System Software

DCAP507

Edited by:
Dr. Avinash Bhagat

LOVELY PROFESSIONAL UNIVERSITY

SYSTEM SOFTWARE

Edited By

Dr. Avinash Bhagat

Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

System Software

Objectives: To enable the student to understand the technicalities of system software. Student will learn various components of programming system, they can design their own assemblers, linker and loader. This course will also help them to understand of formal systems.

Sr. No.	Description
1.	***Introduction to System Software:*** Definition, System software, Machine structure, Components of a programming system, Assemblers, linker, loader, compiler, Macros. **Evolution of Operating System:** Operating system architecture, User function of operating system, Batch control language, OS User Viewpoint: Facilities **Note:** Students should be allocated programs which can simulate the working of system programs and can be implemented using C / C++ programming.
2.	***Machine Structure and Assemblers Basic Functions:*** Machine structure, Approach to new machine, Machine language, Assembly language
3.	***Design of Assembler:*** Design of assembler, Data structure, Format of databases, Algorithm, Look for modularity. Table processing: Linear Search, Binary Search, Sorting, Hash Searching
4.	***Macro Language:*** Macro instructions, Features, Implementation
5.	***Introduction to Linking & Loading:*** Loader Schemes – Compile and Go Loaders, General loader scheme, Absolute Loaders, Subroutine linkage, Relocating loaders, Direct linking loaders, Other loader schemes – Binders, Linkers, Loaders, Overlays, Dynamic Binders
6.	***Design of Absolute Loader:*** Design of Direct-linking loader, Problem specification, Data structures, Format of databases ***Note:*** Students should be allocated programs which can simulate the working of system programs and can be implemented using C / C++ programming.
7.	***Programming Languages Concept:*** Importance of high level languages, Features, Data types and Data structure, Storage allocation, Accessing of pointers and label variables
8.	***Programming Languages Concept:*** Functional, modularity, Asynchronous operation – conditions, signals, multitasking.
9.	***Formal Systems and Programming Languages:*** Uses of formal systems in programming languages, Formal Specification, Formal Grammars
10.	***Formal Systems:*** Hierarchy of Languages, Backus-Naur Form – Backus Normal Form – BNF, Canonic systems – Syntax specification, Specification of translation, Recognition and translation algorithm, Canonic systems and Formal systems

CONTENT

Unit 1:	Introduction to System Software *Avinash Bhagat, Lovely Professional University*	1
Unit 2:	Evolution of Operating System *Avinash Bhagat, Lovely Professional University*	14
Unit 3:	Machine Structure and Assembler Basic Functions *Avinash Bhagat, Lovely Professional University*	48
Unit 4:	Design of Assembler *Manish Kumar, Lovely Professional University*	57
Unit 5:	Table Processing : Searching *Avinash Bhagat, Lovely Professional University*	73
Unit 6:	Table Processing : Sorting *Avinash Bhagat, Lovely Professional University*	82
Unit 7:	Macro Language *Nisha Sethi, Lovely Professional University*	90
Unit 8;	Introduction to Linking and Loading *Nisha Sethi, Lovely Professional University*	109
Unit 9:	Design of an Absolute Loader *Nisha Sethi, Lovely Professional University*	124
Unit 10:	Programming Languages Concept (I) *Avinash Bhagat, Lovely Professional University*	144
Unit 11:	Programming Languages Concept (II) *Avinash Bhagat, Lovely Professional University*	170
Unit 12:	Formal Systems and Programming Languages *Nisha Sethi, Lovely Professional University*	184
Unit 13:	Formal Systems *Nisha Sethi, Lovely Professional University*	197
Unit 14:	Canonic Systems *Nisha Sethi, Lovely Professional University*	209

Unit 1: Introduction to System Software

Notes

CONTENTS

Objectives

Introduction

1.1 System Software

1.1.1 System Software Concept

1.2 Machine Structure

1.2.1 Machine Dependency of System Software

1.2.2 SIC Machine

1.2.3 SIC Machine Architecture

1.2.4 SIC/XE Machine Architecture

1.3 Major Components of a Programming System

1.3.1 Operating System

1.3.2 Language Translators

1.3.3 Loader

1.3.4 Linker

1.3.5 Macros

1.4 Summary

1.5 Keywords

1.6 Review Questions

1.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Define system software
- Understand the concept of system software
- Discuss machine structure
- Illustrate various components of a programming system

Introduction

System software includes numerous programs that assist the operations of a computer. This makes it probable for the consumer to concentrate on an application or other difficulty to be solved, without requiring to know the details of how the machine functions internally. Instances of system software are text-editors, compilers, loaders or linkers, debuggers, assemblers, and operating systems.

Notes

1.1 System Software

System software is defined as software needed to assist the production or implementation of application programs but which is not particular to any specific application.

1.1.1 System Software Concept

System Software is a set of programs to carry out a number of system functions such as file editing, resource organization, I/O management and storage organization. The trait in which system software varies from application software is machine reliance. An application program is chiefly related with the clarification of some problem, by means of the computer as a tool. System programs alternatively are projected to assist the operation and utilization of the computer itself, instead of any specific application. Therefore, they are typically connected to the architecture of the machine on which they are executed.

Example: Assemblers convert mnemonic instructions into machine code. The instruction formats, addressing modes are of direct concern in assembler design.

There are some facets of system software that do not directly rely on the sort of computing system being supported. These are called machine-independent traits.

Example: The common design and logic of an assembler is fundamentally the similar on most computers.

An instruction is a set of programs that has to be fed to the computer for procedure of computer system all together. When you switch on the computer the programs written in ROM is implemented which triggers dissimilar units of your computer and makes it prepared for you to work on it. This set of program can be considered as system software. Thus system software may be defined as a set of one or more programs intended to manage the operation of computer system.

System software are common programs intended for carrying out tasks like handling all operations needed to move data into and out of the computer. It converses with printers, card reader, disk, tapes, etc. monitor the use of a variety of hardware such as memory, CPU, etc. Also system software is necessary for the expansion of applications software.

Did u know? System Software permits application packages to be executed on the computer with less time and effort.

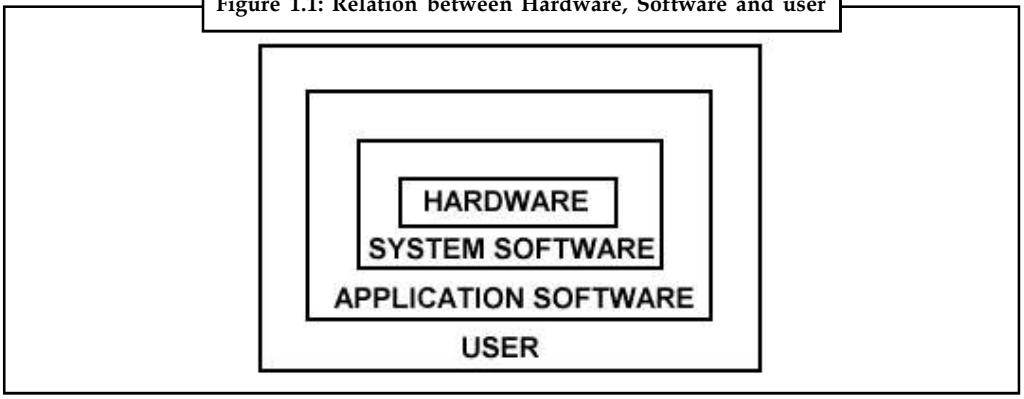
Caution It is not possible to execute application software not including system software.

Development of system software is a multifaceted task and it needs extensive knowledge of computer technology. Because of its intricacy it is not produced in house. Computer manufactures construct and supply this system software with the computer system. DOS, UNIX and WINDOWS are some of the broadly used system software. Out of these UNIX is a multi-user operating system while DOS and WINDOWS are PC-based.

Thus without system software it is impracticable to operate your computer. The following figure is shown as Figure 1.1 relation between hardware, software and you as a user of computer system.

Notes

Figure 1.1: Relation between Hardware, Software and user



The most significant trait in which most system software vary from application software is machine dependency. An application program is chiefly related with the solution to some problem, by means of computer as a tool. The concentration is on the application, not on the application system.

Notes System programs are projected to support the operation and use of the computer itself, instead of any specific application. They are generally related to the architecture of the machine on which they are to run.

Task Differentiate between system software and application software.

Self Assessment

Fill in the blanks:

1. ..... is defined as software needed to assist the production or implementation of application programs but which is not particular to any specific application.
2. An ..... program is chiefly related with the clarification of some problem, by means of the computer as a tool.
3. An ..... is a set of programs that has to be fed to the computer for procedure of computer system all together.
4. The trait in which system software varies from application software is ..... reliance.
5. System software may be defined as a set of one or more programs intended to manage the operation of ..... system.

1.2 Machine Structure

1.2.1 Machine Dependency of System Software

An assembler is considered as a system software which converts mnemonic instructions into machine code; the instruction formats, addressing modes, etc., are of direct relation in assembler design. Likewise, compilers must produce machine language code, considering such hardware

Notes

traits as the number and the types of registers and machine instructions obtainable. Operating systems are directly related with the management of almost all of the resources of a computer system. For example,

1. When you took the initial programming course:
   - Text editor: creates and alters the program
   - Compiler: convert programs into machine language
   - Loader or linker: load machine language program into memory and geared up for implementation
   - Debugger: assist in detecting errors in the program.
2. When you engraved programs in assembler language:
   - Assembler: convert assembly program into machine language
   - Macro processor: convert macros instructions into its definition.
3. When you manage all of these processes
   - By networking with the OS
   - The significant machine structures utilized in the design of system software are:
     - Memory structure
     - Registers
     - Data formats
     - Instruction formats
     - Addressing modes
     - Instruction set.

1.2.2 SIC Machine

SIC points to Simplified Instruction Computer which is a imaginary computer that has been intended to comprise the hardware traits most frequently found on real machines, while averting unusual and immaterial complexities. This permits to evidently separate the central concepts of a system software from the execution details related with a specific machine.

1.2.3 SIC Machine Architecture

- Memory
  - 215 bytes in the computer memory
  - 3 consecutive bytes form a word
  - 8-bit bytes
- Registers

Notes

Mnemonic	Number	Special
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

- Data Formats
  - Integers are accumulated as 24-bit binary numbers; 2's complement demonstration is used for negative values
  - No floating-point hardware.
- Instruction Formats

Opcode(8)	x	Address(15)

- Addressing Modes

Mode	Indication	Target address calculation
Direct	x = 0	TA=address
Indexed	x = 1	TA=address+(X

- Instruction Set
  - integer arithmetic operations: ADD, SUB, MUL, DIV, etc.

    All arithmetic operations entail register A and a word in memory, with the consequence being left in the register.
  - comparison: COMP

    COMP compares the value in register A with a word in memory, this instruction sets a condition code CC to specify the result
  - conditional jump instructions: JLT, JEQ, JGT

    these instructions test the setting of CC and jump accordingly
  - subroutine linkage: JSUB, RSUB

    JSUB jumps to the subroutine, positioning the return address in register L

    RSUB returns by jumping to the address enclosed in register L
- Input and Output
  - Input and output are executed by conveying 1 byte at a time to or from the right-most 8bits of register A
  - The Test Device (TD) instruction tests whether the addressed device is prepared to send or obtain a byte of data
  - Read Data (RD)
  - Write Data (WD)

Notes

- Contents of status word register

Bit position	Field name	Use
0	MODE	0=user mode, 1=supervisor mode
1	IDLE	0=running, 1=idle
2~5	ID	Process identifier
6~7	CC	Condition code
8~11	MASK	Interrupt mask
12~15		Unused
16~23	ICODE	Interruption code

1.2.4 SIC/XE Machine Architecture

- Memory
  - Almost the similar as formerly described for SIC
  - However 1 MB(220 bytes)
- More Registers

Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register
T	5	General working register
F	6	Floating-point acumulator (48bits)

- Data Formats
- Floating-point data type: frac*2(exp-1024)
  - frac: 0~1
  - exp: 0~2047
- Instruction Formats

Format 1		
op(8)		
Format 2		
op(8)	**r1(4)**	**r2(4)**

Format 3						e=0	
op(6)	n	I	x	b	p	e	disp(12)

Format 4						e =1	
op(6)	n	I	x	b	p	e	Address(20)

Notes

- Instruction Set
  - new registers: LDB, STB, etc.
  - floating-point arithmetic: ADDF, SUBF, MULF, DIVF
  - register move: RMO
  - register-register arithmetic: ADDR, SUBR, MULR, DIVR
  - supervisor call: SVC

    produces an interrupt for OS
- Input/Output

  SIO, TIO, HIO: start, test, halt the operation of I/O device
- SIC/XE Instruction formats
  - Bigger memory means an address cannot fit into a 15-bit field

    Expand addressing capacity
  - Use some form of relative addressing-> instruction

    format 3
  - Enlarge the address field to 20 bits-> instruction format 4

    Additional instructions do not reference memory
  - Instruction format 1 & 2

Self Assessment

Fill in the blanks:

6. Operating systems are directly related with the management of almost all of the ..... of a computer system.
7. SIC points to ..... which is a imaginary computer that has been intended to comprise the hardware traits most frequently found on real machines, while averting unusual and immaterial complexities.
8. The ..... instruction tests whether the addressed device is prepared to send or obtain a byte of data.
9. ..... jump instructions JLT, JEQ, JGT test the setting of CC and jump accordingly.
10. Supervisor call ..... produces an interrupt for OS.

1.3 Major Components of a Programming System

The Major Components of a programming system are explained as below:

1. Operating system
2. Language translators
   (a) Compilers
   (b) Interpreters
   (c) Assemblers
   (d) Preprocessors

Notes

3. Loaders
4. Linkers
5. Macro processors.

1.3.1 Operating System

- It is the most significant system program that performs as an interface among the users and the system. It makes the computer simpler to utilize.
- It offers an interface that is more comprehensible than the fundamental hardware.
- The functions of OS are: Process management, Memory management, Resource management, I/O operations, Data management, and offering security to user's job.

1.3.2 Language Translators

Language translator is the program that takes an input program in one language and generates an output in another language.

Source Program → Language Translator → Object Program

Compiler

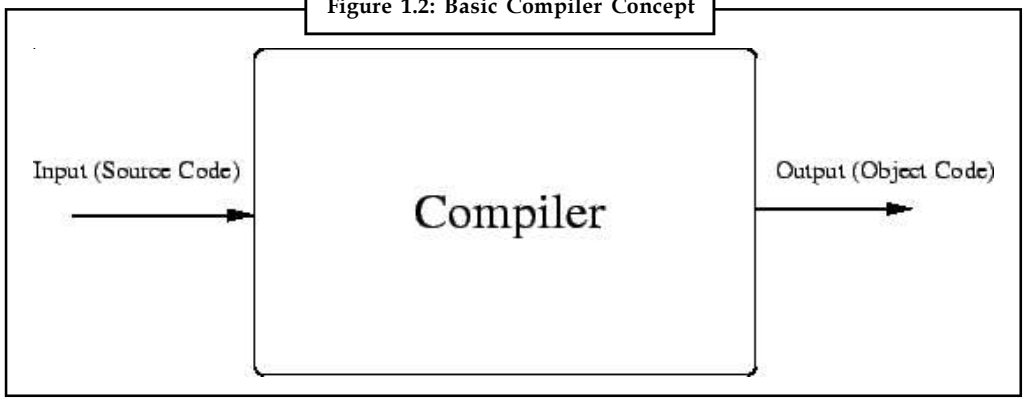
Compiler is a program whose job is to accept as input a source program written in a some high-level language and to generate as output an object code which can be executed on the computer.

A compiler converts programs from one language to the other as displayed in Figure 1.2. It has a nick name known as translator. Usually, a compiler converts programs written in high-level language to assembly languages. Assemblers then translate programs written in assembly languages to machine code. Interpreters such as Java virtual machine executes programs by interpreting intermediate code like Java bytecode. By and large, a program that converts input in some form to output in some form can be known as a compiler. There are some fundamental properties for a compiler.

- A compiler must be error-free.
- A compiler must at all times terminate, regardless of what the input appears like.

Caution A compiler should effort to find as many errors as possible throughout a single compilation pass.

Figure 1.2: Basic Compiler Concept



Notes

Interpreters

Interpreter is a translator program that converts a statement of high-level language to machine language and implements it instantly. The program instructions are taken line by line. The interpreter reads the source program and accumulates it in memory. During interpretation, it takes a source statement, identifies its meaning and carries out actions which increases it. This involves computational and I/Oactions. Program Counter (PC) specifies which statement of the source program is to be interpreted next. This statement would be subjected to the interpretation cycle. The interpretation cycle includes the following steps:

- Fetch the statement.
- Scrutinize the statement and determine its meaning.
- Execute the meaning of the statement.

The following are the traits of interpretation:

- The source program is preserved in the source form itself, no target program exists.
- A statement is examined during the interpretation.

Assembler

An assembly language program is not directly executable. To be executed, first it is required to change it into its machine language equivalent.

An Assembler is a program which is used to translate an assembly language program into its machine level language equivalent. The program in assembly language is termed as source code and its machine language equivalent is called object program.

Notes Once the object program is created, it is transferred into the computer's primary memory using the system's loader. Here, another program called link editor passes computer control to the first instruction in the object program, and then the execution starts and proceeds till the end of the program.

Assembler is a program that automatically converts the source program written in assembly language and to create as output an object code written in binary machine code.

The third party of language translator, an assembler translates only assembly language statements into machine language. Assembly languages are utilized almost completely by professional programmers to write competent programming code. An assembler functions like compiler, generating a stored object module. A computer system usually functions only one assembly language; thus, only one assembler is required. Users of any language translator program should identify the difference between it and the programming language it converts. A programming language is a set of rules for coding program commands, while a language translator is a system software program that translates code written under the rules into the bits and bytes that the computer recognizes. The translation into machine language appears while the program executes. The third type of translator an assembler functions like a computer but it translates only assembly language programs. An assembler is a language translator that translates assembly language instruction into machine language. A language translator is a systems software program that translates the programmer's applications program into machine language.

Example: If you generate a program in he BASIC or COBOL programming languages, you cannot execute it on a computer until you summon the appropriate translator to paraphrase your comments in machine language.

Notes

1.3.3 Loader

Loader is a custom that loads an object program and organizes it for execution.

In a computer operating system, a loader is a constituent that positions a specified program (which can be an application or, in some cases, part of the operating system itself) in offline storage (like a hard disk), loads it into main storage (in a personal computer, it's known as random access memory), and provides that program control of the computer (permits it to execute its instructions).

A program that is loaded may itself enclose components that are not primarily loaded into main storage, but can be loaded if and when their logic is required. In a multitasking operating system, a program that is at times called a dispatcher organizes the computer processor's time between different tasks and calls the loader when a program related with a task is not already in main storage.

1.3.4 Linker

Linking is the process of binding an exterior reference to the accurate link time address. An external reference is said to be uncertain until linking is performed for it. It is said to be resolved when its linking is accomplished.

1.3.5 Macros

A macro displays a generally used group of statements in the source programming language. The macro processor substitutes every macro instruction with the equivalent group of source language statements. This is known as expanding the macros.

Macro instructions permit the programmer to write a shorthand edition of a program, and leave the mechanical details to be managed by the macro processor.

Example: Presume that it is required to save the contents of all registers before calling a subprogram.

Did u know? The most common utilization of macro processors is in assembler language programming.

However, macro processors can also be used with high-level programming languages, operating system command languages, etc. A macro call is abbreviation (or name) for several code. A macro definition is a succession of code that has a name (macro call). A macro processor is a program that replaces and specializes macro definitions for macro call.

Task Define macro instruction.

Self Assessment

Fill in the blanks:

11. ..... is the most significant system program that performs as an interface among the users and the system.

Notes

12. ..... is a program whose job is to accept as input a source program written in a some high-level language and to generate as output an object code which can be executed on the computer.

13. ..... is a translator program that converts a statement of high-level language to machine language and implements it instantly.

14. ..... is a custom that loads an object program and organizes it for execution.

15. The ..... substitutes every macro instruction with the equivalent group of source language statements.

1.4 Summary

- System software is defined as software needed to assist the production or implementation of application programs but which is not particular to any specific application.
- An instruction is a set of programs that has to be fed to the computer for procedure of computer system all together.
- SIC points to Simplified Instruction Computer which is a imaginary computer that has been intended to comprise the hardware traits most frequently found on real machines, while averting unusual and immaterial complexities.
- Operating System is the most significant system program that performs as an interface among the users and the system. It makes the computer simpler to utilize.
- Language translator is the program that takes an input program in one language and generates an output in another language.
- Compiler is a program whose job is to accept as input a source program written in a some high-level language and to generate as output an object code which can be executed on the computer.
- Interpreter is a translator program that converts a statement of high-level language to machine language and implements it instantly.
- An Assembler is a program which is used to translate an assembly language program into its machine level language equivalent.
- Loader is a custom that loads an object program and organizes it for execution and linking is the process of binding an exterior reference to the accurate link time address.
- A macro displays a generally used group of statements in the source programming language.

1.5 Keywords

Assembler: An Assembler is a program which is used to translate an assembly language program into its machine level language equivalent.

Compiler: Compiler is a program whose job is to accept as input a source program written in a some high-level language and to generate as output an object code which can be executed on the computer.

Instruction: An instruction is a set of programs that has to be fed to the computer for procedure of computer system all together.

Notes

Interpreter: It is a translator program that converts a statement of high-level language to machine language and implements it instantly.

Loader: Loader is a custom that loads an object program and organizes it for execution.

Macro: A macro displays a generally used group of statements in the source programming language.

Operating System: It is the most significant system program that performs as an interface among the users and the system. It makes the computer simpler to utilize.

SIC: SIC points to Simplified Instruction Computer which is a imaginary computer that has been intended to comprise the hardware traits most frequently found on real machines, while averting unusual and immaterial complexities.

System Software: It is defined as software needed to assist the production or implementation of application programs but which is not particular to any specific application.

1.6 Review Questions

1. What is system software? Explain the concept of system software with examples.
2. Illustrate the relation between hardware, software, and user of the system.
3. Illustrate the concept of general machine structure.
4. Illustrate the concept of Machine dependency of system software.
5. Explain data formats and instruction sets in SIC Machine architecture.
6. Describe various registers used in SIC/XE Machine Architecture.
7. What do you mean by system programming? Enlighten the components of system programming.
8. Make distinction between compilers and interpreters.
9. Illustrate the function of assemblers with example.
10. Explain the use of loaders and linkers.

Answers: Self Assessment

1. System software
2. application
3. instruction
4. machine
5. computer
6. resources
7. Simplified Instruction Computer
8. Test Device (TD)
9. Conditional
10. SVC
11. Operating System
12. Compiler
13. Interpreter
14. Loader
15. macro processor

1.7 Further Readings

Notes

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online link

www.osdata.com/system/system.htm

Notes

Unit 2: Evolution of Operating System

CONTENTS

Objectives

Introduction

2.1 Evolution of Operating System

2.1.1 Operating Systems Classification

2.2 Operating System Architecture

2.2.1 Monolithic Systems

2.2.2 Layered Systems

2.2.3 Exokernel

2.2.4 Client-server Model

2.2.5 Virtual Machines

2.3 User Function of Operating System

2.3.1 Other Operating System Functions

2.3.2 Operating System Concerns

2.4 Batch Control Language

2.4.1 The Role of Graphics

2.4.2 Potential for Objects in Control

2.5 OS User Viewpoint – Facilities

2.5.1 Program Execution

2.5.2 I/O Operations

2.5.3 File System Manipulation

2.5.4 Communications

2.5.5 Error Detection

2.6 Summary

2.7 Keywords

2.8 Review Questions

2.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concept of operating system
- Discuss operating system architecture
- Illustrate user function of operating system
- Understand batch control language
- Discuss operating system user facilities

Introduction

Notes

If we just build a computer, using physical components, we end up with a lot of assembled metal, plastic and silicon. In this state the computer can do nothing useful. To turn it into one of the most versatile tools known to man we need software. We need applications that allow us to write letters develop (other) programs and calculate cash flow forecasts. But, if all we have are the applications, then each programmer has to deal with the complexities of the hardware. If a program requires data from a disc, the programmer would need to know how every type of disc worked and then be able to program at a low level in order to extract the data. In addition, the programmer would have to deal with all the error conditions that could arise. As an instance, it is a lot easier for a programmer to say READ NEXT RECORD than have to worry about moving the read/write heads, waiting for the correct sector to come around and then reading the data (and this is very simplified view of what actually happens). It was clear, from an early stage in the development of computers, that there needed to be a layer of software that sat between the hardware and the software. This had led to this view of a computer. The bottom layer of the hardware consists of the integrated circuits, the cathode ray tubes, the wire and everything else that electrical engineers use to build physical devices. The next layer of hardware (microprogramming) is actually software (sometimes called firmware) that provides basic operations that allow communication with the physical devices. This software is normally in Read Only Memory (and that is why it is called firmware).

The machine language layer defines the instruction set that are available to the computer. Again, this layer is really software but it is considered as hardware as the machine language will be part and parcel of the hardware package supplied by the hardware supplier. The operating system is a layer between this hardware and the software that we use. It allows us (as programmers) to use the hardware in a user friendly way. Furthermore, it allows a layer of abstraction from the hardware.

Example: We can issue a print command without having to worry about what printer is physically connected to the computer.

In fact, we can even have different operating systems running on the same hardware (e.g., DOS, Windows and UNIX) so that we can use the hardware using an operating system that suits us. On top of the operating system, we have the system software. This is the software that allows us to do useful things with the computer, but does not directly allow us to use the computer for anything that is useful in the real world (this is a broad statement which we could argue about but system software really only allows us to use the computer more effectively). It is important to realize that system software is not part of the operating system. However, much system software is supplied by the computer manufacturer that supplied the operating system. But, system software can be written by programmers and many large companies have teams of programmers (often called system programmers) who's main aim is to make the operating system easier to use for other people (typically programmers) in the organization.

The main difference between the operating system and system software is that the operating system runs in kernel or supervisor mode where as the system software and applications run in user mode. This means that the hardware stops the user programs directly accessing the hardware.

Example: Due to the above restriction, you cannot, say, write your own disc interrupt handler and replace the one in the operating system. However, you can write your own (say) command shell and replace the one supplied with the computer.

Finally, at the top level we have the application programs that, at last, allow us to do something useful.

Notes

2.1 Evolution of Operating System

Operating systems are so ubiquitous in computer operations that one hardly realizes its presence. Most likely you must have already interacted with one or more different operating systems. The names like DOS, UNIX, etc. should not be unknown to you. These are the names of very popular operating systems.

Try to recall when you switch on a computer what all happens before you start operating on it. In a typical personal computer scenario, this is what happens. Some information appears on the screen. This is followed by memory counting activity. Keyboard, disk drives, printers and other similar devices are verified for proper operation. These activities always occur whenever the computer is switched on or reset. There may be some additional activities on some machine also. These activities are called power-on routines. Why do these activities always happen? You will learn about it elsewhere in this unit.

You know a computer does not do anything without properly instructed. Thus, for each one of the above power-on activities also, the computer must have instructions. These instructions are stored in a non-volatile memory, usually in a ROM. The CPU of the computer takes one instruction from this ROM and executes it before taking next instruction. ROMs are of finite size. They can store only a few kilobytes of instructions. One by one the CPU executes these instructions. Once, these instructions are over, the CPU must obtain instructions from somewhere. Where are these instructions stored and what are their functions?

Usually, these instructions are stored on a secondary storage device like hard disk, floppy disk or CD-ROM disk. These instructions are collectively known as operating system and their primary function is to provide an environment in which users may execute their own instructions. Once the operating system is loaded into the main memory, the CPU starts executing it. Operating systems run in an infinite loop, each time taking instructions in the form of commands or programs from the users and executing them. This loop continues until the user terminates the loop when the computer shuts down. In order to exploit the most from a computer, therefore, a deep understanding of operating system is a must.

Previously operating systems have been strongly associated to the computer architecture, it is good thought to learn the history of operating systems from the architecture of the computers on which they execute.

Operating systems have evolved via number of distinct stages or generations which matches approximately to the decades.

The 1940's – First Generation

The most primitive electronic digital computers had no operating systems. Machines of the time were so prehistoric that programs were frequently entered one bit at time on rows of mechanical switches (plug boards). Programming languages were unidentified (not even assembly languages). Operating systems were unheard of.

The 1950's – Second Generation

By the early 1950's, the routine had enhanced to some extent with the introduction of punch cards. The General Motors Research Laboratories executed the first operating systems in early 1950 are for their IBM 701. The system of the 50's usually ran one job at a time. These were known as single-stream batch processing systems since programs and data were submitted in groups or batches.

Notes

The 1960's – Third Generation

The systems of the 1960's were also known as batch processing systems, but they were able to take improved advantage of the computer's resources by executing numerous jobs at once. So operating systems designers produced the concept of multiprogramming in which numerous jobs are in main memory at once; a processor is switched from job to job as required to keep numerous jobs advancing while keeping the peripheral devices in function.

Example: On the system without multiprogramming, when the present job paused to wait for other I/O operation to absolute, the CPU simply sat idle until the I/O finished. The solution for this difficulty that evolved was to partition memory into numerous pieces, with a dissimilar job in every partition. While one job was waiting for I/O to complete, another job could be by means of the CPU.

Another major characteristic in third-generation operating system was the technique known as spooling (concurrent peripheral operations on line). In spooling, a high-speed device akin to a disk interposed among a running program and a low-speed device involved with the program in input/output. Rather than writing directly to a printer, for instance, outputs are written to the disk. Programs can execute to completion faster, and other programs can be initiated sooner when the printer turns out available, the outputs may be printed.

Notes Spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed.

Another trait present in this generation was time-sharing technique, a deviation of multiprogramming technique, in which every user has an online (i.e., directly associated) terminal. As the user is present and interacting with the computer, the computer system must respond quickly to user requests, or else user productivity could undergo. Timesharing systems were produced to multiprogram huge number of concurrent interactive consumers.

Fourth Generation

With the expansion of LSI (Large Scale Integration) circuits, chips, operating system appeared in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the peak that it turn out to be possible to construct desktop computers as influential as the mainframes of the 1970s. Two operating systems have conquered the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines by means of the Intel 8088 CPU and its successors, and UNIX, which is foremost on the huge personal computers using the Motorola 6899 CPU family.

2.1.1 Operating Systems Classification

The variations and differences in the nature of different operating systems may give the impression that all operating systems are absolutely different from each-other. But this is not true. All operating systems contain the same components whose functionalities are almost the same. For instance, all the operating systems perform the functions of storage management, process management, protection of users from one-another, etc. The procedures and methods that are used to perform these functions might be different but the fundamental concepts behind these techniques are just the same.

Notes

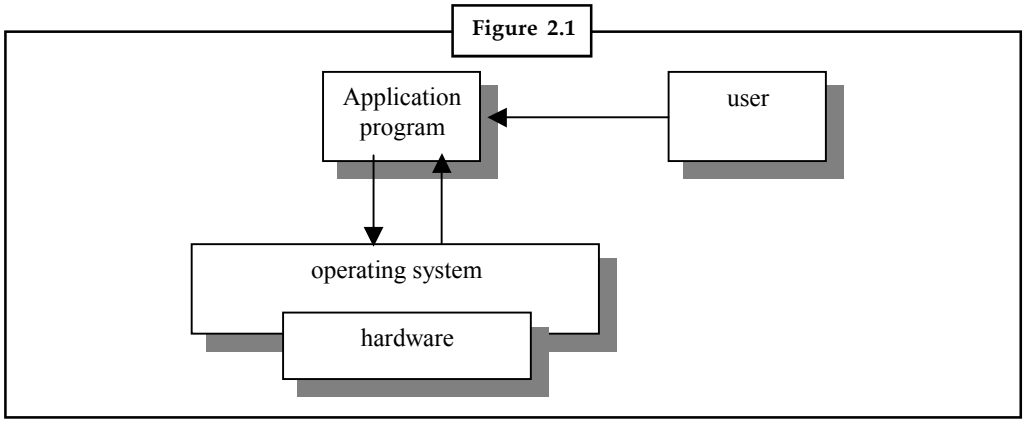
Did u know? Operating systems in general, perform similar functions but may have distinguishing features.

Operating systems can be classified into different categories on different bases. Let us quickly look at the different types of operating systems.

Single User – Single Processing System

The simplest of all the computer systems is a single use-single processor system. It has a single processor, runs a single program and interacts with a single user at a time. The operating system for this system is very simple to design and implement. However, the CPU is not utilized to its full potential, because it sits idle for most of the time.

Figure 2.1



In this configuration, all the computing resources are available to the user all the time. Therefore, operating system has very simple responsibility.

Example: A representative example of this category of operating system is MS-DOS.

Batch Processing Systems

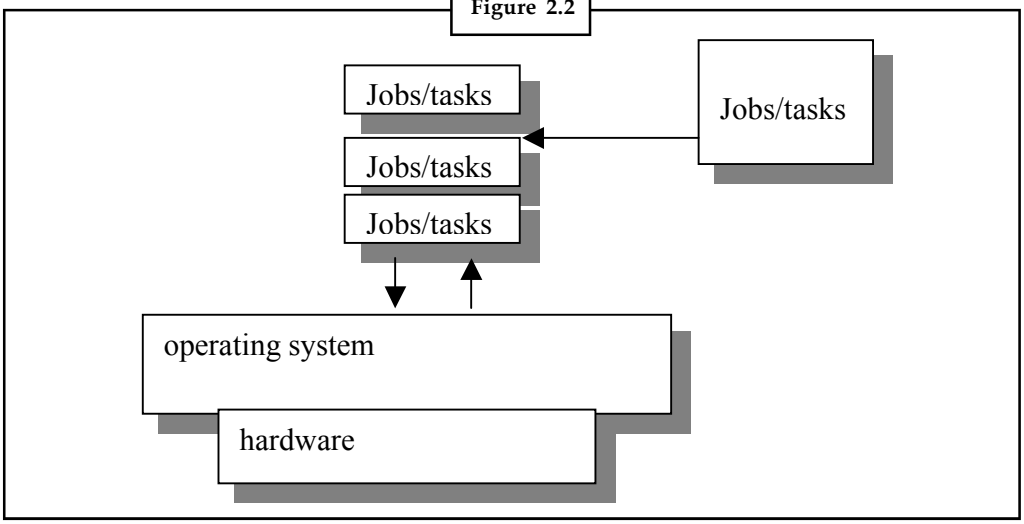
The main function of a batch processing system is to automatically keep executing one job to the next job in the batch. The main idea behind a batch processing system is to reduce the interference of the operator during the processing or execution of jobs by the computer. All functions of a batch processing system are carried out by the batch monitor. The batch monitor permanently resides in the low end of the main store. The current jobs out of the whole batch are executed in the remaining storage area. In other words, a batch monitor is responsible for controlling all the environment of the system operation. The batch monitor accepts batch initiation commands from the operator, processes a job, performs the job of job termination and batch termination.

In a batch processing system, we generally make use of the term 'turn around time'. It is defined as the time from which a user job is given to the time when its output is given back to the user. This time includes the batch formation time, time taken to execute a batch, time taken to print results and the time required to physically sort the printed outputs that belong to different jobs. As the printing and sorting of the results is done for all the jobs of batch together, the turn around time for a job becomes the function of the execution time requirement of all jobs in the batch. You can reduce the turn around time for different jobs by recording the jobs or faster input output media like magnetic tape or disk surfaces. It takes very less time to read a record from

Notes

these media. For instance, it takes round about five milliseconds for a magnetic 'tape' and about one millisecond for a fast fixed-head disk in comparison to a card reader or printer that takes around 50-100 milliseconds. Thus, if you use a disk or tape, it reduces the amount of time the central processor has to wait for an input output operation to finish before resuming processing. This would reduce the time taken to process a job which indirectly would bring down the turnaround times for all the jobs in the batch.

Figure 2.2



Another term that is commonly used in a batch processing system is Job Scheduling. Job scheduling is the process of sequencing jobs so that they can be executed on the processor. It recognizes different jobs on the basis of First-Come-First-Served (FCFS) basis. It is because of the sequential nature of the batch. The batch monitor always starts the next job in the batch. However, in exceptional cases, you could also arrange the different jobs in the batch depending upon the priority of each batch. Sequencing of jobs according to some criteria require scheduling the jobs at the time of creating or executing a batch. On the basis of relative importance of jobs, certain 'priorities' could be set for each batch of jobs. Several batches could be formed on the same criteria of priorities. So, the batch having the highest priority could be made to run earlier than other batches. This would give a better turn around service to the selected jobs.

Now, we discuss the concept of storage management. At any point of time, the main store of the computer is shared by the batch monitor program and the current user job of a batch. The big question that comes in our mind is-how much storage has to be kept for the monitor program and how much has to be provided for the user jobs of a batch. However, if too much main storage is provided to the monitor, then the user programs will not get enough storage. Therefore, an overlay structure has to be devised so that the unwanted sections of monitor code don't occupy storage simultaneously.

Next we will discuss the concept of sharing and protection. The efficiency of utilization of a computer system is recognized by its ability of sharing the system's hardware and software resources amongst its users. Whenever, the idea of sharing the system resources comes in your mind certain doubts also arise about the fairness and security of the system. Every user wants that all his reasonable requests should be taken care of and no intentional and unintentional acts of other users should fiddle with his data. A batch processing system guarantees the fulfillment of these user requirements. All the user jobs are performed one after the other. There is no simultaneous execution of more than one job at a time. So, all the system resources like storage IO devices, central processing unit, etc. are shared sequentially or serially. This is how sharing of resources is enforced on a batch processing system. Now, arises the question of protection.

Notes

Though all the jobs are processed simultaneously, this too can lead to loss of security or protection. Let us suppose that there are two users A and B. User A creates a file of his own. User B deletes the file created by User A. There are so many other similar instances that can occur in our day to day life. So, the files and other data of all the users should be protected against unauthorized usage. In order to avoid such loss of protection, each user is bound around certain rules and regulations. This takes the form of a set of control statements, which every user is required to follow.

Multiprogramming Operating System

The objective of a multiprogramming operating system is to increase the system utilization efficiency. The batch processing system tries to reduce the CPU idle time through operator interaction. However, it cannot reduce the idle time due to IO operations. So, when some IO is being performed by the currently executing job of a batch, the CPU sits idle without any work to do. Thus, the multiprogramming operating system tries to eliminate such idle times by providing multiple computational tasks for the CPU to perform. This is achieved by keeping multiple jobs in the main store. So, when the job that is being currently executed on the CPU needs some IO, the CPU passes its requirement over to the IO processor. Till the time the IO operation is being carried out, the CPU is free to carry out some other job. The presence of independent jobs guarantees that the CPU and IO activities are totally independent of each other. However, if it was not so, then it could lead to some erroneous situations leading to some time-dependent errors. Some of the most popular multiprogramming operating systems are: UNIX, VMS, WindowsNT, etc.

A multiprogramming supervisor has a very difficult job of managing all the activities that take place simultaneously in the system. He has to monitor many different activities and react to a large number of different situations in the course of working. The multiprogramming supervisor has to look through the following control functions:

Time Sharing or Multitasking System

Time sharing, or multitasking, is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running.

An interactive, or hands-on, computer system provides online communication between the user and the system.

Did u know? The user gives instructions to the operating system or to a program directly, and receives an immediate response.

Usually, a keyboard is used to provide input, and a display screen (such as a cathode-ray tube (CRT), or monitor) is used to provide output. When the operating system finishes the execution of one command, it seeks the next "control statement" not from a card reader, but rather from the user's keyboard. The user gives a command, waits for the response, and decides on the next command, based on the result of the previous one. The user can easily experiment, and can see results immediately. Most systems have an interactive text editor for entering programs, and an interactive debugger for assisting in debugging programs.

If users are to be able to access both data and code conveniently, an online file system must be available. A file is a collection of related information defined by its creator. Commonly, files

Notes

represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by its creator and user. The operating system implements the abstract concept of a file by managing mass-storage devices, such as tapes and disks. Files are normally organized into logical clusters, or directories, which make them easier to locate and access. Since multiple users have access to files, it is desirable to control by whom and in what ways files may be accessed. Batch systems are appropriate for executing large jobs that need little interaction. The user can submit jobs and return later for the results; it is not necessary for the user to wait while the job is processed.

Interactive jobs tend to be composed of many short actions, where the results of the next command may be unpredictable. The user submits the command and then waits for the results. Accordingly, the response time should be short-on the order of seconds at most.

An interactive system is used when a short response time is required. Early computers with a single user were interactive systems. That is, the entire system was at the immediate disposal of the programmer/operator. This situation allowed the programmer great flexibility and freedom in program testing and development. But, as we saw, this arrangement resulted in substantial idle time while the CPU waited for some action to be taken by the programmer/operator. Because of the high cost of these early computers, idle CPU time was undesirable. Batch operating systems were developed to avoid this problem. Batch systems improved system utilization for the owners of the computer systems.

Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.

Each user has at least one separate program in memory. A program that is loaded into memory and is executing is commonly referred to as a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output is to a display for the user and input is from a user keyboard.

Since interactive I/O typically runs at people speeds, it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; five characters per second is fairly fast for people, but is incredibly slow for computers. Rather than let the CPU sit idle when this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

A time-shared operating system allows the many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users.

The idea of time-sharing was demonstrated as early as 1960, but since time-shared systems are difficult and expensive to build, they did not become common until the early 1970s. As the popularity of time-sharing has grown, researchers have attempted to merge batch and time-shared systems. Many computer systems that were designed as primarily batch systems have been modified to create a time-sharing subsystem.

Example: IBM's OS/360, a batch system, was modified to support the time-sharing option (TSO).

At the same time, time-sharing systems have often added a batch subsystem. Today, most systems provide both batch processing and time sharing, although their basic design and use tends to be one or the other type.

Notes

Time-sharing operating systems are even more complex than are multi-programmed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory, which requires some form of memory management and protection. So that a reasonable response time can be obtained, jobs may have to be swapped in and out of main memory.

Many universities and businesses have large numbers of workstations tied together with local-area networks. As PCs gain more sophisticated hardware and software, the line dividing the two categories is blurring.

Task Make distinction between multiprogramming system and multitasking system.

Parallel Systems

Most systems to date are single-processor systems; that is, they have only one main CPU. However, there is a trend toward multiprocessor systems. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. These systems are referred to as tightly coupled systems.

There are several reasons for building such systems. One advantage is increased throughput. By increasing the number of processors, we hope to get more work done in a shorter period of time. The speed-up ratio with n processors is not n, however, but rather is less than n. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources lowers the expected gain from additional processors. Similarly, a group of n programmers working closely together does not result in n times the amount of work being accomplished.

Multiprocessors can also save money compared to multiple single systems because the processors can share peripherals, cabinets, and power supplies. If several programs are to operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, rather than to have many computers with local disks and many copies of the data.

Another reason for multiprocessor systems is that they increase reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down. If we have 10 processors and one fails, then each of the remaining nine processors must pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

This ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Systems that are designed for graceful degradation are also called fault-tolerant.

Continued operation in the presence of failures requires a mechanism to allow the failure to be detected, diagnosed, and corrected (if possible). The Tandem system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of two identical processors, each with its own local memory. The processors are connected by a bus. One processor is the primary, and the other is the backup. Two copies are kept of each process; one on the primary machine and the other on the backup. At fixed checkpoints in the execution of the system, the state information of each job (including a copy of the memory image) is copied from the primary machine to the backup. If a failure is detected, the backup copy is activated, and is restarted from the most recent checkpoint.

This solution is obviously an expensive one, since there is considerable hardware duplication. The most common multiple-processor systems now use the symmetric multi-processing model, in which each processor runs an identical copy of the operating system, and the copies

Notes

communicate with one another as needed. Some systems use symmetric multiprocessing, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

Example: An example of the symmetric multiprocessing system is Encore's version of UNIX for the Multimax computer. This computer can be configured to employ dozens of processors, all running a copy of UNIX. The benefit of this model is that many processes can run at once (N processes if there are N CPUs) without causing a deterioration of performance.

However, we must carefully control I/O to ensure that data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. To avoid these inefficiencies, the processors can share certain data structures. A multiprocessor system of this form will allow jobs and resources to be shared dynamically among the various processors, and can lower the variance among the systems. However, such a system must be written carefully.

Asymmetric multiprocessing is more common in extremely large systems, where one of the most time-consuming activities is simply processing I/O. In older batch systems, small processors, located at some distance from the main CPU, were used to run card readers and line printers and to transfer these jobs to and from the main computer. These locations are called remote-job-entry (RJE) sites. In a time-sharing system, a main I/O activity is processing the I/O of characters between the terminals and the computer. If the main CPU must be interrupted for every character for every terminal, it may spend all its time simply processing characters. So that this situation is avoided, most systems have a separate front-end processor that handles all the terminal I/O.

Example: A large IBM system might use an IBM Series/I minicomputer as a front-end.

The front-end acts as a buffer between the terminals and the main CPU, allowing the main CPU to handle lines and blocks of characters, instead of individual characters. Such systems suffer from decreased reliability through increased specialization. It is important to recognize that the difference between symmetric and asymmetric multiprocessing may be the result of either hardware or software.

Special hardware may exist to differentiate the multiple processors, or the software may be written to allow only one master and multiple slaves. For instance, Sun's operating system SunOS Version 4 provides asymmetric multiprocessing, whereas Version 5 (Solaris 2) is symmetric.

As microprocessors become less expensive and more powerful, additional operating system functions are off-loaded to slave-processors, or back-ends.

Example: It is fairly easy to add a microprocessor with its own memory to manage a disk system.

The microprocessor could receive a sequence of requests from the main CPU and implement its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the key strokes into codes to be sent to the CPU. In fact, this use of microprocessors has become so common that it is no longer considered multiprocessing.

Notes

Distributed Systems

A recent trend in computer systems is to distribute computation among several processors. In contrast to the tightly coupled systems, the processors do not share memory or a clock. Instead, each processor has its own memory and clock. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.

These systems are usually referred to as loosely coupled systems, or distributed systems. The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. These processors are referred to by a number of different names, such as sites, nodes, computers, and so on, depending on the context in which they are mentioned.

There are a variety of reasons for building distributed systems, the major ones being:

Resource sharing: If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another.

Example: A user at site A may be using a laser printer available only at site B. Meanwhile, a user at B may access a file that resides at A.

In general, resource sharing in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices (such as a high-speed array processor), and performing other operations.

Computation speedup: If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then a distributed system may allow us to distribute the computation among the various sites - to run that computation concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded, sites. This movement of jobs is called load sharing.

Reliability: If one site fails in a distributed system, the remaining sites can potentially continue operating. If the system is composed of a number of large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, on the other hand, the system is composed of a number of small machines, each of which is responsible for some crucial system function (such as terminal character I/O or the file system), then a single failure may effectively halt the operation of the whole system. In general, if sufficient redundancy exists in the system (in both hardware and data), the system can continue with its operation, even if some of its sites have failed.

Communication: There are many instances in which programs need to exchange data with one another on one system.

Example: Window systems are one example, since they frequently share data or transfer data between displays. When many sites are connected to one another by a communication network, the processes at different sites have the opportunity to exchange information.

Users may initiate file transfers or communicate with one another via electronic mail. A user can send mail to another user at the same site or at a different site.

Real Time Systems

Another form of a special-purpose operating system is the real-time system. A real-time system is used when there are rigid time requirements on the operation of a processor or the flow of data, and thus is often used as a control device in a dedicated application.

Notes

Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and some display systems are real-time systems. Also included are some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems.

A real-time operating system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building. A real-time system is considered to function correctly only if it returns the correct result within any time constraints. Contrast this requirement to a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or to a batch system, where there may be no time constraints at all.

There are two flavors of real-time systems. A hard real-time system guarantees that critical tasks complete on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems. Secondary storage of any sort is usually limited or missing, with data instead being stored in short-term memory, or in read-only memory (ROM). ROM is located on nonvolatile storage devices that retain their contents even in the case of electric outage; most other types of memory are volatile.

Most advanced operating-system features are absent too, since they tend to separate the user further from the hardware, and that separation results in uncertainty about the amount of time an operation will take. For instance, virtual memory is almost never found on real-time systems. Therefore, hard real-time systems conflict with the operation of time-sharing systems, and the two cannot be mixed. Since none of the existing general-purpose operating systems support hard real-time functionality, we do not concern ourselves with this type of system in this text.

A less restrictive type of real-time system is a soft real-time system, where a critical real-time task gets priority over other tasks, and retains that priority until it completes.

