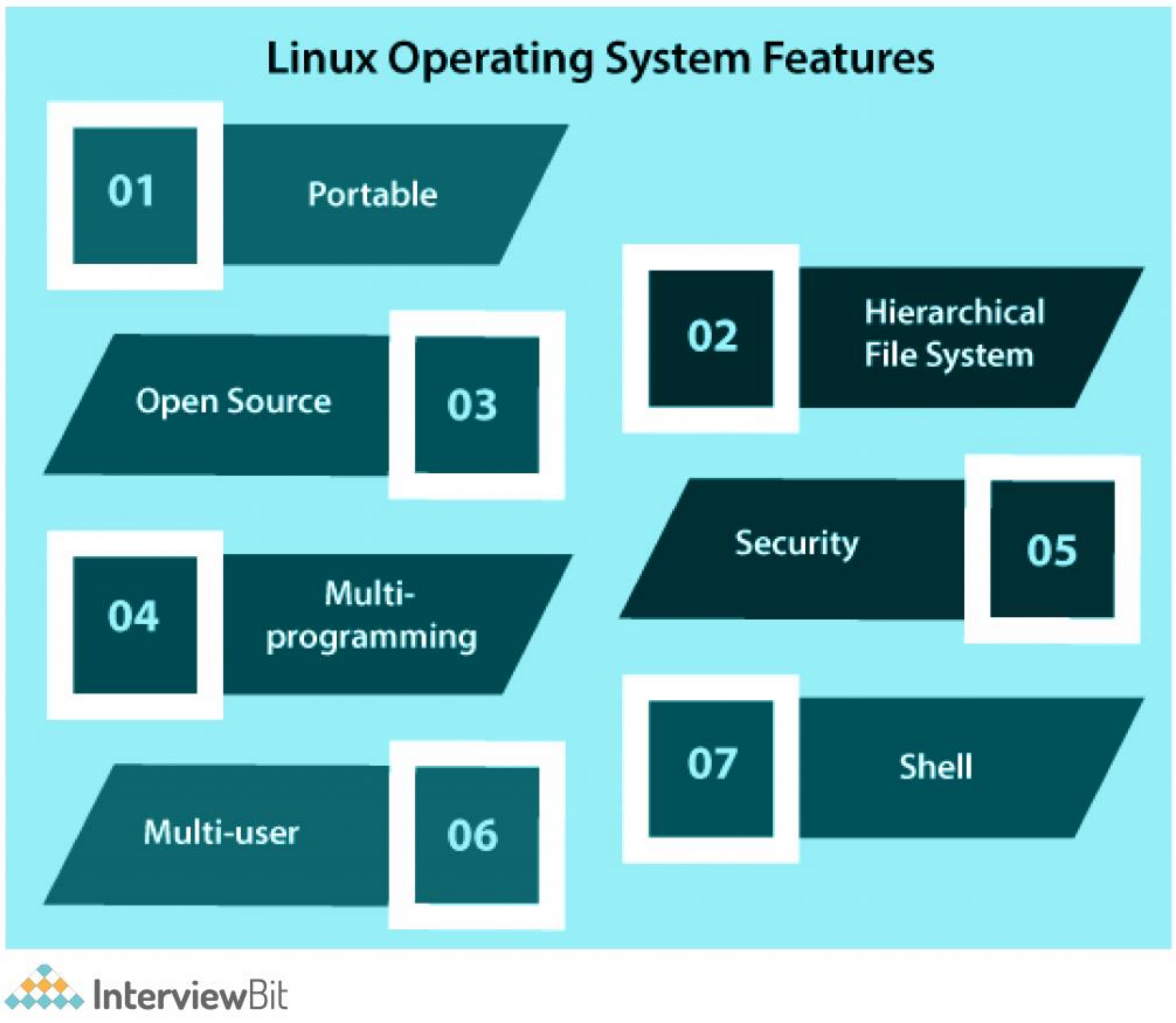
Hardware layer: The hardware layer of Linux is made up of several peripheral devices such as a CPU, HDD, and RAM.