Self Assessment

Fill in the blanks:

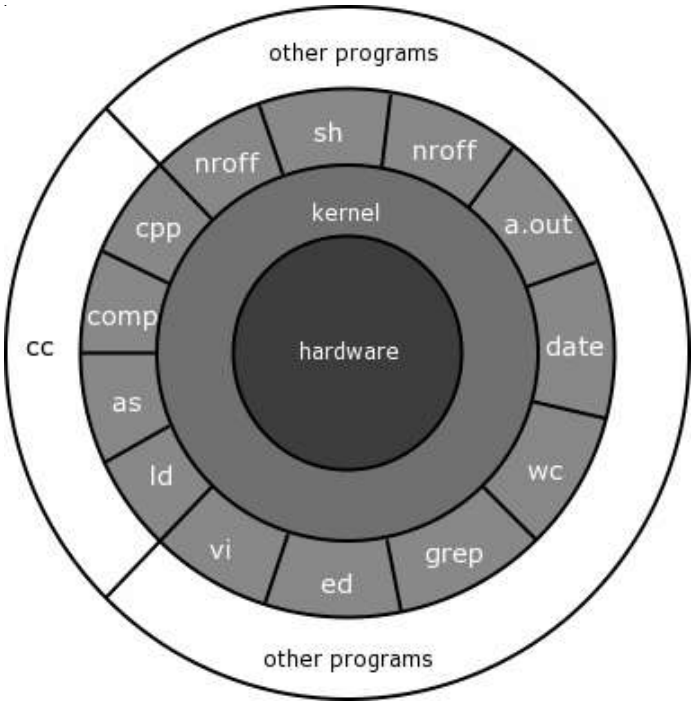
1. The system of the 50's which usually ran one job at a time were known as ..... batch processing systems since programs and data were submitted in groups or batches.
2. The main function of a ..... is to automatically keep executing one job to the next job in the batch.
3. Time sharing, or multitasking, is a logical extension of .....
4. A ..... is used when there are rigid time requirements on the operation of a processor or the flow of data, and thus is often used as a control device in a dedicated application.

2.2 Operating System Architecture

The operating system structure is a container for a collection of structures for interacting with the operating system's file system, directory paths, processes, and I/O subsystem. The types and functions provided by the operating system substructures are meant to present a model for handling these resources that is largely independent of the operating system. There are different types of structure as described above:

Notes

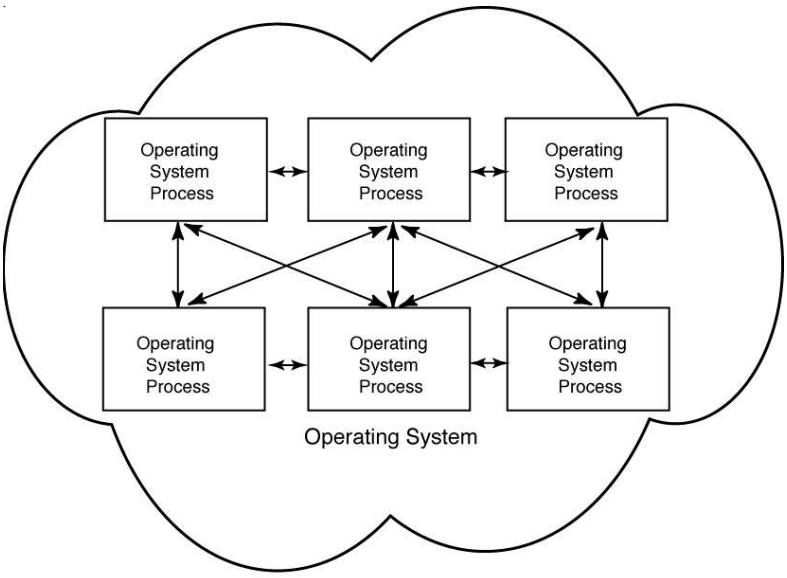
Figure 2.3: Diagram of Unix Operating System



2.2.1 Monolithic Systems

This approach is well known as "The Big Mess". The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

Figure 2.4: Diagram of Monolithic System



Notes

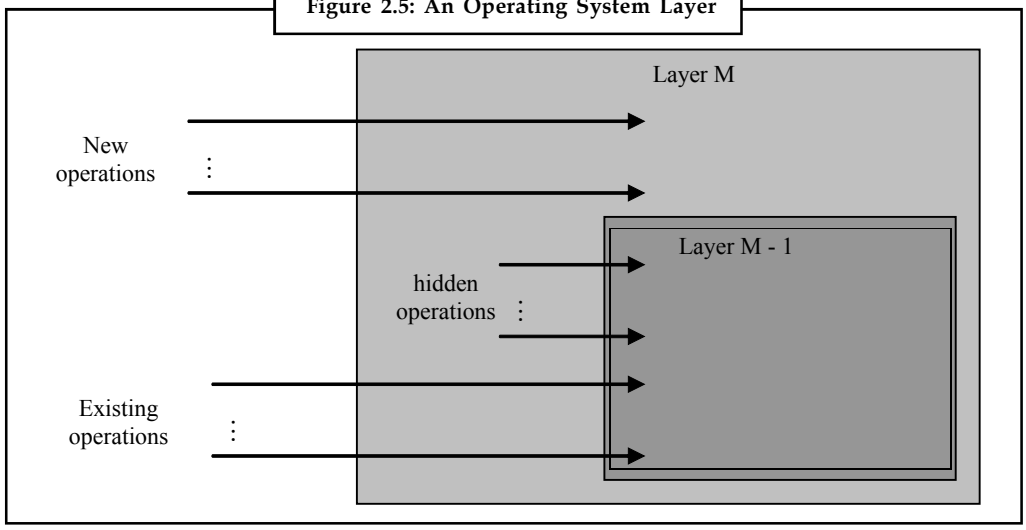
For constructing the actual object program of the operating system when this approach is used, one compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file with the linker. In terms of information hiding, there is essentially none every procedure is visible to every other one i.e. opposed to a structure containing modules or packages, in which much of the information is local to module, and only officially designated entry points can be called from outside the module.

However, even in Monolithic systems, it is possible to have at least a little structure. The services like system calls provide by the operating system are requested by putting the parameters in well-defined places, such as in registers or on the stack, and then executing a special trap instruction known as a kernel call or supervisor call.

2.2.2 Layered Systems

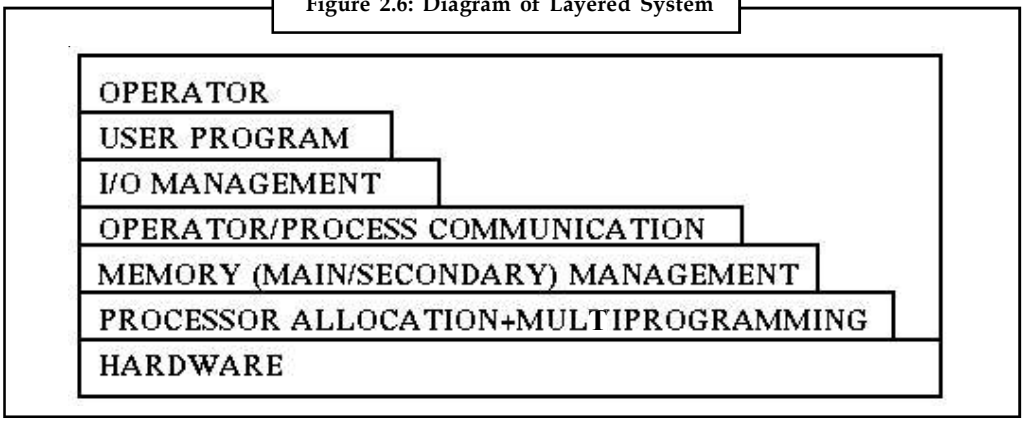
A generalization of the approach as shown below in the figure for organizing the operating system as a hierarchy of layers, each one constructed upon the one below it.

Figure 2.5: An Operating System Layer



The system has 6 layers. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Figure 2.6: Diagram of Layered System



Notes

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512k word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

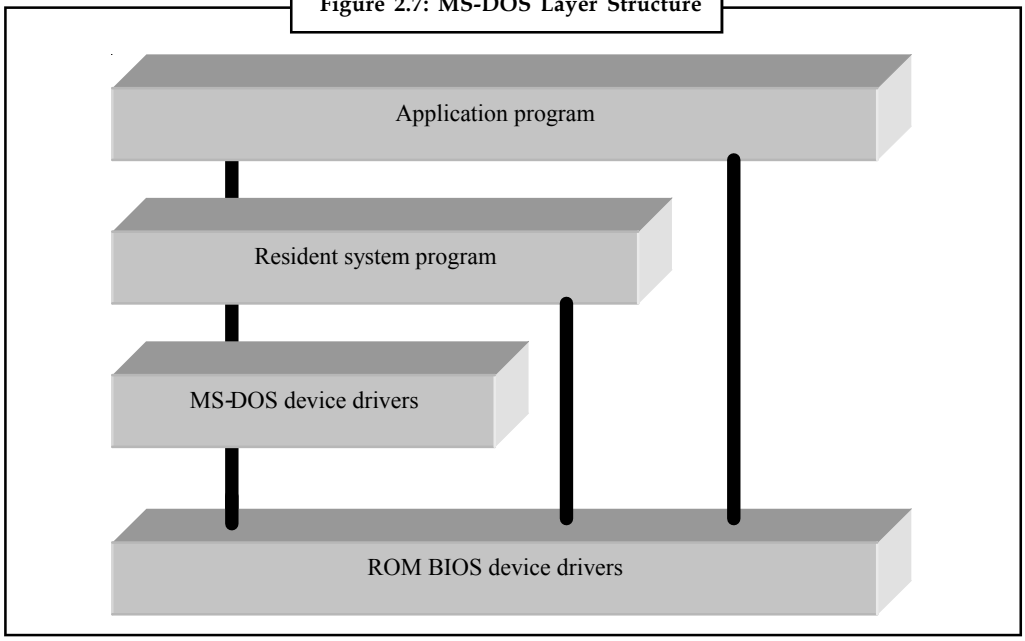
Layer 2 handled communication between each process and the operator console. Above this layer each process effectively had its own operator console.

Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities.

Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management.

The system operator process was located in layer 5.

Figure 2.7: MS-DOS Layer Structure



In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

The main advantage of the layered approach is modularity. The layers are selected such that each uses functions (operations) and services of only lower level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is worked on, and so on. If an error is found during the debugging of a particular layer, we know that the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified when the system is broken down into layers.

Notes

Each layer is implemented using only those operations provided by lower level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The layer approach to design was first used in the operating system at the Technische Hogeschool Eindhoven. The system was defined in six layers. The bottom layer was the hardware. The next layer implemented CPU scheduling. The next layer implemented memory management; the memory-management scheme was virtual memory. Layer 3 contained device driver for the operator's console. Because it and I/O buffering (level 4) were placed above memory management, the device buffers could be placed in virtual memory. The I/O buffering was also above the operator's console, so that I/O error conditions could be output to the operator's console.

This approach can be used in many ways.

Example: The Venus system was also designed using a layered approach.

The lower layers (0 to 4), dealing with CPU scheduling and memory management, were then put into microcode. This decision provided the advantages of additional speed of execution and a clearly defined interface between the microcoded layers and the higher layers.

The major difficulty with the layered approach involves the appropriate definition of the various layers. Because a layer can use only those layers that are at a lower level, careful planning is necessary.

Example: The device driver for the backing store (disk space used by virtual-memory algorithms) must be at a level lower than that of the memory-management routines, because memory management requires the ability to use the backing store.

Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, for a user program to execute an I/O operation, it executes a system call which is trapped to the I/O layer, which calls the memory-management layer, through to the CPU scheduling layer, and finally to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call and the net result is a system call that takes longer than one does on a non-layered system.

These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction. For instance, OS/2 is a descendant of MS-DOS that adds multitasking and dual-mode operation, as well as other new features.

Because of this added complexity and the more powerful hardware for which OS/2 was designed, the system was implemented in a more layered fashion. Contrast the MS-DOS structure to that of the OS/2. It should be clear that, from both the system-design and implementation standpoints, OS/2 has the advantage.

Notes

Example: Direct user access to low-level facilities is not allowed, providing the operating system with more control over the hardware and more knowledge of which resources each user program is using. As a further example, consider the history of Windows NT. The first release had a very layer-oriented organization. However, this version suffered low performance compared to that of Windows 95. Windows NT 4.0 redressed some of these performance issues by moving layers from user space to kernel space and more closely integrating them.

2.2.3 Exokernel

Exokernel is an operating system kernel developed by the MIT Parallel and Distributed Operating Systems group, and also a class of similar operating systems.

The idea behind exokernel is to force as few abstractions as possible on developers, enabling them to make as many decisions as possible about hardware abstractions.

Applications may request specific memory addresses, disk blocks, etc. The kernel only ensures that the requested resource is free, and the application is allowed to access it. This low-level hardware access allows the programmer to implement custom abstractions, and omit unnecessary ones, most commonly to improve a program's performance. It also allows programmers to choose what level of abstraction they want, high, or low.

Exokernels can be seen as an application of the end-to-end principle to operating systems, in that they do not force an application program to layer its abstractions on top of other abstractions that were designed with different requirements in mind.

Example: In the MIT Exokernel project, the Cheetah web server stores preformatted Internet Protocol packets on the disk, the kernel provides safe access to the disk by preventing unauthorized reading and writing, but how the disk is abstracted is up to the application or the libraries the application uses.

Operating systems define the interface between applications and physical resources. Unfortunately, this interface can significantly limit the performance and implementation freedom of applications. Traditionally, operating systems hide information about machine resources behind high-level abstractions such as processes, files, address spaces and interprocess communication. These abstractions define a virtual machine on which applications execute; their implementation cannot be replaced or modified by untrusted applications.

Hardcoding the implementations of these abstractions is inappropriate for three main reasons:

- it denies applications the advantages of domain-specific optimizations,
- it discourages changes to the implementations of existing abstractions, and
- it restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all).

These problems can be solved through application level resource management in which traditional operating system abstractions, such as Virtual Memory (VM) and interprocess communication (IPC), are implemented entirely at application level by untrusted software. In this architecture, a minimal kernel-called an exokernel-securely multiplexes available hardware resources. Library operating systems, working above the exokernel interface, implement higher-level abstractions.

Application writers select libraries or implement their own. New implementations of library operating systems are incorporated by simply relinking application executables. Applications can benefit greatly from having more control over how machine resources are used to implement

Notes

higher-level abstractions. The high cost of general-purpose virtual memory primitives reduces the performance of persistent stores, garbage collectors, and distributed shared memory systems. Application-level control over file caching can reduce application-running time considerably. Application-specific virtual memory policies can increase application performance. The inappropriate file-system implementation decisions can have a dramatic impact on the performance of databases. The exceptions can be made an order of magnitude faster by deferring signal handling to applications.

To provide applications control over machine resources, an exokernel defines a low-level interface. The exokernel architecture is founded on and motivated by a single, simple, and old observation that the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions.

To provide an interface that is as low-level as possible (ideally, just the hardware interface), an exokernel designer has a single overriding goal of separating protection from management. For instance, an exokernel should protect framebuffers without understanding windowing systems and disks without understanding file systems.

One approach is to give each application its own virtual machine. Virtual machines can have severe performance penalties. Therefore, an exokernel uses a different approach - it exports hardware resources rather than emulating them, which allows an efficient and simple implementation.

An exokernel employs three techniques to export resources securely:

- by using secure bindings, applications can securely bind to machine resources and handle events,
- by using visible resource revocation, applications participate in a resource revocation protocol,
- by using an abort protocol, an exokernel can break secure bindings of uncooperative applications by force.

The advantages of exokernel systems among others are:

- exokernels can be made efficient due to the limited number of simple primitives they must provide,
- low-level secure multiplexing of hardware resources can be provided with low overhead,
- traditional abstractions, such as VM and IPC, can be implemented efficiently at application level, where they can be easily extended, specialized, or replaced,
- applications can create special-purpose implementations of abstractions, tailored to their functionality and performance needs.

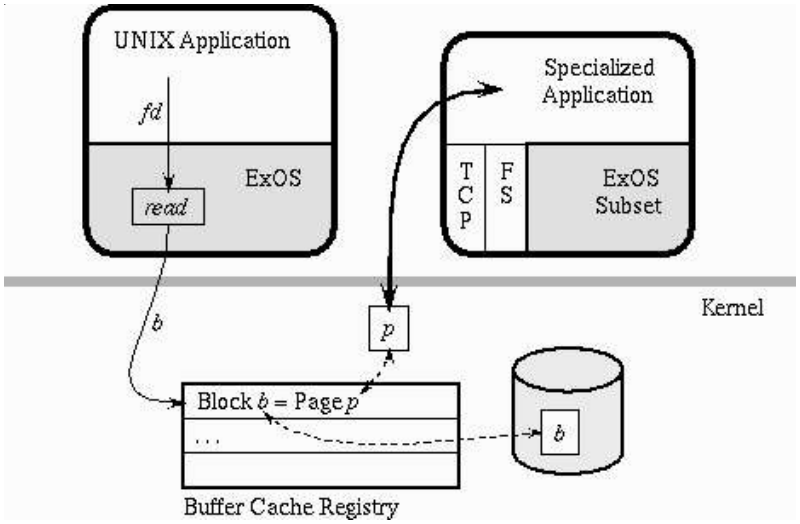
Finally, many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavyweight servers that cannot be bypassed or tailored to application-specific needs. These heavyweight servers can be viewed as fixed kernel subsystems that run in user-space.

2.2.4 Client-server Model

A trend in modern operating systems is to take this idea of moving code up into higher layers even further, and remove as much as possible from the operating system, leaving a minimal kernel. The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (presently known as the client process) sends the request to a server process, which then does the work and sends back the answer.

Notes

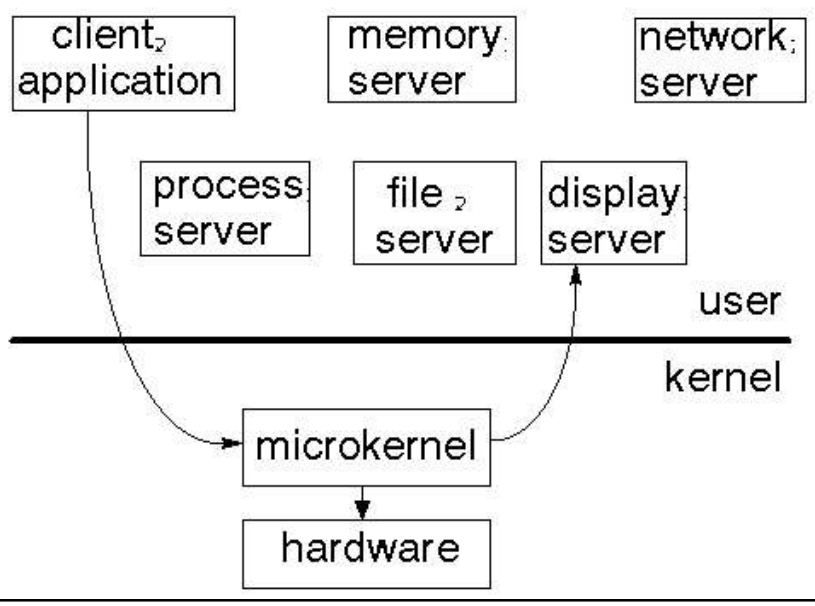
Figure 2.8: Diagram of Exokernel



In client-Server Model, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one fact of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable; furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed system. If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

Figure 2.9: Diagram of Client-Server Model



2.2.5 Virtual Machines

Notes

A virtual machine is a type of computer application used to create a virtual environment, which is referred to as virtualization. Virtualization allows the user to see the infrastructure of a network through a process of aggregation. Virtualization may also be used to run multiple operating systems at the same time. Through the help of a virtual machine, the user can operate software located on the computer platform.

There are different types of virtual machines. Most commonly, the term is used to refer to hardware virtual machine software, also known as a hypervisor or virtual machine monitor. This type of virtual machine software makes it possible to perform multiple identical executions on one computer. In turn, each of these executions runs an operating system. This allows multiple applications to be run on different operating systems, even those they were not originally intended for.

Virtual machine can also refer to application virtual machine software. With this software, the application is isolated from the computer being used. This software is intended to be used on a number of computer platforms. This makes it unnecessary to create separate versions of the same software for different operating systems and computers.

Example: Java Virtual Machine is a very well known example of an application virtual machine.

A virtual machine can also be a virtual environment, which is also known as a virtual private server. A virtual environment is used for running programs at the user level. Therefore, it is used solely for applications and not for drivers or operating system kernels.

A virtual machine may also be a group of computers that work together to create a more powerful machine. In this type of virtual machine, the software makes it possible for one environment to be formed throughout several computers. This makes it appear to the end user as if he or she is using a single computer, when there are actually numerous computers at work.

The heart of the system, known as the virtual machine monitor, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are exact copies of the bare hardware, including kernel/user mod, I/O, interrupts, and everything else the real machine has.

Each virtual machine is identical to the true hardware; therefore, each one can run any operating system that will run directly on the hardware. Different virtual machines can, and usually do, run different operating systems. Some run one of the descendants of OF/360 for batch processing, while other ones run a single-user, interactive system called CMS (conversational Monitor System) fro timesharing users.

Conceptually, a computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system calls for use by outer layers. The systems programs above the kernel are therefore able to use either system calls or hardware instructions, and in some ways these programs do not differentiate between these two. Thus, although they are accessed differently, they both provide functionality that the program can use to create even more advanced functions. System programs, in turn, treat the hardware and the system calls as though they both are at the same level.

Some systems carry this scheme even a step further by allowing the system programs to be called easily by the application programs. As before, although the system programs are at a level higher than that of the other routines, the application programs may view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a virtual machine.

Notes

Example: The VM operating system for IBM systems is the best example of the virtual-machine concept, because IBM pioneered the work in this area.

By using CPU scheduling and virtual-memory techniques, an operating system can create the illusion of multiple processes, each executing on its own processor with its own (virtual) memory. Of course, normally, the process has additional features, such as system calls and a file system, which are not provided by the bare hardware. The virtual-machine approach, on the other hand, does not provide any additional function, but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer.

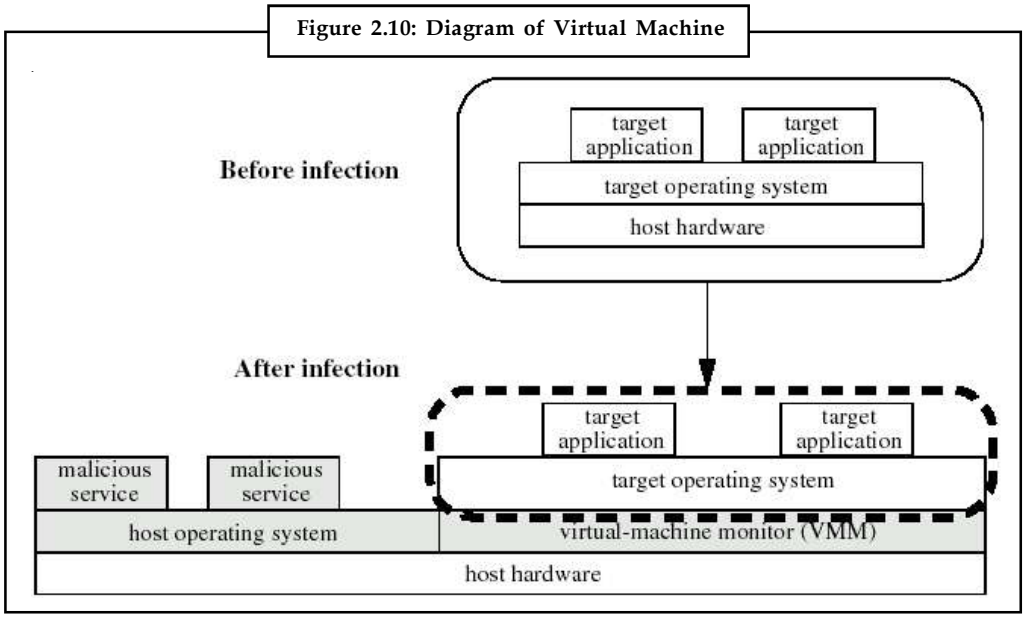
The resources of the physical computer are shared to create the virtual machines. CPU scheduling can be used to share the CPU and to create the appearance that users have their own processor. Spooling and a file system can provide virtual card readers and virtual line printers. A normal user timesharing terminal provides the function of the virtual machine operator's console.

A major difficulty with the virtual-machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines. Clearly, it cannot allocate a disk drive to each virtual machine. Remember that the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks, which are identical in all respects except size; these are termed minidisks in IBM's VM operating system. The system implements each minidisk by allocating as many tracks as the minidisk needs on the physical disks.

Caution The sum of the sizes of all minidisks must be less than the actual amount of physical disk space available.

Users thus are given their own virtual machine. They can then run any of the operating systems or software packages that are available on the underlying machine. For the IBM VM system, a user normally runs CMS, a single-user interactive operating system. The virtual-machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but does not need to consider any user-support software. This arrangement may provide a useful partitioning of the problem of designing a multiuser interactive system into two smaller pieces.

Figure 2.10: Diagram of Virtual Machine



Notes

Task Illustrate the function of virtual machines.

Self Assessment

Fill in the blanks:

5. The services like system calls provide by the operating system are requested by putting the parameters in well-defined places, such as in registers or on the stack, and then executing a special trap instruction known as a .....
6. Layer 2 handled ..... between each process and the operator console.
7. In ....., all the kernel does is handle the communication between clients and servers.
8. A ..... is a type of computer application used to create a virtual environment, which is referred to as virtualization.

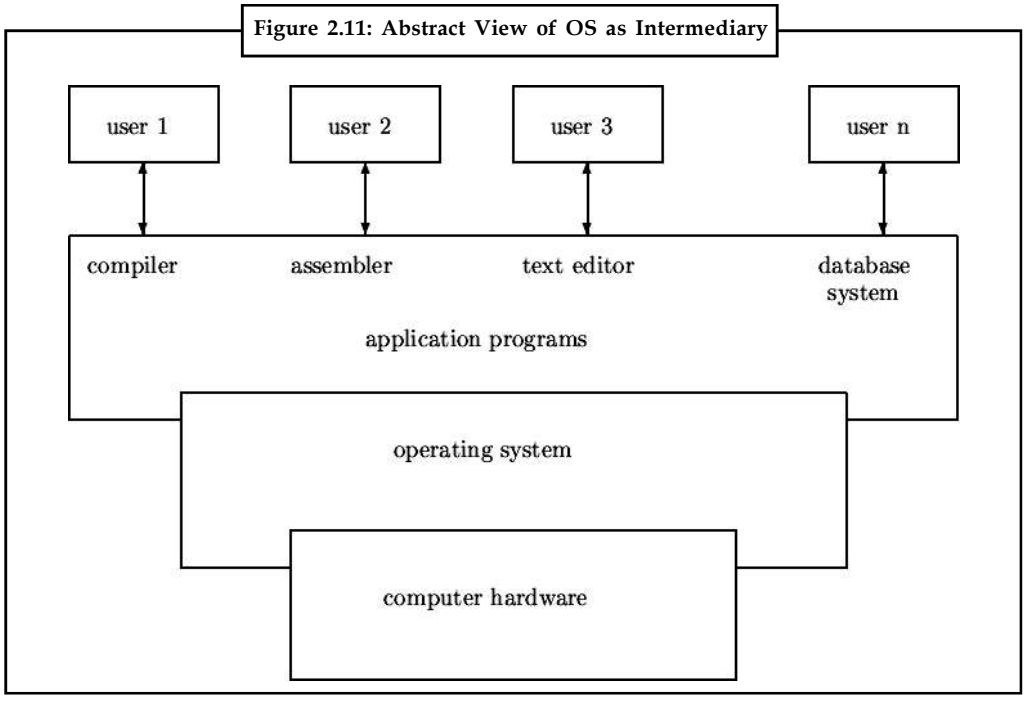
2.3 User Function of Operating System

An operating system manages all aspects of a complex system.

Imagine what will happen if three programs running on a computer all try to print their output simultaneously on the same printer. The first few lines might be from program one and the next few lines might be from program two and so on with the result resulting in chaos. The operating system can bring order to potential chaos by buffering all the output destined for the printer on the disk.

The primary task of operating system is to keep track of who is using which resource, to grant resource request, to account for usage, and to meditate conflicting request from different programs and users.

Figure 2.11: Abstract View of OS as Intermediary



Notes

When a computer has multiple users, the needs for managing and protecting the memory, input/output devices and other resources are vary necessary. These needs arises because it is usually necessary to share expensive resource such as tapes drives and phototypesetters.

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

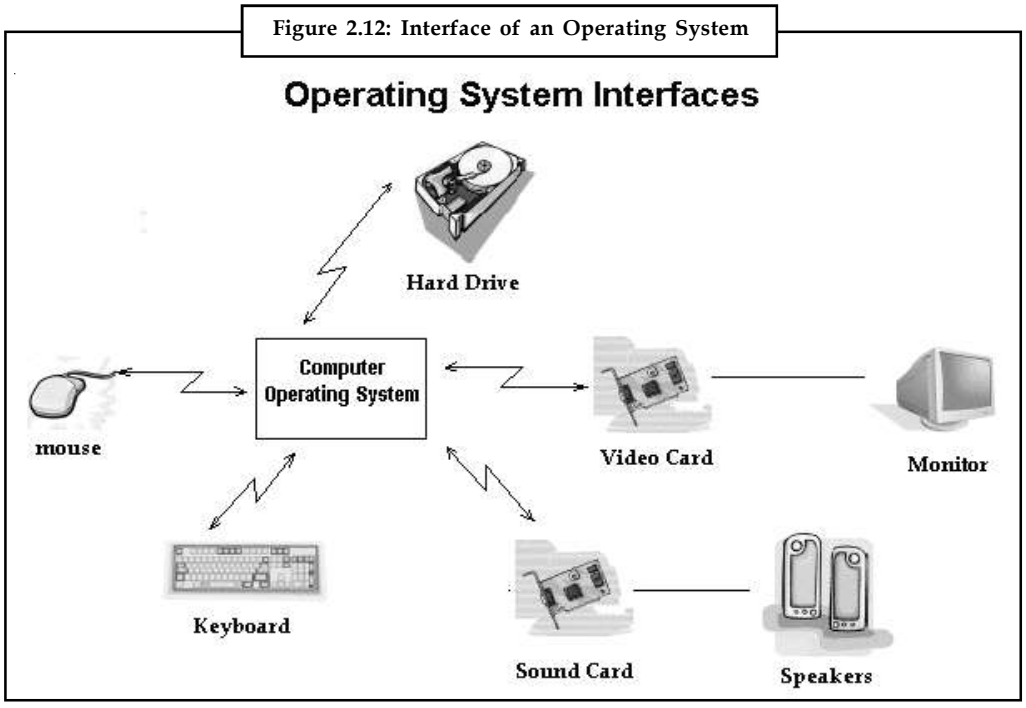
For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop -- it makes sure that different program and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

An operating system performs the following services for applications:

- In a multitasking system where multiple programs are run at the same time, the operating system determines which applications should run in what order and how much time should be allowed for each application before another application gets its turn.
- It sends messages to each application or interactive user (or to a system operator) about the status of operation and any errors that may have occurred.
- It handles input and output to and from attached hardware devices, such as hard disks, printers, and dial-up ports.
- It arranges the sharing of internal memory among multiple applications.
- It can offload the management of what are called batch jobs (for example, printing) so that the initiating application is freed from this work.
- In computers that can provide parallel processing, an operating system manages to divide the instructions of a program so that it runs on more than one processor at a time.

2.3.1 Other Operating System Functions

Figure 2.12: Interface of an Operating System



Notes

The operating system provides for several other functions including:

- System tools (programs) used to monitor computer performance, debug problems, or maintain parts of the system.
- A set of libraries or functions which programs may use to perform specific tasks especially relating to interfacing with computer system components.

The operating system makes these interfacing functions along with its other functions operate smoothly and these functions are mostly transparent to the user.

2.3.2 Operating System Concerns

As mentioned previously, an operating system is a computer program. Operating systems are written by human programmers who make mistakes. Therefore, there can be errors in the code even though there may be some testing before the product is released. Some companies have better software quality control and testing than others so you may notice varying levels of quality from operating system to operating system. Errors in operating systems cause the following main types of problems:

System Crashes and Instabilities: These can happen due to a software bug typically in the operating system, although computer programs being run on the operating system can make the system more unstable or may even crash the system by themselves. This varies depending on the type of operating system. A system crash is the act of a system freezing and becoming unresponsive which would cause the user to need to reboot.

Security Flaws: Some software errors leave a door open for the system to be broken into by unauthorized intruders. As these flaws are discovered, unauthorized intruders may try to use these to gain illegal access to your system. Patching these flaws often will help keep your computer system secure.

Sometimes errors in the operating system will cause the computer not to work correctly with some peripheral devices such as printers.

Self Assessment

Fill in the blanks:

9. In a ..... system where multiple programs are run at the same time, the operating system determines which applications should run in what order and how much time should be allowed for each application before another application gets its turn.
10. A ..... is the act of a system freezing and becoming unresponsive which would cause the user to need to reboot.

2.4 Batch Control Language

The vision of much Batch Control standardization has been the expansion of a common Batch language. Practically, Batch languages are superposed on collections of ancient control data bases and illustrative demonstrations. A common Control Language could incorporate all aspects of control. Such a language could capture the imminents of continuous, logical, and batch control together, and still be calculated against the different standardized Batch Models or diagrams.

It is usual for batch engineers to sight their control as a more common and inclusive practice than nonstop control. Here we will hold that view to its logical ending: It will present a single information model and language framework to incorporate the full range of both control practices.

Notes

The framework expands previous conversation of Idioms, Super Variables, State Based Logic, Theme Statements, etc.

The consequential Control Language varies inherently from conventional computer languages, integrating unique accessibility concepts, including:

- Expression of real-time sequential, parallel, and continuous computation, not just calculate time sequencing.
- Structured, about the process and control organization.
- Leveled to reflect levels of human activity.
- Producing an integrated data base, for plant operation, monitoring, control, and historical recording of operating parameters and production states.
- Formatted, in semigraphic form, for readability.
- Complex hundred parameter Block control data structures substituted by one line statements; the Idioms and Super Variables.

The previous references address component language concepts. The discussion will swap between batch and continuous concerns, and between batch oriented control applied to continuous examples and vice versa.

This aim notwithstanding, batch and continuous perspectives vary fundamentally, requiring a rethought synthesis, an integrated viewpoint permitting each practice to advantage from the imminent of the other. On a theoretical plane the two viewpoints can be summed up as below:

- Continuous control depends on the continuous assessment of all pertinent aspects of the current process state, to recalculate the control actions which regulate that state for best process performance.
- Batch control depends on an unspecified desired production course in process variable space, on the monitoring of events which recognize the progression to later specifies or stages of that trajectory, and on the exercise of controls and control actions to preserve or move the process along that trajectory.

In some respects, the Batch viewpoint is simpler, defining the trajectory clearly, not relying on its implied derivation from interacting state calculations. The application of a fully incessant computation to the classic Batch process would be unfeasibly complex. Alternatively, the lodging of elaborate constraint or recovery controls in a batch program can make a maze of side paths in the logic. The similar constraint controls can expand automatically as a result of a suitable continuous design.

Frequently application goals can be carried from either viewpoint, with interesting comparative benefits.

One reference comprises an example in which the utilization of conventional continuous feed forward methods allows a continuous control system to take the position of three batch phases.

2.4.1 The Role of Graphics

The Language design takes a fundamental position comparative to the typical graphic panaceas. Graphics should improve easiness of use in more basic manners than familiarity. Actually, most control graphic diagrams were intended with a different goal than the specification of an integrated control application:

- Successful graphic figures abstract and abridge specialized areas of computation, and simple designs, not common computations. On the negative side, the effect in process

Notes

control is applications configured as a compilation of weakly compatible expert diagrams (intended from mismatched control points of view?).

- Most standard diagrams were invented to reflect the associations and specifications to execute controls in a specific hardware, instead of stating the proposed control aims in the best hardware independent way.
- Frequently that hardware is so far out of date that many control engineers have only seen its dependent simulation.

We require control demonstrations that reflect the contemporary computerized control system.

- As a consequence, it is fair to inquire if these diagrams are actually the best ease of use demonstration for the application or the modern computer environment.

On the contrary, both the text and graphics elements in the Control Language have been specifically intended to most evidently symbolize the intended application behavior. Particularly the graphics have the following roles:

- Text Key Words are substituted by icons (e.g., Activity Brackets), intended to reflect the underlying application structure visually.
- Icon diagrams are produced automatically (e.g., Idiom diagrams and Sequential Function Charts) to exemplify programmed text.
- Just by their inclusion, diagrams and icons can offer visual structure and landmarks to a program, cueing later searches.
- They are intended to strengthen the learning of a user.

2.4.2 Potential for Objects in Control

Objects symbolize a preferential sight of programming in which (ideally) all application data is cataloged into data structures (the Objects), each type (Class) being defined with its access and processing functions (the Methods). The concept permits programs (enclosing the programs which define the Methods) to attach to the data in any Object via Messages (the calls to the Method functions) to it.

The Object Class concept permits the definitions of alike application structures (the Classes) to be dependent on each other via a concept of Inheritance and Overriding. The strength of this capability is that it permits similar Objects to be considered similarly through Messages by means of the same Method names, while the system makes the low level distinctions, choosing the right Method variant for each Object. Every such Class may then be the foundation of any number of functioning Instances, each accessed via the connected techniques.

Self Assessment

Fill in the blanks:

11. Practically, ... languages are superposed on collections of ancient control data bases and illustrative demonstrations.
12. The ..... Class concept permits the definitions of alike application structures (the Classes) to be dependent on each other via a concept of Inheritance and Overriding.

2.5 OS User Viewpoint – Facilities

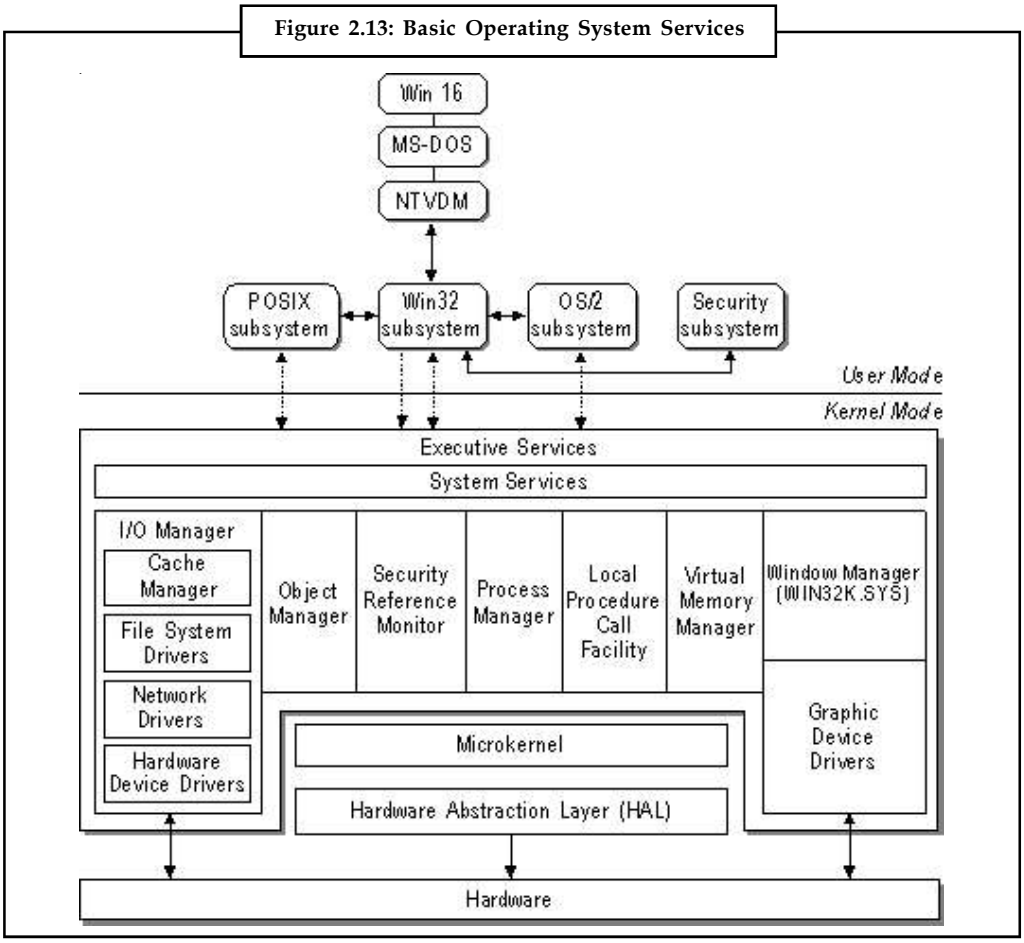
Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending

Notes

output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop - it makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

Figure 2.13: Basic Operating System Services



2.5.1 Program Execution

The purpose of a computer system is to allow the user to execute programs. So the operating system provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems.

Running a program involves the allocating and deallocating memory, CPU scheduling in case of multiprocess. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

In computing, a process is an instance of a computer program that is being sequentially executed. A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program.

Notes

Example: Opening up two windows of the same program typically means two processes are being executed.

A single computer processor executes only one instruction at a time. To allow users to run several programs at once, single-processor computer systems can use time-sharing processes switch between being executed and waiting to be executed. In most cases this is done at a very fast rate, giving an illusion that several processes are executing at once. Using multiple processors achieves actual simultaneous execution of multiple instructions from different processes, but time-sharing is still typically used to allow more processes to run at once.

For security reasons most modern operating systems prevent direct inter-process communication, providing mediated and limited functionality. However, a process may split itself into multiple threads that execute in parallel, running different instructions on much of the same resources and data. This is useful when, for instance, it is necessary to make it seem that multiple things within the same process are happening at once (such as a spell check being performed in a word processor while the user is typing), or if part of the process needs to wait for something else to happen (such as a web browser waiting for a web page to be retrieved).

A multitasking operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one time on a single-core CPU (unless using multi-threading or other similar technology).

It is usual to associate a single process with a main program, and 'daughter' ('child') processes with any spin-off, parallel processes, which behave like asynchronous subroutines. A process is said to own resources, of which an image of its program (in memory) is one such resource.

Notes However, that in multiprocessing systems, many processes may run off of, or share, the same reentrant program at the same location in memory- but each process is said to own its own image of the program.

Processes are often called tasks in embedded operating systems. The sense of 'process' (or task) is 'something that takes up time', as opposed to 'memory', which is 'something that takes up space'. (Historically, the terms 'task' and 'process' were used interchangeably, but the term 'task' seems to be dropping from the computer lexicon.)

The above description applies to both processes managed by an operating system, and processes as defined by process calculi.

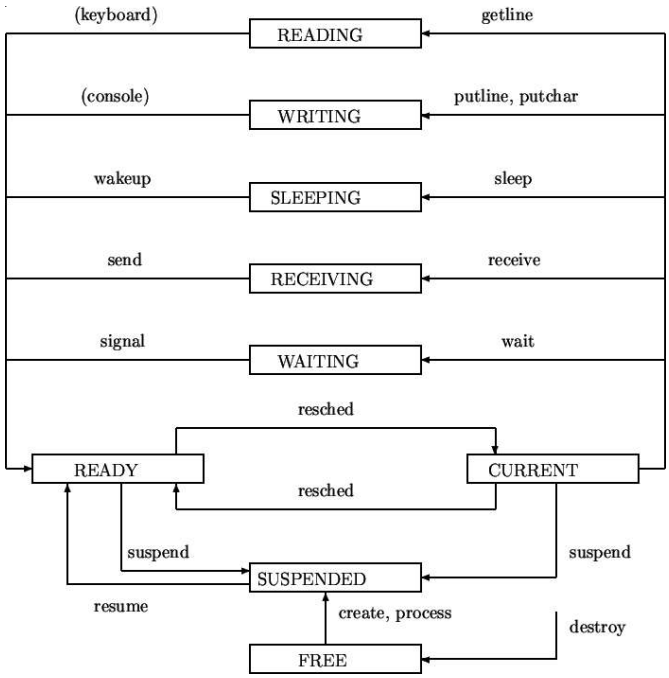
If a process requests something for which it must wait, it will be blocked. When the process is in the Blocked State, it is eligible for swapping to disk, but this is transparent in a virtual memory system, where blocks of memory values may be really on disk and not in main memory at any time.

Caution Even unused portions of active processes/tasks (executing programs) are eligible for swapping to disk.

All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.

Notes

Figure 2.14: Different Stages of Program Execution

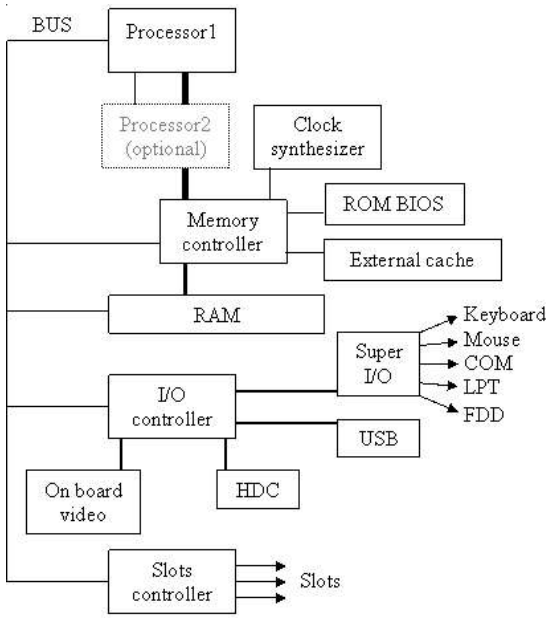


2.5.2 I/O Operations

Each program requires an input and produces output. This involves the use of I/O. The operating systems hides from the user the details of the underlying hardware for the I/O. What the user sees is that the I/O has been performed without any details.

For efficiency and protection, the users cannot control I/O, so this service cannot be provided by user-level programs.

Figure 2.15: I/O Operation Handling



Notes

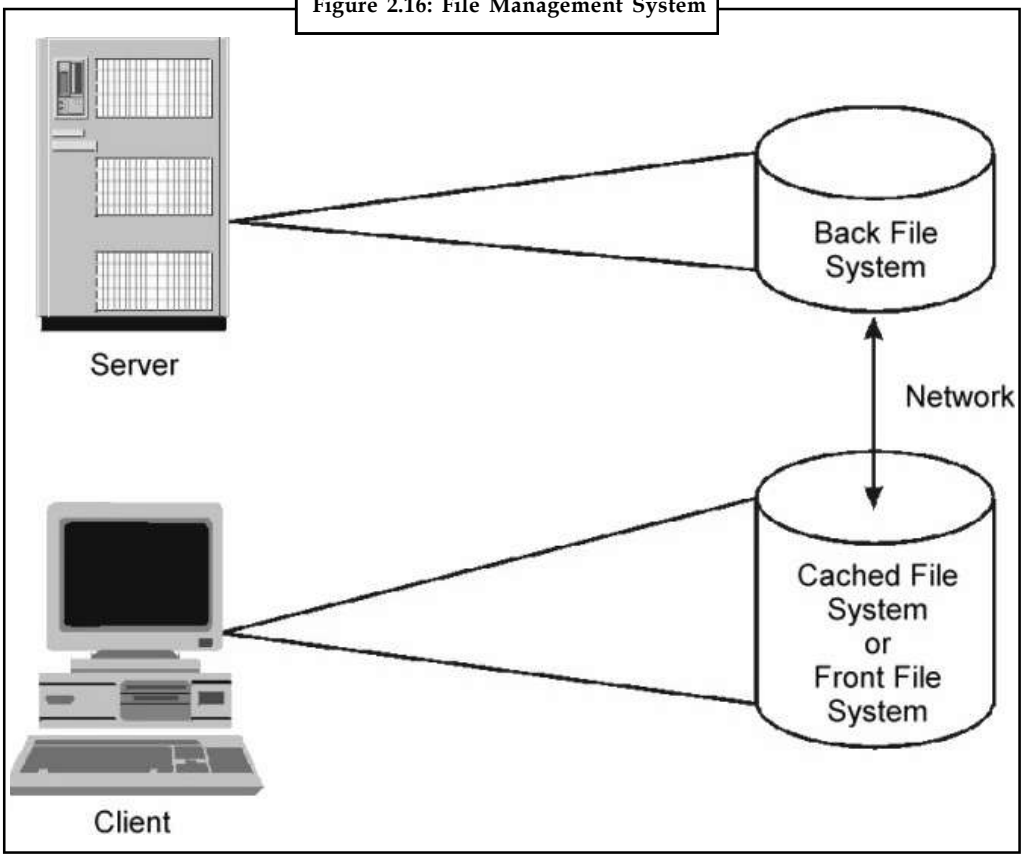
2.5.3 File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating system provides this service.

The user need not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his/her task accomplished. Thus, operating system makes it easier for user programs to accomplish their task.

This service also involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs. Therefore, it is left to the operating systems to manage it rather than giving individual users the control of it.

Figure 2.16: File Management System

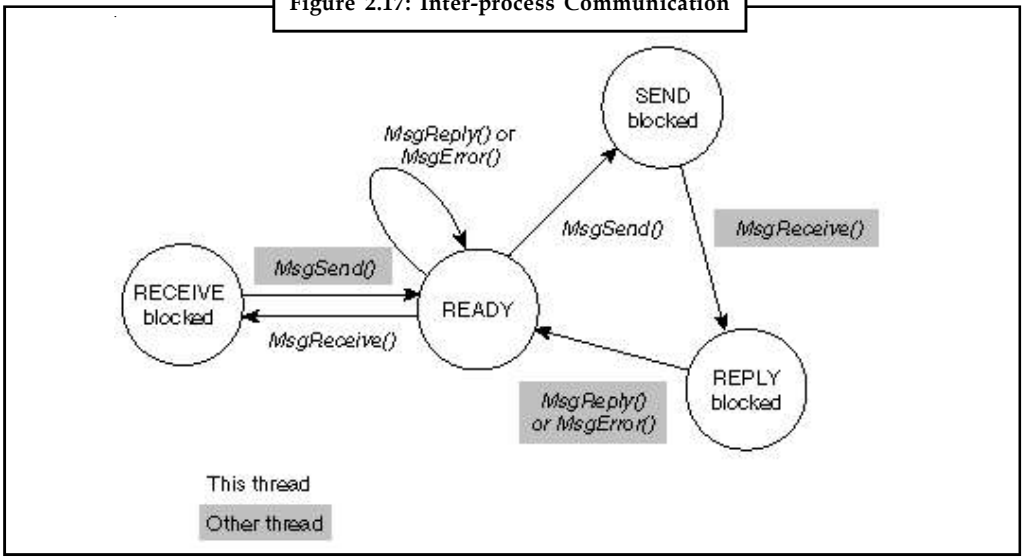


2.5.4 Communications

The processes need to communicate with each other to exchange information. It can be between the processes those are running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes. In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user program may be customized to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

Notes

Figure 2.17: Inter-process Communication

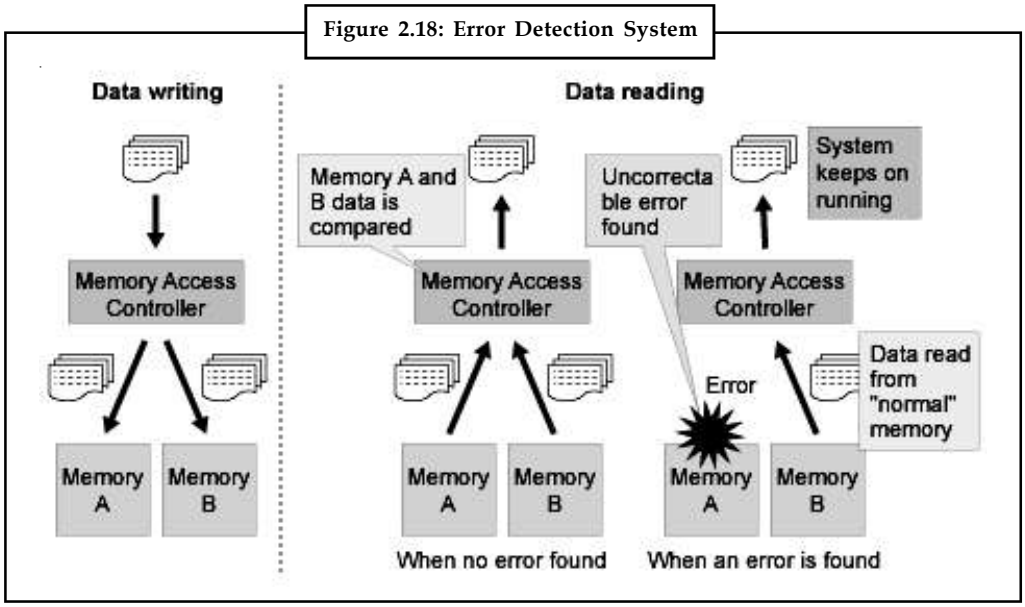


2.5.5 Error Detection

An error is also one part of the system that may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

This service is not allow to be handled by user programs because it involves monitoring and in cases altering area of memory or deallocation of memory for a faulty process. Or may be relinquishing the CPU of a process that goes into an infinite loop. These tasks are too critical to be handed over to the user programs. .

Figure 2.18: Error Detection System



Self Assessment

Notes

Fill in the blanks:

13. In computing, a ..... is an instance of a computer program that is being sequentially executed.
14. The speed of I/O that depends on ..... storage management is critical to the speed of many programs.
15. An ..... is one part of the system that may cause malfunctioning of the complete system.

2.6 Summary

- The main difference between the operating system and system software is that the operating system runs in kernel or supervisor mode where as the system software and applications run in user mode.
- Operating systems have evolved via number of distinct stages or generations which matches approximately to the decades.
- The simplest of all the computer systems is a single use-single processor system which has a single processor, runs a single program and interacts with a single user at a time.
- The main function of a batch processing system is to automatically keep executing one job to the next job in the batch.
- A real-time system is used when there are rigid time requirements on the operation of a processor or the flow of data, and thus is often used as a control device in a dedicated application.
- A virtual machine is a type of computer application used to create a virtual environment, which is referred to as virtualization.
- The operating system structure is a container for a collection of structures for interacting with the operating system's file system, directory paths, processes, and I/O subsystem.
- The primary task of operating system is to keep track of who is using which resource, to grant resource request, to account for usage, and to meditate conflicting request from different programs and users.
- Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.
- The operating system provides an environment where the user can conveniently run programs.

2.7 Keywords

Batch Processing: A mode of data processing in which a number of tasks are lines up and the entire task set is submitted to the computer in one go.

Client-Server Model: In client-Server Model, all the kernels handle the communication between clients and servers by splitting the operating system up into parts, each of which only handles one fact of the system, such as file service, process service, terminal service, or memory service.

Communications: These programs provide the mechanism for creating virtual connections among processes, users, and different computer systems.

Notes

Distributed System: A computer systems in which computation tasks are distributed among several processors.

Error Detection: This is a process where the operating system constantly monitors the system for detecting the malfunctioning of it.

Exokernel: It is an operating system kernel developed by the MIT for Parallel and Distributed Operating Systems group, and also a class of similar operating systems.

I/O Operations: It refers to the communication between an information processing system and the outside world - possibly a human, or another information processing system.

Layered Systems: This is an operating system structure where the operating system is organized as a hierarchy of layers, each one constructed upon the one below it. Here the system had 6 layers.

Monolithic Systems: This is an operating system structure where it is written as a collection of procedures, each of which can call any of the other ones whenever it needs to.

Multiprogramming: A style of programming in which multiple programs can share the resources appearing to be executing simultaneously.

Operating System: An operating system is itself a computer program which must be executed.

Parallel System: A system that is capable of executing a number of programs parallely.

Process Communication: A processes need to communicate with other process or with the user to exchange the information, this is known as Process Communication.

Program Execution: Program execution is a method in which user given commands call up a processes and pass data to them.

Real Time System: A special-purpose operating system in which there are rigid time requirements on the operation of a processor or the flow of data.

Time Sharing: A mode of programming in which the CPU is shared between multiple programs each getting a share of CPU time in turn.

Virtual Machines: The virtual machine monitor runs on the bare hardware at the heart of the system and does the multiprogramming, providing not one, but several virtual machines to the next layer up.

2.8 Review Questions

1. Elucidate process management briefly.
2. Illustrate the process of file management and I/O system management.
3. Do you think a single user system requires process communication? Support your answer with logic.
4. Suppose a user program faced an error during memory access. What will it do then? Will it be informed to the OS? Explain.
5. How information (parameters) is passed between a running program and the operating system?
6. What is Exokernel? How it is structured?
7. Is client-Server Model the only model for all the operating systems which support network services?

Notes

8. Describe the function of the different layers in layered system structure.
9. More number of processors in a system with single user operating system will enhance the overall performance of the system. Comment.
10. Compare and contrast between layered and client-server architectures of an operating system.
11. What is batch system? What are the shortcomings of early batch systems? Explain it.
12. Briefly explain the evolution of the operating system.
13. Write the differences between the time sharing system and distributed system.
14. Depict Real time operating system with example.

Answers: Self Assessment

1.	single-stream	2.	batch processing system
3.	multiprogramming	4.	real-time system
5.	kernel call or supervisor call	6.	communication
7.	client-Server Model	8.	virtual machine
9.	multitasking	10.	system crash
11.	Batch	12.	Object
13.	process	14.	secondary
15.	error		

2.9 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A.Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online link

citeseerx.ist.psu.edu/viewdoc

Notes

Unit 3: Machine Structure and Assembler Basic Functions

CONTENTS

Objectives

Introduction

3.1 Machine Structure

3.1.1 Software Architecture

3.2 New Approach to New Machine

3.3 Machine Level Language (1940-1950)

3.4 Assembly Language (1950-1958)

3.5 Summary

3.6 Keywords

3.7 Review Questions

3.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain Machine structure
- Demonstrate Approach to new machine
- Discuss Machine language
- Scan Assembly language

Introduction

The Assembler translates symbolic source language instructions into an object program. The object program may be either relocatable or absolute. The Loader of the Basic Control System loads and links relocatable programs. The Basic Binary Loader loads programs in absolute form.

The Assembler accepts as input a paper tape containing a control statement and a source language program. The output produced by the Assembler may include a punched paper tape containing the object program and/or an object program listing.

Assembler is a program that converts computer instructions into bits that can be processed by processor in computer to perform certain operation. We have discussed machine structure in unit 1 also.

3.1 Machine Structure

The field of computer science has come across problems linked with complexity since its formation. Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the idea of separation of concerns. Although the term "software architecture" is relatively new to the industry, the basic principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s. Early attempts to capture and explain software architecture of a system were vague and

Notes

disorganized, often characterized by a set of box-and-line diagrams. During the 1990s there was a concentrated effort to define and codify fundamental aspects of the discipline. Initial sets of design patterns, styles, best practices, description languages, and formal logic were developed during that time.

The software architecture discipline is centered on the idea of reducing difficulty through abstraction and separation of concerns. To date there is still no agreement on the precise definition of the term "software architecture".

Notes However, this does not mean that individuals do not have their own definition of what software architecture is. This leads to problems because many people are using the same language to describe differing ideas.

As a growing regulation with no clear rules on the right way to build a system, designing software architecture is still a mix of art and science. The "art" aspect of software architecture is because a profitable software system supports some aspect of a business or a mission. How a system supports key business drivers is described via scenarios as non-functional requirements of a system, also known as quality attributes, decide how a system will behave. This could be thought of as a parallel to a mission statement and value system in business strategy. Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and such other facilities will vary with each implementation. To bring a software architecture user's perspective into the software architecture, it can be said that software architecture gives the direction to take steps and do the tasks involved in each such user's specialty area and interest, e.g., the stakeholders of software systems, the software developer, the software system operational support group, the software maintenance specialists, the deployer, the tester and also the business end user. In this sense software architecture is really the amalgamation of the multiple perspectives a system always embodies.

Caution The fact that those several different perspectives can be put together into a software architecture stands as the vindication of the need and justification of creation of software architecture before the software development in a project attains maturity.

The beginning of software architecture as a concept was first recognized in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. These scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Research institutions have played a well-known role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled Software Architecture: Perspectives on an Emerging Discipline in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles and so on. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

Notes

3.1.1 Software Architecture

Architecture Description Languages

Architecture Description Languages (ADLs) are employed to explain a Software Architecture. Several different ADLs have been developed by dissimilar organizations, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin (developed by Imperial College London), DAOP-ADL (developed by University of Málaga), and ByADL (University of L'Aquila, Italy). Common elements of an ADL are component, connector and configuration.

Views

Software architecture is usually organized in views, which are analogous to the different types of blueprints made in building architecture. A view is a representation of a set of system components and relationships among them. Within the ontology established by ANSI/IEEE 1471-2000, views are responses to viewpoints, where a viewpoint is a specification that describes the architecture in question from the perspective of a given set of stakeholders and their concerns.

Task The viewpoint specifies not only the concerns addressed but the presentation, model kinds used, conventions used and any consistency (correspondence) rules to keep a view consistent with other views. Analyze

Some possible views (actually, viewpoints in the 1471 ontology) are:

- Functional/logic view
- Code/module view
- Development/structural view
- Concurrency/process/runtime/thread view
- Physical/deployment/install view
- User action/feedback view
- Data view/data model.

More than a few languages for describing software architectures ('architecture description language' in ISO/IEC 42010/IEEE-1471 terminology) have been devised, but no agreement exists on which symbol-set or language should be used to describe each architecture view. The UML is a standard that can be used "for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes." Thus, the UML is a visual language that can be used to create software architecture.

Architecture Frameworks

Frameworks related to the domain of software architecture are:

- 4+1
- RM-ODP (Reference Model of Open Distributed Processing)
- Service-Oriented Modeling Framework (SOMF)

Notes

Other architectures such as the Zachman Framework, DODAF, and TOGAF relate to the field of Enterprise architecture.

The Distinction from Functional Design

The IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology defines the following distinctions:

- ***Architectural Design:*** The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- ***Detailed Design:*** The process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation.
- ***Functional Design:*** The process of defining the working relationships among the components of a system.
- ***Preliminary Design:*** The process of analyzing design alternatives and defining the architecture, components, interfaces, and timing/sizing estimates for a system or components.

Software architecture, also explained as strategic design, is an activity concerned with global requirements governing how a solution is implemented such as programming paradigms, architectural styles, component-based software engineering standards, architectural patterns, security, scale, integration, and law-governed regularities. Functional design, also described as tactical design, is an activity concerned with local requirements governing what a solution does such as algorithms, design patterns, programming idioms, refactorings, and low-level implementation.

According to the Intension/Locality Hypothesis, the distinction between architectural and detailed design is defined by the Locality Criterion, according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program which does not.

Example: The client-server style is architectural (strategic) because a program that is built on this principle can be expanded into a program which is not client-server; for example, by adding peer-to-peer nodes.

Architecture is design but not all design is architectural In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases. Examples of rules or heuristics that architects (or organizations) can establish when they want to distinguish between architecture and detailed design include:

- Architecture is driven by non-functional requirements, while functional design is driven by functional requirements.
- Pseudo-code belongs in the detailed design document.
- UML component, deployment, and package diagrams generally appear in software architecture documents; UML class, object, and behavior diagrams appear in detailed functional design documents.

Notes

Self Assessment

Fill in the blanks:

1. The Assembler translates symbolic source language instructions into an …….
2. The output produced by the …… may include a punched paper tape containing the object program and/or an object program listing.
3. Earlier problems of complexity were solved by developers by choosing the right data structures, developing …… , and by applying the concept of separation of concerns.
4. Early attempts to capture and explain software architecture of a system were imprecise and ……, often characterized by a set of box-and-line diagrams.
5. The software architecture discipline is centered on the idea of reducing …… through abstraction and separation of concerns.
6. Software architecture is commonly organized in views, which are analogous to the different types of …… made in building architecture.
7. Architecture is design but not all design is architectural In practice, the architect is the one who draws the line between …… and detailed design.
8. Software architecture, also described as …… , is an activity concerned with global requirements.
9. The …… is a standard that can be used "for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes."
10. …… is the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation.
11. Architecture is driven by …… requirements, while functional design is driven by functional requirements.

3.2 New Approach to New Machine

The following language should be placed in all Request for Proposals (RFP), contracts and purchase orders, for the acquisition and development of new computer systems or software in regards to social security numbers. The vendor of choice must adhere to these guidelines.

- The system must use the social security number only as a data element or alternate key to a database and not as a primary key to a database.
- The system must not display social security numbers visually (such as on monitors, printed forms, system outputs) unless required by law or permitted by this policy.
- Name and directory systems must be capable of being indexed or keyed on the unique identifier, once it is assigned, and not on the social security number.
- For those databases that require social security numbers, the databases may automatically cross-reference between the social security number and other information through the use of conversion tables within a system or other technical mechanisms.

Self Assessment

Fill in the blank:

12. Name and …… systems must be capable of being indexed or keyed on the unique identifier.

Notes

3.3 Machine Level Language (1940-1950)

Computer can understand binary codes (i.e., 1, 0) only. So the instructions given to the computer can only be in 1 or 0. The language which contains binary codes is called machine level language.

A typical machine level language instruction essentially contains two parts:

(a) Operation part : Specifies what is to be performed.

(b) Address part : Specifies the location of data to be manipulated.

Writing programs in machine level language was a tedious task as it is difficult for human beings to remember all the binary codes.

Did u know? **What are high level languages?**

A programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages. In contrast, assembly languages are considered low-level because they are very close to machine languages.

Advantages

(a) Machine level instructions are directly executable.

(b) Machine level language makes most efficient use of computer system resources like storage, register, etc.

(c) Machine language instructions can be used to manipulate individual bits.

Disadvantages

(a) As machine level languages are device dependent, the programs are not portable from one computer to another.

(b) Programming in machine language usually results in poor programmer productivity.

(c) Programs in machine language are more error prone and difficult to debug.

(d) Computer storage locations must be addressed directly, not symbolically.

(e) Machine language requires a high level of programming skills, which increases programmer training costs.

Self Assessment

Fill in the blanks:

13. Machine language instructions can be used to manipulate …….. bits.

14. …….. in machine language are more error prone and difficult to debug.

3.4 Assembly Language (1950-1958)

Assembly languages are also known as second generation languages. These languages substitute alphabetic symbols for the binary codes of machine language.

Notes

In assembly language, symbols are used in place of absolute addresses to represent memory locations.

Mnemonics are used for operation code, i.e., single letters or short abbreviations that help the programmers to understand what the code represents. For example, MOV AX, DX.

Here mnemonic MOV represents 'transfer' operation and AX, DX are used to represent the registers.

Advantages

(a) Assembly language is easier to use than machine language.

(b) An assembler is useful for detecting programming errors.

(c) Programmers do not have to know the absolute addresses of data items.

(d) Assembly languages encourage modular programming.

Disadvantages

(a) Assembly language programs are not directly executable.

(b) Assembly languages are machine dependent and, therefore, not portable from one machine to another.

(c) Programming in assembly language requires a higher level of programming skill.

Self Assessment

Fill in the blank:

15. Assembly languages are …….. dependent and, therefore, not portable from one machine to another.

Caselet

CMC Develops Sports Management Software

CMC Ltd., which is now part of Tata Consultancy Services (TCS), announced that it has developed a Games Management Software that facilitates management of mega sporting events by processing large data and helps disseminate information.

The General Manager, CMC Ltd., Mr Chinta Prakash, addressing a press conference here, said that the Games Management System has been successfully deployed in Andhra Pradesh with the support of the State Government and the organisers at the recently concluded National Games 2002 held at Hyderabad and Visakhapatnam.

The entire IT Infrastructure – Hardware and Software support for the National Games including hiring and setting up of over 300 PCs distributed across different media centres, 26 venues and accreditation centres, extensive networking spread across Hyderabad and Visakhapatnam was achieved in a record 15 days.

Contd...

Notes

CMC was awarded the order just one month before the major sports event and its mandate included establishing and managing a 24-hour operational support for the IT infrastructure, call centre for providing information relating to the event to the general public, implementing its games management system, software package to facilitate accreditation of the participants, providing information from the GMS for delivery over SMS messaging network.

In addition to the software and the infrastructure, CMC provided consultancy for installation and operational support for the much-appreciated video display board at the GMC Balayogi Stadium - the main athletic stadium at Gachi Bowli.

Mr Prakash said that the games management system could enhance the effectiveness in conducting various sporting events, in dissemination of information to various bodies such as organising committees, sport federations while providing administrative support. CMC has enlisted a quite a few clients for this software.

3.5 Summary

- The field of computer science has come across problems associated with complexity since its formation.
- Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns.
- Although the term "software architecture" is relatively new to the industry, the fundamental principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s.
- Computer can understand binary codes (i.e., 1, 0) only.
- So the instructions given to the computer can only be in 1 or 0. The language which contains binary codes is called machine level language.
- Assembly languages are also known as second generation languages.
- These languages substitute alphabetic symbols for the binary codes of machine language.

3.6 Keywords

ADLs: Architecture Description Languages

RFP: Request for proposals

3.7 Review Questions

1. The Assembler accepts as input a paper tape containing a control statement and a source language program. Explain.
2. The object program may be either relocatable or absolute. Discuss.
3. Assembler is a program that converts computer instructions into bits that can be processed by processor in computer to perform certain operation. Comment.
4. The field of computer science has come across problems associated with complexity since its formation. Examine.
5. Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system. Give reasons.

Notes

6. The origin of software architecture as a concept was first identified in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. Explain.
7. A view is a representation of a set of system components and relationships among them. Discuss.
8. Explain the several languages for describing software architectures.
9. Machine level language makes most efficient use of computer system resources. Comment.
10. In assembly language, symbols are used in place of absolute addresses to represent memory locations. Explain briefly.

Answers: Self Assessment

1. object program
2. Assembler
3. algorithms
4. disorganized
5. complexity
6. blueprints
7. software architecture
8. strategic design
9. UML
10. Detailed Design
11. non-functional
12. directory
13. individual
14. Programs
15. machine

3.8 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A.Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online links

http://en.wikipedia.org/wiki/Assembly_language

http://www.tuiasi.ro/users/103/Bind4.pdf

Unit 4: Design of Assembler

Notes

CONTENTS

Objectives
Introduction
4.1 Design of Assembler
4.1.1 Addressing Options
4.1.2 Statement of Problem
4.1.3 Data Structure
4.1.4 Format of the Databases
4.1.5 Algorithm
4.1.6 Look for Modularity
4.2 Summary
4.3 Keywords
4.4 Review Questions
4.5 Further Readings

Objectives

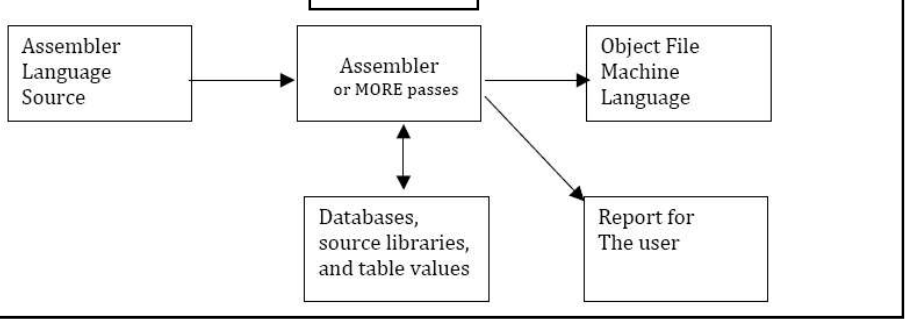
After studying this unit, you will be able to:

- Understand the concept of designing assembler
- Discuss data structure, format of databases, and algorithm in the design of assembler
- Discuss modularity in the design of assembler

Introduction

An assembler is a program that receives as input an assembly language program and generates its machine language correspondent along with information for the linker/loader and report for the consumer. Fundamentally it converts code in one language into another. Assemblers are nothing more than huge translators that convert code from one language to another following recognized syntax rules. In many cases the code is converted into machine code with information concerning the program, position of internal variables that might be referenced by another program, information concerning local addresses that will require to be updated depending on where the operating system allocates the program to memory. These activities are the role of the linker/loader.

Figure 4.1



Notes

4.1 Design of Assembler

4.1.1 Addressing Options

Absolute Addresses: These are the address fields that will not require to be customized by the loader.

Relocatable Addresses: These are the addresses that will require to be accustomed if and only if the loader choose to position this program at a dissimilar memory location than the assembler allocated.

Externally defined Symbols/Labels: The assembler does not recognize the addresses (or values) of these symbols and it depends on the loader to locate the programs comprising these symbols, load them into memory, and position the addresses of these symbols in the calling program.

Indirect Addresses: The address refers to a location in memory that comprises the target address.

Direct Addresses: Address of an explicit position in the physical memory.

Immediate: The normal address will be utilized for an immediate value instead of an address.

Caution Inside the instruction there must be a flag to specify this.

Task Make distinction between direct addresses and indirect addresses.

4.1.2 Statement of Problem

Assume that we are the "assembler" attempting to convert the program in the first column below. We read the START instruction and observe that it is pseudo-op (directive) instruction (to the assembler) providing JOHN as the name of this program; the assembler must pass the name of this program to the loader. Observe that in this example the location counter depends on the number of bytes (4 per word).

Intermediate Steps on Assembling a Program

SOURCE PROGRAM			First Pass			Second Pass		
JOHN	START	0						
	LOAD	l,VAL2	0	0001b	1x,????	1	0001b	1x,C relocatable
	ADD	l,VAL1	4	1000b	1x,????	4	1000b	1x,10 relocatable
	STORE	l,RES	8	0010b	1x,????	8	0010b	1x,14 relocatable
VAL1	DATA	1*4	C	00000004x		C	00000004x	absolute
VAL2	DATA	1*5	10	00000005x		10	00000005x	absolute
RES	DATA	1*1	14	00000001x		14	00000001x	absolute
	END							

The program begins by setting the signed load address at 0. Next appears a LOAD instruction: LOAD l, VAL2. We require to ensure it is valid syntax. Look up the bit/hex configuration for the mnemonic in a table (machine operations table) and put the bit configuration for the LOAD (0001b) in the suitable place in the machine language instruction. Next we require the address of VAL2. Here, however, we do not recognize where VAL2 is, so we cannot supply its address. So we move on, we preserve a location counter representing the relative address of the instruction being processed; this counter is incremented by 4 bytes (length of a LOAD instruction). The next instruction is an ADD instruction. We observe the op-code (1000b), but we do not recognize the

Notes

address for VAL1. The same thing occurs with the STORE instruction. The DATA instruction is a directive asking the assembler to describe some data; for VAL1 we are to generate a '4'. We know that this word will be stored at relative location C, since the location counter now has the value C, having been incremented by the length of each previous instruction. The first instruction, 4 bytes long (1 word), is at relative address 0. The next two instructions are also 4 bytes long. We say that the symbol "VAL1" has the address C. The next instruction has as a label VAL2 and an connected location counter address 10. The label on RES has an connected address of 14.

As the assembler, we can now go back via the program and fill in the missing information in the third column. Since symbols can emerge before they are defined, it is suitable to make two passes over the input (as this example displays). The first pass has only to define the symbols, allocate addresses, and carry out syntax checking; the second pass can then produce the instructions and addresses. Particularly the assembler does the following:

1. Computes the Location Counter (Pass 1)
2. Scans for errors (Pass 1)
3. Produce instructions:
   (a) Evaluate the mnemonic in the operation field to produce its machine code. (Pass 1)
   (b) Evaluate the sub fields-find the value of every symbol, process literals, and assign addresses. (Pass 1 & 2)
4. Process pseudo-ops/directives (Pass 1 & 2)

We can group these tasks into two passes or sequential scans over the input; connected with each task are one or more assembler modules.

Pass 1: Purpose-define Symbols and Literals, and Determine the Location Counter

1. Syntax scan
2. Find out length of machine instructions (Machine_Operations_Table)
3. Maintain track of Location Counter (Location_Counter)
4. Produce Symbol Table with address/value of symbols (Symbol_Table)
5. Memorize literals (Literal_Table came be combined with the Symbol_Table)
6. Process some Directives, e.g., EQU, BEGIN/ORIGIN/Start/PGM
7. Construct an intermediate version (file or structure) to assist pass 2
8. Produce data for DATA/num, SKIP, and literals (DATA_GEN).

Pass 2: Purpose-generate Object Program

1. Observe addresses of symbols (Symbol_Table)
2. Produce instructions (Machine_Operations_Table)
3. Produce reports for the user
4. Produce the object file intended for the linker and then on to the simulator.

4.1.3 Data Structure

The next step in our design process is to institute the databases internal and external. We know for instance we will need a technique to import in the instructions, the bit equivalence, and a flag to specify the format of the operand field. A similar technique is required for directives. Internally we will require a symbol table.

Notes

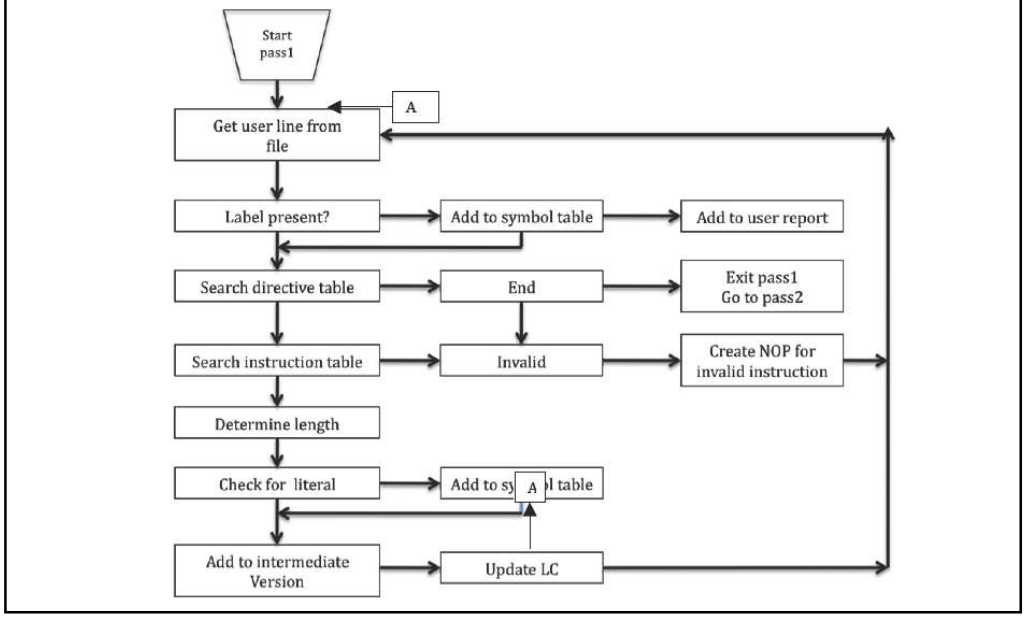
Pass 1 Databases

1. Input source program.
2. Location Counter utilized to maintain track of every instruction's location.
3. Machine-Operation Table signifies the symbolic mnemonic for every instruction and its length (two, four, or six bytes).
4. Directive/Pseudo-Operation Table signifies the symbolic mnemonic and action to be taken for each pseudo-op in pass 1.
5. Symbol Table used to amass each label and its equivalent value.
6. Literal Table utilized to amass each literal encountered and its matching assigned location.
7. An accurate copy of the input is saved for pass 2. This may be accumulated in a secondary storage device, like disk or in a structure in main memory (small programs like the CSE 560 programs) (enlarged with information that pass allocated).

Pass 2 Databases

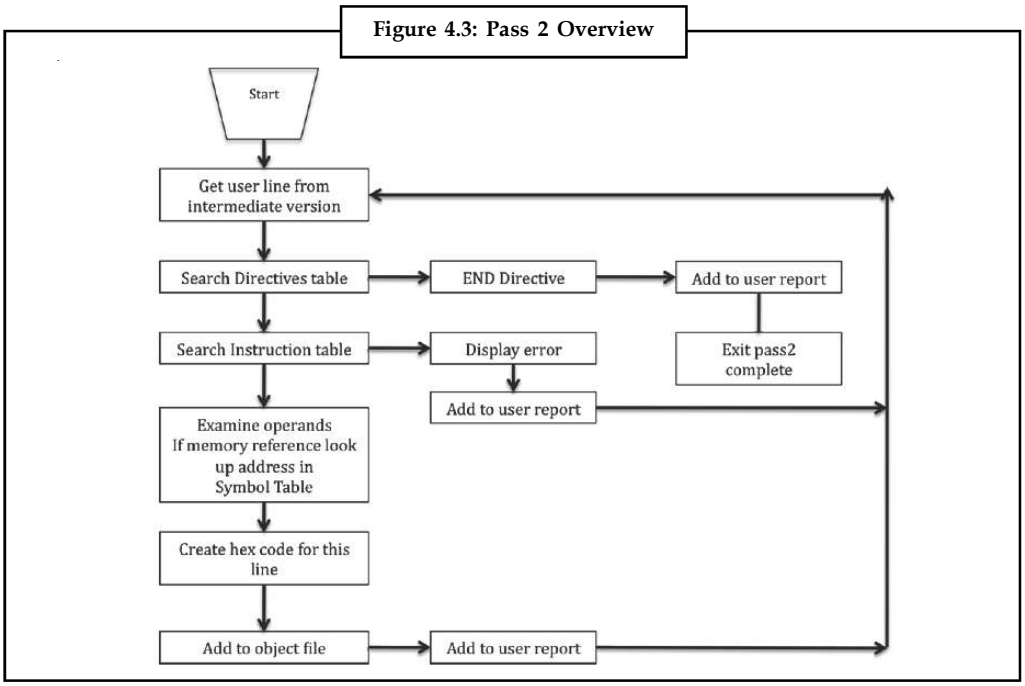
1. Copy of source program (with pass 1 knowledge).
2. Location Counter.
3. Machine Operation Table, which signifies for every instruction: (a) symbolic mnemonic; (b) length; (c) binary machine opcode, and (d) format (e.g., RS, RX, and SSI).
4. Directive/Pseudo-Operation Table that signifies for every directive the symbolic mnemonic and the action to be taken in pass 2.
5. Symbol Table organized by pass 1, comprising each label and its corresponding value.
6. A workspace, INST, is used to hold every instruction as its different parts (e.g., binary op-code, register fields, length fields, displacement field) are being assembled together.
7. An output file of assembled instructions in the format needed by the linker.

Figure 4.2: Assembler Pass 1 Structure Flowchart



Notes

Figure 4.3: Pass 2 Overview



4.1.4 Format of the Databases

The third step in our design process is to identify the format and content of each of the databases – a mission that must be undertaken even before depicting the particular algorithm underlying the assembler design.

Pass 2 needs a Machine Operation Table containing name, length, binary code, and format; Pass 1 requires only name, format, and length. We could use two separate tables with different formats and contents or use the same table for both passes; the same is true of the Directive/ Pseudo Operation Table. By simplifying the table formats, we could merge the tables into one table. For this particular design, we will use disconnected tables.

Once we decide what information fist in each database, it is essential to identify the format of each entry.

Example: In what format are symbols stored (e.g., left justified, padded with blanks, coded in ASCII) and what are the coding conventions?

Example: The Machine-Op Table and Pseudo Op Tables are examples of fixed tables.

The contents of these tables are not filled in or tainted during the assembly process.

Notes In real life, the algorithm, database, and formats are all interconnected. Their specification is in practical designs, circular, in that the designer has in mind some traits of the format and algorithm he/she plans to use and continues to iterate their design until all cases function.

Notes

Machine-op Table for passes 1 and 2

Mnemonic op-code	Binary op-code	Instruction length	Instruction format	Operand format
ADD	1000	1 word	RX	Register, address(index register)
MPY	0101	1 word	RX	Register, address(index register)
STORE	0010	1 word	RX	Register, address(index register)
LOAD	0001	1 word	RX	Register, address(index register)
HALT	1111	1 word	n/a	n/a
MVC	1001	1 word	SSI	address(index register), address(index register)

Directive Table

Directive	Pass 1	Pass 2	Operand Format Code	operand Range low	operand Range high	Size
END	Y	Y	AN2	0	255	0
EQU	Y	N	AN	0	255	0
START or PGM	Y	N	AN2	0	255	0
NUM	Y	Y	AN-Max	-32768	+32767	1 word

The Symbol Table and Literal Table comprise for each entry the name and assembly-time address/value fields but also a length field, usage field, and a relative position indicator. The length field signifies the length (in bytes) of the instruction or data to which the symbol is connected.

Example: Consider

Symbol Table for pass 1 and pass 2

Symbol	Location Addr/Value (hexadecimal)	Length (hexadecimal)	Relocation code	Usage	Value if EQU assigns a constant
JOHN	0000	22 (total pgm length)	R	Label	
VAL1	000C	1 word	R	Data	
VAL2	0010	1 word	R	Data	
RES	0014	1 word	R	Data	

The relative-location column informs the assembler whether the value of the symbol is absolute (does not modify if the program is shifted in memory), or relative to the beginning of the program. If the symbol is defined by equivalence with a constant (e.g., 6) or an absolute symbol, then the symbol is absolute. Or else, it is a relative symbol. The relative-location field in the symbol table will have an R in it if the symbol is relative, or an A if the symbol is absolute.

Did u know? In the definite assembler a considerably more multifaceted algorithm is generally used.

The following assembly program is used to exemplify the use of the variable tables (symbol table and literal table). We are only regarded with the difficulty of assembling this program; its particular function is irrelevant.

Sample Assembly Source Program (generic code)

```
1    PRGAM2    START    0
2              SUB      TOTAL,TOTAL
3    AC        EQU      2
4    INDEX     EQU      3
5    TOTAL     EQU      4
```

Contd...

Notes

```
 6      SETUP       EQU          *
 7                  SUB          INDEX, INDEX
 8      LOOP        LOAD         AC,DATA1(INDEX)
 9                  ADDREG       TOTAL,AC
10                  ADD          AC,=5
11                  STORE        AC,SAVE(INDEX)
12                  ADD          INDEX,=1
13                  COMPARE      INDEX,=50
14                  BNOTEQUAL    LOOP
15                  LOADREG      1,TOTAL
16                  BRANCH       14
17      SAVE        SKS          5
18      DATAAREA    EQU          *
19      DATA1       NUM          25,26,97,101,.....
                                 (50 numbers)
20                  END
```

In preserving with the purpose of pass 1 of an assembler (define symbols and literals), we can generate the symbol and literal tables displayed below.

Symbol Table

Symbol	Location Address (hexadecimal)	Length (hexadecimal)	Relocation code	Usage	Value if EQU assigns a constant
PRGAM2	0	108	R	Pgm name	
AC	n/a	n/a	A	EQU value	2
INDEX	n/a	n/a	A	EQU value	3
TOTAL	n/a	n/a	A	EQU value	4
SETUP	1	1	R	EQU Address	n/a
LOOP	2	1	R	Instruction Label	n/a
SAVE	B	1	R	Data Label	n/a
DATAAREA	C	1	R	EQU Address	n/a
DATA1	C	C8	R	Data Label	n/a

Literal Table

Symbol	Location Address (hexadecimal)	Length (hexadecimal)	Relocation code	Usage	Value
=5	F8	1 word	R	LITERAL	5
=1	FC	1 word	R	LITERAL	1
=50	100	1 word	R	LITERAL	50

We scan the program above along with a local counter. For every symbol in the label field we make an entry in the symbol table. For the symbol PRGAM2 its value is its relative location. We modernize the location counter, observing that the every instruction is 1 word long. Continuing, we discover that the next four symbols are defined by the pseudo-op EQU. These symbols are entered into the symbol to and the connected values given in the argument fields of the EQU statements entered.

Observe that none of the Pseudo-ops encountered effect location counter as they do not effect in any object code. So the location counter has a value 8 when LOOP is encountered. Thus, LOOP is entered into the symbol table with a value 2. It is a relocatable variable. Its length is 1 word as it signifies a location that will enclose an instruction. All other symbols are entered in like manner. In the same pass all literals are renowned and entered into a literal table. The first literal is in statement 10 and its value is the address of the location that comprises the literal. Because this is the first literal, it will have the first address the literal area. The literal table will be positioned at the end of the program.

Notes

For every instruction in pass 2, we generate the equivalent machine language instruction as shown below:

1. Find value of SETUP in symbol table (which is 2)
2. Find value of op-code in machine op table binary op code for LOAD
3. Formulate address
4. Assemble output code in suitable formula.

Likewise, we produce instructions for the remaining code as displayed below:

Generated "machine code" hex loc.	statement no.		Instruction	
0	1	PRGAM2	START	0
0	2		SUB	TOTAL,TOTAL
-	3	AC	EQU	2
-	4	INDEX	EQU	3
-	5	TOTAL	EQU	4
1	6	SETUP	EQU	*
1	7		SUB	INDEX, INDEX
2	8	LOOP	LOAD	AC,DATA1(INDEX)
3	9		ADDREG	TOTAL,AC
4	10		ADD	AC,=5
5	11		STORE	AC,SAVE(INDEX)
6	12		ADD	INDEX,=2
7	13		COMPARE	INDEX,=100
8	14		BNOTEQUAL	LOOP
9	15		LOADREG	1,TOTAL
A	16		BRANCH	14
B	17	SAVE	SKS	5
C	19	DATAAREA	EQU	*
C	20	DATA1	NUM	25,26,97,101..... (50 numbers)
	21		END	

4.1.5 Algorithm

Pass 1: Define Symbols

The reason of the first pass is to allocate a location to every instruction and data defining pseudo-instruction, and thus to define addresses/values for symbols occurring in the label fields of the source program. Primarily, the Location Counter is set to the first location in the program (relative address 0). Then a source statement is read. The operation-code field is scrutinized to find out if it is a directive pseudo-op; if it is not, the table of machine op-codes is investigated to discover a match for the source statement's opcode field. The matched entry states the length of the instruction. The operand field is scanned for the occurrence of a literal. If a new literal is found, it is entered into the Literal Table for afterward processing. The label field of the source statement is then inspected for the presence of a symbol. If there is a label, the symbol is saved in the Symbol Table along with the existing value of the location counter. Lastly, the current value of the location counter is incremented by the length of the instruction and a replica of the source record is saved for use by pass 2. The above sequence is then repeated for the next instruction. The loop depicted is physically a small portion of pass I although it is the most

Notes

significant function. The major sections of pass 1 and pass 2 are dedicated to the particular processing required for the various pseudo-operations. For ease, only a few major pseudos are openly indicated in the flowchart; the others are processed in a uncomplicated way. We now consider what must be performed to process a pseudo op. Pass 1 is only regarded with pseudo-ops that define symbols (labels) or affect the location counter.

In the case of the EQU pseudo-op during pass 1, we are related only with defining the symbol in the label field. This needs evaluating the expression in the operand field. (The symbols in the operand field of an EQU statement must have been defined before.) The SKS and NUM pseudo ops can affect both the location counter and the definition of symbols in pass 1.

Caution The operand field must be scrutinized to find out the number of bytes/words of storage needed.

Because of requirements for certain alignment conditions (e.g., full words must begin on a byte whose address is a multiple of two), it may be essential to adjust the location counter before defining the symbol. When the END pseudo-op is encountered, pass 1 is finished. Before transferring control to pass 2, there are different "housekeeping" operations that must be done. These include allocating locations to literals that have been collected during pass 1, a process that is very similar to that for the NUM/CHC pseudo op. Lastly, conditions are re-initialized for processing by pass 2.

Pass 1

Read source file line by line

Save it in enlarged intermediate file (required for pass 2 report), validate it if symbol check symbol table

Is it already present?

==> yes==> set error code

==>no==> enter into table

enter LC in HEX into table

Check instruction

does it consume memory

==> yes==> update LC

Check Pseudo-op

Label? ==> place in symbol table if not already entered

Consume memory? ==> yes==> update LC

Process as needed

if END finish up pass 1 check for more records (error)

start pass 2

Check syntax of operand field

save source line (with added information) for pass 2

appended with LC, op-code, r, s, x, error codes

Notes

In-between Passes

sort symbol table

save intermediate source file

Construct general object file information

Pass 2: Generate Code

After all the symbols have been depicted by pass 1, it is probable to terminate the assembly by processing each record and determining values for its operation code and its operand field. Additionally, pass 2 must structure the generated code into the suitable format for later processing by the loader, and print an assembly listing comprising the original source and the hexadecimal equivalent of the bytes produced. A record is read from the source file left by pass 1. The operation code field is scrutinized to find out if it is a pseudo-op; if it is not, the table of machine op codes (MOT) is hunted to discover a match for the op-code field.

The matching MOT entry indicates the length, binary op-code, and the format type of the instruction.

Did u know? The operand fields of the different instruction format types need somewhat different processing.

For the RR-format instructions, each of the two register specification fields is evaluated. This evaluation may be very straightforward, as in:

AR 2,3 [RR Format]

MPY 1,EVEN [RX format]

MPY 1,EVEN(3) [RX format]

or more complex, as in:

MPY l,EVEN+EVEN+I

MOVE MUD, DIRT [SSI format]

MOVE MUD(3),DIRT [SSI format]

The two fields are inserted into their individual fields in the RR-instruction. For RX format instructions, the register and index fields are assessed and processed in the similar manner as the register specifications for RR-format instructions. The storage address operand is evaluated to produce an Effective Address EA). Only the RR and RX instruction types are overtly displayed in the flowchart. The other instruction formats are managed similarly. After the instruction has been assembled, it is put into the essential format for later processing by the loader. Usually, numerous instructions are placed on a single record. A listing line containing a replica of the source card, its allocated storage location, and its hexadecimal representation is then printed. Lastly, the location counter is incremented and processing is continued with the next record. As in pass 1, each of the pseudo-ops calls for special processing.

Notes The EQU pseudo-op needs very little processing in pass 2, since symbol definition was completed in pass 1. It is essential only to print the EQU record as part of the printed listing.

Notes

The SKS and NUM/CHC pseudo-ops are processed essentially as in pass 1. In pass 2, though, actual code must be produced for the NUM/CHC pseudo-op. Based upon the data types specified, this includes diverse conversions (e.g., floating point character to binary representation) and symbol evaluations (e.g., address constants). The END pseudo-op signifies the end of the source program and finishes the assembly. Different "housekeeping" tasks must now be performed."

For example, code must be produced for any literals remaining in the Literal Table.

Pass 2

Read improved source file

if instruction

==> yes==> instruction

use op-code (binary) saved from pass 1

Begin building binary version of the instruction

Add r & X field binary

Transform to hex

Search S symbol in symbol/literal table

Append hex address to end of instruction

Find out A/R/E...then add A/R/E to end of hex op-code

Append symbol if E

Write valid machine code instruction

==> no==> pseudo-op

Define hex equivalent for CHR, NUM and ADR

Write valid hex to object file

Process other pass 2 formatting pseudo ops

END

Finish object file

Finish output

Print symbol table

4.1.6 Look for Modularity

We now evaluate our design, searching functions that can be isolated. Usually, such functions fall into two categories: (1) multi-use and (2) unique. Listed below are some of the functions that may be isolated in the two passes. Observe a common table building/searching, hex integer bit conversions.

Pass 1

1.	READ1	Read the next assembly source card.
2.	POT_TABLE	Search the pass 1 Pseudo-Op Table (POT) for a match with the operation field of the current source card.

Notes

3.	MOT_TABLE	Search the Machine-Op Table (MOT) for a match with the operation of the current source card.
4.	SYM_TABLE	Store a label and its associated value into the Symbol Table (ST). If the symbol is already in the table, return error indication (multiply defined symbol).
5.	LIT_TABLE	Store a literal into the Literal Table (LT); do not store the same literal twice.
6.	WRITE1	Write a copy of the assembly source record on a storage device for use by pass 2.
7.	DLENGTH	Scan operand field of space consuming pseudo-ops to determine the amount of storage required.
8.	EVAL	Evaluate an arithmetic expression consisting of constants and symbols (e.g. 6, ALPHA, BETA+14*GAMMA).
9.	STGET	Search the Symbol Table (ST) for the entry corresponding to a specific symbol (used by SYM_TABLE, and EVAL).
10.	LIT_LC	Assign storage locations to each literal in the literal table (may use DLENGTH).

Pass 2

1.	READ2	Read the next assembly source record from the file copy.
2.	POT_TABLE	Same as in pass 1
3.	MOT_TABLE	Same as in pass 1 (evaluate expressions).
4.	EVAL	Same as in pass 1 (evaluate expressions).
5.	OBJECT	Transform produced instruction to object record format; write the record when it is filled with data.
6.	PRINT	Transform relative location and produced code to character format; print the line along with copy of the source card.
7.	DATAGEN	Process the fields of the data-generating pseudo-op to produce object code (uses EVAL).
8.	DLENGTH	Similar as in pass 1.
9.	LIT_GEN	Produce code for literals (uses DATAGEN).

Each of these functions should separately go via the whole design process (problem statement, data basics, algorithm, modularity, etc.). These functions can be executed as disconnected external subroutines, as internal subroutines, or as sections of the pass 1 and pass 2 programs. In any case, the aptitude to tract functions separately makes it much simpler to design the structure of the assembler and each of its parts. So, instead of observing the assembler as a single program (of 1,000 to 10,000 source statements), we examine it as a coordinated collection of routines each of relatively minor size and complexity. We will not attempt to scrutinize all of these functional routines in detail since they are quite uncomplicated. There are two particular observations of interest:

1. Many of the routines involve the scanning or evaluation of fields (e.g., DLENGTH, EVAL, DATA_GEN);

Notes

2. Many other routines entail the processing of tables by storing or searching (e.g., POT_TABLE, MOT_TABLE, LIT_TABLE, SYSTO, STGET).

Tables

Assemblers depend chiefly on Tables. Obtaining the tables established correctly is critical to the development of a straightforward assembler. Most assemblers depend on:

Machine Operations table comprises all the instructions, binary equivalent, length, and valid instruction format (parsing information).

Pseudo-op table that comprises all the pseudo-ops length (if necessary), format (parsing information) Symbol Table that includes all program defined labels/variables, length, types (label, numeric, character, address), relative (to begin of the program) memory location, and location(s) where utilized.

Literal table (relies on how literal are implemented) which includes all program defined literals, length, type (numeric, character, address), relative (to start of the program) memory location, and location(s) where utilized.