Shell: Different operating systems are classified as graphical shells and command-line shells. A graphical shell is an interface between the kernel and the user. It provides kernel services, and it runs kernel operations. There are two types of graphical shells, which differ in appearance. These operating systems are divided into two categories, which are the graphical shells and command-line shells.

The graphical line shells allow for graphical user interfaces, while the command line shells enable for command line interfaces. As a result, both of these shells operate. However, graphical user interfaces performed using the graphical line shells are faster than those using the command line shells.

Linux Operating System Features

A list of some of the significant attributes of Linux is provided.



Linux Architecture

A computer’s operating system interface to the hardware is referred to as a software application. A number of software applications are run on operating systems to manage hardware resources on a computer.

The diagram illustrates the structure of the Linux system, according to the layers concept.

The Linux architecture is largely composed of elements such as the Kernel, System Library, Hardware layer, System, and Shell functions.

IMPORTANT QUESTIONS:

Short questions

1. Write about Linux?

2. Discuss about Linux architecture?

3. What are the Linux administration tools?

4. Write the commands to be used in Linux?

Long questions

5. Write about Ram, Disk drivers?

6. Write about the architecture of device drivers and its design issues?

S.V.U.COLLEGE OF COMMERCE MANAGEMENT

AND COMPUTER SCIENCE::TIRUPATI

Department of Computer Science

Time: 2HR INTERNAL EXAMINATION-1 Max .Marks :30M

MCA 402: SYSTEM PROGRAMMING

SECTION-A

Answer any five of the following 5*2=10

1. What is system software and its types?

2. Define assembler options?

3. What are the features of system software?

4. Explain about SIC architecture?

5. Write about macro processors?

6. What are the functions of basic macro processors?

7. Write the difference between linkers and loaders?

8. Define basic loader junction?

SECTION-B

Answer any one question from each unit 2*10=20

8. Write about machine architecture and explain about RISC, CISC?

OR

9. Define assembler design options and its implementation?

UNIT-2

10. Explain about the features of independent macro processors?

OR

11. Define macro processor design implementation?

S.V.U.COLLEGE OF COMMERCE MANAGEMENT

AND COMPUTER SCIENCE::TIRUPATI

Department of Computer Science

Time: 2HR INTERNAL EXAMINATION-2 Max.Marks:30M

MCA 402: SYSTEM PROGRAMMING

SECTION-A

Answer any five of the following 5*2=10

1. Write about compiler and interpreter?

2. Define about text editors?

3. Define debugging system?

4. Explain types of drivers?

5. Write about RAM, DISK driver?

6. Write device drivers?

7. Write about the Linux?

8. Define shell concept?

SECTION-B

Answer any one question from each unit 2*10=20

UNIT-1

9. Write about machine dependent and independent?

OR

Design interactive debugging system?

UNIT-2

10. Explain about block driver-1 and its design issues?

OR

.Define executing Linux shell scripts?

12-00-4-02R

MASTER OF COMPUTER APPLICATIONS DEGREE EXAMINATIONS-JULY-2020

FOURTH SEMESTER

Paper -MCA-402: SYSTEM PROGRAMMING

(Under CBCS New Regulation w.e.f2020-2021)

(Common paper to University and all Affiliated Colleges)

Time: 3Hours
Max.Marks:70

PART-A

Answer any five of the following questions .Each question carries 4marks. (5*4=20)

1. a) What is system software and its types?

b) Define assembler options?

c) What are the functions of basic macro processors?

d) Write the difference between linkers and loaders?

e) Write about compiler and interpreter?

f) Define about text editors?

g) Explain types of drivers?

h) Write about RAM, DISK driver?

I) Write about the Linux?

j) Define shell concept?

PART-B

Answer five questions, choosing one question from each Unit. Each question carries 10 marks.

UNIT-1

2.Explain about SIC,RISC,CISC architecture?

OR

4. Define some assembler design options?

UNIT-2

5. Explain dependent and independent macro processor?

OR

6. Explain about loaders and its types with implementation?

UNIT-3

7. Write about different types of debugging system?

OR

8. Explain text editors and its features?

UNIT-4

9. Write about RAM, DISK drivers?

OR

10. Write about architecture of device drivers and its design issues?

UNIT-5

11. Write about shell programming with an example?

OR

12. Explain Linux administration tools?