Data Files/Tables

name		Pass 1	Pass 2
Source	as provided by programmer	*reviewed*	*augmented*
MOT	as provided by assembler developer	*searched*	*not needed*
POT	as provided by assembler developer	*searched*	*searched*
Symbol Table	format defined by the developers	*entries made*	*searched*
Literal Table	format defined by the developers	*entries made*	*searched*
Intermediate source	*created* Assembler builds this source image plus pass 1 information	*searched*	

Task Illustrate the function of tables in designing assemblers.

Self Assessment

Fill in the blanks:

1. ..... converts code in one language into another.

Notes

2. ..... are the address fields that will not require to be customized by the loader.
3. ..... are the addresses that will require to be accustomed if and only if the loader choose to position this program at a dissimilar memory location than the assembler allocated.
4. ..... Addresses refers to a location in memory that comprises the target address.
5. The ..... instruction is a directive asking the assembler to describe some data.
6. Since symbols can emerge before they are defined, it is suitable to make two passes over the .....
7. Pass 2 needs a ..... Table containing name, length, binary code, and format.
8. ..... requires only name, format, and length.
9. By ..... the table formats, we could merge the tables into one table.
10. Once we decide what information fist in each database, it is essential to identify the ..... of each entry.
11. The contents of fixed tables are not filled in or tainted during the ..... process.
12. The ..... field signifies the length (in bytes) of the instruction or data to which the symbol is connected.
13. For every instruction in ....., we generate the equivalent machine language instruction.
14. The reason of the ..... pass is to allocate a location to every instruction and data defining pseudo-instruction, and thus to define addresses/values for symbols occurring in the label fields of the source program.
15. When the ..... pseudo-op is encountered, pass 1 is finished.

4.2 Summary

- An assembler is a program that receives as input an assembly language program and generates its machine language correspondent along with information for the linker/ loader and report for the consumer.
- Absolute addresses are the address fields that will not require to be customized by the loader.
- Relocatable addresses are the addresses that will require to be accustomed if and only if the loader choose to position this program at a dissimilar memory location than the assembler allocated.
- Since symbols can emerge before they are defined, it is suitable to make two passes over the input (as this example displays).
- Once we decide what information fist in each database, it is essential to identify the format of each entry.
- The Symbol Table and Literal Table comprise for each entry the name and assembly-time address/value fields but also a length field, usage field, and a relative position indicator.
- The reason of the first pass is to allocate a location to every instruction and data defining pseudo-instruction, and thus to define addresses/values for symbols occurring in the label fields of the source program.

Notes

- After all the symbols have been depicted by pass 1, it is probable to terminate the assembly by processing each record and determining values for its operation code and its operand field.
- Instead of observing the assembler as a single program (of 1,000 to 10,000 source statements), we examine it as a coordinated collection of routines each of relatively minor size and complexity.

4.3 Keywords

Absolute Addresses: These are the address fields that will not require to be customized by the loader.

Assembler: An assembler is a program that receives as input an assembly language program and generates its machine language correspondent along with information for the linker/loader and report for the consumer.

Pass 1: Pass 1 requires only name, format, and length.

Pass 2: Pass 2 needs a Machine Operation Table containing name, length, binary code, and format.

Relocatable Addresses: These are the addresses that will require to be accustomed if and only if the loader choose to position this program at a dissimilar memory location than the assembler allocated.

4.4 Review Questions

1. What is an assembler? Discuss the function of assembler.
2. Illustrate various Addressing Options used in designing assembler.
3. Enlighten the steps on assembling a program with example.
4. How can we group tasks into two passes? Illustrate the purpose of pass 1 and pass 2.
5. Make distinction between pass 1 database and pass 2 databases.
6. Draw the Structure Flowchart for assembler pass 1 and assembler pass 2.
7. Explain the process of identifying the format and content of each of the databases.
8. How do you define symbols in pass 1? Write an algorithm to explain.
9. Explain the process of generating code with the help of an algorithm.
10. Illustrate the concept of modularity in designing assembler.

Answers: Self Assessment

1.	Assembler	2.	Absolute addresses
3.	Relocatable addresses	4.	Indirect
5.	DATA	6.	input
7.	machine operation	8.	Pass 1
9.	simplifying	10.	format
11.	assembly	12.	length

Notes

13. pass 2
14. first
15. END

4.5 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A.Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online link

www.iamroot.org

Unit 5: Table Processing: Searching

Notes

CONTENTS

Objectives

Introduction

5.1 Linear Search

5.2 Binary Searching

5.3 Hash Searching

5.4 Summary

5.5 Keywords

5.6 Review Questions

5.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concept of Linear Search
- Illustrate Binary Search
- Discuss Hash Searching

Introduction

The difficulty of searching is as follows: specified a keyword, locate an entry in the table that matches, and returns its value. The particular problems more than one entry with the similar keyword, and no entry discovered need individual conduct relying on the function of the table. For an assembler's symbol table, these particular cases match to multiply defined symbols and undefined symbols.

5.1 Linear Search

Linear search is a simple searching method. This method accesses table in which the items have not been ordered. One manner to search for a specified keyword is to compare every entry in the table with the specified keyword.

As you can see in the figure 5.1, the symbols and values are accumulated in contiguous locations in an array named SYMTBL and defined by a DS. The word LAST includes the location of the current "end of table."

The loop illustrated will match up the keyword (in the location SYMBOL) with each succeeding item in the table. When a match is found, array named SYMFOUND is used to make exit; if no match is discovered by the end of the table, then execution will move to location NOTFOUND.

Generally, we would search half the table, by means of a linear search, before locating an entry. Thus, the average length of time to locate an entry is

$$T(avg) = [\text{overhead associated with entry probe}] \times \frac{N}{2}$$

Notes

Linear search is demonstrated in Figure 5.1.

Figure 5.1: Sample Linear Search Program

```
             LA     4, SYMTBL        Start of table
LOOP         CLC    0(8,4), SYMBOL   Compare symbols
             BE     SYMFOUND         Equal
             A      4,= F'14'        Move to next symbol
             C      4, LAST          Are we at end of table
             BNE    LOOP             Loop back if not
NOTFOUND            (Symbol not found)
                ⋮
SYMFOUND            (Symbol found)
                ⋮
SYMBOL       DS     CL14             Symbol to be searched for, character string of length 14
SYMTBL       DS     100CL14          Symbol table space (14 bytes per entry)
LAST         DC     A(-----)         Address of current end of symbol table
```

This kind of linear search procedure is okay for short tables and has an ease in the procedure, but it can be very slow for long tables. It is analogous to searching for a word in a dictionary whose contents are not in an ordered manner. It would give slight relieve to recognize that on an average, you only have to look out for half of the dictionary.

Did u know? Linear search is also known as sequential search as it compares the consecutive elements of the specified set by means of the search key.

Self Assessment

Fill in the blanks:

1. Linear search method accesses table in which the items have not been …… .
2. …… process can be very slow for long tables.
3. Linear search is also known as …… search as it compares the consecutive elements of the specified set by means of the search key.
4. A specified keyword can be searched by comparing every …… in the table with the specified keyword.

5.2 Binary Searching

Another relatively simple method of accessing a table is the binary search method. The entries in the table are stored in alphabetically or numerically increasing order.

The drawback of sequential search can be eliminated if it becomes possible to eliminate large portions of the list from consideration in subsequent iterations. Binary search requires sorted data to operate on. In this method, search begins by examining the records in the middle of file rather than the one at one of the ends as in sequential search.

Let us assume that file being searched is sorted on increasing value of key. Based on the result of the comparison with the middle key, Km, following conclusions can be drawn.

Notes

- If K < Km then if the record being searched for is in the file, it must be in the lower numbered half of the file.
- If K = Km then the middle record is the one being searched for.
- If K > Km then if the record being searched for is in the file it must be in the higher numbered half of the file.

A search for a particular item which contains key value resembles the search for a name in a telephone directory. The approximate middle entry of the table is located, and its key value is examined. If its value is too high, then the key value of the middle entry of the first half of the table is examined and the procedure is repeated on the first half until the required item is found. If the value is too low, then the key of middle entry of the second half of the table is tried and the procedure is repeated on the second half.

Example: As an example of binary search, let us consider a table of 15 items as shown in Figure 5.2.

For example, suppose that we are looking for item IF (for ease the values are not displayed). We first compare IF with the middle item LO and discover that IF must be in the top half of the table; a second comparison with the middle item in this half of the table, FU shows IF to be in the second quarter; a third comparison with W shows IF to be in the third eighth of the table (that is, between items 4 and 6). and a final comparison is done with the item in position 5. A comparison failure on the fourth probe would have discovered that the item is not present in the table.

This kind of bracketing technique of looking for a table should be clear in principle even though its accomplishment may be a little more complex. We call this as a binary search or a logarithmic search, you should know that as the effective table is halved on each probe, a maximum of about $\log_2(N)$ probes is needed to search it.

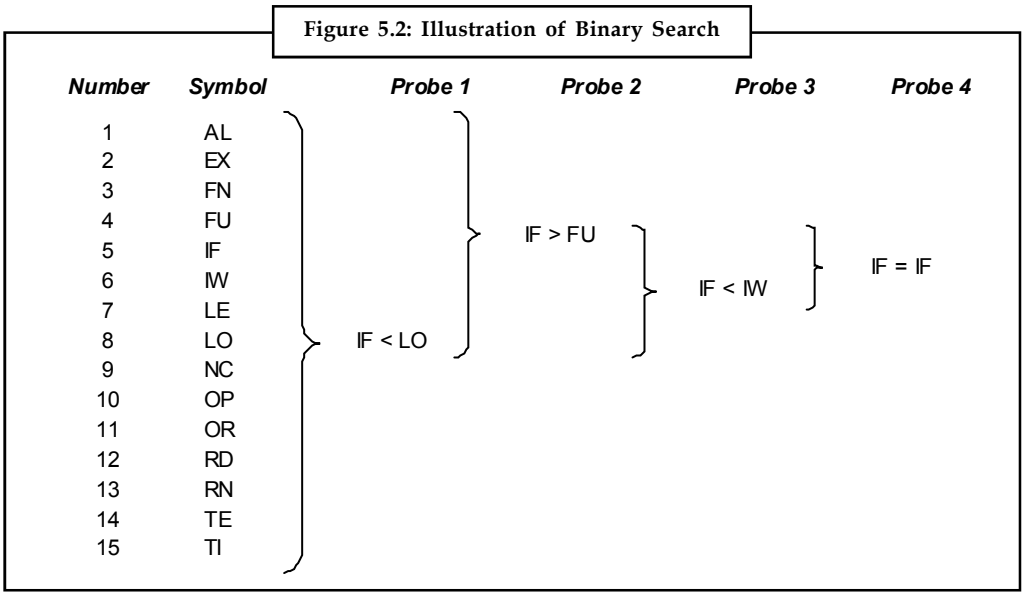
On comparing the times of the linear search with those of the binary search where A and B are overhead times connected with each table probe, we get:

$$T(tin) = A*N$$

$$T(bin) = B* \log_2 N;$$

As the binary search is more complex, the constant B can be considerably larger than A.

Figure 5.2: Illustration of Binary Search



Notes

Let us now see an example of binary search program as below.

Example: Figure 5.3 illustrates an example of binary search program. As the binary process frequently divides by 2, for effectiveness and simplicity we suppose table size is a power of 2. This condition is simply achieved by just adding enough "dummy" entries to the end of the table (such as entries for the symbol ZZZZ ZZZZ).

Figure 5.3: Sample Binary Search Program

```
                L          5,SIZE          Set table size (2N * 14 bytes)
                SRL        5,1             Divide by 2 by shifting
                LR         6,5             Copy into register 6
LOOP            SRL        6,1             Divide table size in half again
                LA         4,SYMTBL(5)     Set address of table entry
                CLC        0(8,4),SYMBOL   Compare with symbol
                BE         FOUND           Symbols match, entry found
                BH         TOOHIGH         SYMTBL entry > SYMBOL
TOOLOW          AR         5,6             Move higher in table
                B          TESTEND
TOOHIGH         SR         5,6             Move lower in table
TESTEND         LTR        6,6             Test if remaining size is 0
                BNZ        LOOP            No, look at next entry
NOTFOUND        (Symbol not found)
                           ⋮
FOUND           (Symbol found)
```

Self Assessment

Fill in the blanks:

5. In …… searching, the entries in the table are stored in alphabetically or numerically increasing order.
6. Binary search requires …… data to operate on.
7. In binary search method, search begins by examining the records in the middle of file rather than the one at one of the ends as in …… search.
8. A search for a particular item which contains key value resembles the search for a …… in a telephone directory.
9. In binary search, an estimated …… entry of the table is located, and its key value is examined.
10. Binary search is also known as …… search.
11. As the effective table is halved on each probe, a maximum of about …… probes is needed to search it.

5.3 Hash Searching

Notes

Hash searching is also termed as Random Entry Searching. Binary search algorithms, being fast, can only function on tables that are ordered and packed, that is, tables that have contiguous items ordered by keywords.

Did u know? These kind of search procedures may have to be utilized in combination with a sort algorithm which orders the data and packs the data as well.

In fact, it is needless for the table to be ordered and packed to attain good speed in searching. As for now, it is likely to perform considerably better with an unpacked, unordered table. The number of storage spaces assigned to it goes beyond the number of items to be accumulated.

Putting elements in order slows down the procedure. A significant enhancement can be attained by inserting elements in a random manner. The random entry-number K is developed from the key. If the Kth position is void, then the new element is placed there; if not, then some other cell must be found for the insertion.

The primary problem is the production of a random number from the key. Certainly we don't in fact need a random number in the sense that a specified keyword may give up one position today and another tomorrow. What we need is a process that will produce pseudo-random, reliable table positions for keywords.

One quite good outlook for four character EBCDIC keywords is to merely divide the keyword by the table length N and use the remainder. This method functions well providing N and the key size (32 bits in our case) have no common factors. For a specified group of M keywords, the remainders should be fairly equally distributed over O"{N-l). Another method is to consider the keyword as a binary fraction and multiply it by another binary fraction:

```
L    1,SYMBOL
M    0,RHO
```

The outcome is a 64-bit product in registers 0 and 1. If RHO is selected cautiously in the low order 31 bits will be equally distributed among 0 to 1, and a second .multiplication by N will produce a number equally distributed over 0···(N–l). This is called the power residue technique.

It has the benefit that the 31 ·bit first result can be used to generate another equally distributed number (by multiplying again by RHO) in the event that the first probe of the table is ineffective.

The second problem is the process to be followed when the first tryout entry results in a filled position. There are a number of techniques of resolving this problem. These are:

1. ***Random Entry with Replacement:*** A series of random numbers is generated from the keyword (like by the power residue method). From each of these a number between 1 and N is formed and the table is probed at that position. Probings are terminated when a void space is found.

Note Observe that the random numbers generated are independent and it is perfectly possible (but not likely) to probe the same position twice.

2. ***Random Entry without Replacement:*** This is the similar as above excluding that any effort to probe the same position two times is avoided. This technique holds benefit over the above only when probes are luxurious, e.g., for files on tape or drum.

Notes

3. ***Open addressing:*** If the first probe provides a position K and that position is filled, then the next location K + 1 is probed, and so on in anticipation of a free position. If the hunt runs off the bottom of the table, then it is transformed at the top. Specifically, the table is considered to be cyclic.

Caution Out of these three techniques, the open addressing scheme is the simplest.

Example: Let us see an example to illustrate this technique.

Consider a table of 17 positions (N= 17) in which the following twelve numbers are to be accumulated: 19. 13, 05, 27, 01, 26, 31, 16, 02, 09, 11.21. These items are to be placed in the table at a position defined by the remainder after division by 17; if that position is filled, then the next position is observed, etc. Figure 5.4 displays the progress of entry for the 12 items. Observe the resolution of disagreemenst on items 02. 09, and 11. The column 'Probes to find' provides the number of probes essential to locate the corresponding items in the table; so it takes 3 probes to locate items 09, and 1 to locate item 26. The column 'Probes to locate not' provides the number of probes needed to find out that an item is not in the table. So, the search for the number S4 would provide an initial position of 3 and it would take 4 probes to locate that the item is not present. The item is recognized not to be present when a void position came across (position 6 in this case). Observe here that the following figures hold'

Length of table	N = 17
Items stored	M = 12
Density	ρ = 12/17 = 0.705
Probes to store	T_s = 16
Average probes to find	T_p = 16/12 = 1.33
Average probes to find not	T_n = 54/16 = 3.31

The proportional times for a packed table, by means of radix exchange sort (explained in the next unit) and binary search are as follows:

Probes to store and sort	$T_s = M+M-\log_2(M) = 55$
Average probe to find	$T_p = \log_2(M) = 3.58$
Average probe to find not	$T_n = \log_2(M) = 3.58$

Therefore, it would emerge that the open addressing plan holds significant benefit in speed, but it pays for this by having a table almost 50 percent longer than required. Moreover, the table cannot be compacted after its initial allotment nor can the allotted area be simply shared between several tables. One final drawback is that it is very complex to delete any item from the table.

Note One cannot just zero out that location since this might break the addressing chain.

Random entry with replacement is considered to be a simpler method to assess.

Notes

Figure 5.4: Example of Open Addressing

Position	Item	Probes to find	Probes to find not
0		1	1
1	01	1	6
2	19,02*	2	5
3	02	1	4
4	21	1	3
5	05		2
6			1
7			1
8			1
9	26.09*	1	7
10	27,09*	1	6
11	09,11*	3	5
12	11	2	4
13	13	1	3
14	31	1	2
15			1
16	16	1	1
		16	54

Task Make distinction between 'Random entry with replacement' and 'Random entry without replacement' techniques.

Self Assessment

Fill in the blanks:

12. Hash searching is also termed as …… Searching.
13. In random entry with replacement technique, the random numbers generated are independent and it is perfectly possible to probe the same position …… .
14. In case of ……, any effort to probe the same position two times is avoided.
15. In case of ……, if the first probe provides a position K and that position is filled, then the next location K + 1 is probed, and so on in anticipation of a free position.

5.4 Summary

- In linear searching, to locate a given item, start your search at the beginning of the data collection and persist to observe until you have either located the goal or exhausted the search space.
- Linear search is as, in the average case, one-half of the items in the search space will be scrutinized before a match is located.
- In binary searching, the entries in the table are stored in alphabetically or numerically increasing order.
- In binary search, search begins by examining the records in the middle of file rather than the one at one of the ends as in sequential search.

Notes

- We call a binary search as logarithmic search also, as when the effective table is halved on each probe, a maximum of about $\log_2(N)$ probes is needed to search it.
- In random entry with replacement technique, the random numbers generated are independent and it is perfectly possible (but not likely) to probe the same position twice.
- In case of random entry without replacement, any effort to probe the same position two times is avoided.
- In case of open addressing, if the first probe provides a position K and that position is filled, then the next location K + 1 is probed, and so on in anticipation of a free position.

5.5 Keywords

Binary Search: In binary searching, the entries in the table are stored in alphabetically or numerically increasing order.

Linear Search: In linear searching, to locate a given item, start your search at the beginning of the data collection and persist to observe until you have either located the goal or exhausted the search space.

Random Entry Searching: These kinds of search procedures may have to be utilized in combination with a sort algorithm which orders the data and packs the data as well.

5.6 Review Questions

1. Illustrate the problem of searching in table processing.
2. Explain the concept of linear search with example.
3. What is binary searching? Illustrate the concept.
4. Give an example of linear search program and illustrate.
5. Give an example of a binary search program. Explain.
6. In binary search, search begins by examining the records in the middle of file rather than the one at one of the ends as in sequential search. Comment.
7. Illustrate the process of hash searching with example.
8. What is open addressing? Illustrate the concept with example.
9. It is needless for the table to be ordered and packed to attain good speed in searching. Comment.
10. Explain the concept of power residue method.

Answers: Self Assessment

1. ordered
2. Linear search
3. sequential
4. entry
5. binary
6. sorted
7. sequential
8. name
9. middle
10. logarithmic
11. $\log_2(N)$
12. Random Entry

13. twice
14. random entry without replacement
15. open addressing

Notes

5.7 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Shi-kuo Chang, *Data Structures and Algorithms,* World Scientific

Online link

www.ethoberon.ethz.ch/WirthPubl/AD.pdf

Notes

Unit 6: Table Processing: Sorting

CONTENTS

Objectives

Introduction

6.1 Sorting

6.1.1 Classification of Sorting

6.2 Interchange Sort

6.3 Shell Sort

6.4 Bucket Sort

6.5 Radix Exchange Sort

6.6 Summary

6.7 Keywords

6.8 Review Questions

6.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concept of sorting
- Discuss interchange sort, shell sort, bucket sort, etc.

Introduction

It appears evidently that binary search is more competent than linear search, but such a search needs an ordered table which may not be simply obtainable. Here you will recognize the interchange sort, shell sort, bucket sort, etc.

6.1 Sorting

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data or alphabetically, with character data.

Let P be a list of n elements P_1, P_2, P_3...P_n in memory. Sorting P refers to the operation of rearranging the contents of P so that they are increasing in order (numerically or lexicographically), that is,

$$P_1 < P_2 < P_3 ... < P_n$$

Since P has n elements, there are n! ways that the contents can appear in P. These ways correspond precisely to the n! permutations of 1, 2, ...n.

Caution Accordingly, each sorting algorithm must take care of these n! possibilities.

Notes

6.1.1 Classification of Sorting

Sorting can be classified in two types:

(i) Internal Sorting

(ii) External Sorting

Internal Sorting: If the records that have to be sorted are in the internal memory.

External Sorting: If the records that have to be sorted are in secondary memory.

Efficiency Considerations

The most important considerations are:

1. The amount of machine time necessary for running the program.
2. The amount of space necessary for the program.

In most of the computer applications one is optimized at the expense of another. The actual time units to sort a file of size n varies from machine to machine, from one program to another, and from one set of data to another.

So, we find out the corresponding change in the amount of time required to sort a file induced by a change in the file size n.

Note The time efficiency is not measured by the number of time units required but rather by the number of critical operations performed.

Critical operations are those that take most of the execution time.

Example: Key comparisons (that is, the comparisons of the key of two records in the file to determine which is greater), movement of records or pointer to record, or interchange of two records.

Task Make distinction between internal sorting and external sorting.

Self Assessment

Fill in the blanks:

1. If the records that have to be sorted are in the internal memory, it is called …… sorting.
2. If the records that have to be sorted are in secondary memory, it is called …… sorting.
3. …… refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data or alphabetically, with character data.

Notes

6.2 Interchange Sort

Here we will recognize how to sort a table by means of interchange sort. There are a many ways of performing this, some simple and some complex. Figure 6.1 is a segment of coding that executes an interchange sort.

Did u know? Interchange sort is also known as a bubble sort, a sinking sort or a sifting sort.

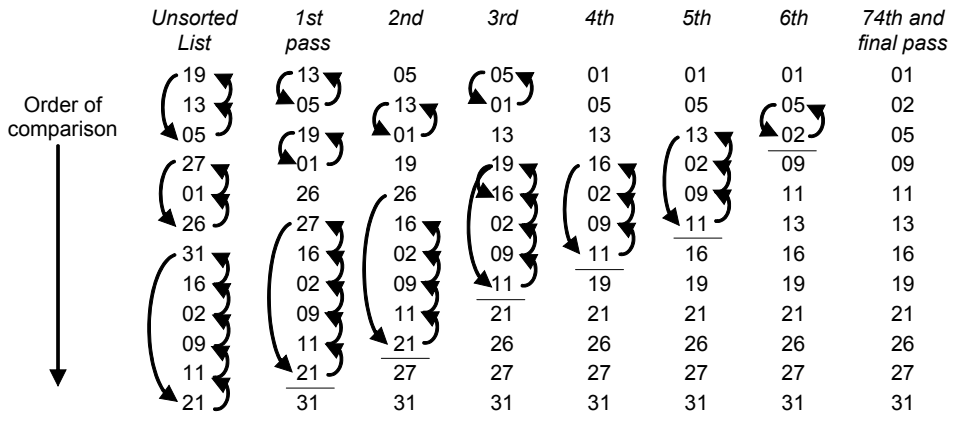
This simple sort considers contiguous pairs of items in the table and places them in order (interchanges them) as necessary. Such a sorting algorithm is not very competent, but it is easy.

Figure 6.1: Example of Interchange Sort in 360 Assembly Code

```
         L      5,LAST
         LA     4,SYMTBL
LOOP     CLC    0(8,4),14(4)    Compare adjacent symbols – 8 bytes
         BNH    OK              Correct order
         MVC    TEMP(14),0(4)   Switch entries
         MVC    0(14,4),14(4)   · · ·
         MVC    14(14,4),TEMP   · ·
OK       A      4,=F'14'        Move to next entry
         C      4,LAST          Is it last entry
         BNE    LOOP            No
          ⋮
SYMTBL   DS     0F              Symbol table
         DS     100CL14         14 bytes per entry
TEMP     DS     CL14            Temporary entry
LAST     DC     A(-----)        Location of next free entry in table
```

Example: Now we show an example to see how it works. Consider the table of 12 numbers displayed in Figure 6.2.

Figure 6.2: Illustration of Interchange Sort



Notes

In the figure 6.2, every column signifies one pass over the numbers interchanging any two contiguous numbers that are out of order. This specific table is totally sorted in only seven passes over the data.

In the most awful case, N-I (here, 11) passes would be essential. The Interchange sort is therefore proficient to exploit whatsoever natural order there may be in the table. Furthermore, on each pass via the data at least one item is added to the base of the list in perfect order (here the 31 first, then, 27, then 26, etc.).

Thus, the sort could be made more competent by:

1. Cutting the portion of the sorted list on each pass; and
2. Verifying for early completion.

Caution Such an optimized sort should need approximately N*(N–l)/2 comparisons and so should take a time nearly proportional to N^2.

We would like an even improved sorting method which needs less time. Sorting methods consists of basic types:

1. Distributive sorts which sort by probing entries one digit at a time;
2. Comparative sorts which sort by contrasting keywords two at a time; and
3. Address calculation sorts that convert a key into an address close to where the symbol is predictable to end up.

Self Assessment

Fill in the blanks:

4. …… sort considers contiguous pairs of items in the table and places them in order (interchanges them) as necessary.
5. …… sorts are sorted by probing entries one digit at a time.
6. …… sorts sorted by contrasting keywords two at a time.
7. …… sorts are sorted by converting a key into an address close to where the symbol is predictable to end up.

6.3 Shell Sort

A rapid comparative sort algorithm is because of D.L. Shell and is known as a Shell sort. It leads to best possible performance for a comparative type of sort. The Shell sort is analogous to the interchange sort as it shifts data items by exchanges. However, it starts by contrasting items a distance "d" apart. This signifies that items that are out of place will be moved more speedily than a simple interchange sort. In each pass the value of d is decreased generally;

$$d_{i+1} = \frac{d_i + 1}{2}$$

Every item is contrasted with the one located d positions further in the vector of items. If the higher item consists of a lower value, an exchange is' performed. The sort prolongs by comparing the next item in the vector with an item d locations away (if one exists). If an exchange is again specified, it is performed and the comparison is attempted again with the next entry. This continues

Notes

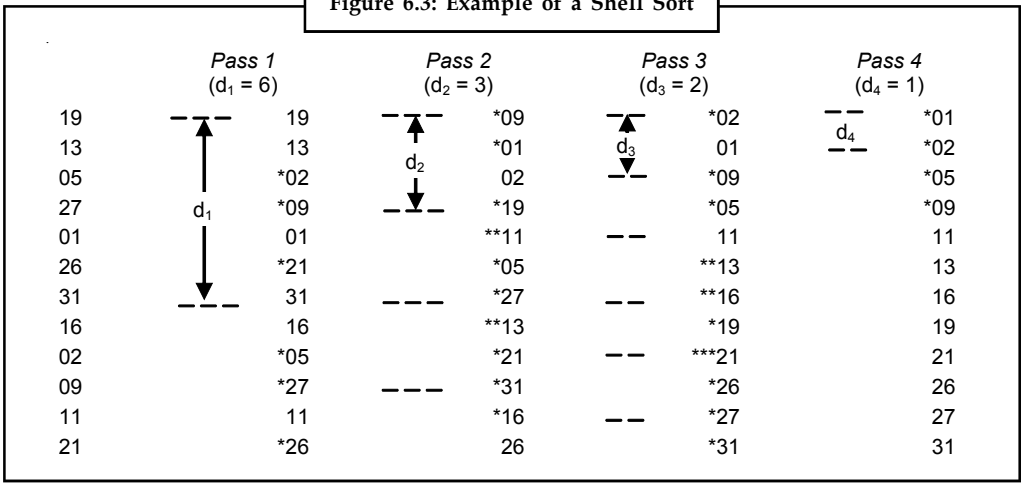
until no lower items remain, This process is known as bubbling; if you think of low valued items floating to the top, the behavior in the subprocess is like that of a bubble water lank.

Did u know? After bubbling can no longer take place with a fixed value of d, the process starts again with a new d.

It is complicated to guess the time requirements of the Shell sort as it is difficult to demonstrate the effect of one pass on another. It should be clear that if the above method of calculating d is utilized, the number of passes will be about $\log_2(d)$ as bubbling when d=t will complete the sort. Empirical studies have exposed that the Shell sort takes about $B*N*(\log_2 N)^2$ units of time for an N element vector. The constant of proportionality B is quite small, thus for small N the Shell sort will outperform the radix exchange sort illustrated later in the unit.

Example: We show an example of a shell sort as below in Figure 6.3.

Figure 6.3: Example of a Shell Sort



* = Exchange; ** = Dual exchange; *** = Triple exchange

Self Assessment

Fill in the blanks:

8. ..... leads to best possible performance for a comparative type of sort.
9. The Shell sort is analogous to the interchange sort as it shifts data items by..... .
10. Shell sort takes about ..... units of time for an N element vector.

6.4 Bucket Sort

A simple distributive sort is known as the radix sort or bucket sort. The sort includes probing the least important digit of the keyword first, and the item is then allocated to a bucket exclusively dependent on the value of the digit. After distributing all items, the "buckets" items arc combined in order and then the process is repeated until no more digits are left. A number system of base P needs P buckets.

Notes

Example: Consider the radix sorting of the numbers as displayed in Figure 6.4. You should be able to discover rather quickly the functioning of this sort. Actually, this is exactly the method utilized on a card sorting machine.

Figure 6.4: Demonstration of Radix Sorting

Original table	*First distribution*	*Merge*	*Second distribution*	*Final merge*
19		01		01
13	0)	31	0) 01,02,05,09	02
05	1) 01,31,11,21	11	1) 11,13,16,19	05
27	2) 02	21	2) 21,26,27	09
01	3) 13	02	3) 31	11
26	4)	13	4)	13
31	5) 05	05	5)	16
16	6) 26,16	26	6)	19
02	7) 27	16	7)	21
09	8)	27	8)	26
11	9) 19,09	19	9)	27
21		09		31
↑		↑		
Separate, based on last digit		Separate, based on first digit		

However, there are some drawbacks to using it internally on a digital computer

1. It takes two different processes, a separation and a merge; and
2. It needs a lot of extra storage for the buckets. However, this last drawback can be conquered by chaining records inside a logical "bucket" instead of pre-allocating maximum size buckets.

Self Assessment

Fill in the blanks:

11. Bucket sort is also known as …… sort.
12. A number system of base P needs P …… .

6.5 Radix Exchange Sort

A significantly better distributive sort is the radix exchange sort which is valid when the keys are expressed, (or are expressible) in binary. Sorting is achieved by considering groups with the similar (M) first bits and ordering that group regarding the (M+ l)st bit. The ordering of a group on a specified bit is achieved by scanning down from the top of the group for a one bit and up from the bottom for a zero bit; these two are exchanged and the sort goes on.

This algorithm needs the program to continue with a huge number of groups and, coded in a bad form, could need an additional table N long. However, with optimal coding it is probable to follow the groups by just observing the top of the table and a list of break points, one for each bit of the key- word. (Therefore with 32 bit words a table of only 33 entries is needed.)

Notes

If the sort algorithm is programmed to quit sorting when a group includes only one item, then the time needed for the radix exchange sort is proportional to N*log(N) as compared to $N*\log_p(K)$ for the bucket sort (here K is the maximum key size and p is the radix).

Note The radix exchange sort does not need additional table space for "buckets."

Task Make distinction between radix sort and radix exchange sort.

Self Assessment

Fill in the blanks:

13. A …… sort is valid when the keys are expressed, (or are expressible) in binary.
14. Sorting is achieved by considering groups with the similar (M) first bits and ordering that group regarding the …… bit.
15. If the sort algorithm is programmed to quit sorting when a group includes only one item, then the time needed for the radix exchange sort is proportional to N*log(N) as compared to …… for the bucket sort.

6.6 Summary

- Efficient and reliable data processing depends upon sorted data.
- The internal and external sorting methods have their relative efficiencies in different applications.
- Interchange sort is a simple sort that considers contiguous pairs of items in the table and places them in order (interchanges them) as necessary.
- Shell sort leads to best possible performance for a comparative type of sort.
- The Shell sort is analogous to the interchange sort as it shifts data items by exchanges.
- Bucket sort includes probing the least important digit of the keyword first, and the item is then allocated to a bucket exclusively dependent on the value of the digit.
- A significantly better distributive sort is the radix exchange sort which is valid when the keys are expressed, (or are expressible) in binary.
- The radix exchange sort does not need additional table space for "buckets."

6.7 Keywords

Bucket Sort: Bucket sort includes probing the least important digit of the keyword first, and the item is then allocated to a bucket exclusively dependent on the value of the digit.

Interchange Sort: Interchange sort is a simple sort that considers contiguous pairs of items in the table and places them in order (interchanges them) as necessary.

Radix Exchange Sort: A significantly better distributive sort is the radix exchange sort which is valid when the keys are expressed, (or are expressible) in binary.

Notes

Shell Sort: The Shell sort is analogous to the interchange sort as it shifts data items by exchanges.

Sorting: Sorting is the operation of arranging data in some given order such as increasing or decreasing, with numerical data or alphabetically, with character data.

6.8 Review Questions

1. What is sorting? Discuss the need for sorting.
2. Discuss the significant considerations followed in sorting.
3. How shell sort is comparable to the interchange sort? Illustrate.
4. Illustrate how the elements are reorganized in case of interchange sort.
5. Why is radix exchange sort considered as an improved distributed sort?
6. What steps are required to make interchange sort more proficient? Discuss.
7. Illustrate the process of shell sort with example.
8. Demonstrate the concept of bucket sort with example.
9. The radix exchange sort does not need additional table space for "buckets". Comment.
10. Illustrate the drawbacks that appear on using bucket sort internally on a digital computer.

Answers: Self Assessment

1. internal
2. external
3. Sorting
4. Interchange
5. Distributive
6. Comparative
7. Address calculation
8. Shell sort
9. exchanges
10. $B*N*(\log_2 N)^2$
11. Radix
12. buckets
13. radix exchange
14. (M+ l)st
15. $N*\log_p(K)$

6.9 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Shi-kuo Chang, *Data Structures and Algorithms,* World Scientific

Online link

www.ethoberon.ethz.ch/WirthPubl/AD.pdf

Notes

Unit 7: Macro Language

CONTENTS

Objectives

Introduction

7.1 Macro Instructions

7.1.1 Macro Processor

7.1.2 Format of Macro Definition

7.1.3 Basic Macro Processor Functions

7.1.4 Macro Processor Algorithm and Data Structures

7.2 Machine Independent Macro Processor Features

7.2.1 Concatenation of Macro Parameter

7.2.2 Generation of Unique Labels

7.2.3 Conditional Macro Expansion

7.2.4 Keyword Macro Parameters

7.3 Implementation

7.3.1 Implementation of Conditional Macro Expansion (IF-ELSE-ENDIF Structure)

7.3.2 Implementation of Conditional Macro Expansion (WHILE ENDW Structure)

7.4 Summary

7.5 Keywords

7.6 Review Questions

7.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concept of macro instructions
- Discuss features of macro processors
- Understand the implementation of macro processors

Introduction

In this unit you will understand the concept of macro instructions which has been replaced by macro processor. You will discuss macro processor functions, algorithms, and data structures. Also you will study various features and implementations of macro processor with their respective examples.

7.1 Macro Instructions

Notes

A macro instruction (macro) is just a notational expediency for the programmer to write a shorthand edition of a program. It symbolizes a usually used group of statements in the source program. It is substituted by the macro processor with the equivalent group of source language statements.

Did u know? The above explained operation is known as "expanding the macro".

Example: Suppose it is essential to save the contents of all registers before calling a subroutine. This needs a series of instructions. We can define and utilize a macro, SAVEREGS, to signify this sequence of instructions.

7.1.1 Macro Processor

A macro processor functions fundamentally engross the replacement of one group of characters or lines for another. Usually, it performs no analysis of the text it manages. It doesn't regard the meaning of the involved statements throughout macro expansion. Thus, the design of a macro processor usually is machine independent.

Notes Macro processors are used in assembly language, high-level programming languages, e.g., C or C++, OS command languages, and universal purpose.

7.1.2 Format of Macro Definition

A macro can be defined as follows MACRO - MACRO pseudo-op displays beginning of macro definition. Name [List of Parameters] - Macro name with a list of formal parameters.

…….

……

…….. Sequence of assembly language instructions.

MEND- MEND (MACRO-END) Pseudo displays the end of macro definition.

Example:

```
MACRO
        SUM X,Y
        LDA X
        MOV BX,X
        LDA Y
        ADD BX
        MEND
```

Notes

7.1.3 Basic Macro Processor Functions

The fundamental functions familiar to all macro processors are:

1. Macro Definition
2. Macro Invocation
3. Macro Expansion

Macro Definition and Expansion

- Two new assembler directives are utilized in macro definition:
  - MACRO: identify the beginning of a macro definition
  - MEND: identify the end of a macro definition
- Prototype for the macro:
  - Each parameter begins with '&

```
'label    op          operands
  name  MACRO       parameters
  :
  Body
  :
  MEND
```

- ***Body:*** The statements that will be produced as the expansion of the macro.

```
COPY      START   0               COPY FILE FROM INPUT TO OUTPUT
RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
.
.         MACRO TO READ RECORD INTO BUFFER
.
          CLEAR   X               CLEAR LOOP COUNTER
          CLEAR   A
          CLEAR   S
          +LDT    #4096           SET MAXIMUM RECORD LENGTH
          TD      =X'&INDEV'      TEST INPUT DEVICE
          JEQ     *-3             LOOP UNTIL READY
          RD      =X'&INDEV'      READ CHARACTER INTO REG A
          COMPR   A,S             TEST FOR END OF RECORD
          JEQ     *+11            EXIT LOOP IF EOR
          STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
          TIXR    T               LOOP UNLESS MAXIMUM LENGTH
```

Contd...

Notes

```
        JLT     *-19            HAS BEEN REACHED
        STX     &RECLTH         SAVE RECORD LENGTH
        MEND
WRBUFF  MACRO   &OUTDEV,&BUFADR,&RECLTH
.
.       MACRO TO WRITE RECORD FROM BUFFER
.
        CLEAR   X               CLEAR LOOP COUNTER
        LDT     &RECLTH
        LDCH    &BUFADR,X       GET CHARACTER FROM BUFFER
        TD      =X'&OUTDEV'     TEST OUTPUT DEVICE
        JEQ     *-3             LOOP UNTIL READY
        WD      =X'&OUTDEV'     WRITE CHARACTER
        TIXR    T               LOOP UNTIL ALL CHARACTERS
        JLT     *-14              HAVE BEEN WRITTEN
        MEND
```

- It displays an example of a SIC/XE program using macro Instructions.
- This program defines and utilizes two macro instructions, RDBUFF and WRDUFF.
- The functions and logic of RDBUFF macro are alike to those of the RDBUFF subroutine.
- The WRBUFF macro is alike to WRREC subroutine.
- Two Assembler directives (MACRO and MEND) are utilized in macro definitions.
- The first MACRO statement determines the start of macro definition.
- The Symbol in the label field (RDBUFF) is the name of macro, and entries in the operand field determine the parameters of macro instruction.
- In our macro language, every parameter starts with character &, which facilitates the substitution of parameters during macro expansion.
- The macro name and parameters define the outline or prototype for the macroinstruction utilized by the programmer. The macro instruction definition has been removed since they have been no longer needed after macros are expanded.
- Every macro invocation statement has been extended into the statements that form the body of the macro, with the arguments from macro invocation substituted for the parameters in macro prototype.

Caution The arguments and parameters are connected with one another according to their positions.

Notes

Task What does macro name and parameters define?

Macro Invocation

- A macro invocation statement (a macro call) provides the name of the macroinstruction being summoned and the arguments in mounting the macro.
- The processes of macro invocation and subroutine call are pretty different.
  - Statements of the macro body are expanded every time the macro is invoked.
  - Statements of the subroutine emerge only one; regardless of how many times the subroutine is called.

Notes The macro invocation statements considered as comments and the statements produced from macro expansion will be assembled as although they had been written by the programmer.

```
.       MAIN PROGRAM
.
FIRST   STL     RETADR          SAVE RETURN ADDRESS
CLOOP   RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
        LDA     LENGTH          TEST FOR END OF FILE
        COMP    #0
        JEQ     ENDFIL          EXIT IF EOF FOUND
        WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
        J       CLOOP           LOOP
ENDFIL  WRBUFF  05,EOF,THREE    INSERT EOF MARKER
        J       @RETADR
EOF     BYTE    C'EOF'
THREE   WORD    3
RETADR  RESW    1
LENGTH  RESW    1               LENGTH OF RECORD
BUFFER  RESB    4096            4096-BYTE BUFFER AREA
        END     FIRST
```

Macro Expansion

- Each macro invocation statement will be expanded into the statements that produces the body of the macro.

Notes

- Arguments from the macro invocation are replaced for the parameters in the macro prototype.
  - The arguments and parameters are related with one another according to their positions.
    - The first argument in the macro invocation communicates to the first parameter in the macro prototype, etc.
- Comment lines inside the macro body have been deleted, but comments on individual statements have been preserved.

Did u know? Macro invocation statement itself has been incorporated as a comment line.

Example of a macro expansion

```
COPY     START   0                   COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR              SAVE RETURN ADDRESS
.CLOOP   RDBUFF  F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
CLOOP    CLEAR   X                   CLEAR LOOP COUNTER
         CLEAR   A
         CLEAR   S
        +LDT     #4096               SET MAXIMUM RECORD LENGTH
         TD      =X'F1'              TEST INPUT DEVICE
         JEQ      *-3                LOOP UNTIL READY
         RD      =X'F1'              READ CHARACTER INTO REG A
         COMPR    A,S                TEST FOR END OF RECORD
         JEQ     *+11                EXIT LOOP IF EOR
         STCH    BUFFER,X            STORE CHARACTER IN BUFFER
         TIXR    T                   LOOP UNLESS MAXIMUM LENGTH
         JLT     *-19                  HAS BEEN REACHED
         STX     LENGTH              SAVE RECORD LENGTH
```

- In expanding the macro invocation on line 190, the argument F1 is replaced for the parameter and INDEV wherever it appears in the body of the macro.
- Likewise BUFFER is substituted for BUFADR and LENGTH is substituted for RECLTH.
- Lines 190a via 190m show the complete expansion of the macro invocation on line 190.
- The label on the macro invocation statement CLOOP has been preserved as a label on the first statement generated in the macro expansion.
- This permits the programmer to use a macro instruction in exactly the same manner as an assembler language mnemonic.
- After macro processing the expanded file can be used as input to assembler.
- The macro invocation statement will be considered as comments and the statements produced from the macro expansions will be assembled precisely as though they had been written directly by the programmer.

Notes

7.1.4 Macro Processor Algorithm and Data Structures

- It is simple to design a two-pass macro processor in which all macro definitions are accessed throughout the first pass, and all macro invocation statements are extended during second pass.
- Such a two pass macro processor would not permit the body of one macroinstruction to enclose definitions of other macros.

Example:

```
1  MACROS   MACRO     {Defines SIC standard version macros}
2  RDBUFF   MACRO     &INDEV,&BUFADR,&RECLTH
                .
                .     {SIC  standard version}
                .
3                MEND      {End of RDBUFF}
4  WRBUFF   MACRO     &OUTDEV,&BUFADR,&RECLTH
                .
                .     {SIC standard version}
                .
5                MEND      {End of WRBUFF}
                .
                .
                .
6                MEND      {End of MACROS}
```

Example:

```
1  MACROS   MACRO     {Defines SIC standard version macros}
2  RDBUFF   MACRO     &INDEV,&BUFADR,&RECLTH
                .
                .     {SIC  standard version}
                .
3                MEND      {End of RDBUFF}
4  WRBUFF   MACRO     &OUTDEV,&BUFADR,&RECLTH
                .
                .     {SIC standard version}
                .
5                MEND      {End of WRBUFF}
                .
                .
                .
6                MEND      {End of MACROS}
```

Notes

- Defining MACROS or MACROX does not define RDBUFF and the other macroinstructions. These definitions are developed only when an invocation of MACROS or MACROX is expanded.
- A one pass microprocessor that can swap between macro definition and macro expansion is able to manage macros like these.
- There are three main data structures included in our macro processor.

Definition Table (DEFTAB)

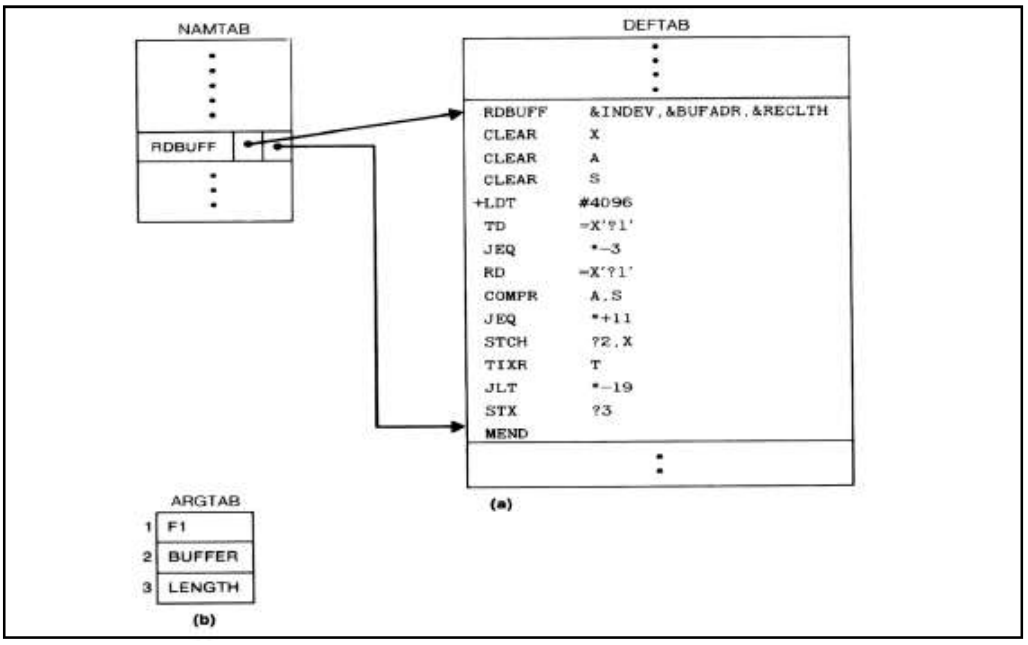
1. The macro definition themselves are accumulated in definition table (DEFTAB), which comprises the macro prototype and statements that make up the macro body.
2. Comment lines from macro definition are not entered into DEFTAB since they will not be a part of macro expansion.

Name Table (NAMTAB)

1. References to macro instruction parameters are transformed to a positional entered into NAMTAB, which provides the index to DEFTAB.
2. For every macro instruction defined, NAMTAB comprises pointers to beginning an end of definition in DEFTAB.

Argument Table (ARGTAB)

1. The third Data Structure in an argument table (ARGTAB), which is utilized throughout expansion of macro invocations.
2. When macro invocation statements are predictable, the arguments are amassed in ARGTAB according to their position in argument list.
3. As the macro is expanded, arguments from ARGTAB are replaced for the corresponding parameters in the macro body.



Notes

- The position notation is utilized for the parameters. The parameter &INDEV has been transformed to ?1, &BUFADR has been converted to ?2.
- When the ?n notation is renowned in a line from DEFTAB, a simple indexing operation provides the property argument from ARGTAB.

Algorithm:

- The procedure DEFINE, which is called when the start of a macro definition is recognized, makes the suitable entries in DEFTAB and NAMTAB.
- EXPAND is called to set up the argument values in ARGTAB and expand macro invocation statement.
- The procedure GETLINE gets the next line to be processed.
- This line may occur from DEFTAB or from the input file, based upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

```
begin {macro processor}
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end {while}
end {macro processor}


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end {PROCESSLINE}

procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition {prototype} from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
```

Contd...

Notes

```
        begin
            GETLINE
            PROCESSLINE
        end {while}
    EXPANDING := FALSE
end {EXPAND}


procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notat
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

Self Assessment

Fill in the blanks:

1. A ..... functions fundamentally engross the replacement of one group of characters or lines for another.
2. The design of a macro processor usually is ..... independent.
3. The first MACRO statement determines the start of .....
4. A ..... statement (a macro call) provides the name of the macroinstruction being summoned and the arguments in mounting the macro.
5. Each macro invocation statement will be expanded into the statements that produce the ..... of the macro.
6. Arguments from the macro invocation are replaced for the ..... in the macro prototype.
7. It is simple to design a ..... macro processor in which all macro definitions are accessed throughout the first pass, and all macro invocation statements are extended during second pass.
8. References to macro instruction parameters are transformed to a positional entered into ....., which provides the index to DEFTAB.

7.2 Machine Independent Macro Processor Features

Machine independent macro processor features are comprehensive traits that are not directly connected to architecture of computer for which the macro processor is written.

Notes

7.2.1 Concatenation of Macro Parameter

- Most Macro Processor permits parameters to be concatenated with other character strings.
- A program comprises a set of series of variables:

  XA1, XA2, XA3,…

  XB1, XB2, XB3,…
- If similar processing is to be performed on each series of variables, the programmer might want to fit in this processing into a macro instruction.
- The parameter to such a macro instruction could specify the series of variables to be functioned on (A, B, C …).
- The macro processor builds the symbols by concatenating X, (A, B, …), and (1,2,3,…) in the macro expansion.
- Let such parameter is named &ID, the macro body may comprise a statement: LDA X&ID1, in which &ID is concatenated after the string "X" and before the string "1".
  - LDA XA1 (&ID=A)
  - LDA XB1 (&ID=B)
- Ambiguity problem: E.g., X&ID1 may mean "X" + &ID + "1""X" + &ID1. This problem appears since the end of the parameter is not marked.

Macro Definition

```
1  SUM     MACRO    &ID
2          LDA      X&ID→1
3          ADD      X&ID→2
4          ADD      X&ID→3
5          STA      X&ID→S
6          MEND
```

Macro Invocation Statements

```
SUM        A
  ↓
LDA        XA1
ADD        XA2
ADD        XA3
STA        XAS
```

Notes

- The macroprocessor removes all incidences of the concatenation operator immediately after carrying out parameter substitution, so the character 'will not appear in the macro expansion.

7.2.2 Generation of Unique Labels

- Labels in the macro body may reason "duplicate labels" problem if the macro is invocated and expanded multiple times.
- Utilization of relative addressing at the source statement level is very inconvenient, error-prone, and hard to read.
- It is extremely attractive to let the programmer use label in the macro body.
- Labels utilized within the macro body start with $.2. Let the macro processor produce unique labels for every macro invocation and expansion.
- Throughout macro expansion, the $ will be substituted with $xx, where xx is a two-character alphanumeric counter of the number of macroinstructions extended.
- XX=AA, AB, AC …….

  Consider the definition of WRBUFF

```
COPY        START       0
TD                      =X'&OUTDEV'
JEQ                     *-3
JLT                     *-14
END                     FIRST
```

- If a label was positioned on the TD instruction on line 135, this label would be defined twice, once for every invocation of WRBUFF.
- This duplicate definition would avert correct assembly of the resulting expanded program.
- The jump instructions on line 140 and 155 are written by means of the relative operands *-3 and *-14, since it is not possible to position a label on line 135 of the macro definition.
- This relative addressing may be suitable for short jumps like " JEQ *-3"
- For longer jumps spanning several instructions, such notation is very inopportune, error-prone and difficult to read.
- Many microprocessors avoid these problems by permitting the creation of special types of labels inside macro instructions.

Notes

RDBUFF definition

```
RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH
         CLEAR   X            CLEAR LOOP COUNTER
         CLEAR   A
         CLEAR   S
        +LDT     #4096        SET MAXIMUM RECORD LENGTH
$LOOP    TD      =X'&INDEV'   TEST INPUT DEVICE
         JEQ     $LOOP        LOOP UNTIL READY
         RD      =X'&INDEV'   READ CHARACTER INTO REG A
         COMPR   A,S          TEST FOR END OF RECORD
         JEQ     $EXIT        EXIT LOOP IF EOR
         STCH    &BUFADR,X    STORE CHARACTER IN BUFFER
         TIXR    T            LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP          HAS BEEN REACHED
$EXIT    STX     &RECLTH      SAVE RECORD LENGTH
         MEND
```

- Labels inside the macro body start with the special character $.

Macro Expansion

```
.        RDBUFF  F1,BUFFER,LENGTH


         CLEAR   X            CLEAR LOOP COUNTER
         CLEAR   A
         CLEAR   S
        +LDT     #4096        SET MAXIMUM RECORD LENGTH
$AALOOP  TD      =X'F1'       TEST INPUT DEVICE
         JEQ     $AALOOP      LOOP UNTIL READY
         RD      =X'F1'       READ CHARACTER INTO REG A
         COMPR   A,S          TEST FOR END OF RECORD
         JEQ     $AAEXIT      EXIT LOOP IF EOR
         STCH    BUFFER,X     STORE CHARACTER IN BUFFER
         TIXR    T            LOOP UNLESS MAXIMUM LENGTH
         JLT     $AALOOP        HAS BEEN REACHED
$AAEXIT  STX     LENGTH       SAVE RECORD LENGTH
```

- Unique labels are produced within macro expansion.
- Each symbol starting with $ has been modified by replacing $ with $AA.
- The character $ will be substituted by $xx, where xx is a two-character alphanumeric counter of the number of macro instructions extended.

Notes

- For the first macro expansion in a program, xx will have the value AA. For subsequent macro expansions, xx will be set to AB, AC, etc.

7.2.3 Conditional Macro Expansion

- Arguments in macro invocation can be utilized to:
  - Replace the parameters in the macro body without varying the sequence of statements expanded.
  - Amend the sequence of statements for conditional macro expansion (or conditional assembly when related to assembler).

  This capability adds greatly to the power and flexibility of a macro language.

Example: Consider the example

```
RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
         IF      (&EOR NE '')
&EORCK   SET     1
         ENDIF
         CLEAR   X               CLEAR LOOP COUNTER
         CLEAR   A
         IF      (&EORCK EQ 1)
         LDCH    =X'&EOR'        SET EOR CHARACTER
         RMO     A,S
         ENDIF
         IF      (&MAXLTH EQ '')
        +LDT     #4096           SET MAX LENGTH = 4096
         ELSE
        +LDT     #&MAXLTH        SET MAXIMUM RECORD LENGTH
         ENDIF
$LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
         JEQ     $LOOP           LOOP UNTIL READY
         RD      =X'&INDEV'      READ CHARACTER INTO REG A
         IF      (&EORCK EQ 1)
         COMPR   A,S             TEST FOR END OF RECORD
         JEQ     $EXIT           EXIT LOOP IF EOR
         ENDIF
         STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
         TIXR    T               LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP             HAS BEEN REACHED
$EXIT    STX     &RECLTH         SAVE RECORD LENGTH
         MEND
```

- Two additional parameters utilized in the example of conditional macro expansion:
  - &EOR: signifies a hexadecimal character code that marks the end of a record.
  - &MAXLTH: signifies the maximum length of a record.

Notes

- Macro-time variable (SET symbol) can be used to
  - accumulate working values during the macro expansion
  - amass the evaluation result of Boolean expression
  - manage the macro-time conditional structures
  - starts with "&" and that is not a macro instruction parameter
  - be initialized to a value of 0
  - be set by a macro processor directive, SET
- Macro-time conditional structure
  - IF-ELSE-ENDIF
  - WHILE-ENDW

7.2.4 Keyword Macro Parameters

- Positional parameters
  - Parameters and arguments are connected as per their positions in the macro prototype and invocation.

Caution The programmer must state the arguments in appropriate order.

- If an argument is to be misplaced, a null argument should be used to preserve the proper order in macro invocation statement.

Example: Presume a macro instruction GENER has 10 possible parameters, but in a specific invocation of the macro only the 3rd and 9th parameters are to be mentioned.

  - The statement is GENER , DIRECT,,,,,,3.
  - It is not appropriate if a macro has a huge number of parameters, and only anew of these are agreed values in a classic invocation.
- Keyword parameters
  - Every argument value is written with a keyword that names the corresponding parameter.
  - Arguments may emerge in any order.
  - Null arguments no longer need to be used.
  - If the 3rd parameter is named &TYPE and 9th parameter is named &CHANNEL, the macro invocation would be GENER TYPE=DIRECT,CHANNEL=3.
  - It is simpler to read and much less error-prone than the positional method.

Consider the Example

- Here every parameter name is followed by equal sign, which determines a keyword parameter and a default value is mentioned for some of the parameters.

Notes

```
RDBUFF   MACRO   &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
         IF      (&EOR NE '')
&EORCK   SET     1
         ENDIF
         CLEAR   X             CLEAR LOOP COUNTER
         CLEAR   A
         IF      (&EORCK EQ 1)
         LDCH    =X'&EOR'      SET EOR CHARACTER
         RMO     A,S
         ENDIF
        +LDT     #&MAXLTH      SET MAXIMUM RECORD LENGTH
$LOOP    TD      =X'&INDEV'    TEST INPUT DEVICE
         JEQ     $LOOP         LOOP UNTIL READY
         RD      =X'&INDEV'    READ CHARACTER INTO REG A
         IF      (&EORCK EQ 1)
         COMPR   A,S           TEST FOR END OF RECORD
         JEQ     $EXIT         EXIT LOOP IF EOR
         ENDIF
         STCH    &BUFADR,X     STORE CHARACTER IN BUFFER
         TIXR    T             LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP           HAS BEEN REACHED
$EXIT    STX     &RECLTH       SAVE RECORD LENGTH
         MEND

.        RDBUFF  BUFADR=BUFFER,RECLTH=LENGTH


         CLEAR   X             CLEAR LOOP COUNTER
         CLEAR   A
         LDCH    =X'04'        SET EOR CHARACTER
         RMO     A,S
        +LDT     #4096         SET MAXIMUM RECORD LENGTH
$AALOOP  TD      =X'F1'        TEST INPUT DEVICE
         JEQ     $AALOOP       LOOP UNTIL READY
         RD      =X'F1'        READ CHARACTER INTO REG A
         COMPR   A,S           TEST FOR END OF RECORD
         JEQ     $AAEXIT       EXIT LOOP IF EOR
         STCH    BUFFER,X      STORE CHARACTER IN BUFFER
         TIXR    T             LOOP UNLESS MAXIMUM LENGTH
         JLT     $AALOOP         HAS BEEN REACHED
$AAEXIT  STX     LENGTH        SAVE RECORD LENGTH
```

Here the value if &INDEV is stated as F3 and the value of &EOR is stated as null.

Task Make distinction between Positional parameters and Keyword parameters.

Notes

Self Assessment

Fill in the blanks:

9. Most Macro Processor permits parameters to be concatenated with other ..... strings.
10. ..... in the macro body may reason "duplicate labels" problem if the macro is invocated and expanded multiple times.
11. Macro-time variable (SET symbol) starts with "..... " and that is not a macro instruction parameter.
12. Parameters and arguments are connected as per their ..... in the macro prototype and invocation.
13. Every ..... value is written with a keyword that names the corresponding parameter.

7.3 Implementation

7.3.1 Implementation of Conditional Macro Expansion (IF-ELSE-ENDIF Structure)

- A symbol table is sustained by the microprocessor.
  - This table comprises the values of all macro-time variables used.
  - Entries in this table are made or customized when SET statements are processed.
  - This table is utilized to look up the current value of a macro-time variable whenever it is needed.
- The testing of the condition and looping are completed while the macro is being expanded.
- When an IF statement is encountered throughout the expansion of a macro, the specified Boolean expression is evaluated. If value is
  - TRUE
    - The macro processor persists to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
    - If ELSE is encountered, then skips to ENDIF.
  - FALSE
    - The macro processor omits ahead in DEFTAB until it finds the next ELSE or ENDLF statement.

7.3.2 Implementation of Conditional Macro Expansion (WHILE ENDW Structure)

- When an WHILE statement is encountered throughout the expansion of a macro, the particular Boolean expression is assessed. If value is
  - TRUE
    - The macro processor persists to process lines from DEFTAB until it encounters the subsequent ENDW statement.

Notes

- ♦ When ENDW is encountered, the macro processor returns to the previous WHILE, reevaluates the Boolean expression, and takes action again.

❖ FALSE

- ♦ The macro processor skips ahead in DEFTAB until it locates the next ENDW statement and then recommences normal macro development.

Self Assessment

Fill in the blanks:

14. When an IF statement is encountered throughout the expansion of a macro, the specified ..... is evaluated.

15. When ENDW is encountered, the macro processor returns to the previous ....., reevaluates the Boolean expression, and takes action again.

7.4 Summary

- A macro instruction (macro) is just a notational expediency for the programmer to write a shorthand edition of a program.
- A macro processor functions fundamentally engross the replacement of one group of characters or lines for another.
- The first MACRO statement determines the start of macro definition.
- The macro name and parameters define the outline or prototype for the macroinstruction utilized by the programmer.
- A macro invocation statement (a macro call) provides the name of the macroinstruction being summoned and the arguments in mounting the macro.
- Each macro invocation statement will be expanded into the statements that produces the body of the macro.
- Most Macro Processor permits parameters to be concatenated with other character strings.
- Labels in the macro body may reason "duplicate labels" problem if the macro is invocated and expanded multiple times.
- Parameters and arguments are connected as per their positions in the macro prototype and invocation.

7.5 Keywords

Labels: Labels in the macro body may reason "duplicate labels" problem if the macro is invocated and expanded multiple times.

Macro Expansion: Each macro invocation statement will be expanded into the statements that produces the body of the macro.

Macro Instruction: A macro instruction (macro) is just a notational expediency for the programmer to write a shorthand edition of a program.

Macro Invocation: A macro invocation statement (a macro call) provides the name of the macroinstruction being summoned and the arguments in mounting the macro.

Macro Processor: A macro processor functions fundamentally engross the replacement of one group of characters or lines for another.

Notes

Answers: Self Assessment

1.	macro processor	2.	machine
3.	macro definition	4.	macro invocation
5.	body	6.	parameters
7.	two-pass	8.	NAMTAB
9.	character	10.	Labels
11.	&	12.	positions
13.	argument	14.	Boolean expression
15.	WHILE		

7.6 Review Questions

1. What is macro instruction? Illustrate with example.
2. What is macro processor? Also discuss the forma of macro definition.
3. Explain various functions of micro processors.
4. Explain the concept of macro definition with example.
5. Describe the process of macro invocation with example.
6. What is Macro Expansion? Illustrate with the example of Macro Expansion.
7. What are the main data structures included in our macro processor? Illustrate.
8. Explain various features of macro processors with their respective examples.
9. What are different Keyword Macro Parameters? Discuss with examples.
10. Explain the concept of Implementation of Conditional Macro Expansion (WHILE ENDW Structure.

7.7 Further Readings

Books

Chattopadhyay, *System Software*, PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online link

www.classle.net/book/macros-and-system-software

Unit 8: Introduction to Linking and Loading

Notes

CONTENTS

Objectives

Introduction

8.1 Loader Schemes

8.1.1 "Compile and Go" Loader

8.1.2 General Loader Scheme

8.1.3 Absolute Loaders

8.1.4 Subroutine Linkage

8.1.5 Relocating Loaders

8.1.6 Direct Linking Loaders

8.2 Other Loader Schemes

8.2.1 Binders

8.2.2 Linkers

8.2.3 Loaders

8.2.4 Overlay Structures and Dynamic Loading

8.2.5 Dynamic Binders

8.3 Summary

8.4 Keywords

8.5 Review Questions

8.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concepts of linking and loading
- Study various loader schemes such as compile and go loaders, general loader scheme, etc.

Introduction

Loader is defined as a utility program which considers object code as input organizes it for implementation and loads the executable code into the memory. Thus, loader, is in fact accountable for initiating the implementation process.

The loader is accountable for the activities like allocation, linking, relocation and loading:

1. It assigns the space for program in the memory, by computing the size of the program. This activity is known as allocation.
2. It solves the symbolic references (code/data) among the object modules by allocating all the user subroutine and library subroutine addresses. This activity is known as linking.

Notes

3. There are some address reliant positions in the program, such address constants must be accustomed as per to allocated space, such activity performed by loader is known as relocation.
4. Lastly it positions all the machine instructions and data of analogous programs and subroutines into the memory. Thus, program now turns out to be prepared for execution; this activity is known as loading.

8.1 Loader Schemes

Depending on the different functionalities of loader, there are different types of loaders:

8.1.1 "Compile and Go" Loader

In "Compile and go" loader , the instruction is read line by line, its machine code is attained and it is directly placed in the major memory at some recognized address. That means the assembler executes in one part of memory and the assembled machine instructions and data is directly placed into their allocated memory locations. After finishing of assembly process, allocate starting address of the program to the location counter.

Example: The usual example is WATFOR-77, it's a FORTRAN compiler which utilizes such "load and go" scheme.

Did u know? This loading scheme is also known as "assemble and go".

Advantage

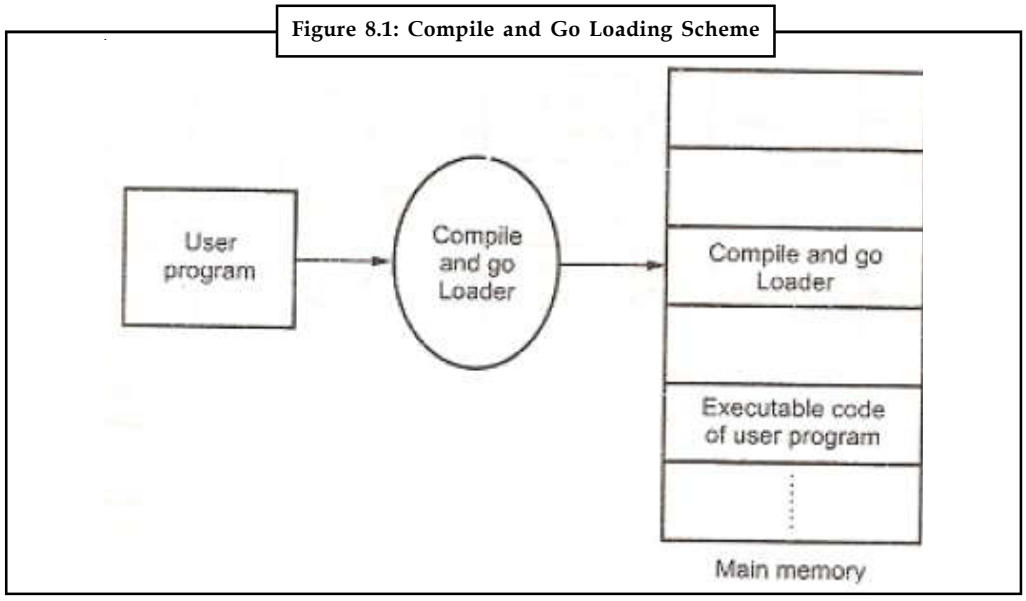
- This scheme is easy to execute. Since assembler is positioned at one part of the memory and loader just loads assembled machine instructions into the memory.

Disadvantages

- In this scheme some part of memory is engaged by assembler which is just a wastage of memory. As this scheme is mixture of assembler and loader activities, this combination program contains huge block of memory.
- There is no production of .obj file, the source code is directly transformed to implementable form. Thus although there is no modification in the source program it requires to be assembled and executed every time, which then turns out to be a time consuming activity.
- It cannot manage multiple source programs or multiple programs written in dissimilar languages. This is because assembler can transform one source language to other goal language.
- For a programmer it is very hard to make an arranged modulator program and also it turns out to be difficult to preserve such program, and the "compile and go" loader cannot manage such programs.
- The implementation time will be more in this scheme as each time program is assembled and then implemented.

Notes

Figure 8.1: Compile and Go Loading Scheme



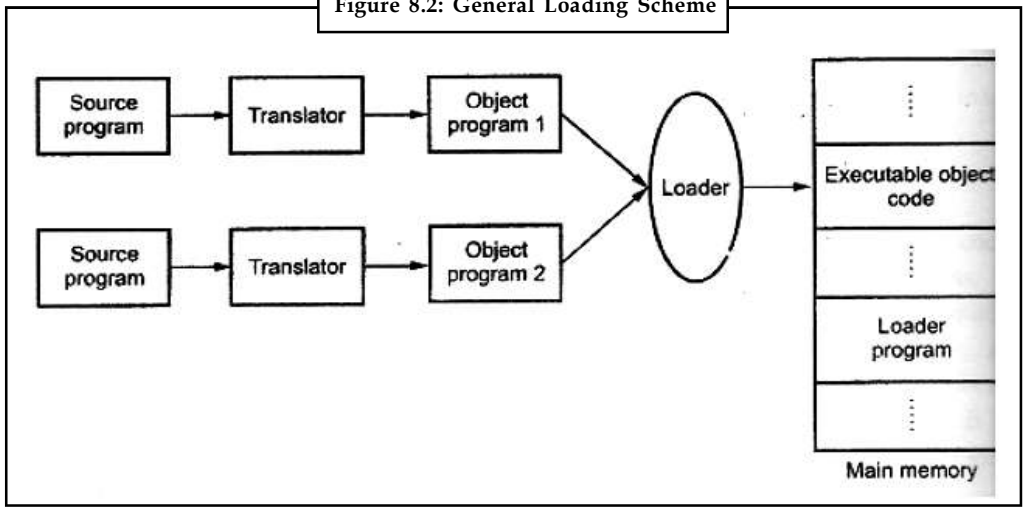
8.1.2 General Loader Scheme

In General Loader Scheme loader scheme, the source program is transformed to object program by some translator (assembler). The loader admits these object modules and puts machine instruction and data in an implementable form at their allocated memory. The loader occupies some part of main memory.

Advantages

- The program require not be retranslated every time while executing it. This is because primarily when source program gets executed an object program gets created. Of program is not customized, then loader can utilize this object program to translate it to executable form.

Figure 8.2: General Loading Scheme



- There is no depletion of memory, since assembler is not positioned in the memory, instead of it, loader contains some part of the memory. And size of loader is minor than assembler, so more memory is obtainable to the user.

Notes

- It is possible to write source program with numerous programs and multiple languages, since the source programs are first transformed to object programs always, and loader accepts these object modules to translate it to executable form.

8.1.3 Absolute Loaders

Absolute loader is a type of loader in which relocated object files are produced, loader accepts these files and positions them at particular locations in the memory. This type of loader is known as absolute since no relocation information is required; rather it is attained from the programmer or assembler. The beginning address of every module is recognized to the programmer, this corresponding beginning address is accumulated in the object file, then task of loader turns out to be very simple and that is to simply position the executable form of the machine instructions at the locations declared in the object file. In this scheme, the programmer or assembler should have information of memory management. The resolution of external references or linking of dissimilar subroutines are the concerns which require to be managed by the programmer. The programmer should pay attention to two things: first thing is specification of beginning address of each module to be utilized. If some alteration is performed in some module then the length of that module may differ. This causes a variation in the starting address of instant next modules, its then the programmer's duty to make essential variations in the starting addresses of respective modules. Second thing is while branching from one division to another the absolute starting address of individual module is to be recognized by the programmer so that such address can be specified at individual JMP instruction.

Example:

```
Line number
1          MAIN        START          1000
.                      .
.                      .
.                      .
15                     JMP            5000
16                     STORE          ;instruction at location 2000
                       END
1          SUM         START          5000
2
20                     JMP            2000
21                     END
```

This example includes two segments, which are inter-reliant. At line number 1 the assembler directive START states the physical starting address that can be used throughout the execution of the first segment MAIN. Then at line number 15 the JMP instruction is provided which states the physical starting address that can be utilized by the second segment. The assembler makes the object codes for these two segments by taking into account the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as mentioned by the programmer. Thus, the problem of

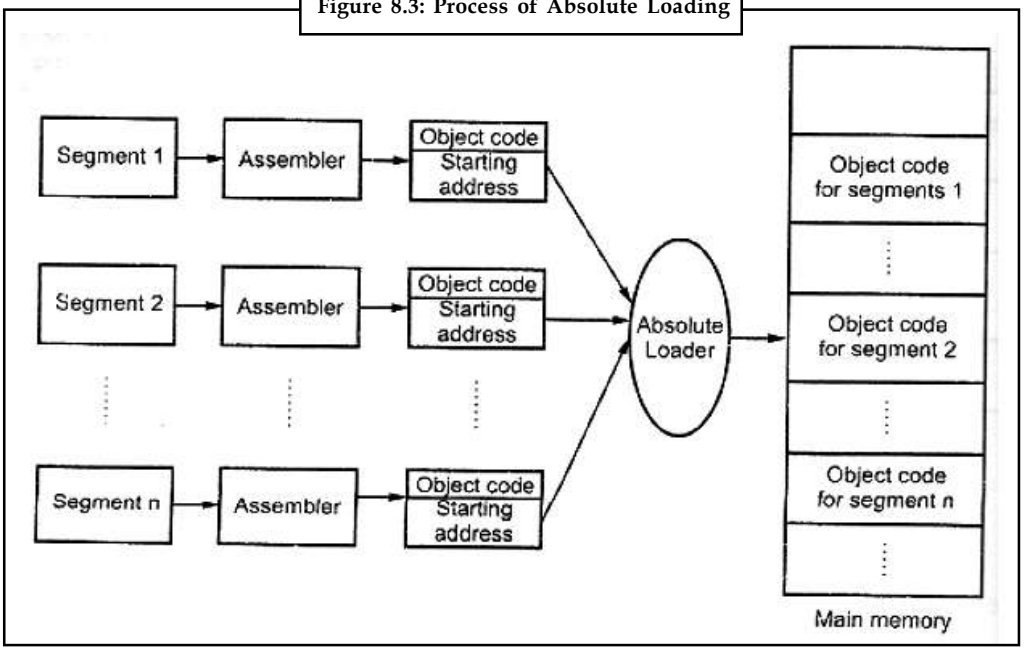
Notes

linking is manually solved by the programmer itself by taking care of the mutually dependant dresses. As you can observe that the control is suitably transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, essentially at location 2000 the instruction STORE of line number 16 is present. So, resolution of mutual references and linking is done by the programmer.

Notes The task of assembler is to generate the object codes for the above segments and along with the information like starting address of the memory where in fact the object code can be positioned at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information regarding their starting addresses, it will in fact place (load) them in the memory at mentioned addresses.

The whole process is modeled in the following figure.

Figure 8.3: Process of Absolute Loading



Thus, the absolute loader is simple to execute in this scheme:

1. Allocation is performed by either programmer or assembler
2. Linking is performed by the programmer or assembler
3. Resolution is performed by assembler
4. Simply loading is performed by the loader.

As the name recommends, no relocation information is required, if at all it is necessary then that task can be completed by either a programmer or assembler.

Advantages

1. It is easy to implement.
2. This scheme permits multiple programs or the source programs written dissimilar languages. If there are various programs written in dissimilar languages then the respective

Notes

language assembler will translate it to the language and a familiar object file can be organized with all the ad resolution.

3. The task of loader turns out to be simpler as it simply obeys the instruction concerning where to position the object code in the main memory.
4. The procedure of execution is efficient.

Disadvantages

1. In this scheme it is the programmer's responsibility to regulate all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.
2. If at all any modification is done the some segments, the starting addresses of immediate next segments may get changed, the programmer has to take care of this concern and he requires to update the corresponding beginning addresses on any modification in the source.

Algorithm for Absolute Loader

Input: Object codes and beginning address of program segments.

Output: An executable code for matching source program. This executable code is to be positioned in the main memory.

Method: Begin

For each program segment do Begin

Read the first line from object module to obtain information about memory location. The starting address say S in corresponding object module is the memory location where executable code is to be placed.

Hence

Memory_location = S

Line counter = 1; as it is first line While (! end of file)

For the current object code

do Begin

1. Read next line
2. Write line into location S
3. S = S + 1
4. Line counter Line counter + 1

8.1.4 Subroutine Linkage

To recognize the concept of subroutine linkages, first consider the below scenario:

"In Program A a call to subroutine B is prepared. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Notes

Nothing is wrong in it. But from assembler's point of vision while producing the code for B, as B is not defined in the segment A, the assembler can not locate the value of this symbolic reference and therefore it will state it as an error. To conquer problem, there should be some method by which the assembler should be explicitly learned that segment B is really defined in some other segment C.

Caution Whenever segment B is used in segment A and if at all B is defined in C, then B must be stated as an external routine in A.

To state such subroutine as external, we can use the assembler directive EXT. Thus, the statement such as EXT B should be added at the beginning of the segment A. This really helps to notify assembler that B is defined somewhere else. Likewise, if one subroutine or a variable is defined in the existing segment and can be pointed by other segments then those should be stated by means of pseudo-ops INT. Thus, the assembler could inform loader that these are the subroutines or variables used by other segments.

Did u know? This general process of establishing the relations among the subroutines can be theoretically known as a_ subroutine linkage.

Example:

```
MAIN      START
              EXT  B
               .
               .
               .
              CALL B
               .
               .
              END
B         START
               .
               .
          RET
          END
```

At the beginning of the MAIN the subroutine B is stated as external. When a call to subroutine B is prepared, before creating the unconditional jump, the recent content of the program counter should be accumulated in the system stack preserved internally. Likewise while returning from the subroutine B (at RET) the pop is executed to restore the program counter of caller routine with the address of next instruction to be implemented.

Notes

8.1.5 Relocating Loaders

Relocation is the procedure of modernizing the addresses used in the address sensitive instructions of a program. It is essential that such an alteration should aid to carry out the program from preferred area of the memory.

The assembler produces the object code. This object code gets executed after loading at storage positions. The addresses of such object code will get specified only after the assembly process is over. Thus, after loading,

Address of object code = Mere address of object code + relocation constant.

There are two kinds of addresses being produced: Absolute address and, Relative address. The absolute address can be directly utilized to map the object code in the main memory. While the relative address is only after the addition of relocation constant to the object code address. This sort of adjustment needs to be completed in case of relative address before definite execution of the code.

Example: The classic example of relative reference is : addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals, redefinable symbols. Likewise, the classic example of absolute address is the constants which are produced by assembler are absolute.

The assembler computes which addresses are absolute and which addresses are relative throughout the assembly process. Throughout the assembly process the assembler computes the address with the aid of simple expressions.

Example: LOADA(X)+5

The expression A(X) means the address of variable X. The meaning of the above instruction is that loading of the contents of memory location which is 5 more than the address of variable X.

Let us consider if the address of X is 50 then by above command we attempt to get the memory location 50+5=55. Thus as the address of variable X is relative A(X) + 5 is also relative.

To compute the relative addresses the simple expressions are permitted. It is expected that the expression should possess at the most addition and multiplication operations. A straightforward exercise can be carried out to find out whether the specified address is absolute or relative. In the expression if the address is absolute then put 0 over there and if address is relative then put lover there. The expression then obtains transformed to sum of O's and l's. If the resultant value of the expression is 0 then expression is absolute. And if the consequential value of the expression is 1 then the expression is relative. If the resultant is other than 0 or 1 then the expression is illegal.

Example:

Expression	Computation	Relocation Attribute
A - B	1 - 1 = 0	Absolute
A + B - C	1 + 1-1= 1	Relative
A - B + 5	1 -1 + 0 = 0	Absolute
A + B	1 + 1 = 2	Illegal

In the above expression the A, B and C are the variable names. The assembler is to consider the relocation attributes and regulates the object code by relocation constant. Assembler is then accountable to convey the information loading of object code to the loader.

Notes

Task Make distinction between absolute address and relative address.

8.1.6 Direct Linking Loaders

The direct linking loader is the main common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to position the object code in the memory there are two circumstances: either the address of the object code could be absolute which then can be directly positioned at the specified location or the address can be relative. If at all the address is relative then it is the assembler who notifies the loader regarding the relative addresses.

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 111 the present segment but can be utilized in the current segment.
3. The list of all the symbols which are defined in the existing 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 USE table preserves the information such as name of the symbol, address, address relativity.

The list of symbols which are defined in the current segment and can be referred by the other segments are accumulated in a data structure known as DEFINITION table. The definition table preserves the information like symbol, address.

Self Assessment

Fill in the blanks:

1. ..... is defined as a utility program which considers object code as input organizes it for implementation and loads the executable code into the memory.
2. In ..... loader, the instruction is read line by line, its machine code is attained and it is directly placed in the major memory at some recognized address.
3. In ..... Scheme loader scheme, the source program is transformed to object program by some translator (assembler).
4. ..... is a type of loader in which relocated object files are produced, loader accepts these files and positions them at particular locations in the memory.
5. To state a subroutine as external, we can use the assembler directive .....
6. The general process of establishing the relations among the subroutines can be theoretically known as ..... linkage.
7. ..... is the procedure of modernizing the addresses used in the address sensitive instructions of a program.
8. The ..... address can be directly utilized to map the object code in the main memory.
9. The ..... address is only after the addition of relocation constant to the object code address.

Notes

8.2 Other Loader Schemes

8.2.1 Binders

The implementation of any program requires four basic functionalities and those are allotment, relocation, linking and loading. As we have also observed in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Furthermore if the program contains many subroutines or functions and the program requires to be executed repeatedly then this activity turns out to be annoyingly complex. Each time for implementation of a program, the allocation, relocation linking and loading requires to be done. Now performing these activities each time augments the time and space complexity. In fact, there is no requirement to redo all these four activities each time. Rather, if the results of some of these activities are accumulated in a file then that file can be utilized by other activities. And doing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Therefore dividing the essential four functions in groups decreases the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, generates linked executable text and accumulates this text in a file in some systematic manner. Such type of module organized by the binder execution is called load module. This load module can then be in fact loaded in the main memory by the loader. This loader is also recognized as module loader. If the binder can generate the exact replica of executable code in the load module then the module loader just loads this file into the main memory which eventually decreases the overall time complexity. But in this process the binder should knew the existing positions of the main memory. Although the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the area of main memory obtainable for loading the program is determined by the host operating system. The binder should also know which memory area is allocated to the loading program and it should adjust the relocation information consequently. The binder which performs the linking function and generates adequate information concerning allocation and relocation and writes this information along with the program code in the file is known as linkage editor. The module loader then accepts this rile as input, reads the information accumulated in and depending on this information about allocation and relocation it performs the task of loading in the main memory.

Notes Although the program is repeatedly executed the linking is finished only once. Furthermore, the flexibility of allocation and relocation aids efficient utilization of the main memory.

Task Illustrate the function of binders.

8.2.2 Linkers

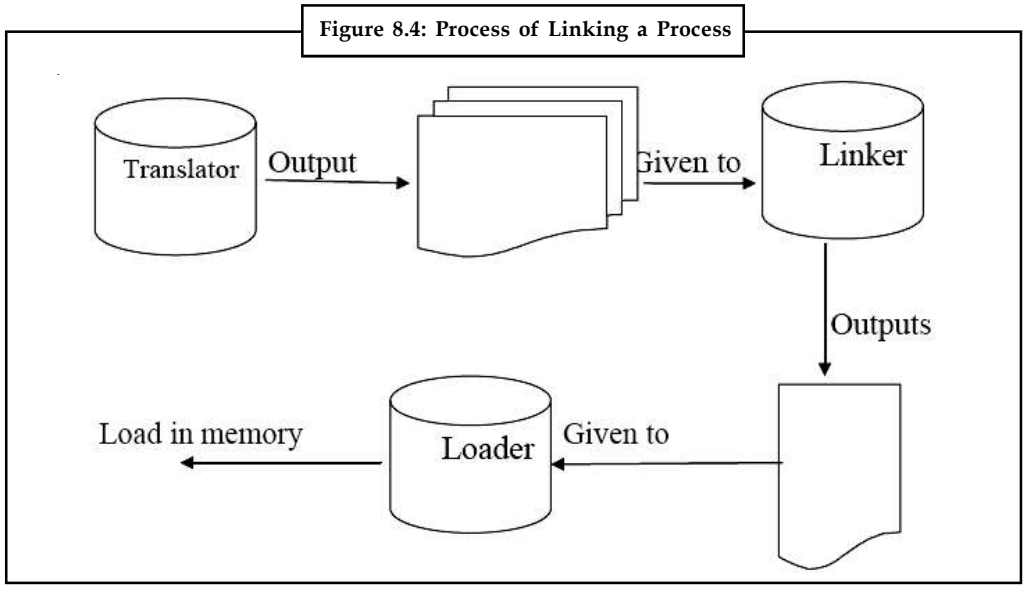
As we have discussed earlier, the execution of program can be performed with the assistance of following steps:

1. Translation of the program (carried out by assembler or compiler).
2. Linking of the program with all other programs which are required for execution. This also includes preparation of a program called load module.
3. Loading of the load module prepared by linker to some mentioned memory location.

Notes

The output of translator is a program known as object module. The linker processes these object modules binds with essential library routines and makes a ready to implement program. Such a program is known as binary program. The "binary program also comprises some necessary information regarding allocation and relocation. The loader then loads this program into memory for implementation reason.

Figure 8.4: Process of Linking a Process



Different tasks of linker are:

1. Prepare a single load module and regulate all the addresses and subroutine references with respect to the offset position.
2. To prepare a load module concatenate all the object modules and regulate all the operand address references in addition to external references to the offset position.
3. At accurate locations in the load module, copy the binary machine instructions and constant data so as to prepare ready to implement module.

The linking process is done in two passes. Two passes are essential since the linker may encounter a forward reference before recognizing its address. So it is essential to scan all the DEFINITION and USE table at least once. Linker then builds the Global symbol table with the help of USE and DEFINITION table. In Global symbol table name of every externally referenced symbol is involved along with its address relative to starting of the load module. And throughout pass 2, the addresses of external references are substituted by attaining the addresses from global symbol table.

8.2.3 Loaders

As we turn on the computer there is nothing significant in the main memory (RAM). A small program is written and amassed in the ROM. This program primarily loads the operating system from secondary storage to main memory. The operating system then takes the on the whole control. This program which is accountable for booting up the system is known as bootstrap loader.

Caution This is the program which must be implemented first when the system is first powered on.

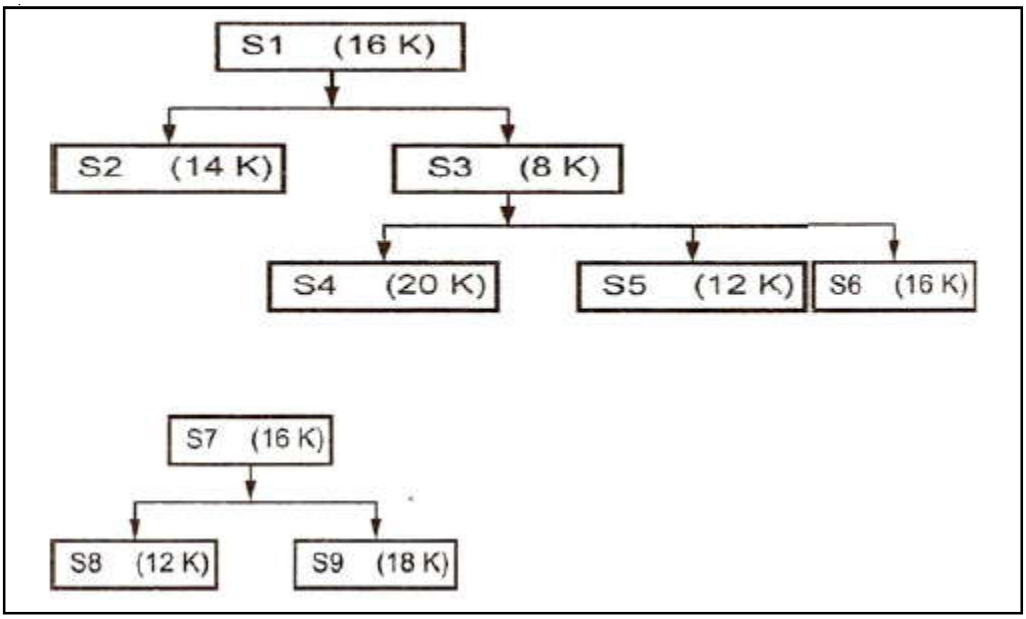
Notes

If the program begins from the location x then to implement this program the program counter of this machine should be loaded with the value x. Thus, the task of setting the initial value of the program counter is to be performed by machine hardware. The bootstrap loader is a very small program which is to be fixed in the ROM. The job of bootstrap loader is to load the essential portion of the operating system in the main memory. The initial address at which the bootstrap loader is to be loaded is usually the lowest (may be at 0th location) or the highest location.

8.2.4 Overlay Structures and Dynamic Loading

At times a program may need more storage space than the obtainable one implementation of such program can be probable if all the segments are not needed at the same time to be present in the main memory. In such situations only those segments are resident in the memory that is in fact wanted at the time of execution But the question occurs what will happen if the needed segment is not there in the memory? Obviously the execution process will be delayed until the needed segment gets loaded in the memory. The overall effect of this is competence of execution process gets degraded. The efficiency can then be enhanced by carefully choosing all the interdependent segments. Of course the assembler can not do this task. Only the user can specify such dependencies. The inter dependency of the segments can be declared by a tree like structure called static overlay structures. The overlay structure comprises multiple root/nodes and edges. Each node represents the segment. The specification of necessary amount of memory is also essential in this structure. The two segments can lie concurrently in the main memory if they are on the same path.

Example: Let us consider an example to recognize the concept. Numerous segments along with their memory necessities is as shown below.



8.2.5 Dynamic Binders

In dynamic linking, the binder first makes a load module in which along with program code the allocation and relocation information is accumulated. The loader just loads the main module in the main memory. If any external reference to a subroutine comes, then the execution is suspended

Notes

for a while, the loader brings the necessary subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is performed dynamically.

Self Assessment

Fill in the blanks:

10. The program which performs allocation, relocation and linking is called .....
11. The binder which performs the linking function and generates adequate information concerning allocation and relocation and writes this information along with the program code in the file is known as .....
12. The output of translator is a program known as .....
13. The program which is accountable for booting up the system is known as .....
14. The ..... structure comprises multiple root/nodes and edges.
15. In ..... linking, the binder first makes a load module in which along with program code the allocation and relocation information is accumulated.

8.3 Summary

- Loader is defined as a utility program which considers object code as input organizes it for implementation and loads the executable code into the memory.
- In "Compile and go" loader, the instruction is read line by line, its machine code is attained and it is directly placed in the major memory at some recognized address.
- In General Loader Scheme loader scheme, the source program is transformed to object program by some translator (assembler).
- Absolute loader is a type of loader in which relocated object files are produced, loader accepts these files and positions them at particular locations in the memory.
- The general process of establishing the relations among the subroutines can be theoretically known as a_ subroutine linkage.
- Relocation is the procedure of modernizing the addresses used in the address sensitive instructions of a program.
- The absolute address can be directly utilized to map the object code in the main memory.
- The relative address is only after the addition of relocation constant to the object code address.
- The direct linking loader is the main common type of loader which is a relocatable loader.
- In dynamic linking, the binder first makes a load module in which along with program code the allocation and relocation information is accumulated.

8.4 Keywords

Absolute Address: The absolute address can be directly utilized to map the object code in the main memory.

Absolute Loader: Absolute loader is a type of loader in which relocated object files are produced, loader accepts these files and positions them at particular locations in the memory.

Notes

Compile and go Loader: In "Compile and go" loader , the instruction is read line by line, its machine code is attained and it is directly placed in the major memory at some recognized address.

Direct Linking Loader: The direct linking loader is the main common type of loader which is a relocatable loader.

General Loader: In General Loader Scheme loader scheme, the source program is transformed to object program by some translator (assembler).

Loader: Loader is defined as a utility program which considers object code as input organizes it for implementation and loads the executable code into the memory.

Relocation: Relocation is the procedure of modernizing the addresses used in the address sensitive instructions of a program.

Relative Address: The relative address is only after the addition of relocation constant to the object code address.

Subroutine Linkage: The general process of establishing the relations among the subroutines can be theoretically known as a_ subroutine linkage.

8.5 Review Questions

1. What is a loader? Illustrate the functions of loader.
2. Describe the concept of "Compile and go" loader scheme with example.
3. Enlighten the advantages and disadvantages of "Compile and go" loader scheme.
4. Explicate the loader scheme of transforming the source program to object program by some translator (assembler).
5. What is absolute loader? Illustrate its advantages and disadvantages.
6. Write an algorithm for absolute Loader.
7. Explain the concept of subroutine linkage with example.
8. What are relocating loaders? Illustrate the concept.
9. Make distinction between binders and linkers.
10. The overlay structure comprises multiple root/nodes and edges. Comment.

Answers: Self Assessment

1. Loader
2. "Compile and go"
3. General Loader
4. Absolute loader
5. EXT
6. a_ subroutine
7. Relocation
8. absolute
9. relative
10. binder
11. linkage editor
12. object module
13. bootstrap loader
14. overlay
15. dynamic

8.6 Further Readings

Notes

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online link

uotechnology.edu.iq

Notes

Unit 9: Design of an Absolute Loader

CONTENTS

Objectives

Introduction

9.1 Design of an Absolute Loader

9.2 Design of a Direct Linking Loader

9.2.1 Specification of Problem

9.2.2 Specification of Data Structures

9.2.3 Algorithm

9.3 Summary

9.4 Keywords

9.5 Review Questions

9.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the design of Absolute Loader
- Discuss the design of direct linking loader

Introduction

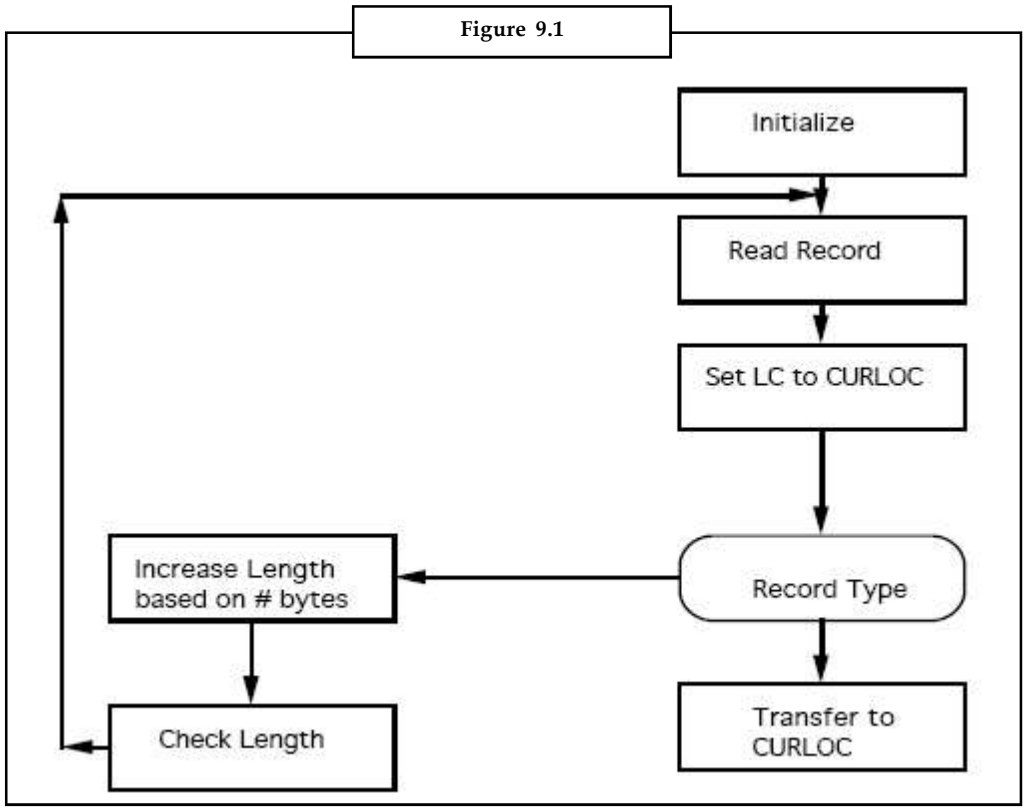
By means of an absolute loading scheme the programmer and the assembler execute the tasks of location, relocation, and linking. Thus, it is only essential for the loader to read records of the object file and shift the text on the records into the absolute locations declared by the assembler. In this unit, you will understand the design of Absolute Loader and design of direct linking loader.

9.1 Design of an Absolute Loader

There are two kinds of information that the object file must contain from the assembler to the loader. Primarily, it must transmit the machine instructions that the assembler has formed along with the allocated core locations. Next, it must transmit the entry point of the program, which is where the loader is to transport control when all instructions are loaded. The algorithm for an absolute loader is pretty simple. The object file for this loader comprises of a series of text records completed by a transfer record. So, the loader should read one record at a time, moving the text to the location stated on the record, until the transfer record is accomplished. Here the assembled instructions are in core, and it is only essential to transmit to the entry point stated on the transfer record. A flowchart for this process is depicted in figure 9.1.

Notes

Figure 9.1



Example: Write a C Program for the Implementation of an Absolute Loader.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char  input[10],label[10],ch1,ch2;
int  addr,w=0,start,ptaddr,l,length=0,end,count=0,k,taddr,address,i=0;
FILE  *fp1,*fp2;
void  check();
void  main()
{
clrscr();
fp1=fopen("INPUT.dat","r");
fp2=fopen("OUTPUT.dat","w");
fscanf(fp1,"%s",input);
printf("\n\n\t\t\tABSOLUTE  LOADER\n");
fprintf(fp2,"\n--------------------\n");
fprintf(fp2,"MEMORY  ADDRESS\t\t\tCONTENTS");
fprintf(fp2,"\n--------------------\n");
while(strcmp(input,"E")!=0)
```

Notes

```
{
if(strcmp(input,"H")==0)
{
fscanf(fp1,"%s  %x  %x  %s",label,&start,&end,input);
address=start;
}
else  if(strcmp(input,"T")==0)
{
l=length;
ptaddr=addr;
fscanf(fp1,"%x  %x  %s",&taddr,&length,input);
addr=taddr;
if(w==0)
{
ptaddr=address;
w=1;
}
for(k=0;k<(taddr-(ptaddr+l));k++)
{
address=address+1;
fprintf(fp2,"xx");
count++;
if(count==4)
{
fprintf(fp2,"   ");
i++;
if(i==4)
{
fprintf(fp2,"\n\n%x\t\t",address);
i=0;
}
count=0;
}
}
if(taddr==start)
fprintf(fp2,"\n\n%x\t\t",taddr);
fprintf(fp2,"%c%c",input[0],input[1]);
check();
fprintf(fp2,"%c%c",input[2],input[3]);
check();
```

Notes

```
fprintf(fp2,"%c%c",input[4],input[5]);
check();
fscanf(fp1,"%s",input);
}
else
{
fprintf(fp2,"%c%c",input[0],input[1]);
check();
fprintf(fp2,"%c%c",input[2],input[3]);
check();
fprintf(fp2,"%c%c",input[4],input[5]);
check();
fscanf(fp1,"%s",input);
}
}
fprintf(fp2,"\n--------------------\n");
fcloseall();
printf("\n\n The contents of output file:\n\n");
fp2=fopen("OUTPUT.DAT","r");
ch2=fgetc(fp2);
while(ch2!=EOF)
{
printf("%c",ch2);
ch2=fgetc(fp2);
}
fcloseall();
getch();
}
void check()
{
count++;
address++;
taddr=taddr+1;
if(count==4)
{
fprintf(fp2,"   ");
i++;
if(i==4)
{
fprintf(fp2,"\n\n%x\t\t",taddr);
```

Notes

```
i=0;
}
count=0;
}
}
```

Self Assessment

Fill in the blanks:

1. By means of an ..... loading scheme the programmer and the assembler execute the tasks of location, relocation, and linking.
2. There are two kinds of information that the object file must contain from the assembler to the .....
3. The ..... for absolute loader comprises of a series of text records completed by a transfer record.

9.2 Design of a Direct Linking Loader

The obscure traits have been omitted, and where substitute formats are probable, only the simplest is provided. The design steps followed will parallel those taken in the plan of an assembler.

Notes Since the direct-linking loader requires to know the absolute (load time) values of some external symbols before it can carry out the alterations on address constants, it needs two passes.

9.2.1 Specification of Problem

The organization helps the tasks to be done by its relocating loader. A direct access machine was essential to relocate the address part on almost all instructions. In the mainframe system, instruction relocation is finished by the utilization of the base register, which is set by neither the assembler nor the loader. Thus, the mainframe relocating loader can treat instructions precisely like non relocatable data (full word constants, characters, etc.). Nevertheless, address constants must still be relocated.

Example: The following instructions:

			Might be assembled as
TEST	Start		
	Load Reg1,DAWG		O_{16} LOAD Reg1,96_{16} (0)
	.....		
DAWG	DATA 5		96_{16} 5_{16}
	END		

Irrespective of where the program is loaded, the LOAD instruction will be unaffected as long as DATA remains 96 bytes from the starting of the program. The real load address may effect in a different address for the S-field, based upon the program's load location.

Notes

Alternatively, consider adjusting the above example:

```
DAWG                DATA 5
CAT     ADDR        A(DAWG)
END
```

CAT must enclose the absolute address of DAWG. The assembler recognizes only that DAWG is 96 bytes from the starting of the program, so the loader must add to this the load address of the program in locating the absolute address to be included in CAT.

Let us elucidate the scope of the address constant problem. An address constant may be

1. absolute;
2. simple relocatable; or
3. complex relocatable.

The direct linking loader processes programs produced by the assembler, FORTRAN compiler, or some other compiler.

Did u know? Neither the original source program nor the assembler symbol table is obtainable to the loader.

Caution The object file must enclose all information required for relocation and linking.

There are four segments to the object file (and four corresponding formats):

1. External Symbol Dictionary (ESD)
2. Instructions and data records, called "text" of program (TXT)
3. Relocation and Linkage Directory (RLD)
4. End (END)

The ESD records enclose the information essential to build the external symbol dictionary or symbol table. External symbols are symbols that can be referred away from the subroutine level. Only the assembler utilizes the normal labels in the source program, and information regarding them is not incorporated in the object file.

```
A               START
                EXTERNAL    NAMES

                LOAD        1,ADDRNAME      get address of NAME table

ADDRNAME        ADDRESS     Addr(NAMES)
                END
################################################
B               START
                ENTRY       NAMES

NAMES           DATA        nn*0
                END
```

Notes

Presume program B has table known as NAMES; it can be accessed by program A as follows:

There are three types of external symbols, as depicted:

1. *Segment Definition/Program Name (SD)*-name on START
2. *Load Definition (LD)* – stated on ENTRY. There must be a label in similar program with similar name.
3. *External Reference (ER)* – stated on EXTERNAL. There must be a matching ENTRY, or START in another program with the similar name.

The TXT records enclose blocks of data and the relative address at which the data is to be positioned. Once the loader has decided where to load the program, it simply adds the Program Load Address (IPLA) to the relative address and moves the data into the consequential location. The data on the TXT record may be instructions, non-relocated data, or initial values of address constants.

The RLD records enclose the following information:

1. The location and length of each address constant that requires to be changed for relocation or linking.
2. The external symbol by which the address constant should be altered (added or subtracted).
3. The operation to be done (add or subtracted).

Variable	flag	Length	Rel. loc
A	+	4	52
NAMES	+	4	56

This RLD information notifies the loader to add the absolute load address of A to the contents of relative location 52 and then add the absolute load address of NAMES to the contents of relative location 56.

The END record states the end of the object file. If the assembler END record has a symbol in the operand field, it states a start of execution point for the whole program (all subroutines). This address is recorded on the END record. There is a final record needed to state the end of a collection of object files. The mainframe loaders typically use either a loader terminate (LOT) or End of File (EOF) record.

	ESD
Subroutine A	TXT
	RLD
	END
	ESD
Subroutine B	TXT
	RLD
	END
	ESD
Subroutine C	TXT
	RLD
	END
	EOF

Notes

The simple programs PGA and PGB demonstrate a broad range of relocation and linking situations. The display shows the ESD, TXT, and RLD records formed by the assembler for PGA and PGB, respectively. Lastly, we show, the contents of main storage after the programs have been assigned space, relocated, linked, and loaded.

You should scrutinize these figures cautiously and validate the accuracy and reasons for each value. A few particular points in these examples should be observed. Both PGA and PGB enclose an address constant of the form A(A2-A1-3).

```
1     0     PGA     START
2                   ENTRY       A1,A2
3                   EXTERNAL    B1,PGB
4     20    A1
5     30    A2
6     40              ADDR        A(A1)
7     44              ADDR        A(A2+15)
8     48              ADDR        A(A2-A1-3)
9     52              ADDR        A(PGB)
10    56              ADDR        A(B1+PGB-A1+4)
11                    END
-------------------------------------------------
12    0     PGB     START
13                  ENTRY       B1
14                  EXTERNAL    A1,A2
15    16    B1
16    24              ADDR        A(A1)
17    28              ADDR        A(A2+15)
18    32              ADDR        A(A2-A1-3)
19                    END
```

You should observe that both instances of this address constant (location 152, 200) have the similar value. Since both A2 and A1 are symbols internal to PGA, the assembler processing PGA, can work out the whole expression and find out the value of 7. We observe that the TXT record for location 48-51 includes the 7 and there are no connected RLD records for address constant. Alternatively, these symbols are external PGB; thus, the assembler, while processing PGB has no means of assessing the address constant. This is demonstrated also. The TXT record for relative location 32-35 comprises a -3, the only part of the address constant that can be gauged by the assembler. The last two RLD records inform the loader to add the load address of A2 to location 32-35 and then subtract the load address of A1. When processed by the loader, this address constant in PGB will certainly have the similar value as the one in PGA. As the direct linking loader may come across external references in an object file which can not be assessed until a later object file is processed, this type of loader needs two passes. Their functions are very alike to those of the two passes of an assembler. The main function of pass 1 of a direct linking loader is to assign and assign each program location in core and create a symbol table filling in the values of the external symbols. The main function of pass 2 is to load the definite program text and carry out the relocation modification of any address constants requiring to be altered. The first pass assigns storage locations to all segments and accumulates the values of an external symbols in a symbol table.

Did u know? The external symbol occurs as local definitions on the ESD records of another assembled program.

Caution For every external reference symbol there must be an analogous internal symbol in some other program.

Notes

Notes The loader inserts the absolute address of all of the external symbols in the symbol table. In the second pass the loader positions the text into the allocated locations and performs the relocation task, amend relocatable constants.

Task Illustrate the function of TXT records.

ESD Records for PGA

Variable Name	Type	Address	Length	Reference: to source line
PGA	Program Name	0	60	1
A1	Local variable	20		2
A2	Local variable	30		2
PGB	External Var			3
B1	External Var			3

TXT Records for PGA

Relative Address	Contents	What the assembler did	Reference to source line
40	20		6
44	45	=30+15	7
48	7	30-20-3	8
52	0	unknown	9
56	-16	-20+4	10

RLD Records for PGA

Relative Address	Arithmetic Operator	Variable Name	Length	Reference to source line
40	+	PGA	4	6
44	+	PGA	4	7
52	+	PGB	4	9
56	+	B1	4	10
56	+	PGB	4	10
56	-	PGA	4	10

ESD Records for PGB

Variable Name	Type	address	Length	Reference: to source line
PGB	Program Name	0	36	12
B1	Local Variable	16	N/A	13
A1	External Var	unknown	N/A	14
A2	External Var	unknown	N/A	14

TXT Records for PGB

Relative Address	Contents	What the assembler did	Reference to source line
24	0	=0 A1 unknown	16
28	15	=15 A2 unknown	17
32	-3	=-3 A1,A2 unknown	18

RLD Records for PGB

Relative Address	Arithmetic Operator	Variable Name	Length	Reference to source line
24	+	A1	4	16
28	+	A2	4	17
32	+	A2	4	18
32	-	A1	4	18

Main storage after loading programs PGA and PGB

PGA loaded at location 104 PGB loaded at location 168.

Notes

Final memory memory address	contents of memory	
104		
..		
..		
..		
144	124	6
148	149	7
152	7	8
156	168	9
160	232	10
164	unused	
168	load point of PGB	
..		
..		
192	124	16
196	149	17
200	7	18

9.2.2 Specification of Data Structures

The next step in our design process is to recognize the databases necessary by each pass of the loader.

Format of Databases

Pass 1 databases:

1. Input object files.
2. A parameter, the Initial Program Load Address (IPLA) provided by the programmer or the operating system that specifies the address to load the first segment.
3. A Program Load Address (PLA) counter, utilized to keep track of each segment's assigned location.
4. A table, the Global External Symbol Table (GEST) that is used to amass each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2. This may be amassed on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object file may be reread by the loader a second time for pass 2.
6. A printed listing, the load map, that states each external symbol and its assigned value.

Pass 2 databases:

1. Copy of object program to be inputted to pass 1.
2. The Initial Program Load Address parameter (IPLA).
3. The Program Load Address counter (PLA).
4. The Global External Symbol Table (GEST'), arranged by pass 1, comprising each external symbol and its corresponding absolute address value.

Notes

5. An array, the Local External Symbol Array (LESA), which is utilized to establish a correspondence among the NAMES, used on ESD and RLD records, and the analogous external symbol's absolute address.

Global External Symbol Table

The Global External Symbol Table (GEST) is used to amass the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) record. When these symbols are encountered throughout pass 1, they are allocated an absolute core address; this address is stored, along with the symbol, in the GEST as demonstrated.

Task Compare pass 1 databases and pass 2 databases.

9.2.3 Algorithm

The following two flowcharts illustrate an algorithm for a direct linking loader. While most of the logical processes are involved, these flowcharts are still a generalization of the operations carried out in a complex loader. Particularly, many special traits, like COMMON segments library processing dynamic loading dynamic linking are not overtly included.

Pass 1 – Allocate Segments and Define Symbols

The purpose of the first pass is to assign a location to each segment, and thus to define the values of all external symbol.

Since we wish to minimize the amount of core storage required for the total program, we will assign each segment the next table location after the preceding segment. It is necessary for the loader to know where. It can load the first segment. This address the Initial Program Load Address (IPLA), is normally determined by the operating system. In some systems the programmer may specify the IPLA; in either case we will assume that the IPLA is a parameter supplied to the loader. Initially, the Program Load Address (PLA) is set to the Initial Program Load address IPLA. An object record is then read and a copy written for use by pass 2. The record can be one of five types, ESD, TXT, RLD, END, or LDT/EOF.

If it is a TXT or RLD record, there is no processing required during pass 1 so the next record is read. An ESD record is processed in different ways depending upon the type of external symbol, SD, LD or ER. If a segment definition

ESD record is read, the length field, LENGTH, from the record is temporarily saved in the variable, SLENGTH . The value, VALUE, to be assigned to this symbol is set to the value of the PLA. The symbol and its assigned value are then stored in the GEST; if the symbol already existed in the GEST, there must have been a previous SD or LD ESD with the same name - this is an error. The symbol and its value are printed as part of the load map. A similar process is used for LD symbols; the value to be assigned is set to the current PLA plus the relative address, ADDR, indicated on the ESD record. The ER symbols do not require any processing during pass 1. When an END record is encountered, the program load address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment. When the LDT or EOF record is finally read, pass 1 is completed and control transfer to pass 2.

Notes

Example: Write a C program to execute pass 1 of direct linking loader.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#define MAX 10
struct estab
{     char csect[10];
char sym_name[10];
long int add;
int length;
}table[MAX];
void main()
{    FILE *fp1, *fp2;
char input[10];
long int i, count =0, start, length, loc;
clrscr();
fp1=fopen("linkin.dat", "r");
fp2=fopen("linkout.dat", "w");
printf("Enter the location where the program has to be loaded:");
scanf("%lx", &start);
fprintf(fp2, "CSECT \t symname \t address \t length\n");
rewind(fp1);
while(!feof(fp1))
{
fscanf(fp1, "%s", input);
if(strcmp(input, "H")==0)
{
fscanf(fp1, "%s", input);
strcpy(table[count].csect, input);
strcpy(table[count].sym_name, "");
fscanf(fp1, "%s", input);
table[count].add=atoi(input)+start;
fscanf(fp1, "%s", input);
length=atoi(input);
table[count++].length=atoi(input);
fscanf(fp1, "%s", input);
}
if(strcmp(input, "D")==0)
```

Notes

```
{
fscanf(fp1, "%s%lx", input, &loc);
while((strcmp(input, "R")!=0))
{
strcpy(table[count].csect, "");
strcpy(table[count].sym_name, input);
table[count].add=loc+start;
table[count++].length=0;
fscanf(fp1, "%s%lx", input, &loc);
}
while(strcmp(input, "T")!=0)
fscanf(fp1, "%s", input);
}
if(strcmp(input, "T")==0)
while(strcmp(input, "E")!=0)
fscanf(fp1, "%s", input);
fscanf(fp1, "%s", input);
start=start+length;
}
for(i=0; i<count; i++)
fprintf(fp2, "%s\t%s\t%lx\t%d\n", table[i].csect, table[i].sym_name,
table[i].add, table[i].length);
printf("\nPass 1 of Direct Linking Loaded is completed");
getch();
}
```

Pass 2 – Load Text and Relocate/Link Address Constants

After all the segments have been allocate locations and the external symbols have been defined by pass 1, it is possible to complete the loading by loading the text and changing (relocation or linking) address constants. At the end of pass 2, the loader will transmit control to the loaded program. The following simple rule is frequently used to find out where to begin execution:

1. If an address is stated on the END record, that address is used as the implementation start address.
2. Or else, execution will begin at the beginning of the first segment

At the starting of pass 2 the program load address is initialized as in pass 1, and the execution start address

(EXADDR) is set to IPLA. The records are read one by one from the object file left by pass 1.

All of the five types of records is processed in a different way, as follows:

ESD Records

Each of the ESD record types is processed in a different way.

Notes

SD-type ESD The LENGTH of the segment is momentarily saved in the variable SLENGTH. The suitable entry in the local external symbol array, LESA(ID), is set to the current value of the Program Load Address.

LD type ESD The LD-type ESD does not require any processing during pass 2.

ER type ESD The Global External Symbol Table (GEST) is searched for match with the ER symbol. If it is not found, the corresponding segment or entry must be missing - this is a mistake. If the symbol is found in the GEST, its value is extracted and the corresponding Local External Symbol Array entry, LESA(ID), is set equal to it. TXT record. When a TXT record is read, the text is copied from the record to the suitable relocated core location (PLA + ADDR).

RLD Record

The value to be utilized for relocation and linking is extracted from the load external symbol array as specified).

Based upon the flag setting (plus or minus) the value is either added to or subtracted from the address constant. The definite relocated address of the address constant is computed as the sum of the PLA and the ADDR field stated on the RLD record.

END Record

If an execution start address is stated on the END record, it is saved in the variable EXADDR after being relocated by the PLA. The Program Load Address is augmented by the length of the segment and saved in SLENGTH, turning out to be the PLA for the next segment.

Example: Write a C program to execute pass 2 of direct linking loader.

Algorithm

STEP 1: Start the program execution.

STEP 2: Find the control section length from the header record.

STEP 3: In the text record, character form is converted to machine representation.

STEP 4: Move the object code from record to memory location.

STEP 5: In the modification record, search for symbol to be modified from external table.

Found symbol address is added or subtracted with corresponding symbol address

and value at starting location.

STEP 6: Stop the program execution.

Program

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct  ext_table
{  char  csect[10];
```

Notes

```
char sname[10];
int padd;
int plen;
} estab[20];
struct object_code
{ char code[15];
int add;
}
obcode[500];
void main()
{
FILE *fp1,*fp2,*fp3;
int
i,j,n=0,num=0,inc=0,count=0,record=0,pstart,exeloc,start,textloc,loc,mloc[30],textlen,ml
en[30],length,location,x,y;
long int newadd;
char *add1,operation,lbl[10],input[10],label[30][10],address[10];
clrscr();
fp1=fopen("link1.c","r");
fp2=fopen("link2.c","r");
fp3=fopen("link3.c","w");
rewind(fp1);
rewind(fp2);
rewind(fp3);
while(!feof(fp2))
{
fscanf(fp2,"%s%s%d
%d",estab[num].csect,estab[num].sname,&estab[num].padd,&estab[num].plen);
num++;
} exeloc=estab[0].padd;
loc=exeloc;
start=loc;
while(!feof(fp1))
{
fscanf(fp1,"%s",input);
if(strcmp(input,"H")==0)
{
fscanf(fp1,"%s",input);
for(i=0;i<num;i++)
if(strcmp(input,estab[i].csect)==0)
```

Notes

```
{
pstart=estab[i].padd;
break;
}
while(strcmp(input,"T")!=0)
fscanf(fp1,"%s",input);
}
do
{  if(strcmp(input,"T")==0)
{
fscanf(fp1,"%d",&textloc);
textloc=textloc+pstart;
for(i=0;i<(textloc-loc);i++)
{s
trcpy(obcode[inc].code,"xx");
obcode[inc++].add=start++;
}
fscanf(fp1,"%d",&textlen);
loc=textloc+textlen;
}  else  if(strcmp(input,"M")==0)
{
fscanf(fp1,"%d",&mloc[record]);
mloc[record]=mloc[record]+pstart;
fscanf(fp1,"%d",&mlen[record]);
fscanf(fp1,"%s",label[record++]);
}  else
{  length=strlen(input);
x=0;
for(i=0;i<length;i++)
{
obcode[inc].code[x++]=input[i];
if(x>1)
{
obcode[inc++].add=start++;
x=0;
}}}
fscanf(fp1,"%s",input);
}
while(strcmp(input,"E")!=0);
if(strcmp(input,"E")==0)
```

Notes

```
fscanf(fp1,"%s",input);
}
for(n=0;n<record;n++)
{
operation=label[n][0];
length=strlen(label[n]);
for(i=1;i<length;i++)
lbl[i-1]=label[n][i];
lbl[length-1]='\0';
length=0;
strcpy(address,"\0");
location=mloc[n]-exeloc;
loc=location;
count=0;
while(length<mlen[n])
{s
trcat(address,obcode[location++].code);
count++;
length+=2;
}
for(i=0;i<num;i++)
if(strcmp(lbl,estab[i].sname)==0)
break;
switch(operation)
{ case '+':newadd=strtol(address,&add1,10)+(long int)estab[i].padd;
break;
case '-':newadd=strtol(address,&add1,10)-(long int)estab[i].padd;
break;
} ltoa(newadd,address,10);
x=0;
y=0;
while(count>0)
{
obcode[loc].code[x++]=address[y++];
if(x>1)
{
x=0;
loc++;
count--;
}}} count=0;
n=0;
fprintf(fp3,"%d\t",obcode[0].add);
for(i=0;i<inc;i++)
```

Notes

```
{
fprintf(fp3,"%s",obcode[i].code);
n++;
if(n>3)
{
fprintf(fp3,"%s",obcode[i].code);
n++;
if(n>3)
fprintf(fp3,"\t");
n=0;
count++;
}  if(count>3)
{
fprintf(fp3,"\n%d\t",obcode[i+1].add);
count=0;
}}
getch();
}
```

Self Assessment

Fill in the blanks:

4. Since the ..... loader requires to know the absolute (load time) values of some external symbols before it can carry out the alterations on address constants, it needs two passes.
5. The mainframe relocating loader can treat instructions precisely like ..... data.
6. The ..... records enclose the information essential to build the external symbol dictionary or symbol table.
7. ..... are symbols that can be referred away from the subroutine level.
8. The ..... records enclose blocks of data and the relative address at which the data is to be positioned.
9. The ..... record states the end of the object file.
10. The main function of ..... of a direct linking loader is to assign and assign each program location in core and create a symbol table filling in the values of the external symbols.
11. The main function of ..... is to load the definite program text and carry out the relocation modification of any address constants requiring to be altered.
12. The ..... is used to amass the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) record.
13. Each of the ESD record types is processed in a ..... way.

Notes

14. The definite relocated address of the address constant is computed as the sum of the PLA and the ADDR field stated on the ..... record.

15. The Program Load Address is augmented by the length of the segment and saved in ....., turning out to be the PLA for the next segment.

9.3 Summary

- By means of an absolute loading scheme the programmer and the assembler execute the tasks of location, relocation, and linking.
- The object file for absolute loader comprises of a series of text records completed by a transfer record.
- The direct linking loader processes programs produced by the assembler, FORTRAN compiler, or some other compiler.
- The ESD records enclose the information essential to build the external symbol dictionary or symbol table.
- External symbols are symbols that can be referred away from the subroutine level.
- The TXT records enclose blocks of data and the relative address at which the data is to be positioned.
- The END record states the end of the object file. If the assembler END record has a symbol in the operand field, it states a start of execution point for the whole program (all subroutines).
- The Global External Symbol Table (GEST) is used to amass the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) record.

9.4 Keywords

Absolute Loading: By means of an absolute loading scheme the programmer and the assembler execute the tasks of location, relocation, and linking.

External Symbols: External symbols are symbols that can be referred away from the subroutine level.

Global External Symbol Table: The Global External Symbol Table (GEST) is used to amass the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) record.

Object File: The object file for absolute loader comprises of a series of text records completed by a transfer record.

9.5 Review Questions

1. Describe functions of absolute loader.
2. Illustrate the concept of Design of an Absolute Loader.
3. Describe databases required by each pass of loader.
4. Explain the design of direct linking loader.
5. What are the various segments to the object file? Illustrate.

Notes

6. Discuss the format of Data Bases in the design of a direct linking loader.
7. What is purpose of ID number on ESD cards? Why it is not needed for locally defined symbols?
8. Describe the function of each of the RLS, ESD, TXT and END records.
9. Illustrate the different types of external symbols.
10. What is Global External Symbol Table? Discuss.

Answers: Self Assessment

1.	absolute	2.	loader.
3.	object file	4.	direct-linking
5.	non-relocatable	6.	ESD
7.	External symbols	8.	TXT
9.	END	10.	pass 1
11.	pass 2	12.	Global External Symbol Table (GEST)
13.	different	14.	RLD
15.	SLENGTH		

9.6 Further Readings

Books

Chattopadhyay, *System Software*, PHI Learning Pvt. Ltd.

Donovan, *Systems Programming*, Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming*, Technical Publications

M. Joseph, *System Software*, Firewall Media

Online link

www.csie.ntu.edu.tw/~pjcheng/course/asm2008/asm_linkload.pdf

Notes

Unit 10: Programming Languages Concept (I)

CONTENTS

Objectives
Introduction
10.1 Programming Languages
10.1.1 Need for Computer Languages
10.1.2 Selection Criteria of Programming Language
10.1.3 Classification of Programming Languages
10.1.4 Generation of Programming Languages
10.2 High Level Languages
10.2.1 Importance of High Level Languages
10.2.2 Popular High Level Programming Languages
10.2.3 Features of High Level Languages
10.3 Data Types
10.3.1 Primitive Data Types
10.3.2 User-defined Data Types
10.3.3 Structure Data Types
10.4 Data Structures
10.5 Storage Allocation
10.5.1 Segments and Addresses
10.5.2 Simple Storage Layout
10.5.3 Multiple Segment Types
10.5.4 Segment and Page Alignment
10.6 Variable
10.6.1 Accessing of Pointers
10.6.2 Accessing of Label Variable
10.7 Summary
10.8 Keywords
10.9 Review Questions
10.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concept of programming languages
- Discuss importance of high level languages

- Illustrate data types and data storage
- Understand storage allocation
- Discuss accessing of pointers and label variables

Notes

Introduction

Computer is an electronic device which works on the instructions provided by the user. As the computer does not understand natural language, it is required to provide the instructions in some computer - understandable language. Such a computer understandable language is known as Programming language.

A computer programming language consists of a set of symbols and characters, words, and grammar rules that permit people to construct instructions in the format that can be interpreted by the computer system.

10.1 Programming Languages

10.1.1 Need for Computer Languages

A language is a system of communication. A language consists of all the symbols, characters and usage rules that permit people to communicate with each other. The special language by which we can communicate with the computer is called computer language or programming language. A programming language is a systematic notation by which we describe computational process. Computational process means steps for solving a problem.

Understanding of computer software is imperfect without a basic knowledge of programming language. Programming languages allow the programmers and end users to develop the programs that are executed by a computer. In today's world, several computer languages exists. Some of these are created to serve a special purpose (like controlling a robot), while others are more flexible. General purpose tools that are suitable for many types of applications. However, every computer language must have instructions that fall into following categories like:

- Input/Output.
- Calculations/Text manipulation.
- Logic/Comparison.
- Storage/Retrieval.

10.1.2 Selection Criteria of Programming Language

- Language should be simple and best known to programmers.
- It should perform well in the concerning application area.
- It should be able to support latest programming paradigm like structured programming, modular programming, etc.
- There should be an ease of program modification as to err is human.
- Language should supported by standardization and could be able to improvement or self enhancement.

There is no best programming language any more than there is a best natural language. As said by Charles V "I speak Spanish to God, Italian to women, French to men and German to my horse."

Notes

Caution A programming language must be chosen according to the purpose intended.

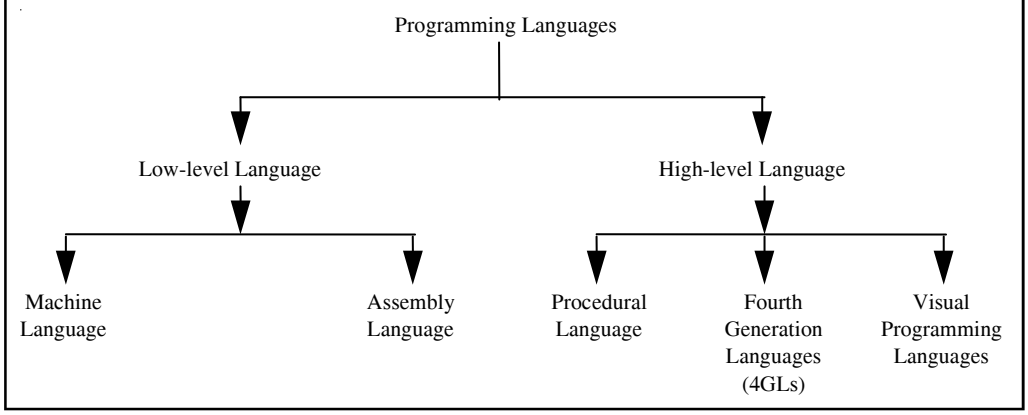
10.1.3 Classification of Programming Languages

Though all programming languages have an instruction set that permits these familiar operations to be performed, but there is a marked difference found in the symbols, characters and syntax of computer languages, computer languages can be classified broadly into the following categories:

- First generation languages/Lowest level languages/Machine language.
- Second generation languages/Low level language/Assembly language.
- Third generation language/High level language.
- Fourth generation language/Very high level language.
- Fifth generation language/Natural language.

Types of programming languages can be easily explained by the following figure:

Figure 10.1: Types of Programming Languages



The programming languages are said to be low or high or very high or natural, depending on how far these are from the internal architecture of the machine or how close they are to the user as far as the convenience of the user is concern. The languages that operates in close proximity of the external architecture of a machine are called as low level/lowest level languages. Machine language and assembly languages fall in this category.

The languages that put a user far away from the internal details of the computer, thereby creating more user friendly environment are called as high level or very high level or natural languages.

Generation wise categorization of the language is in the basis of their introduction to the programming world.

10.1.4 Generation of Programming Languages

On the basis of development, programming languages can be divided into 5 generations:

1.	First Generation Language	Machine Languages (1940-1950)
2.	Second Generation Language	Assembly Languages (1950-1958)

Contd...

Notes

3.	Third Generation Language	Procedural Languages (1958-1985)
4.	Fourth Generation Language	4GLs (1985 Onwards)
5.	Fifth Generation Language	Visual/Graphic Languages (1990 onwards)

Machine language and Assembly language are considered as first generation language and second generation language respectively. These languages have been discussed in the previous units.

Self Assessment

Fill in the blanks:

1. The special language by which we can communicate with the computer is called .....
2. The languages that operates in close proximity of the external architecture of a machine are called as .....

10.2 High Level Languages

Unlike the assembly program, that works with the specific make the development of mnemonic techniques and micro instructions that could work with different makes of computer with little modifications has led to the development of high level languages.

High level languages basically consist of English like instructions rather than mnemonic codes or binary digits of specific computer, so this language is easier to learn than assembly language. It requires less time to write as four or five low level instructions are produced to a simple high level statements.

These are the third generation languages. These are procedure-oriented languages and are machine independent. Programs are written in English like statements. As high level languages are not directly executable, translators (compilers and interpreters) are used to convert them in machine language equivalent.

Notes Naturally, a source program written in a high level language must also be translated into a machine usable code. A translating program that can perform this operation is called a compiler. A compiler may generate many lines of machine code for each source program statement.

10.2.1 Importance of High Level Languages

1. These are easier to learn than assembly language.
2. Less time is required to write programs.
3. These provide better documentation.
4. These are easier to maintain.
5. These have an extensive vocabulary.
6. Libraries of subroutines can be incorporated and used in many other programs.

Notes

7. Programs written in high level languages are easier to debug as translators display all the errors with proper error messages at the time of translation.
8. Programs written in high level languages are largely machine independent. Therefore, programs developed on one computer can be run on another with little or no modifications.

10.2.2 Popular High Level Programming Languages

ADA

ADA was named after Lady Augusta Ada Byron (the first computer programmer). It was designed by the US Defence Department for its real time applications. It is suitable for parallel processing.

APL

Developed by Dr Kenneth Aversion at IBM, APL is a convenient, interactive programming language suitable for expressing complex mathematical expressions in compact formats. It requires special terminals for use.

BASIC (Beginners All-purpose Symbolic Instruction Code)

BASIC was developed by John Kemeny & Thomas Karthy at Dartmouth College. It is a widely known and accepted programming language. It is easy to use and is almost coded in real-time conversational mode. This language provides good error diagnostics but has no self-structuring or self-documentation.

Pascal

Developed in 1968, Pascal was named after a French inventor Blaise Pascal and was developed by a SWISS programmer NiKolus Wirth. Pascal was the first structured programming language and it is used for both scientific and business data processing applications.

FORTRAN (FORmula TRANslation)

- Developed by IBM in 1957, it is one of the oldest and most widely used high level languages. It is widely used by scientists and engineers as this language has huge libraries of engineering and scientific programs.
- This language is suitable for expressing formulae, writing equations, and performing iterative calculations.

  Various versions of FORTRAN are:

  FORTRAN I (1957)

  FORTRAN II (1958)

  FORTRAN IV (1962)

  FORTRAN 77 (1978)

COBOL (Common Business Oriented Language)

COBOL is a structured and self documented language. It was developed by a committee of business, industry, government, and academic representatives called CODASYL (COnference

Notes

on DAta SYstem Languages) commissioned by US government in 1959. Statements of COBOL language resemble English language expressions and it makes them easy to understand and use.

C

Developed by Denis Ricthie at the Bell Laboratories in 1970, it is a general purpose programming language, which features economy of expression, modern control flow and data structures, and a rich set of operators. Its programs are highly portable (machine independent).

Did u know? C language supports modular programming through the use of functions, subroutines, and macros.

LISP (LISt Processor)

Developed in 1960 by Prof. John McGrthy, Lisp and Prolog (PROgramming LOGic) are the primary languages used in artificial intelligence research and applications.

RPG (Report Program Generator)

RPG is an important business oriented programming language developed by IBM in late 1960s. It is primarily used for preparing written reports. Advantages:

(a) RPG is problem oriented.

(b) It is easy to learn and use.

(c) Limited programming skills are required.

10.2.3 Features of High Level Languages

Abstraction

- ***Data Abstraction:*** The machine's illustration of information is unknown.
- ***Computation Abstraction:*** Machine instructions are concealed by higher-level operations and procedural abstraction.

Precise Representation

Algorithms articulated in a form appropriate for computer processing. Syntax** and Semantics must be followed.

Translation

- ***Compiled:*** Program text is transformed to executable code, which is then implemented.
- ***Interpreted:*** Program text is scrutinized and implemented one step at a time.

Self Assessment

Fill in the blanks:

3. High level languages are procedure-oriented languages and are ..... independent.

Notes

4. ..... language is suitable for expressing formulae, writing equations, and performing iterative calculations.

10.3 Data Types

10.3.1 Primitive Data Types

The kind of data that variables may hold is called as Data type. There are four basic data types used in both languages:

Data Types

	Pascal	C
(i)	char	char
(ii)	integer	int
(iii)	real	float
(iv)	boolean	double

1. char a character
2. int an integer, in the range -32,767 to 32,767
3. long int a larger integer (up to +-2,147,483,647)
4. float a floating-point number
5. double a floating-point number, with more precision and perhaps greater range than float.

The type char

This type is defined as the set of characters available for the language. A particular value of type char is denoted by enclosing the characters in single quotes.

For example,

'A', '4', '+', '?'

The type int

This type represents any positive and negative number without a decimal point.

For example,

5, -25, -67, 100005

The type real & float

This type allows any signed or unsigned numbers having a decimal point.

For example,

10.367, -0.893, +2., 25.0,

Notes

The type boolean

This type allows only two values true or false. So a boolean variables can assume one of two values true or false.

The type double

It is an advancement of the type float. In this case, the length of signed or unsigned decimal numbers can be larger than that of float type. These are called as double precision numbers.

Example: of this data type are:

23512.6732469, -6712345.98765

10.3.2 User-defined Data Types

Both languages provide facilities for declaring and defining new data types by the user.

In PASCAL, these are:

(i) Enumerated data type

(ii) Subrange data type

In case of language C, these are:

1. Enumerated data type
2. By using typedef statement

Enumerated Data Type

An enumerated data type is a user created data type in which the values, appropriate to the datatype, are specified in a user-defined list. The syntax is:

PASCAL:

Type Enumerated-type-identifier (i1, i2, i3,à.in);

Type	:	Reserved word
Enumerated-type-identifier	:	Identifier
i1, i2, i3,à.in	:	Item list

The item-list constitutions the allowed values which the enumerated type variable can assume. For example,

Type

MONTHS= (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC);

Enumerated data type item list

Now, variables of type MONTHS can be defined as

Var A, B, C: MONTHS;

Thus A, B & C can have the values defined by the item-list.

Notes

C:

Enumerated-type-identifier (i1, i2, i3,à.in);

enum

item-list

Identifier for

enumerated type

reserved word

for example,

enum time of day {am, pm};

enumerated data type item-list

once variables have been declared as enumerated data types, they can only have the values defined in the item-list. For example,

enum time of day X, Y;

enum time of day X, Y;

data - type variables

declares the variables X and Y to be of type time of day and can assume values from the item-list.

Subrange Data Type

When a data item takes a range of values which is a subrange of the values described by some existing type i.e. primitive or enumerated, its type may be defined as a subrange of that host type. The syntax is as follows:

type

type-identifier = lowerlimit..upperlimit

where lowerlimit and upperlimit must be constant, or previously defined constant identifier (of type integer) or item of an enumerated-type-list. Moreover, lowerlimit is always less than the upperlimit.

Example: of subrange type declaration are:

```
const
M = 15;
N = 2;
type
VALUE = 'N'.. 'M' ;
NUMBER = 1..100 ;
LETTER = 'A'.. 'Z' ;
Variables of these type may be declared as
Var
RESULT, answer : NUMBER;
```

Notes

X,Y, Z : VALUE;

W: LETTER;

Thus defined variables RESULT and answer can have values from 1 to 100, similarly X,Y, Z, can have from 2 to 15 and W can have from the list : 'A', 'B', 'C', … 'Z'.

The typedef Statement

In addition to creating new data types, C allows both standard and user-defined data types to be renamed using typedef statement. The syntax is:

Typedef data - type data - identifier

New name

Predefined data-type

Reserved word

Example:

Typedef float REAL;

Makes the name REAL a synonym for float. The name REAL can now be used in place of the term float. For example, the declaration

REAL val;

The typedef does not create a new data type; it creates a new name for an existing data type.

The #define statement is used to create synonym for an existing data type Similarly # define preprocessor command is used to equate constants and operators to create symbolic names.

Example:

#define PI 3,1416 - equate symbolic name PI to the number 3.1416.

#define TIMES * - equates symbolic name TIMES to the operator *. The substitutions are made by the C preprocessor just prior to program compilation. The symbolic name is referred to as a macro.

10.3.3 Structure Data Types

Structure data types are formed from primitive data types. Arrays, Records, Files are examples in PASCAL and Arrays, Structures, Unions, Files in C language. Records, Structures & Unions, etc., are discussed in the following section.

The RECORD and STRUCTURE

A collection of related data items (or fields), treated as a unit, is called a record. Whereas a set of related records is referred to as a file. To understand this concept, suppose a file of students at a NDIMCRC centre having data items registration no. (reg.no.), name , course & telephone.

Since array can have data-items only of identical type, the above information can be stored in record type structure. So in record type structure, one can store related the data-items which are distinct in nature and type.

Notes

REG NO. NAME COURSE TELEPHONE

STDREC à

{ { { {

Field Field Field Field

The type and variable declaration for the above record can be done as follows:

C:

Struct STDREC] ® Structure-type-identifier

{

int REGNO;

char NAME [15];

char course [8]; ® members

int TELEPHONE;

} SR1, SR2, SR#;] ® List of structure variables.

ARRAYS

So far, we've been declaring simple variables: the declaration

int i;

declares a single variable, named i, of type int. It is also possible to declare an array of several elements. The declaration

int a[10];

declares an array, named a, consisting of ten elements, each of type int. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric subscript. (Arrays in programming are similar to vectors or matrices in mathematics.)

In C, arrays are zero-based: the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is a[0], the second element is a[1], etc. You can use these "array subscript expressions" anywhere you can use the name of a simple variable.

Example:

a[0] = 10;

a[1] = 20;

a[2] = a[0] + a[1];

Notice that the subscripted array references (i.e. expressions such as a[0] and a[1]) can appear on either side of the assignment operator.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression.

Example: it's common to loop over all elements of an array:

Notes

```
int i;
for(i = 0; i < 10; i = i + 1)
        a[i] = 0;
```

This loop sets all ten elements of the array a to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
a = 0;                /* WRONG */
```

and

```
int b[10];
b = a;                /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
int b[10];
for(i = 0; i < 10; i = i + 1)
        b[i] = a[i];
```

Remember that for an array declared

```
int a[10];
```

there is no element a[10]; the topmost element is a[9]. This is one reason that zero-based loops are also common in C. Note that the for loop

```
for(i = 0; i < 10; i = i + 1)
        ...
```

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has i set to 9. (The comparison i <= 9 would also work, but it would be less clear and therefore poorer style.)

In the little examples so far, we've always looped over all 10 elements of the sample array a. It's common, however, to use an array that's bigger than necessarily needed, and to use a second variable to keep track of how many elements of the array are currently in use.

Example: we might have an integer variable

```
int na;               /* number of elements of a[] in use */
```

Then, when we wanted to do something with a (such as print it out), the loop would run from 0 to na, not 10 (or whatever a's size was):

```
for(i = 0; i < na; i = i + 1)
        printf("%d\n", a[i]);
```

Naturally, we would have to ensure that na's value was always less than or equal to the number of elements actually declared in a.

Arrays are not limited to type int; you can have arrays of char or double or any other type.

Notes

Example: Here is a slightly larger example of the use of arrays. Suppose we want to investigate the behavior of rolling a pair of dice. The total roll can be anywhere from 2 to 12, and we want to count how often each roll comes up. We will use an array to keep track of the counts: a[2] will count how many times we've rolled 2, etc.

We'll simulate the roll of a die by calling C's random number generation function, rand(). Each time you call rand(), it returns a different, pseudo-random integer. The values that rand() returns typically span a large range, so we'll use C's modulus (or "remainder") operator % to produce random numbers in the range we want. The expression rand() % 6 produces random numbers in the range 0 to 5, and rand() %

6 + 1 produces random numbers in the range 1 to 6.

Here is the program:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
        int i;
        int d1, d2;
        int a[13];    /* uses [2..12] */
        for(i = 2; i <= 12; i = i + 1)
                a[i] = 0;
        for(i = 0; i < 100; i = i + 1)
                {
                d1 = rand() % 6 + 1;
                d2 = rand() % 6 + 1;
                a[d1 + d2] = a[d1 + d2] + 1;
                }
        for(i = 2; i <= 12; i = i + 1)
                printf("%d: %d\n", i, a[i]);
        return 0;
}
```

We include the header <stdlib.h> because it contains the necessary declarations for the rand() function. We declare the array of size 13 so that its highest element will be a[12]. (We're wasting a[0] and a[1]; this is no great loss.) The variables d1 and d2 contain the rolls of the two individual dice; we add them together to decide which cell of the array to increment, in the line

```
        a[d1 + d2] = a[d1 + d2] + 1;
```

After 100 rolls, we print the array out. Typically (as craps players well know), we'll see mostly 7's, and relatively few 2's and 12's.

By the way, it turns out that using the % operator to reduce the range of the rand function is not always a good idea.

Notes

Array Initialization

Although it is not possible to assign to all elements of an array at once using an assignment expression, it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array.

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0.

Example:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6};
```

would initialize a[7], a[8], and a[9] to 0. When an array definition includes an initializer, the array dimension may be omitted, and the compiler will infer the dimension from the number of initializers.

Example:

```
int b[] = {10, 11, 12, 13, 14};
```

would declare, define, and initialize an array b of 5 elements (i.e. just as if you'd typed int b[5]). Only the dimension is omitted; the brackets [] remain to indicate that b is in fact an array.

In the case of arrays of char, the initializer may be a string constant:

```
char s1[7] = "Hello,";

char s2[10] = "there,";

char s3[] = "world!";
```

As before, if the dimension is omitted, it is inferred from the size of the string initializer. (We haven't covered strings in detail yet--we'll do so in chapter 8--but it turns out that all strings in C are terminated by a special character with the value 0. Therefore, the array s3 will be of size 7, and the explicitly-sized s1 does need to be of size at least 7. For s2, the last 4 characters in the array will all end up being this zero-value character.)

Arrays of Arrays ("Multidimensional" Arrays)

When we said that "Arrays are not limited to type int; you can have arrays of... any other type," we meant that more literally than you might have guessed. If you have an "array of int," it means that you have an array each of whose elements is of type int. But you can have an array each of whose elements is of type x, where x is any type you choose. In particular, you can have an array each of whose elements is another array! We can use these arrays of arrays for the same sorts of tasks as we'd use multidimensional arrays in other computer languages (or matrices in mathematics). Naturally, we are not limited to arrays of arrays, either; we could have an array of arrays of arrays, which would act like a 3-dimensional array, etc.

The declaration of an array of arrays looks like this:

```
int a2[5][7];
```

Notes

You have to read complicated declarations like these "inside out." What this one says is that a2 is an array of 5 somethings, and that each of the somethings is an array of 7 ints. More briefly, "a2 is an array of 5 arrays of 7 ints," or, "a2 is an array of array of int." In the declaration of a2, the brackets closest to the identifier a2 tell you what a2 first and foremost is. That's how you know it's an array of 5 arrays of size 7, not the other way around. You can think of a2 as having 5 "rows" and 7 "columns," although this interpretation is not mandatory. (You could also treat the "first" or inner subscript as "x" and the second as "y." Unless you're doing something fancy, all you have to worry about is that the subscripts when you access the array match those that you used when you declared it, as in the examples.)

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array a2 using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)
                a2[i][j] = 10 * i + j;
        }
```

This pair of nested loops sets a[1][2] to 12, a[4][1] to 41, etc. Since the first dimension of a2 is 5, the first subscripting index variable, i, runs from 0 to 4. Similarly, the second subscript varies from 0 to 6.

We could print a2 out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)
                printf("%d\t", a2[i][j]);
        printf("\n");
        }
```

(The character \t in the printf string is the tab character.)

Just to see more clearly what's going on, we could make the "row" and "column" subscripts explicit by printing them, too:

```
for(j = 0; j < 7; j = j + 1)
        printf("\t%d:", j);
printf("\n");

for(i = 0; i < 5; i = i + 1)
        {
        printf("%d:", i);
        for(j = 0; j < 7; j = j + 1)
                printf("\t%d", a2[i][j]);
        printf("\n");
        }
```

Notes

This last fragment would print

	0:	1:	2:	3:	4:	5:	6:
0	0	1	2	3	4	5	6
1:	10	11	12	13	14	15	16
2:	20	21	22	23	24	25	26
3:	30	31	32	33	34	35	36
4:	40	41	42	43	44	45	46

Finally, there's no reason we have to loop over the "rows" first and the "columns" second; depending on what we wanted to do, we could interchange the two loops, like this:

```
for(j = 0; j < 7; j = j + 1)
        {
        for(i = 0; i < 5; i = i + 1)
                printf("%d\t", a2[i][j]);
        printf("\n");
        }
```

Notice that i is still the first subscript and it still runs from 0 to 4, and j is still the second subscript and it still runs from 0 to 6.

Task What are Structure Data Types? Give examples.

Self Assessment

Fill in the blanks:

5. The kind of data that variables may hold is called as .....
6. An ..... data type is a user created data type in which the values, appropriate to the data type, are specified in a user-defined list.
7. Structure data types are formed from ..... data types.

10.4 Data Structures

All high level languages share a set of intercepted structure of data structure that generates the languages. These normal data structures are strings, arrays, I/O, Stacks, Queues, Linked Lists, Trees, Graphs, Hash tables, and Vectors. Based on the architectural reasons of the high level languages, these data structures are more complicated in design and more expressive in the languages whose reasons are more defined or drawn out toward computing a set of troubles.

Caution A programmer must own a practiced knowledge of a programming language data structure so as to fully discover all of the properties of the language in the execution of an application.

Each computer programming language that was ever designed belongs to one of these classes of paradigm that defines its reason: imperative, object-oriented, event-driven, simultaneous distributed, generic, array-oriented, procedural, reflective, and functional.

Notes

An essential paradigm is identical with procedural pattern, which is the notion that each statement is calculated up and down and every state modifies the state of the programs to appear at the preferred results. The Object Oriented Paradigm breaks down every sub-problem and locates their commonality and considers them as objects that can be encapsulated and modularized.

Notes Each of the above paradigms is articulated in each programming languages for the reason of solving dissimilar problem domain that frequently occurs in the field of computer programming.

Self Assessment

Fill in the blanks:

8. An essential paradigm is identical with ..... pattern, which is the notion that each statement is calculated up and down and every state modifies the state of the programs to appear at the preferred results.
9. The ..... Paradigm breaks down every sub-problem and locates their commonality and considers them as objects that can be encapsulated and modularized.

10.5 Storage Allocation

A linker or loader's first main task is storage allocation. Once storage is assigned, the linker can carry on to following stages of symbol binding and code fixups. Most of the symbols defined in a linkable object file are defined comparative to storage regions inside the file, so the symbols cannot be resolved until the areas addresses are recognized.

As is the case with many other features of linking, the basic concerns in storage allocation are straightforward, but the details to manage peculiarities of computer architecture and programming language semantics (and the interactions among the two) can get complicated. Most of the job of storage allocation can be managed in an elegant and relatively architecture-independent manner, but there are perpetually some details that need ad hoc machine specific hackery.

10.5.1 Segments and Addresses

Each object or executable file utilizes a model of the target address space. Generally, the target is the target computer's application address space, but there are cases where it's something else, such as a shared library. The fundamental concern in a relocating linker or loader is to make sure that all the sections in a program are defined and have addresses, but that addresses don't overlie where they're not supposed to.

Each of the linker's input files comprises a set of segments of numerous types. Dissimilar kinds of segments are considered in dissimilar manners Most usually all sections of a particular type like executable code, are concatenated into a single segment in the output file. At times segments are merged one on top of another, as for Fortran common blocks, and in an rising number of cases, for shared libraries and C++ particular traits, the linker itself requires to generate some segments and lay them out.

Did u know? Storage layout is a two-pass process, as the location of every segment can't be allocated until the sizes of all segments that logically precede it are recognized.

Notes

10.5.2 Simple Storage Layout

In an uncomplicated but not impractical situation, the input to a linker includes a set of modules, call them M1 through Mn, each of which includes a single segment starting at location 0 of length L1 through Ln, and the target address space also begins at zero.

The linker or loader inspects each module in turn, allocating storage in sequence. The beginning address of Mi is the sum of L1 through Li-1, and the length of the linked program is the sum of L1 through Ln.

Most architectures need that data be aligned on word boundaries, or at least run faster if data is aligned, so linkers usually round each Li up to a multiple of the most stringent position that the architecture needs, usually 4 or 8 bytes.

Example: Presume a main program known as main is to be linked with three subroutines called calif, mass, and newyork. (It assigns venture capital geographically). The sizes of every routine are (in hex): l r. name size _ main 1017 calif 920 mass 615 newyork 1390 Suppose that storage allocation begins at location 1000 hex, and that the alignment is four bytes. Then the assignments might be: l r. name location _ main 1000 - 2016 calif 2018 - 2937 mass 2938 - 2f4c newyork 2f50 - 42df Due to alignment, one byte at 2017 and three bytes at 2f4d are wasted, not sufficient to worry about.

10.5.3 Multiple Segment Types

In all but the easiest object formats, there are numerous kinds of segment, and the linker requires to group matching segments from all of the input modules jointly. On a Unix system with text and data segments, the linked file is required to have all of the text composed together, followed by all of the data, followed reasonably by the BSS. (Although the BSS doesn't take space in the output file, it is required to have space assigned to resolve BSS symbols, and to designate the size of BSS to allocate when the output file is loaded.) This needs a two-level storage allocation strategy.

Now every module Mi has text size Ti, data size Di, and BSS size Bi.

As it reads every input module, the linker assigns space for each of the Ti, Di, and Bi as though each segment were separately assigned at zero. After reading all of the input files, the linker now recognizes the total size of each of the three segments, Ttot, Dtot, and Btot. As the data segment follows the text segment, the linker adds Ttot to the address allocated for each of the data segments, and as the BSS segment follows both the text and data segments, the linker adds the sum of Ttot and Dtot to the allocated BSS segments.

Again, the linker typically needs to round up every assigned size.

10.5.4 Segment and Page Alignment

If the text and data segments are loaded into disconnected memory pages, as is usually the case, the size of the text segment has to be rounded up to a full page and the data and BSS segment positions equally adjusted. Many Unix systems use a hoax that saves file space by beginning the data instantly after the text in the object file, and mapping that page in the file into virtual memory twice, once read-only for the text and once copy-on-write for the data. Then, the data addresses logically begin precisely one page beyond the end of the text, so instead of rounding up, the data addresses start exactly 4K or whatever the page size is beyond the end of the text.

Notes

Example: We expand on Example 1 so that every routine has a text, data, and bss segment. The word alignment remains 4 bytes, but the page size is 0x1000 bytes. l r r r. name text data bss _ main 1017 320 50 calif 920 217 100 mass 615 300 840 newyork 1390 1213 1400 (all numbers hex)

The linker first lays out the text, then the data, then the bss. Observe that the data section begins on a page boundary at 0x5000, but the bss starts immediately after the data, since at run time data and bss are rationally one segment. l r r r. name text data bss _ main 1000 - 2016 5000 - 531f 695c - 69ab calif 2018 - 2937 5320 - 5446 69ac - 6aab mass 2938 - 2f4c 5448 - 5747 6aac - 72eb newyork 2f50 - 42df 5748 - 695a 72ec - 86eb There's wasted space at the end of the page between 42e0 and 5000. The bss segment ends in mid-page at 86eb, but usually programs assign heap space beginning instantly after that.

Self Assessment

Fill in the blanks:

10. Most of the symbols defined in a ..... object file are defined comparative to storage regions inside the file, so the symbols cannot be resolved until the areas addresses are recognized.
11. Each ..... file utilizes a model of the target address space.
12. The fundamental concern in a ..... linker or loader is to make sure that all the sections in a program are defined and have addresses, but that addresses don't overlie where they're not supposed to.

10.6 Variable

It is an identifier, which refers to computer memory space where some value can be stored. A variable can have different values at different stages of the program execution.

All variables must be declared before their use in a program. You may wonder why variables must be declared before use. There are two reasons:

It makes things somewhat easier on the compiler, it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.

It forces a bit of useful discipline on the programmer: you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

The syntax and examples of variable declaration are:

Example:

```
int real;

char choice;
```

A declaration for a variable can also contain an initial value. This initializer consists of an equals sign and an expression, which is usually a single constant:

```
int i = 1;

int i1 = 10, i2 = 20;
```

Notes

10.6.1 Accessing of Pointers

Pointers are often thought to be the most difficult aspect of C. It's true that many people have various problems with pointers, and that many programs founder on pointer-related bugs. Actually, though, many of the problems are not so much with the pointers per se but rather with the memory they point to, and more specifically, when there isn't any valid memory, which they point to. As long as you're careful to ensure that the pointers in your programs always point to valid memory, pointers can be useful, powerful, and relatively trouble-free tools.

A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type "pointer to int," it might point to the int variable i, or to the third cell of the int array a. Given a pointer variable, we can ask questions like, "What's the value of the variable that this pointer points to?"

When we declare a variable

int a = 10;

In the main memory following things happen

a 10

1001

where a is the name of the variable & 1001 is the address of the variable in the memory.

If we declare a variable to store 1001, that variable is known as a pointer. Thus a pointer is a variable that stores the address of another variable.

Basic Pointer Operations

The first things to do with pointers are to declare a pointer variable, set it to point somewhere, and finally manipulate the value that it points to. A simple pointer declaration looks like this:

```
int *ip;
```

This declaration looks like our earlier declarations, with one obvious difference: that asterisk. The asterisk means that ip, the variable we're declaring, is not of type int, but rather of type pointer-to-int. (Another way of looking at it is that *ip, which as we'll see is the value pointed to by ip, will be an int.)

We may think of setting a pointer variable to point to another variable as a two-step process: first we generate a pointer to that other variable, then we assign this new pointer to the pointer variable. We can say (but we have to be careful when we're saying it) that a pointer variable has a value, and that its value is "pointer to that other variable". This will make more sense when we see how to generate pointer values.

Pointers (that is, pointer values) are generated with the "address-of" operator &, which we can also think of as the "pointer-to" operator. We demonstrate this by declaring (and initializing) an int variable i, and then setting ip to point to it:

```
int i = 5;

ip = &i;
```

The assignment expression ip = &i; contains both parts of the "two-step process": &i generates a pointer to i, and the assignment operator assigns the new pointer to (that is, places it "in") the variable ip. Now ip "points to" i.

Notes

We discover the value pointed to by a pointer using the “contents-of” operator, *. Placed in front of a pointer, the * operator accesses the value pointed to by that pointer. In other words, if ip is a pointer, then the expression *ip gives us whatever it is that's in the variable or location pointed to by ip.

Example: We could write something like

```
printf("%d\n", *ip);
```

which would print 5, since ip points to i, and i is (at the moment) 5.

(You may wonder how the asterisk * can be the pointer contents-of operator when it is also the multiplication operator. There is no ambiguity here: it is the multiplication operator when it sits between two variables, and it is the contents-of operator when it sits in front of a single variable. The situation is analogous to the minus sign: between two variables or expressions it's the subtraction operator, but in front of a single operator or expression it's the negation operator. Technical terms you may hear for these distinct roles are unary and binary: a binary operator applies to two operands, usually on either side of it, while a unary operator applies to a single operand.)

The contents-of operator * does not merely fetch values through pointers; it can also set values through pointers. We can write something like

```
*ip = 7;
```

which means ``set whatever ip points to 7.'' Again, the * tells us to go to the location pointed to by ip, but this time, the location isn't the one to fetch from--we're on the left-hand sign of an assignment operator, so *ip tells us the location to store to.

If we called printf("%d\n", *ip) again, it would now print 7.

At this point, you may be wondering why we're going through this rigmarole--if we wanted to set i to 7, why didn't we do it directly? We'll begin to explore that next, but first let's notice the difference between changing a pointer (that is, changing what variable it points to) and changing the value at the location it points to. When we wrote *ip = 7, we changed the value pointed to by ip, but if we declare another variable j:

```
int j = 3;
```

and write

```
ip = &j;
```

we've changed ip itself

We have to be careful when we say that a pointer assignment changes ''what the pointer points to.'' Our earlier assignment

```
*ip = 7;
```

changed the value pointed to by ip, but this more recent assignment

```
ip = &j;
```

has changed what variable ip points to. It's true that ''what ip points to'' has changed, but this time, it has changed for a different reason. Neither i (which is still 7) nor j (which is still 3) has changed. (What has changed is ip's value.) If we again call

```
printf("%d\n", *ip);
```

this time it will print 3.

Notes

We can also assign pointer values to other pointer variables. If we declare a second pointer variable:

```
int *ip2;
```

then we can say

```
ip2 = ip;
```

Now ip2 points where ip does;

Now, if we set ip to point back to i again:

```
ip = &i;
```

We can now see that the two assignments

```
ip2 = ip;
```

and

```
*ip2 = *ip;
```

do two very different things. The first would make ip2 again point to where ip points (in other words, back to i again). The second would store, at the location pointed to by ip2, a copy of the value pointed to by ip; in other words (if ip and ip2 still point to i and j respectively) it would set j to i's value, or 7.

It's important to keep very clear in your mind the distinction between a pointer and what it points to. The two are like apples and oranges (or perhaps oil and water); you can't mix them. You can't ``set ip to 5'' by writing something like

```
ip = 5;                         /* WRONG */
```

5 is an integer, but ip is a pointer. You probably wanted to ``set the value pointed to by ip to 5,'' which you express by writing

```
*ip = 5;
```

Similarly, you can't ``see what ip is'' by writing

```
printf("%d\n", ip);             /* WRONG */
```

Again, ip is a pointer-to-int, but %d expects an int. To print what ip points to, use

```
printf("%d\n", *ip);
```

Finally, a few more notes about pointer declarations. The * in a pointer declaration is related to, but different from, the contents-of operator *. After we declare a pointer variable

```
int *ip;
```

the expression

```
ip = &i
```

sets what ip points to (that is, which location it points to), while the expression

```
*ip = 5
```

sets the value of the location pointed to by ip. On the other hand, if we declare a pointer variable and include an initializer:

```
int *ip3 = &i;
```

Notes

we're setting the initial value for ip3, which is where ip3 will point, so that initial value is a pointer. (In other words, the * in the declaration int *ip3 = &i; is not the contents-of operator, it's the indicator that ip3 is a pointer.)

If you have a pointer declaration containing an initialization, and you ever have occasion to break it up into a simple declaration and a conventional assignment, do it like this:

```
int *ip3;

ip3 = &i;
```

Don't write

```
int *ip3;

*ip3 = &i;
```

or you'll be trying to mix oil and water again.

Also, when we write

```
int *ip;
```

although the asterisk affects ip's type, it goes with the identifier name ip, not with the type int on the left. To declare two pointers at once, the declaration looks like

```
int *ip1, *ip2;
```

Some people write pointer declarations like this:

```
int* ip;
```

This works for one pointer, because C essentially ignores whitespace. But if you ever write

```
int* ip1, ip2;                /* PROBABLY WRONG */
```

it will declare one pointer-to-int ip1 and one plain int ip2, which is probably not what you meant.

What is all of this good for? If it was just for changing variables like i from 5 to 7, it would not be good for much. What it's good for, among other things, is when for various reasons we don't know exactly which variable we want to change, just like the bank didn't know exactly which club member it wanted to send the statement to.

Pointer Variable

Pointer variable is a variable which holds the address of another variable. This concept allows the indirect access to the objects. A variable can be declared as pointer variable and can point to the starting byte address of any data-type variable. Thus, the value associated with the pointer variables is not the content of the memory locations to which it points, but the memory address of that location.

Following is the way to declare and refer a pointer type variable:

C:

Base-type * pointer -variable-identifier;

The address of any variable can be got by preceding the variable name by an ampersand (&).

Notes

Example:

Char *C; Pointer-variable

&C - C is a variable, and &C is the address in which the value of C is stored.

*C - content of the location

10.6.2 Accessing of Label Variable

Labels are mainly used with a statement, so that the statement can be referenced in future during the execution.

Syntax : Label = prefix:

PASCAL: A label-prefix is a non-zero positive unsigned integer. The maximum size can be 4 digits.

C : A label-prefix can be any identifier but must be unique.

Task Illustrate how to declare and refer a pointer type variable.

Self Assessment

Fill in the blanks:

13. A ..... is an identifier, which refers to computer memory space where some value can be stored.
14. A ..... is a variable that points at, or refers to, another variable.
15. ..... are mainly used with a statement, so that the statement can be referenced in future during the execution.

10.7 Summary

- A language consists of all the symbols, characters and usage rules that permit people to communicate with each other.
- The programming languages are said to be low or high or very high or natural, depending on how far these are from the internal architecture of the machine or how close they are to the user as far as the convenience of the user is concern.
- High level languages basically consist of English like instructions rather than mnemonic codes or binary digits of specific computer, so this language is easier to learn than assembly language.
- The kind of data that variables may hold is called as Data type.
- All high level languages share a set of intercepted structure of data structure that generates the languages. These normal data structures are strings, arrays, I/O, Stacks, Queues, Linked Lists, Trees, Graphs, Hash tables, and Vectors.
- A linker or loader's first main task is storage allocation. Once storage is assigned, the linker can carry on to following stages of symbol binding and code fixups.

Notes

- A variable is an identifier, which refers to computer memory space where some value can be stored.
- A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type ``pointer to int,`` it might point to the int variable i, or to the third cell of the int array a.
- Labels are mainly used with a statement, so that the statement can be referenced in future during the execution.

10.8 Keywords

Data Type: The kind of data that variables may hold is called as Data type.

High Level Language: High level languages basically consist of English like instructions rather than mnemonic codes or binary digits of specific computer, so this language is easier to learn than assembly language.

Labels: Labels are mainly used with a statement, so that the statement can be referenced in future during the execution.

Pointer: A pointer is a variable that points at, or refers to, another variable. That is, if we have a pointer variable of type ``pointer to int,`` it might point to the int variable i, or to the third cell of the int array a.

Variable: A variable is an identifier, which refers to computer memory space where some value can be stored.

10.9 Review Questions

1. Give two instances of programming where you will prefer a high level language over a low level language.
2. What are high level languages? Illustrate the Importance of high level languages.
3. Discuss some popular high level programming languages.
4. Enlighten the various features of high level languages.
5. List the various data types. Enumerate with example.
6. What are User-defined Data Types? Illustrate different User-defined Data Types.
7. A programmer must own a practiced knowledge of a programming language data structure so as to fully discover all of the properties of the language in the execution of an application. Comment.
8. Explicate the concept of storage allocation with example.
9. Illustrate the process of accessing of pointers with examples.
10. Make distinction between Enumerated Data Type and subrange data type.

Answers: Self Assessment

1. programming language
2. low level/lowest level languages
3. machine
4. FORTRAN
5. Data type
6. enumerated

7.	primitive	8.	procedural
9.	Object Oriented	10.	linkable
11.	object or executable	12.	relocating
13.	variable	14.	pointer
15.	Labels		

Notes

10.10 Further Readings

Books

Chattopadhyay, *System Software*, PHI Learning Pvt. Ltd.

Donovan, *Systems Programming*, Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming*, Technical Publications

M. Joseph, *System Software*, Firewall Media

Online link

https://www.cs.uiowa.edu/~slonnegr/plc/

Notes

Unit 11: Programming Languages Concept (II)

CONTENTS

Objectives

Introduction

11.1 Functional Modularity

11.1.1 Advantages of Modular Approach

11.1.2 Top-down Approach

11.1.3 Bottom-up Approach

11.2 Asynchronous Operation

11.2.1 Beginning an Asynchronous Operation

11.2.2 Ending an Asynchronous Operation

11.2.3 Asynchronous Operations Conditions and Signals

11.3 Multitasking

11.4 Summary

11.5 Keywords

11.6 Review Questions

11.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the concept of functional modularity
- Discuss asynchronous operations
- Understand multitasking

Introduction

The real power of partitioning comes if a system is partitioned into modules so that the modules are solvable and modifiable separately. In this unit you will understand the concept of modularity. Also you will study the concept of Asynchronous Operations. Lastly we will discuss how to execute multitasking.

11.1 Functional Modularity

A system/program is considered modular if it consists of direct modules so that each modules can be separately implemented, and a change in one module has minimal impact on other modules. Before we discuss it in detail, let us define what we meant by a module.

A module is a logically separable part of a program. In other words, a set of related procedures with the data they manipulate is called a module. In terms of common programming language construct, a module can be a macro, a function, a procedure, a process or a package. It is based on the following principle:

"Decide which module you want; partition the program so that data is hidden in modules."

Notes

Caution To produce modular design, some criteria must be used to select modules so that the module support well defined abstractions and are solvable and modifiable separately.

A program can be logically separated into the following functional module:

- Initialization
- Input
- Input data validation
- Processing
- Output
- Error handling
- Closing Procedure.

The modules reflect a logical flow for a computer program. After initialization, processing proceeds logically with input, input validation, various processing modules and output. Error handling may be required during execution of any modules.

11.1.1 Advantages of Modular Approach

- Testing of individual modules in isolation makes tracing errors easier.
- Modules can be kept separately in a library and used anywhere in the program without rewriting them.
- A module can use other modules.
- The documentation of a large program is simplified by the documentation of individual modules.
- We can develop modules, which contain standard procedures throughout the program, thus saves development time.

A program consists of modules, which may have modules of their own: indeed a program is hierarchy of modules. Almost all design contains hierarchies, as creating a hierarchy is a natural way to manage complexity. Moreover the highest level modules correspond to the entire program. The question arises at this time is, "In what order should the modules be built-starting from the top level or starting from the Bottom level?" To design such hierarchies there are two approaches: Top down and Bottom up.

11.1.2 Top-down Approach

A top-down design approach starts by identifying the major modules of the program i.e. decomposing them into their lower level modules and iterating until the desired level of details is achieved. First the main module is implemented and then their subroutines and so on. Top down design method often results in some forms of stepwise refinement.

Starting from an abstract design, in each step the design is refined to a more concrete level until we reach a level where no more refinement is required and the design can be implemented directly.

Notes

Advantages of Top-down Approach

- Easy to trace the bug. As at each stage, the sub programs are tested individually and then the main is tested. If any errors occur, it will be probably in the related module, which makes the debugging process easy.
- It is advantageous to use top down approach when the design is not detailed enough.
- In case if some of the decisions have to be made during development, it could be easily down in top down implementation, i.e. decision making is easy.
- Using this approach, we can look at the program as a whole instantly, i.e. Easy Program's traversal.
- Each module can be independently worked out and elaborated as convenient.

Notes A top down approach is suitable only if the specifications of the system are clearly known and the system development is from the scratch. Hence, it is a reasonable approach if a stepwise type of proven model is being used.

11.1.3 Bottom-up Approach

A bottom-up design approach starts with designing the most basic or primitive module and proceeds to higher level module that use these lower level components. Firstly, the modules at the bottom of the hierarchy is implemented, then its higher levels are implemented until it reaches the top.

Bottom up method works with layer of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operation of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

Advantages of Bottom-up Approach

- It can handle the increasing complexity of program that are reliable and maintainable.
- It enables programmers to write moderately complex programs fairly easily.
- It is advantageous to use bottom up approach when designing the complex systems like operating system, networking software systems, etc.
- Bottom-up approach conceptualize layered architecture which is generally best for the implementation of complex program.
- More close to the real world as it starts from some existing components.

If a system has to develop from the existing system a bottom-up approach is more suitable. Without having a good idea about the operation needed at the higher level, it is difficult to determine what operations the current layer should support.

Pure top-down or pure bottom-up approaches are often not practical. In practice, in large systems, a combination of two approaches is used, called as hybrid approach. The top modules of the system gives some feedback regarding the functionality of the system and checks whether the "look and feel" of the system is OK. For this, it is best if development proceeds with top-down

Notes

approach. On the other hand, bottom level modules provide the basic operation used by higher level modules. It is therefore important to ensure that these bottom level modules are working correctly before these are used by other modules. This suggests that the development should proceed in a Bottom up manner. As both issues are important in a large project, so it may be best to follow a combination approach for such systems.

Self Assessment

Fill in the blanks:

1. A system/program is considered modular if it consists of ..... modules so that each modules can be separately implemented, and a change in one module has minimal impact on other modules.
2. A set of related procedures with the data they manipulate is called a .....
3. A ..... approach (Pure top-down or pure bottom-up) starts by identifying the major modules of the program.
4. A ..... approach starts with designing the most basic or primitive module and proceeds to higher level module that use these lower level components.
5. In practice, in large systems, a combination of two approaches is used, called as ..... approach.

11.2 Asynchronous Operation

An asynchronous operation that utilizes the IAsyncResult design pattern is executed as two methods named BeginOperationName and EndOperationName that start and end the asynchronous operation OperationName correspondingly.

Example: The FileStream class offers the BeginRead and EndRead methods to asynchronously read bytes from a file. These methods execute the asynchronous edition of the Read method.

After calling BeginOperationName, an application can carry on implementing instructions on the calling thread while the asynchronous operation occurs on a dissimilar thread. For every call to BeginOperationName, the application should also call EndOperationName to obtain the results of the operation.

Task Make distinction between BeginOperationName and EndOperationName.

11.2.1 Beginning an Asynchronous Operation

The BeginOperationName method starts asynchronous operation OperationName and returns an object that executes the IAsyncResult interface. IAsyncResult objects accumulate information regarding an asynchronous operation. The following table displays information regarding an asynchronous operation.

Notes

Member	Description
AsyncState	An optional application-specific object that contains information regarding the asynchronous operation.
AsyncWaitHandle	A WaitHandle that can be used to block application implementation until the asynchronous operation completes.
CompletedSynchronously	A value that indicates whether the asynchronous operation completed on the thread used to call **Begin***Operation Name* instead of completing on a separate ThreadPool thread.
IsCompleted	A value that signifies whether the asynchronous operation has finished.

A Begin Operation Name method takes any parameters stated in the signature of the synchronous edition of the method that are passed by value or by reference. Any out parameters are not part of the Begin Operation Name method signature. The BeginOperationName method signature also involves two additional parameters. The first of these defines an AsyncCallback delegate that references a technique that is called when the asynchronous operation finishes. The caller can specify null (Nothing in Visual Basic) if it does not want a technique invoked when the operation completes. The second additional parameter is a user-defined object. This object can be used to pass application-specific state information to the technique raised when the asynchronous operation completes. If a BeginOperationName method takes additional operation-specific parameters, like a byte array to amass bytes read from a file, the AsyncCallback and application state object are the last parameters in the BeginOperation Name method signature.

Begin OperationName returns control to the calling thread instantly. If the BeginOperationName technique throws exceptions, the exceptions are thrown previous to the asynchronous operation is started.

Did u know? If the BeginOperationName method throws exceptions, the callback method is not invoked.

11.2.2 Ending an Asynchronous Operation

The End Operation Name technique ends asynchronous operation OperationName. The return value of the EndOperationName method is the same type returned by its synchronous counterpart and is particular to the asynchronous operation.

Example: The EndRead method returns the number of bytes read from a FileStream and the EndGetHostByName method returns an IPHostEntry object that includes information regarding a host computer.

The EndOperationName method takes any out or ref parameters stated in the signature of the synchronous edition of the method. In addition to the parameters from the synchronous method, the EndOperationName method also contains an IAsyncResult parameter.

Caution Callers must pass the instance returned by the analogous call to BeginOperationName.

Notes

If the asynchronous operation demonstrated by the IAsyncResult object has not accomplished when EndOperationName is called, EndOperationName blocks the calling thread until the asynchronous operation is complete. Exceptions thrown by the asynchronous operation are thrown from the EndOperationName method. The result of calling the EndOperationName method various times with the same IAsyncResult is not defined. Similarly, calling the EndOperationName method with an IAsyncResult that was not returned by the associated Begin method is also not defined.

Caution For any of the undefined scenarios, implementers should consider throwing InvalidOperationException.

Notes Implementers of this design outline should notify the caller that the asynchronous operation accomplished by setting IsCompleted to true, calling the asynchronous callback method (if one was stated) and signaling the AsyncWaitHandle.

Application developers have numerous design options for processing the results of the asynchronous operation.

Did u know? The accurate choice is based on whether the application has directions that can implement while the operation finishes.

If an application cannot execute any additional work until it obtains the results of the asynchronous operation, the application must block until the results are obtainable. To obstruct until an asynchronous operation completes, you can utilize one of the following strategies:

- Call EndOperationName from the application's main thread, obstructing application execution until the operation is finished.
- Use the AsyncWaitHandle to obstruct application execution until one or more operations are complete.

Applications that do not necessitate blocking while the asynchronous operation accomplishes can use one of the following strategies:

- Poll for operation completion status by verifying the IsCompleted property periodically and calling EndOperationName when the operation is complete.
- Use an AsyncCallback delegate to state a method to be invoked when the operation is complete.

11.2.3 Asynchronous Operations Conditions and Signals

An asynchronous operation is initiated by a function that is named with the prefix async_. These functions will be pointed to as initiating functions.

All initiating functions take a function object meeting handler needs as the final parameter. These handlers accept as their first parameter an value of type const error_code.

Executions of asynchronous operations may call the Application Programming Interface (API) offered by the operating system. If such an operating system API call effect in an error, the handler will be invoked with a const error_code l value that evaluates to true. Or else the handler will be invoked with a const error_code l value that evaluates to false.

Notes

Unless otherwise observed, when the behaviour of an asynchronous operation is defined "as if" executed by a POSIX function, the handler will be invoked with a value of type error_code that matches to the failure condition depicted by POSIX for that function, if any. Or else the handler will be invoked with an implementation-defined error_code value that reflects the operating system error.

Asynchronous operations will not stop working with an error condition that signifies interruption by a signal (POSIX EINTR). Asynchronous operations will not fail with any error condition connected with non-blocking operations (POSIX EWOULDBLOCK, EAGAIN or EINPROGRESS; Windows WSAEWOULDBLOCK or WSAEINPROGRESS).

All asynchronous operations have an connected io_service object. Where the starting function is a member function, the connected io_service is that returned by the io_service() member function on the same object. Where the starting function is not a member function, the related io_service is that returned by theio_service() member function of the first argument to the starting function.

Arguments to starting functions will be considered as follows:

- If the parameter is stated as a const reference or by-value, the program is not needed to confirm the validity of the argument after the initiating function finishes. The execution may make copies of the argument, and all copies will be destroyed no later than instantly after invocation of the handler.
- If the parameter is stated as a non-const reference, const pointer or non-const pointer, the program must confirm the validity of the argument until the handler is invoked.

The library implementation is only allowed to make calls to an initiating function's arguments' copy constructors or destructors from a thread that fulfills one of the following conditions:

- The thread is implementing any member function of the related io_service object.
- The thread is implementing the destructor of the related io_service object.
- The thread is implementing one of the io_service service access functions use_service, add_service or has_service, where the first argument is the associated io_service object.
- The thread is implementing any member function, constructor or destructor of an object of a class defined in this clause, where the object's io_service() member function returns the associated io_service object.
- The thread is implementing any function defined in this clause, where any argument to the function has an io_service() member function that returns the associatedio_service object.

The above needs are proposed to avert these hidden threads from making calls to program code. This indicates that a program can, for instance, use thread-unsafe reference counting in handler objects, provided the program makes sure that all calls to an io_service and associated objects take place from the one thread.

The io_service object related with an asynchronous operation will have uncompleted work, as if by sustaining the subsistence of one or more objects of classio_service::work constructed by means of the io_service, until instantly after the handler for the asynchronous operation has been invoked.

When an asynchronous operation is accomplished, the handler for the operation will be invoked as if by:

1. Building a bound completion handler bch for the handler, as described below.
2. Calling ios.post(bch) to schedule the handler for delayed invocation, where ios is the associated io_service.

Notes

This shows that the handler must not be called directly from within the initiating function, although the asynchronous operation completes instantly.

A bound completion handler is a handler object that includes a copy of a user-supplied handler, where the user-supplied handler accepts one or more arguments. The bound completion handler does not accept any arguments, and includes values to be passed as arguments to the user-supplied handler. The bound completion handler forwards the handler_allocate(), handler_deallocate(), and handler_invoke() calls to the analogous functions for the user-supplied handler. A bound completion handler fulfills the requirements for a completion handler.

Example: A bound completion handler for a ReadHandler may be executed as follows:

```
template<class ReadHandler>
struct bound_read_handler
{
  bound_read_handler(ReadHandler handler, const error_code& ec, size_t s)
    : handler_(handler), ec_(ec), s_(s)
  {
  }

  void operator()()
  {
    handler_(ec_, s_);
  }

  ReadHandler handler_;
  const error_code ec_;
  const size_t s_;
};

template<class ReadHandler>
void* asio_handler_allocate(size_t size,
                    bound_read_handler<ReadHandler>* this_handler)
{
  using namespace boost::asio;
  return asio_handler_allocate(size, &this_handler->handler_);
}

template<class ReadHandler>
void asio_handler_deallocate(void* pointer, std::size_t size,
```

Notes

```
asio_handler_deallocate(pointer, size, &this_handler->handler_);
}
template<class F, class ReadHandler>
void handler_invoke(const F& f,
  bound_read_handler<ReadHandler>* this_handler)
{
  using namespace boost::asio;
handler_invoke(f, &this_handler->handler_);
}
```

If the thread that begins an asynchronous operation finishes before the related handler is invoked, the behaviour is implementation-defined. Particularly, onWindows versions prior to Vista, unfinished operations are cancelled when the initiating thread exits.

The handler argument to an initiating function defines a handler identity. Specifically, the original handler argument and any copies of the handler argument will be measured equivalent. If the implementation requires to allocate storage for an asynchronous operation, the execution will perform asio_handler_allocate(size, &h), where size is the required size in bytes, and h is the handler. The execution will carry out asio_handler_deallocate(p, size, &h), where p is a pointer to the storage, to deallocate the storage before the invocation of the handler via asio_handler_invoke. Numerous storage blocks may be assigned for a single asynchronous operation.

Task Illustrate the arguments to initiating functions.

Self Assessment

Fill in the blanks:

6. The ..... class offers the BeginRead and EndRead methods to asynchronously read bytes from a file.
7. For every call to BeginOperationName, the application should also call ..... to obtain the results of the operation.
8. The BeginOperationName method starts asynchronous operation OperationName and returns an object that executes the ..... interface.
9. If the BeginOperationNamemethod throws exceptions, the ..... method is not invoked.
10. An asynchronous operation is initiated by a function that is named with the prefix async_. These functions will be pointed to as .....
11. A ..... handler is a handler object that includes a copy of a user-supplied handler, where the user-supplied handler accepts one or more arguments.

11.3 Multitasking

Notes

To understand the power of assembly language we discuss how to execute multitasking. In the debugger our thread of instructions was broken by the debugger; it got the control, used all registers, exhibited an detailed interface, waited for the key, and then reinstated processor state to what was instantly before disruption. Our program resumed as if nothing happened. The program execution was in the similar logical flow.

If we have two dissimilar programs A and B. Program A is busted, its state saved, and returned to B rather than A. By looking at the instruction set, we can instantly say that nothing can prevent us from doing that. IRET will return to whatever CS and IP it locates on the stack. Now B is interrupted someway, its state saved, and we return back to A. A will have no way of recognizing that it was interrupted as its whole environment has been restored. It never knew the debugger took control when it was debugged. It sill has no way of gaining this knowledge. If this function of breaking and restoring programs is performed at high speed the user will sense that all the programs are executing simultaneously where actually they are being switched to and forth at high speed.

In real meaning multitasking is simple, even though we have to be enormously cautious when implementing it. The surroundings of a program in the very simple case is all its registers and stack. Now to obtain control from the program without the program knowing regarding it, we can utilize the IRQ 0 highest priority interrupt that is from time to time coming to the processor.

Example: Now we show a very basic example of multitasking. We have two subroutines written in assembly language. All the techniques discussed here are pertinent to code written in higher level languages also. However, the code to control this multitasking cannot be simply written in a higher level language so we write it in assembly language. The two subroutines rotate bars by altering characters at the two corners of the screen and have infinite loops. By hooking the timer interrupt and saving and reinstating the registers of the tasks one by one, it occurs that both tasks are executing concurrently.

Example:

```
001  ; elementary multitasking of two threads[org 0x0100] jmp start ;
002  ax,bx,ip,cs,flags storage areataskstates: dw 0, 0, 0, 0, 0 ; task0 regsdw 0, 0, 0, 0,
003  0 ; task1 regsdw 0, 0, 0, 0, 0 ; task2 regs current: db 0 ; index of current
004  taskchars: db '\|/-' ; shapes to form a bar ; one task to be multitaskedtaskone:
005  mov al, [chars+bx] ; read the next shapemov [es:0], al ; write at top left of
006  screen
007
008
009
010
011
012
013
014
015
```

Contd...

Notes

016 017 018 019	inc and jmp		bx bx, 3taskone	; increment to next shape; taking modulus by 4; infinite task
020 021 022 023 024 025 026	; second task to be multitaskedtasktwo: mov al, [chars+bx]mov [es:158], alinc bx and bx, 3jmp tasktwo			; read the next shape; write at top right of screen; increment to next shape; taking modulus by 4; infinite task
027 028 029 030	; timer interrupt service routinetimer: push axpush bx			
031 032 033 034 035	mov mov mul mov		bl, [cs:current]ax, 10bl bx, ax	; read index of current task; space used by one task; multiply to get start of task; load start of task in bx
036 037 038 039 040 041 042 043 044 045 046	popmov popmov popmov popmov popmov		ax ; read original value of bx[cs:taskstates+bx+2], ax ; space for current taskax ; read original value of ax[cs:taskstates+bx+0], ax ; space for current taskax ; read original value of ip[cs:taskstates+bx+4], ax ; space for current taskax ; read original value of cs[cs:taskstates+bx+6], ax ; space for current taskax ; read original value of flags[cs:taskstates+bx+8], ax ; space for current task	
047 048 049 050 051	inc byte [cs:current] ; update current task indexcmp byte [cs:current], 3 ; is task index out of rangejne skipreset ; no, proceedmov byte [cs:current], 0 ; yes, reset to task 0			
052 053 054 055 056	skipreset:	mov mov mul mov	bl, [cs:current]ax, 10bl bx, ax	; read index of current task; space used by one task; multiply to get start of task; load start of task in bx
057 058 059		mov out	al, 0x200x20, al	; send EOI to PIC

Notes

```
060 push word [cs:taskstates+bx+8] ; flags of new taskpush word
061 [cs:taskstates+bx+6] ; cs of new taskpush word [cs:taskstates+bx+4] ; ip of new
062 taskmov ax, [cs:taskstates+bx+0] ; ax of new task mov bx, [cs:taskstates+bx+2] ;
063 bx of new taskiret ; return to new task
064
065
066
067 start: mov word [taskstates+10+4], taskone ; initialize
068 mov ip[taskstates+10+6], cs ; initialize csword
069 mov [taskstates+10+8], 0×0200 ; initialize flagsword
070 mov [taskstates+20+4], tasktwo ; initialize
071 mov ip[taskstates+20+6], cs ; initialize csword
072 mov [taskstates+20+8], 0×0200 ; initialize flagsword
073 mov [current], 0 ; set current task index
074
075 xor mov cli ax, ax es, ax ; point es to IVT base
076
077
078 mov mov mov mov word [es:8*4], ; hook timer interrupt ;
079 xor sti timer[es:8*4+2], csax, point es to video base;
080 0xb800 es, axbx, bx initialize bx for tasks
081
082
083
084
085 jmp $ ; infinite loop
```

The space where all registers of a task are accumulated is known as the process control block or PCB. Actual PCB comprises a few more things that are not applicable to us now. INT 08 that is saving and restoring the registers is known as the scheduler and the whole event is known as a context switch.

Self Assessment

Fill in the blanks:

12. In real meaning multitasking is simple, even though we have to be enormously cautious when ..... it.
13. We prefer multitasking to understand the power of .....
14. If we have two ..... programs A and B. Program A is busted, its state saved, and returned to B rather than A.
15. To obtain control from the program without the program knowing regarding it, we can utilize the ..... highest priority interrupt that is from time to time coming to the processor.

Notes

11.4 Summary

- A system/program is considered modular if it consists of direct modules so that each modules can be separately implemented, and a change in one module has minimal impact on other modules.
- To produce modular design, some criteria must be used to select modules so that the module support well defined abstractions and are solvable and modifiable separately.
- A top-down design approach starts by identifying the major modules of the program i.e. decomposing them into their lower level modules and iterating until the desired level of details is achieved.
- A bottom-up design approach starts with designing the most basic or primitive module and proceeds to higher level module that use these lower level components.
- An asynchronous operation that utilizes the IAsyncResult design pattern is executed as two methods named BeginOperationName and EndOperationName that start and end the asynchronous operation OperationName correspondingly.
- If an application cannot execute any additional work until it obtains the results of the asynchronous operation, the application must block until the results are obtainable.
- An asynchronous operation is initiated by a function that is named with the prefix async_. These functions will be pointed to as initiating functions.
- In real meaning multitasking is simple, even though we have to be enormously cautious when implementing it.

11.5 Keywords

Asynchronous Operation: Asynchronous operation is initiated by a function that is named with the prefix async_. These functions will be pointed to as initiating functions.

Bottom-up Approach: A bottom up design approach starts with designing the most basic or primitive module and proceeds to higher level module that use these lower level components.

Top-down Approach: A top down design approach starts by identifying the major modules of the program i.e. decomposing them into their lower level modules and iterating until the desired level of details is achieved.

11.6 Review Questions

1. What is functional module? When a system/program is considered as modular? Illustrate.
2. Enlighten the advantages of Modular Approach.
3. Make distinction between Top-down approach and Bottom-up approach.
4. What are the methods that start and end the asynchronous operation? Discuss.
5. Illustrate the concept of beginning an Asynchronous Operation with example.
6. Elucidate the process of ending an Asynchronous Operation.
7. Depict the various conditions used in asynchronous operations.
8. Explain how to execute multitasking.

Notes

9. Apply top-down and bottom-up approaches to solve the following integral equation problem:

   i. $Z = \int x^2\, dx + \int 3p^4 dp - \int Cos3ydy$

10. Pure top-down or pure bottom-up approaches are often not practical. Comment.

Answers: Self Assessment

1.	direct	2.	module
3.	top down design	4.	bottom up design
5.	hybrid	6.	FileStream
7.	EndOperationName	8.	IAsyncResult
9.	callback	10.	initiating functions
11.	bound completion	12.	implementing
13.	assembly language	14.	dissimilar
15.	IRQ 0		

11.7 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online link

https://www.cs.uiowa.edu/~slonnegr/plc/

Notes

Unit 12: Formal Systems and Programming Languages

CONTENTS

Objectives

Introduction

12.1 Use of Formal Systems in Programming Languages

12.2 Formal Specifications

12.2.1 Advantages of Formal Specifications

12.2.2 Disadvantages of Formal Specifications

12.2.3 Notations

12.2.4 Developing a Simple Formal Specification

12.2.5 Pre and Post-Conditions

12.2.6 The Development of an Axiomatic Spec of a Function

12.2.7 The Use of an Error Predicate

12.2.8 The Use of Structure

12.2.9 Theoretical Aspects: Formally

12.2.10 Theoretical Aspects: Less Formally

12.3 Formal Grammars

12.3.1 Vocabulary

12.3.2 Parse Representation

12.3.3 More Definitions

12.4 Summary

12.5 Keywords

12.6 Review Questions

12.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the use of formal systems in programming languages
- Discuss the concept of formal specification
- Explain formal grammars

Introduction

In programming language semantics we deal with the thorough mathematical study of the meaning of programming languages. The meaning of a language is provided by a formal system that depicts the probable calculations expressible inside that language. In computer science and software engineering, formal methods are procedures for the specification, expansion and verification of software and hardware systems depending on formal systems.

Notes

A formal system comprises a formal language and a set of inference rules. The formal language includes primitive symbols that make up well formed formulas and the inference rules are utilized to obtain expressions from other expressions inside the formal system. A formal system may be formulated and considered for its intrinsic properties, or it may be proposed as a depiction (i.e. a model) of external phenomena. In order to be truly functional in computer science, we need our formal systems to be machine executable.

12.1 Use of Formal Systems in Programming Languages

Implementation concerned formally declared models can be measured as machine-independent specifications of program behavior. They can proceed as "yard sticks" for the accuracy of program conversions and optimizations.

Verification foundation of methods for analysis about program properties (e.g., equivalence) and program specifications (program correctness). Language Design can bring to light vagueness and unexpected subtleties in programming language builds.

Formal methods are system design methods that use thoroughly declared mathematical models to construct software and hardware systems. In distinction to other design systems, formal methods use mathematical evidence as a match to system testing so as to make sure correct behavior. As systems become more complex, and security becomes a more significant issue, the formal approach to system design provides another level of insurance.

Formal methods vary from other design systems via the use of formal verification systems, the fundamental principles of the system must be confirmed accurate before they are accepted. Traditional system design has utilized extensive testing to confirm behavior, but testing is competent of only finite conclusions. Dijkstra and others have illustrated that tests can only show the conditions where a system won't fail, but cannot say anything concerning the behavior of the system outside of the testing situations. In distinction, once a theorem is confirmed true it remains true.

Notes It is very significant to observe that formal verification does not prevent the necessitate for testing. Formal verification cannot fix bad suppositions in the design, but it can assist recognize errors in reasoning which would otherwise be left unconfirmed. In numerous cases, engineers have reported locating flaws in systems once they reviewed their designs formally

Approximately speaking, formal design can be viewed as a three step process, following the sketch given here:

1. ***Formal Specification:*** Throughout the formal specification phase, the engineer thoroughly defines a system by means of a modeling language. Modeling languages are fixed grammars which permit users to model complex structures out of predefined types. This process of formal specification is comparable to the process of translating a word problem into algebraic notation. In numerous ways, this step of the formal design process is comparable to the formal software engineering method developed by Rumbaugh, Booch and others. At the minimum, both methods help engineers to evidently define their problems, goals and solutions. However, formal modeling languages are more thoroughly defined: in a formal grammar, there is a difference between WFFs (well-formed formulas) and non-WFFs (syntactically incorrect statements). Yet at this stage, the difference between WFF and non-WFF can help to state the design. Several engineers who have used formal specifications say that the clearness that this stage generates is a advantage in itself.

Notes

2. ***Verification:*** As affirmed above, formal methods vary from other specification systems by their heavy stress on provability and accuracy. By building a system with a formal specification, the designer is in fact developing a set of theorems about his system. By proving these theorems accurate, the formal Verification is a tricky process, largely since even the simplest system has several dozen theorems, each of which has to be proven. Even a traditional mathematical proof is a multifaceted affair.

Example: Wiles' proof of Fermat's Last Theorem, for example, took some years after its declaration to be completed. Given the demands of complexity and Moore's law, approximately all formal systems use an automated theorem proving tool of some form. These tools can show simple theorems, verify the semantics of theorems, and give assistance for verifying more complex proofs.

3. ***Implementation:*** Once the model has been stated and verified, it is implemented by transforming the specification into code. As the dissimilarity between software and hardware design increases, narrower, formal methods for producing embedded systems have been developed.

Example: LARCH, for example, has a VHDL implementation. Likewise hardware systems like the VIPER and AAMP5 processors have been produced using formal approaches.

An substitute to this approach is the lightweight approach to formal design. In a lightweight design, formal methods are applied cautiously to a system. This approach provides the benefits of formal specification, but also averts some of the difficulties.

Formal methods are observed with a definite degree of suspicion. While formal methods research has been succeeding since 1960's, formal methods are only being slowly acknowledged by engineers. There are numerous reasons for this, but most of the problems appear to be a result of misapplication. Most formal systems are tremendously descriptive and all-encompassing, modeling languages have usually been judged by their capacity to model anything. Regrettably, these same qualities make formal methods very tricky to use, especially for engineers untaught in the type theory required for most formal systems.

In opposition, it is noticeable that some form of formal specification is essential: complex systems need formal models. In addition, the mathematics needed for formal methods is becoming a more prominent fixture of engineering curricula, engineering schools in Europe are already requiring courses in VDM, Z and alike formal specifications. Eventually, formal methods will acquire some form of acceptance, but compromises will be made in both directions: formal methods will turn out to be simpler and formal methods training will turn out to be more common.

Formal methods propose additional benefits outside of provability, and these advantages do deserve some mention. However, most of these advantages are available from other systems, and usually without the steep learning curve that formal methods require.

- ***Discipline:*** By virtue of their rigor, formal systems require an engineer to think out his design in a more thorough fashion. In particular, a formal proof of correctness is going to require a rigorous specification of goals, not just operation. This thorough approach can help identify faulty reasoning far previous than in traditional design. The discipline included in formal specification has proved functional even on already obtainable systems.

Example: Engineers by means of the PVS system, for example, reported identifying numerous microcode errors in one of their microprocessor designs.

Notes

- ***Precision:*** Conventionally, disciplines have moved into jargons and formal notation as the weak spots of natural language descriptions turn out to be more clearly obvious. There is no motive that systems engineering should vary, and there are numerous formal methods which are used almost exclusively for notation.

For engineers designing safety-critical systems, the advantages of formal methods lie in their clearness. Unlike many other design strategies, the formal verification needs very clearly defined goals and strategies.

Did u know? In a safety critical system, vagueness can be extremely dangerous, and one of the main benefits of the formal approach is the removal of ambiguity.

Task Make distinction between verification and implementation.

Self Assessment

Fill in the blanks:

1. A formal system comprises a formal language and a set of ..... rules.
2. ..... are system design methods that use thoroughly declared mathematical models to construct software and hardware systems.
3. Formal methods vary from other design systems via the use of formal ..... systems.
4. Modeling languages are ..... grammars which permit users to model complex structures out of predefined types.
5. Once the model has been stated and verified, it is implemented by transforming the ..... into code.

12.2 Formal Specifications

- A formal specification is a requirement articulated in a language whose vocabulary, syntax and semantics are formally defined.

Caution It cannot be depend on natural language but as an alternative must be based on mathematics.

12.2.1 Advantages of Formal Specifications

- The development of a formal specification offers insights into and understanding of the software needs and the software design.
- Given a formal system requirement and a complete formal programming language definition, it may be probable to establish that a program conforms to its specification.
- Formal specifications have the possibility of automated processing.

Notes

- Animation of a specification to offer a prototype system, and mathematical study.
- Formal specs may be used as a conduct to the creation of test cases for any specific component of the system.

12.2.2 Disadvantages of Formal Specifications

- Software management is intrinsically conservative and is unwilling to adopt new techniques if the payoff is not instantly obvious.
- Most analysts and programmers have not been skilled in formal specification techniques.
- Clients are not probable to be recognizable with formal specifications and may be unwilling to fund expansion activities that they do not understand and thus cannot control.
- Some classes of systems are still complicated to specify by means of current techniques.
- There is widespread unawareness of existing specification techniques and their uses.
- Most research efforts have been directed at the expansion of notations and not with tool support.

12.2.3 Notations

- Notations like VDM involve the use of a number of particular symbols which must be memorized and this leads to a important learning curve.
- Some notations include a mnemonic notation that is less alien and can be typed in by means of a standard keyboard.
- The specification language, Z (pronounced 'zed') uses graphics to structure specifications.
- This enhances its readability and encourages incremental growth of specifications.

12.2.4 Developing a Simple Formal Specification

- The simplest form is axiomatic specification where a system is signified as a set of functions (which are stateless), and
- This technique is at present used only for small systems or system components.

Caution Each function is specified using pre and post-conditions.

12.2.5 Pre and Post-Conditions

- These conditions are predicates over the inputs and outputs of a function.
- A predicate is just a boolean expression which is true or false and whose variables are the parameters of the function being specified.
- Predicates include:
  - operators (like =, >, <, not, and, or),
  - the universal and existential quantifiers, and
  - the operator in which is used to choose the range over which the quantifier applies.

Notes

12.2.6 The Development of an Axiomatic Spec of a Function

- Institute the range of the input parameters over which the function is proposed to behave properly. Identify the input parameter constraints as a predicate.
- State a predicate defining a condition which must hold on the output of the function if it behaves properly.
- Establish what changes (if any) are made to the function's input parameters and state these.
- Merge these into pre- and post-conditions for the function.

Example: An Example: Search

function Search (X:INTEGER_ARRAY;

Key:INTEGER) return INTEGER;

Pre: $\exists$ i $\in$ X'FIRST..X'LAST:X(i) = Key

Post: X''(Search(X,Key)) = Key and X = X''

- The input array is unaffected by the search.
- Returns the value of the index of the element which is equivalent to the key.9

Example: Search of an Ordered Array

The specification now involves an additional clause in the pre-condition.

function Search (X:INTEGER_ARRAY;

Key:INTEGER) return INTEGER;

Pre: $\exists$ i $\in$ X'FIRST..X'LAST: X(i) = Key $\wedge$

$\forall$ i,j $\in$ X'FIRST..X'LAST: i < j $\wedge$ X(i) < = X(j)

Post: X''(Search(X,Key)) = Key and X = X''

12.2.7 The Use of an Error Predicate

- Specifications should also set out the behaviour of a constituent if it is presented with unforeseen input.
- One strategy is to have a number of pre/post-condition pairs depending on the number of erroneous input ranges.

The Use of an Error Predicate function Search (X:INTEGER_ARRAY; Key:INTEGER) return INTEGER;

Pre: $\exists$ i $\in$ X'FIRST..X'LAST: X(i) = Key

Post: X''(Search(X,Key)) = Key $\wedge$ X = X''

Error: Search(X,Key) = X'LAST + 1

Notes

12.2.8 The Use of Structure

- Large specifications are rigid to understand so it is significant that a specification language comprises structuring facilities which permit specifications to be developed incrementally.

12.2.9 Theoretical Aspects: Formally

- A formal specification language is a triple:

  < Syn, Sem, Sat >

  where Syn and Sem are sets and Sat, a subset of Syn X Sem, is a relation between them.
- Syn is called the language's syntactic domain, Sem is the semantic domain and Sat is the satisfies relation.
- Given a specification language,

  <Syn, Sem, Sat>
- if Sat(syn,sem) then syn is a specification of sem and sem is a specific and of syn.

12.2.10 Theoretical Aspects: Less Formally

- A formal specification language offers
- A notation (its syntactic domain),
- A universe of objects (its semantic domain), and
- A precise rule defining which objects satisfy each requirement.
- A specification is a sentence written in terms of the elements of the syntactic domain; it indicates a specific and set, a subset of the semantic domain.
- A specific and is an object pleasing a specification – the satisfies relation offers the meaning for the syntactic elements.

Example: An Example of a Simple Specification Language

- Backus-Naur form:
  - syntactic domain → a set of grammars
  - semantic domain → a set of strings
  - Every string is a particular and of each grammar that produces it.
  - Every specific and set is a formal language.

Self Assessment

Fill in the blanks:

6. A ..... is a requirement articulated in a language whose vocabulary, syntax and semantics are formally defined.
7. ..... conditions are predicates over the inputs and outputs of a function.
8. The specification language, Z (pronounced 'zed') uses ..... to structure specifications.

Notes

9. A ..... is a sentence written in terms of the elements of the syntactic domain; it indicates a specific and set, a subset of the semantic domain.

10. A ..... is just a boolean expression which is true or false and whose variables are the parameters of the function being specified.

12.3 Formal Grammars

A grammar is a controlling tool for depicting and analyzing languages. It is a set of rules by which applicable sentences in a language are created.

Example: Here's a trivial example of English grammar:

sentence → <subject> <verb-phrase> <object>

subject → This | Computers | I

verb-phrase → <adverb> <verb> | <verb>

adverb → never

verb → is | run | am | tell

object → the <noun> | a <noun> | <noun>

noun → university | world | cheese | lies

By means of the above rules or productions, we can obtain simple sentences like these:

This is a university.

Computers run the world.

I am the cheese.

I never tell lies.

Here is a leftmost derivation of the first sentence using these productions.

sentence → <subject> <verb-phrase> <object>

→ This <verb-phrase> <object>

→ This <verb> <object>

→ This is <object>

→ This is a <noun>

→ This is a university

In addition to numerous reasonable sentences, we can also obtain nonsense like "Computers run cheese" and "This am a lies". These sentences don't make semantic logic, but they are syntactically accurate since they are of the sequence of subject, verb-phrase, and object. Formal grammars are a tool for syntax, not semantics. In the syntax analysis phase, we confirm structure, not meaning.

12.3.1 Vocabulary

We want to review some definitions before we can continue: grammar: a set of rules by which applicable sentences in a language are constructed.

Notes

Nonterminal: A grammar symbol that can be substituted/expanded to a sequence of symbols.

Terminal: An actual word in a language; these are the symbols in a grammar that cannot be substituted by anything else. "Terminal" is supposed to summon up the idea that it is a dead-end-no further expansion is possible.

Production: A grammar rule that describes how to replace/exchange symbols. The general form of a production for a nonterminal is: X → Y1Y2Y3...Y. The nonterminal X is declared equivalent to the concatenation of the symbols Y1Y2Y3...Yn. The production means that anywhere where we encounter X, we may replace it by the string Y1Y2Y3...Yn. Eventually we will have a string containing nothing that can be expanded further, i.e., it will include only terminals. Such a string is called a sentence.

Did u know? In the context of programming languages, a sentence is a syntactically accurate and complete program.

Derivation: A sequence of applications of the rules of a grammar that generates a finished string of terminals. A leftmost derivation is where we always alternate for the leftmost nonterminal as we apply the rules (we can similarly define a rightmost derivation). A derivation is also known as a parse.

Start symbol: A grammar has a single nonterminal (the start symbol) from which all sentences derive: S → X1X2X3...Xn

All sentences are obtained from S by successive replacement by means of the productions of the grammar.

Null Symbol: ε it is sometimes functional to specify that a symbol can be replaced by nothing at all. To specify this, we use the null symbol ε, e.g., A → B | ε.

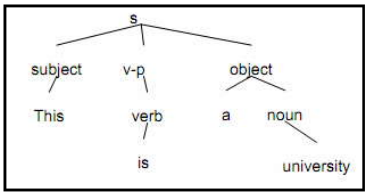
BNF: A method of specifying programming languages by means of formal grammars and production rules with a particular form of notation (Backus-Naur form).

Task Compare start symbol and null symbol.

12.3.2 Parse Representation

In functioning with grammars, we can signify the application of the rules to derive a sentence in two manners. The first is a derivation as exposed previously for "This is a university" where the rules are applied gradually and we substitute for one nonterminal at a time.

Consider a derivation as a history of how the sentence was parsed since it not only includes which productions were applied, but also the order they were applied (i.e., which nonterminal was chosen for expansion at each step). There can many different derivations for the same sentence (the leftmost, the rightmost, and so on). A parse tree is the second method for representation. It diagrams how every symbol derives from other symbols in a hierarchical manner. Here is a parse tree for "This is a university":



Notes

Notes Even though the parse tree involves all of the productions that were applied, it does not encode the order they were applied. For an explicit grammar, there is precisely one parse tree for a particular sentence.

12.3.3 More Definitions

Here are some other definitions we will require, illustrated in reference to this example grammar:

$S \rightarrow AB$

$A \rightarrow Ax \mid y$

$B \rightarrow z$

The alphabet is {S, A, B, x, y, z}. It is divided into two disjoint sets. The terminal alphabet includes terminals, which appear in the sentences of the language:

{x, y, z}. The remaining symbols are the nonterminal alphabet; these are the symbols that occur on the left side of productions and can be substituted throughout the course of a derivation: {S, A, B}. Formally, we use V for the alphabet, T for the terminal alphabet and N for the nonterminal alphabet giving us: $V = T \cup N$.

Context-free Grammar

To define a language, we require a set of productions, of the common form: $u \rightarrow v$. In a context-free grammar, u is a single nonterminal and v is a random string of terminal and nonterminal symbols. When parsing, we can substitute u by v where it occurs. We shall refer to this set of productions symbolically as P.

Formal Grammar

We officially define a grammar as a 4-tuple {S, P, N, T}. S is the start symbol and $S \in N$, P is the set of productions, and N and T are the nonterminal and terminal alphabets. A sentence is a string of symbols in T obtained from S by means of one or more applications of productions in P. A string of symbols derived from S but possibly including non-terminals is known as a sentential form or a working string.

A production $u \rightarrow v$ is used to reinstate an occurrence of u by v. Officially, if we apply a production $p \in P$ to a string of symbols w in V to yield a new string of symbols z in V, we say that z obtained from w using p, written as follows: $w \rightarrow z$. We also use: $w \rightarrow z$ z derives from w (production unspecified).

$w \rightarrow^* z$ z derives from w by means of zero or more productions $w \rightarrow^+ z$ z derives from w by means of one or more productions equivalence. The language L(G) defined by grammar G is the set of sentences derivable by means of G. Two grammars G and G' are said to be equivalent if the languages they generate, L(G) and L(G'), are the similar.

Self Assessment

Fill in the blanks:

11. ..... is a set of rules by which applicable sentences in a language are created.

12. ..... is a grammar rule that describes how to replace/exchange symbols.

Notes

13. ..... is a method of specifying programming languages by means of formal grammars and production rules with a particular form of notation.

14. ..... is a grammar symbol that can be substituted/expanded to a sequence of symbols.

15. Formal grammars are a tool for syntax, not .....

Case Study

Programming Language: ALGOL

Algol is of interest to us because it was the first programming language to be defined using a grammar. It grew out of an international effort in the late 1950's to create a "universal programming language" that would run on all machines. At that time, FORTRAN and COBOL were the prominent languages, with new languages sprouting up all around. Programmers became increasingly concerned about portability of programs and being able to communicate with one another on programming topics. Consequently the ACM and GAMM (Gesellschaft für angewandte Mathematik und Mechanik) decided to come up with a single programming language that all could use on their computers, and in whose terms programs could be communicated between the users of all machines. Their first decision was not to use FORTRAN as their universal language. This may seem surprising to us today, since it was the most commonly used language back then. However, as Alan J. Perlis, one of the original committee members, puts it:

"Today, FORTRAN is the property of the computing world, but in 1957, it was an IBM creation and closely tied to IBM hardware. For these reasons, FORTRAN was unacceptable as a universal language."

ALGOL-58 was the first version of the language, followed up very soon after by ALGOL-60, which is the version that had the most impact. As a language, it introduced the following features:

(i) block structure and nested structures, (ii) strong typing, (iii) scoping, (iv) procedures and functions, (v) call by value, call by reference, (vi) side effects (is this good or bad?), (vii) recursion.

It may seem surprising that recursion was not present in the original FORTRAN or COBOL. You probably know that to implement recursion we need a runtime stack to store the activation records as functions are called. In FORTRAN and COBOL, activation records were created at compile time, not runtime. Thus, only one activation record per subroutine was created. No stack was used. The parameters for the subroutine were copied into the activation record and that data area was used for subroutine processing. The ALGOL report was the first time we see BNF to describe a programming language. Both John Backus and Peter Naur were on the ALGOL committees. They derived this description technique from an earlier paper written by Backus. The technique was adopted because they needed a machine-independent method of description. If one looks at the early definitions of FORTRAN, one can see the links to the IBM hardware. With ALGOL, the machine was not relevant. BNF had a huge impact on programming language design and compiler construction. First, it stimulated a large number of studies on the formal structure of programming languages laying the groundwork for a theoretical approach to language design. Second, a formal syntactic description could be used to drive a compiler directly (as we shall see). ALGOL had a tremendous impact on programming language design, compiler construction, and language theory, but the language itself was a commercial failure. Partly this was due to design decisions (overly complex features, no IO) along with the politics of the time (popularity of Fortran, lack of support from the all-powerful IBM, resistance to BNF).

Notes

12.4 Summary

- A formal system comprises a formal language and a set of inference rules.
- Formal methods are system design methods that use thoroughly declared mathematical models to construct software and hardware systems. In distinction to other design systems, formal methods use mathematical evidence as a match to system testing so as to make sure correct behavior.
- Formal methods vary from other design systems via the use of formal verification systems.
- Throughout the formal specification phase, the engineer thoroughly defines a system by means of a modeling language.
- A formal specification is a requirement articulated in a language whose vocabulary, syntax and semantics are formally defined.
- Large specifications are rigid to understand so it is significant that a specification language comprises structuring facilities which permit specifications to be developed incrementally.
- A specification is a sentence written in terms of the elements of the syntactic domain; it indicates a specific and set, a subset of the semantic domain.
- A grammar is a set of rules by which applicable sentences in a language are created.

12.5 Keywords

Formal System: A formal system comprises a formal language and a set of inference rules.

Formal Specification: A formal specification is a requirement articulated in a language whose vocabulary, syntax and semantics are formally defined.

Grammar: A grammar is a set of rules by which applicable sentences in a language are created. Here's a trivial example of English grammar.

Specification: A specification is a sentence written in terms of the elements of the syntactic domain; it indicates a specific and set, a subset of the semantic domain.

12.6 Review Questions

1. What are formal systems? Illustrate the use of formal systems in programming languages.
2. Elucidate the steps included in formal design.
3. Discuss why formal methods are only being slowly acknowledged by engineers.
4. What is formal specification? Illustrate with example.
5. Illustrate the advantages and disadvantages of formal specification.
6. Enlighten the Pre and Post-Conditions used in formal specification.
7. What is grammar? Give an example of an English grammar.
8. Define a grammar for strings where the number of a's is equal to the number b's.
9. Define a grammar where the number of a's is not equal to the number b's.
10. Explain the concept of Parse Representation with example.

Notes

Answers: Self Assessment

1.	inference	2.	Formal methods
3.	verification	4.	fixed
5.	specification	6.	formal specification
7.	Pre and Post	8.	graphics
9.	specification	10.	predicate
11.	Grammar	12.	Production
13.	BNF (Backus-Naur form)	14.	Non-terminal
15.	semantics		

12.7 Further Readings


Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media


Online link

www.springer.com

Unit 13: Formal Systems

Notes

CONTENTS

Objectives

Introduction

13.1 Hierarchy of Languages

13.1.1 Issues in parsing context-free Grammars

13.2 Backus Naur Form – Backus Normal Form (BNF)

13.2.1 The Purpose and Function of the Notation

13.2.2 The Fundamental Idea

13.2.3 Extensions

13.2.4 Data Type Definitions for Extensions

13.2.5 The Name of the Notation

13.3 Summary

13.4 Keywords

13.5 Review Questions

13.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand hierarchy of languages
- Discuss Backus Naur Form or Backus Normal Form

Introduction

We possess a lot of our perceptive of grammars to the work of the American linguist Noam Chomsky. We will discuss in this unit different types of grammars and their issues. Also you will understand the concept of Backus Naur Form (BNF) which was originally called as Backus Normal Form (BNF).

13.1 Hierarchy of Languages

We possess a lot of our perceptive of grammars to the work of the American linguist Noam Chomsky. We will discuss in this unit different types of grammars and their issues. Also you will understand the concept of Backus Naur Form (BNF) which was originally called as Backus Normal Form (BNF).

Type 0 (free or unobstructed grammars)

These are the most familiar. Productions are of the appearance u -> v where both u and v are random strings of symbols in V, with u non-null.

Notes

Caution There are no limitations on what occurs on the left or right-hand side except the left hand side must be non-empty.

Type 1 (context-sensitive grammars)

Productions are of the appearance uXw → uvw where u, v and w are random strings of symbols in V, with v non-null, and X a single nonterminal. Alternatively, X may be substituted by v but only when it is bounded by u and w. (i.e., in a specific context).

Type 2 (context-free grammars)

Productions are of the appearance X → v where v is an random string of symbols in V, and X is a single nonterminal. Wherever you locate X, you can substitute with v (in spite of context).

Type 3: Regular Grammars

Productions are of the appearance X → a, X → aY, or X → ε where X and Y are non-terminals and a is a terminal. Specifically, the left-hand side must be a single nonterminal and the right-hand side can be either unfilled, a single terminal by itself or with a single nonterminal. These grammars are the most limited in terms of important power.

Did u know? Each type 3 grammar is a type 2 grammar, and each type 2 is a type 1 and so on.

Type 3 grammars are chiefly simple to parse due to the lack of recursive builds. Competent parsers survive for many classes of Type 2 grammars. Even though Type 1 and Type 0 grammars are more influential than Type 2 and 3, they are far less functional as we cannot generate competent parsers for them. In scheming programming languages by means of formal grammars, we will utilize Type 2 or context-free grammars, frequently just shortened as CFG.

Task Make distinction between unobstructed grammars and regular grammars.

13.1.1 Issues in parsing Context-free Grammars

There are numerous competent approaches to parsing most Type 2 grammars. Though, there are concerns that can interfere with parsing that we must take into consideration when scheming the grammar. Let's discuss three of them: ambiguity, recursive rules, and left-factoring.

Ambiguity

If a grammar allows more than one parse tree for a number of sentences, it is said to be ambiguous.

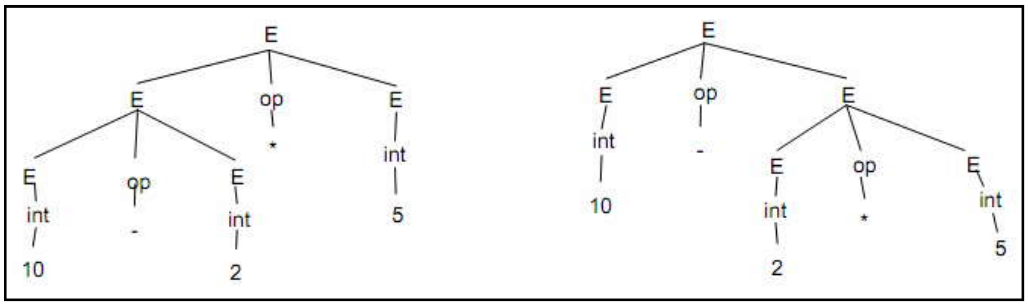
Example: Consider the following typical arithmetic expression grammar:

E → E op E | (E) | int op → + | - | * | /

This grammar indicates expressions that contain integers connected by binary operators and perhaps counting parentheses. As defined above, this grammar is uncertain since for certain sentences we can build more than one parse tree.

Notes

Example: Consider the expression 10 - 2 * 5. We parse by primarily applying the production E → E op E. The parse tree on the left selects to expand that first op to *, the one on the right to -.We have two entirely different parse trees. Which one is correct?



Both trees are authorized in the grammar as affirmed and therefore either interpretation is valid. Even though natural languages can stand some type of ambiguity (e.g., puns, plays on words, etc.), it is not suitable in computer languages. We don't want the compiler just randomly deciding which way to understand our expressions! Given our expectations from algebra about precedence, only one of the trees appears to be right. The right-hand tree fits our anticipation that * "binds tighter" and for that consequence to be calculated first then integrated in the outer expression which has a lower preference operator. It's moderately simple for a grammar to turn out to be ambiguous if you are not cautious in its construction. Regrettably, there is no supernatural method that can be used to resolve all varieties of vagueness. It is an undecidable difficulty to conclude whether any grammar is vague, much less to effort to mechanically take away all ambiguity. Though, that doesn't signify in practice that we cannot notice vagueness or do something regarding it. For programming language grammars, we typically take pains to build an unambiguous grammar or initiate additional disambiguating rules to throw away the unwanted parse trees, leaving only one for each sentence. By means of the above ambiguous expression grammar, one method would leave the grammar as is, but add disambiguating rules into the parser execution. We could code into the parser knowledge of precedence and associatively to break the tie and force the parser to construct the tree on the right instead of the left. The benefit of this is that the grammar remains easy and less complicated. But as a downside, the syntactic structure of the language is no longer specified by the grammar alone.

Another strategy is to alter the grammar to only permit the one tree that appropriately reflects our purpose and eradicate the others. For the expression grammar, we can disconnect expressions into multiplicative and additive subgroups and force them to be prolonged in the preferred order.

E → E

t_op E | T t_op → + | -

T → T

f_op T | F f_op → * | /

F → (E) | int

Terms are addition/subtraction expressions and factors utilized for multiplication and division. As the base case for expression is a term, addition and subtraction will emerge higher in the parse tree, and therefore obtain lower precedence. After confirming that the above rewritten grammar has only one parse tree for the previous ambiguous expression, you might thing we were home free, but now consider the expression 10 –2 –5. The recursion on both sides of the binary operator permits either side to match repetitions. The arithmetic operators typically connect to the left, so by substituting the right-hand side with the base case will force the repetitive matches onto the left side. The concluding result is:

Notes

```
E        -->    E t_op T | T
t_op     -->    + | -
T        -->    T f_op F | F
f_op     -->    * | /
F        -->    (E) | int
```

The comprehensible difficulty of modifying the grammar to take away ambiguity is that it may make difficult and difficult to understand the original grammar definitions. There is no mechanical means to alter any ambiguous grammar into an unmistakable one.

Notes Though, most programming languages have only restricted concerns with ambiguity that can be resolved by means of ad hoc techniques.

Recursive Productions

Productions are frequently defined in terms of themselves.

Example: A list of variables in a programming language grammar could be declared by this production:

variable_list → variable | variable_list , variable

Such productions are considered to be recursive. If the recursive nonterminal is at the left of the right-side of the production, e.g. A → u | Av, we call the production left-recursive. Likewise, we can define a right-recursive production: A → u | vA. Some parsing methods have trouble with one or the other alternatives of recursive productions and so at times we have to massage the grammar into a dissimilar but equivalent form.

Caution Left-recursive productions can be particularly troublesome in the top-down parsers.

Usefully, there is an easy technique for rewriting the grammar to shift the recursion to the other side.

Example: Consider this left-recursive rule:

X → Xa | Xb | AB | C | DEF

To translate the rule, we introduce a new nonterminal X' that we attach to the end of all non-left-recursive productions for X. The expansion for the new nonterminal is fundamentally the reverse of the original left-recursive rule. The rewritten productions are:

X → ABX' | C

X' | DEFX' X' → aX' | bX' | ε

It occurs we just exchanged the left-recursive rules for a corresponding right-recursive edition.

This might appears to be pointless, but some parsing algorithms favor or even need only left or right recursion.

Notes

Task Illustrate how to define right-recursive production.

Left-factoring

The parser typically reads tokens from left to right and it is suitable if, upon reading a token, it can make an instant decision regarding which production from the grammar to develop. However, this can be difficulty if there are productions that have general first symbol(s) on the right side of the productions.

Example: Here is an example we frequently observe in programming language grammars:

Stmt → if Cond then Stmt else Stmt | if Cond then Stmt | Other | ....

The common prefix is if Cond then Stmt. This causes troubles since when a parser encounter an "if", it does not recognize which production to utilize. A functional technique known as left-factoring allows us to reorganize the grammar to evade this circumstance. We rephrase the productions to postpone the decision regarding which of the choices to choose until we have seen sufficient of the input to make the suitable choice. We factor out the common division of the two choices into a shared rule that both will utilize and then add a new rule that picks up where the tokens deviate.

Stmt → if Cond then Stmt

OptElse | Other | … OptElse -> else S | ε

In the rewritten grammar, upon reading an "if" we increase first production and stay until if Cond then Stmt has been observed to decide whether to expand OptElse to else or ?

Hidden left-factors and Hidden left recursion

A grammar may not emerge to have left recursion or left factors, yet still have concerns that will obstruct with parsing. This may be since the concerns are concealed and require to be first exposed through replacement.

Example: Consider this grammar:

A → da | acB

B → abB | daA | Af

A brief examination of the grammar may not sense that the first and second productions of B partly cover the third. We alternate the expansions for A into the third production to represent at this:

A → da | acB

B → abB | daA | daf | acBf

This swaps the original third production of B for numerous new productions, one for every productions for A. These directly demonstrate the overlap, and we can then left factor:

A → da | acB

B → aM | daN

M → bB | cBf

N → A | f

Notes

Likewise, the following grammar does not emerge to have any left-recursion:

$S \rightarrow Tu \mid wx$

$T \rightarrow Sq \mid vvS$

Yet after replacement of S into T, the left-recursion comes to light:

$S \rightarrow Tu \mid wx$

$T \rightarrow Tuq \mid wxq \mid vvS$

If we then eliminate left-recursion, we obtain:

$S \rightarrow Tu \mid wx$

$T \rightarrow wxqT' \mid vvST'$

Self Assessment

Fill in the blanks:

1. In ....., productions are of the appearance u -> v where both u and v are random strings of symbols in V, with u non-null.
2. In ....., productions are of the appearance uXw $\rightarrow$ uvw where u, v and w are random strings of symbols in V, with v non-null, and X a single nonterminal.
3. In ..... productions are of the appearance X$\rightarrow$ v where v is an random string of symbols in V, and X is a single nonterminal.
4. In Type 3 (regular grammars), productions are of the appearance X$\rightarrow$ a, X$\rightarrow$ aY, or X$\rightarrow$ ε where X and Y are .....and a is a terminal.
5. Even though Type 1 and Type 0 grammars are more influential than Type 2 and 3, they are far less functional as we cannot generate competent ..... for them.
6. If a grammar allows more than one parse tree for a number of sentences, it is said to be .....
7. If the recursive nonterminal is at the left of the right-side of the production, e.g. A -> u | Av, we call the production .....
8. Each type 3 grammar is a type 2 grammar, and each type 2 is a ..... and so on.

13.2 Backus Naur Form – Backus Normal Form (BNF)

BNF, formerly was named as "Backus Normal Form". It is a formal meta syntax used to articulate context-free grammars. Backus Normal Form was renamed Backus Naur Form at the suggestion of Donald Knuth.

The syntax of a programming language states the structure and organization of the elements in the text of programs written in that language.

Did u know? The designers and implementers of FORTRAN, the first high-level programming language, depicted its syntax in English prose, supplemented by instances.

Early FORTRAN programs were specified on punched cards, and the syntax of the language reflects this. A FORTRAN program comprises more or less self-governing statements, usually in one-to-one correspondence with cards. Every type of statement has an idiosyncratic syntax, which is either fully fixed or has only a finite number of variants. FORTRAN statements are not nested.

Notes

Example: FORTRAN has an analogue of the for-statement as we recognize it from languages such as Java and C, but in FORTRAN the body of the loop is not measured to be piece of the for-statement; it's just a series of statements, like any other segment of a FORTRAN program. To organize for the recurrence, the FORTRAN programmer adds a label to the last declaration in the sequence and prepends a particular DO-statement at the commencement.

The DO-statement states four things: the loop-control variable, its initial value, its final value, and the label of the last statement. The FORTRAN compiler substitutes the DO-statement with the code for initializing the loop control variable and adds, after the labelled statement, code for modernizing and testing the loop-control variable and for making the jump back to the top of the loop.

The statements making up the sequence to be frequent were not essentially apparent as "belonging" to the loop, and actually it was not atypical for programmers to direct power out of this area and back in again if they required executing some additional statements that it was suitable to inscribe elsewhere.

This notion of the program as a flat list of autonomous statements turned out to be excessively restricting in some respects and excessively flexible in others.

13.2.1 The Purpose and Function of the Notation

The syntactic components of a program generate a hierarchy, with the elements (like identifiers, operator symbols, keywords, and literal constants) at the bottom and increasingly larger elements at higher levels. These elements are considered as belonging syntactic categories, with members of the similar category having similar syntactic tasks and functions. One can symbolize the arrangement of a program as a syntax tree in which the leaf nodes are syntactic elements and every interior node is labelled with the syntactic category of the element generated by the leaves that descend from it.

The Backus-Naur formalism offers a brief manner of describing potential modes of mixture of elements. It has a supplementary advantage for implementers of languages: It displays exactly how to construct the recursive data structures for syntax trees.

13.2.2 The Fundamental Idea

The basic idea is to portray the syntax of a programming language as a set of rules (at times known as productions). Each rule depicts one possible method of building a constituent belonging to a specific syntactic category, which is named at the commencement of the rule. We'll write the names of syntactic categories inside angle brackets. The right hand side of the rule then lists, in order, the components of the constituents; these may comprise syntactic elements, other syntactic groups, or both.

Any number of components can emerge in the right-hand side of a rule. The syntactic category at the foundation of the rule is alienated from the right-hand side by the (more or less random) symbol ::=.

Two or more rules starting with the similar syntactic categories may be united into one by concatenating their right-hand sides, separating the substitute constructions with vertical bars, therefore:

<expression> ::= <identifier> | <lambda-expression> | <procedure-call>

The reader must identify that the vertical bar in such cases, like the ::= symbol, is portion of the meta-notation, not part of the programming language that is being depicted. Certainly, all of these extensions entail additions to the meta-notation; when they are all deployed immediately, it can at

Notes

times be difficult to differentiate the BNF symbols from syntactic elements of the object language. This is one cause why some authors desire to include syntactic elements in quotation marks.

There are numerous trivial variants of this notation. At times syntactic elements are included in quotation marks and the names of syntactic categories are written without the angle brackets. Sometimes an arrow of some sort is utilized in lieu of the ::= symbol.

Example: Here's the tangible syntax of expressions of the λ-calculus:

<lc-expression> ::= <lc-identifier>

| (lambda (<lc-identifier>) <lc-expression>)

| (<lc-expression> <lc-expression>)

Dropping all of the fixed symbols provides us the equivalent abstract

syntax:

<lc-expression> ::= <lc-identifier>

| <lc-identifier> <lc-expression>

| <lc-expression> <lc-expression>

When we construct our individual syntax trees in Scheme, we'll write a data type definition for every non-trivial syntactic category, with a variation for every rule. The right-hand side of the BNF rule informs us what the fields of the analogous variant should be-every field matches to one of the non-fixed constituents.

It is hypothetically probable for the right-hand side of a BNF rule to be

empty:

<empty> ::=

Practically, though, this possibility is rarely used. (For one thing, it messes up the association with syntax trees, as a node labelled with a syntactic category such as empty is technically a leaf node but gives no syntactic element to the noticeable text of the program.)

13.2.3 Extensions

Plain vanilla BNF is rather unwieldy. For convenient use, the notation is usually extended in different manners.

It frequently occurs that two rules for the similar syntactic category vary only in the presence or absence of some optional constituent.

Example: We show an example from Java:

<break-statement> ::= break ;

| break <identifier> ;

Such a pair of rules can be united by including the possible component in braces and writing a question mark after it, therefore:

<break-statement> ::= break {<identifier>}? ;

Again, it frequently occurs that a constituent may include any number of consecutive components of the similar category. We'll state this in our comprehensive Backus-Naur formalism by including the repeatable constituent in braces and writing a star after it:

<procedure-call> ::= (<expression> {<expression>}*)

Notes

In unextended BNF, one would get approximately the similar effect by offering a separate group for arbitrary-length series of the repeatable component and examining it therefore:

<procedure-call> ::= (<expression> <expression-sequence>)

<expression-sequence> ::= <empty>

| <expression> <expression-sequence>

This is much like the recursive definition of a list as either null or a pair comprising one element (the car) and a list (the cdr).

Notes The syntax trees constructed as per the unextended BNF rule appear different from those constructed as per the star rule-none of the expression nodes for the operands will be children of the procedure-call node if the unextended BNF rule is utilized.

The star notation permits for the possibility that the repeatable constituent does not occur at all. (In our example, a procedure call may enclose no operands.) If we wish to state a repeatable component that must happen at least once, we'll use a plus sign on behalf of the star.

Example: We show example from the Icon programming language:

<program> ::= {<declaration>}+

Alternatively, an Icon program includes one or more declarations. Again, we could obtain a similar effect in unextended BNF:

<program> ::= <declaration-sequence>

<declaration-sequence> ::= <declaration>

| <declaration> <declaration-sequence>

Lastly, it frequently happens that a creation can comprise any number of consecutive occurrences of some component, apart from that if there are two or more of them, any two that are contiguous must be separated by some fixed symbol.

Example: The compound statement in Algol 60 usually comprises numerous statements separated by semicolons:

begin

temp := first;

first := second;

second := temp

end

Observe that the semicolon is a separator, not a terminator. It would be an error to put a semicolon among the second occurrence of temp and the keyword end. A compound statement in Algol 60 can enclose just one statement in its body:

begin

temp := first

end

And it may enclose none at all:

begin

end

Notes

No semicolon is required in these cases, as there are no two adjacent component statements to be kept disconnected.

We'll state this separated-repetition rule by means of a modification of the star notation, positioning the separator symbol in braces after the star:

<compound-statement> ::= begin {<statement>}*{;} end

13.2.4 Data Type Definitions for Extensions

All of the extensions just illustrated can be managed within the framework offered by define-datatypes and cases.

For the question-mark expansion, it's frequently simpler to develop it into two variants:

(define-datatype break-expression break-expression?

(simple-break ())

(targeted-break (target identifier?)))

When there is more than one optional constituent in the similar rule, so that separating all the variants would lead to a combinatorial explosion, one can offer a single variant but permit some type of an absence marker to any or all of the possible locations:

<for-stmt> ::= for ({<expr>}? ; {<expr>}? ; {<expr>}?) <stmt>

(define-datatype for-stmt for-stmt?

(a-for-stmt (initializer (maybe expr?))

(entry-test (maybe expr?))

(updater (maybe expr?))

(for-body stmt?)))

Given a unary predicate, the perhaps procedure constructs and returns a new predicate that is contented by everything that satisfies the specified predicate, but also by the symbol absent (conventionally signalling an absent optional component of a syntactic construction).

13.2.5 The Name of the Notation

Backus-Naur form is named after John Backus (1924-2007) and Peter Naur (b. 1928). Backus, who used to lead the team that produced the first high-level programming language (FORTRAN), invented the formalism in 1959. Naur assisted to make it likable by adapting and utilizing it in the definition of the Algol 60 programming language.

Self Assessment

Fill in the blanks:

9. BNF is a formal meta syntax used to articulate ..... grammars.
10. Backus Normal Form was renamed as ..... at the suggestion of Donald Knuth.
11. The Backus-Naur formalism displays exactly how to construct the recursive data structures for syntax .....
12. Each rule depicts one possible method of building a constituent belonging to a specific ..... category, which is named at the commencement of the rule.

Notes

13. At times syntactic elements are included in quotation marks and the names of syntactic categories are written without the .....

14. It frequently occurs that two ..... for the similar syntactic category vary only in the presence or absence of some optional constituent.

15. Backus-Naur form is named after John Backus and .....

13.3 Summary

- There are four types of formal grammars in the Chomsky Hierarchy, they extent from Type 0, the most common, to Type 3, the most limiting.
- In Type 0 (free or unobstructed grammars) , productions are of the appearance $u \rightarrow v$ where both u and v are random strings of symbols in V, with u non-null.
- In Type 1 (context-sensitive grammars), productions are of the appearance $uXw \rightarrow uvw$ where u, v and w are random strings of symbols in V, with v non-null, and X a single nonterminal.
- In Type 2 (context-free grammars), productions are of the appearance $X \rightarrow v$ where v is an random string of symbols in V, and X is a single nonterminal.
- In Type 3: regular grammars, productions are of the appearance $X \rightarrow a$, $X \rightarrow aY$, or $X \rightarrow \varepsilon$ where X and Y are non-terminals and a is a terminal.
- BNF, formerly was named as "Backus Normal Form" which is a formal meta syntax used to articulate context-free grammars.
- The syntax of a programming language states the structure and organization of the elements in the text of programs written in that language.
- Plain vanilla BNF is unwieldy. For convenient use, the notation is usually extended in different manners.

13.4 Keywords

Backus Normal Form: BNF, formerly was named as "Backus Normal Form" which is a formal meta syntax used to articulate context-free grammars.

Type 0 (free or unobstructed grammars): In Type 0 (free or unobstructed grammars) , productions are of the appearance $u \rightarrow v$ where both u and v are random strings of symbols in V, with u non-null.

Type 1 (context-sensitive grammars): In Type 1 (context-sensitive grammars), productions are of the appearance $uXw \rightarrow uvw$ where u, v and w are random strings of symbols in V, with v non-null, and X a single nonterminal.

Type 2 (context-free grammars): In Type 2 (context-free grammars), productions are of the appearance $X \rightarrow v$ where v is an random string of symbols in V, and X is a single nonterminal.

Type 3 (regular grammers) $\rightarrow$ In Type 3 (regular grammars), productions are of the appearance $X \rightarrow a$, $X \rightarrow aY$, or $X \rightarrow \varepsilon$ where X and Y are non-terminals and a is a terminal.

13.5 Review Questions

1. Elucidate the concept of languages hierarchy.
2. Make distinction between context-sensitive grammars and context-free grammars.

Notes

3. Illustrate the process of parsing context-free grammars.
4. Even though natural languages can stand some type of ambiguity (e.g., puns, plays on words, etc.), it is not suitable in computer languages. Comment.
5. What are recursive productions? Illustrate with examples.
6. What is Backus Naur Form (BNF)? Explicate with examples.
7. Describe the concept of Extensions in BNF. Also discuss Data type definitions for extensions with example.
8. Describe the Algol 60 compound statement in unextended BNF.
9. The variable-declaration part of a Pascal program, procedure definition, or function definition consists of the keyword var followed by one or more declarations, each consisting of one or more identifiers (separated by commas), a colon, and either an identifier or a constructed type. Each declaration is terminated by a semicolon.
10. Translate this description into one or more BNF rules. (You can assume that the category <constructed-type> is defined elsewhere.) Then write a data type definition for it.

Answers: Self Assessment

1. Type 0 (free or unobstructed grammars)
2. Type 1 (context-sensitive grammars)
3. Type 2 (context-free grammars),
4. non-terminals
5. parsers
6. ambiguous
7. left-recursive
8. type 1
9. context-free
10. Backus Naur Form
11. trees
12. syntactic
13. angle brackets
14. rules
15. Peter Naur

13.6 Further Readings


Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media


Online link

www.springer.com

Unit 14: Canonic Systems

Notes

CONTENTS

Objectives
Introduction
14.1 Syntax Specification
14.2 Specification Translation
14.3 Canonic Recognition
14.3.1 Benefits
14.4 Canonic Translation Algorithm
14.5 Canonic Systems and Formal Systems
14.6 Summary
14.7 Keywords
14.8 Review Questions
14.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan canonic systems
- Describe syntax specification
- Demonstrate specification of translation
- Explain recognition and translation algorithm
- Discuss canonic systems and formal systems

Introduction

When using autoconf, there are three system definitions (or machine definitions) that are used to identify the "actors" in the build process; each definition relates to a similarly-named variable. These three definitions are:

- ***host (CHOST):*** The system that is going to run the software once it is built, which is the main actor. Once the software has been built, it will execute on this particular system.
- ***build (CBUILD):*** The system where the build process is being executed. For most uses this would be the same as the host system, but in case of cross-compilation the two obviously differ.
- ***target (CTARGET):*** The system against which the software being built will run on. This actor only exists, or rather has a meaning, when the software being built may interact specifically with a system that differs from the one it's being executed on (our host). This is the case for compilers, debuggers, profilers and analyzers and other tools in general.

To identify the current actors involved in the build process, autoconf provides three macros that take care of finding the so-called "canonic" values: AC_CANONIC_HOST, AC_CANONIC_BUILD and AC_CANONIC_TARGET. These three macros then provide to the configure script the variables with the name of the actor ($host, $build and $target), and three parameters with the same name to the configure script so that the user can override the default discovered values.

Notes

The most basic autoconf based build systems won't need to know any of these values, at least directly. Some other tools, such as libtool, will require discovery of canonic systems by themselves. Since adding these macros unconditionally adds direct and indirect code to the configure script (and a dependency on the two support files config.sub and config.guess); it is recommended not to call them unconditionally.

It is actually quite easy to decide whether canonic system definitions are needed or not. We just have to look for the use of the related actor variable. For instance, if the configure.ac script uses the $build variable, we would need to call AC_CANONIC_BUILD to discover its value. If the system definition variables are used in a macro instead, we should use the AC_REQUIRE macro to ensure that they are executed before entering. Don't fear calling them in more than one place.

One common mistake is to "go all the way" and always use the AC_CANONIC_TARGET macro, or its misnamed predecessor AC_CANONIC_SYSTEM. This is particularly a problem; because most of the software will not have a target actor at all. This actor is only meaningful when the software that is being built manages data that is specific to a different system than the one it is being executed on (the host system).

Notes In practice, the only places where the target actor is meaningful are to the parts of a compile toolchain: assemblers, linkers, compilers, debuggers, profilers, analysers, etc. For the rest of the software, the presence of an extraneous – target option to configure is likely to just be confusing. Especially for software that processes the output of the script to identify some information about the package being built.

14.1 Syntax Specification

Advanced Query Syntax (AQS) is the default query syntax used by Windows Search to query the index and to refine and narrow search parameters. AQS is employed by developers to build queries programmatically (and by users to narrow their search parameters). Canonic AQS was introduced in Windows 7 and must be used in Windows 7 and later to programmatically generate AQS queries.

Canonic syntax was introduced for keywords in Windows 7. An example of a query with a canonic property is System.Message.FromAddress:=me@microsoft.com. When coding queries in applications running on Windows 7 and later, you must use canonic syntax to programmatically generate AQS queries. If you do not use canonic syntax and your application is deployed in a locale or UI language different from the language in the application code, your queries will not be interpreted correctly.

The conventions for canonic keyword syntax are as follows:

- The canonic syntax for a property is its canonic name, such as System.Photo.LightSource. Canonic names are not case sensitive.
- The canonic syntax for the Boolean operators consists of the keywords AND, OR, and NOT, in all uppercase.
- The operators <, >, =, and so forth, are not localized and are thus also part of the canonic syntax.
- If a property P has enumerated values or ranges named N1 through Nk, the canonic syntax for the Ith value or range is the canonic name for P, followed by the character #, followed by NI, as illustrated in the following example:

  System.Photo.LightSource#Daylight, System.Photo.LightSource#StandardA, and so forth.

Notes

- For a defined semantic type T with values or ranges named N1 through Nk, the canonic syntax for the Ith value or range is the canonic name for T, followed by the character #, followed by NI, as illustrated in the following example:

  System.Devices.LaunchDeviceStageFromExplorer:=System.StructuredQueryType.Boolean#True

- For literal values such as words or phrases, the canonic syntax is the same as the regular syntax.

Examples: Queries with literal values in canonic syntax are:

- System.Author:sanjay
- System.Keywords:"Animal"
- System.FileCount:>100

Notes There is no canonic syntax for numbers in Windows 7 and later. Because floating point formats vary among locales, the use of a canonic query that involves a floating point constant is not supported. Integer constants, in contrast, can be written using only digits (no separators for thousands) and can be safely used in canonic queries in Windows 7 and later.

Example: The following table shows some examples of canonic properties and the syntax for using them.

Types of canonic property	Examples	Syntax
String value	System.Author	The string value is searched for in the author property: *System.Author:Jacobs*
Enumeration range	System.Priority	The priority property can have a numerical value range: *System.Priority:System.Priority#High*
Boolean	System.IsDeleted	Boolean values can be used with any Boolean property: *System.IsDeleted:System.StructuredQueryType.Boolean#True, andSystem.IsDeleted:System.StructuredQueryType.Boolean#False*
Numerical	System.Size	It is not possible to write safely a canonic query that involves a floating point constant, because floating point formats vary among locales. Integers must be written with no separators for thousands. For example: System.Size:<12345

Self Assessment

Fill in the blanks:

1. …….. is the system against which the software being built will run on.
2. It is not possible to write safely a canonic query that involves a …….. constant, because floating point formats vary among locales.
3. To identify the …….. actors involved in the build process, autoconf provides three macros that take care of finding the so-called "canonic" values.
4. Some other tools, such as ……., will require discovery of canonic systems by themselves.

Notes

5. One common mistake is to "go all the way" and always use the …….. macro, or its misnamed predecessor AC_CANONIC_SYSTEM.
6. Canonic syntax was introduced for …….. in Windows 7.
7. When coding queries in applications running on Windows 7 and later, you must use canonic syntax to programmatically generate …….. .

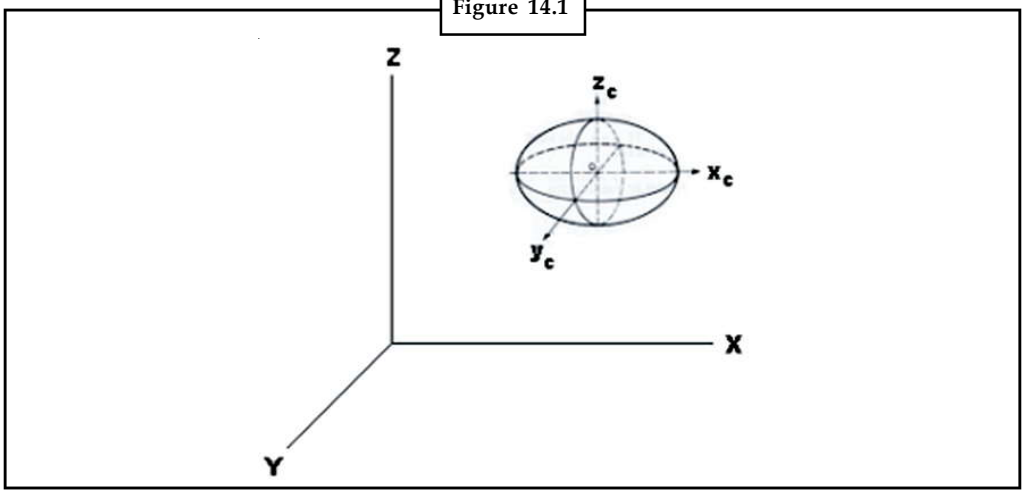
14.2 Specification Translation

Reduction of the general equation of the second degree to canonic form

Any equation of the second degree

1. $f(x, y, z) = ax^2 + by^2 + cz^2 + 2fyz + 2gxz + 2hxy + 2px + 2qy + 2rz + d = 0$

Figure 14.1



can be reduced to one of 17 different canonic forms by a suitable translation and rotation. Each canonic form represents a quadric surface. Figure 14.1 shows a quadric surface (an ellipsoid) along with its canonic coordinate system x_c-y_c-z_c located at some point (x_0, y_0, z_0) in space (as referred to the X-Y-Z system). Reduction of a particular second degree equation to canonic form involves the following steps:

1. Determining the location (x_0, y_0, z_0) of the origin of the canonic system x_c-y_c-z_c of the surface.
2. Determining the orientation of the x_c-y_c-z_c system (as referred to the X-Y-Z system).
3. Determining the expression for our equation as expressed with respect to the x_c-y_c-z_c system by performing those substitutions associated with a translation of the X-Y-Z system to the point (x_0, y_0, z_0) and then a rotation to the orientation to the x_c-y_c-z_c system.

Determining the Origin (x_0, y_0, z_0) of the Canonic System

How is the point (x_0, y_0, z_0) found? If the quadratic surface has a center, point (x_0, y_0, z_0) corresponds to a center. A quadric surface may have a single center, a line of centers, or a plane of centers. If there is more than one center, translation to any center will do. Of the 17 quadric surfaces, 14 have centers. We compute the coordinates of a center using the equation:

$$\begin{bmatrix} a & h & g \\ h & b & f \\ g & f & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} = 0$$

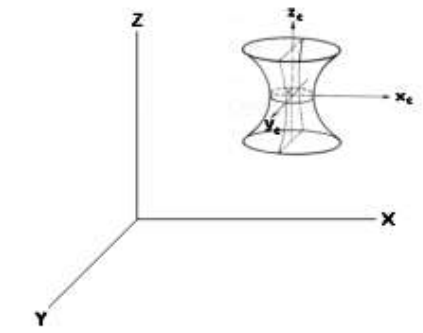
Notes

Three surfaces, the elliptic and hyperbolic paraboloids and the parabolic cylinder, do not have centers. In the case of these three, the point (x_0, y_0, z_0) corresponds to their vertices. The elliptic and hyperbolic paraboloids have a single vertex and the parabolic cylinder has a line of vertices. In the case of a line of vertices, translation to any vertex will do. These vertices must be found by some technique or procedure.

Determining the Orientation of the Canonic System

In the problem of translation to the canonic system origin we have noted that there may not be a single point that we must translate to but instead we may have a range of points that we can translate to (as in the case of a line of centers or a plane of centers). The same kind of situation exists in the problem of finding the orientation of the canonic coordinate system. In some cases there is a range of orientations that we can rotate to instead of just a single orientation. Consider the surface of revolution shown in Figure 14.2.

Figure 14.2



Notes Note that we can turn the canonic system x_c-y_c-z_c about the z_c axis through any angle from 0° to 360° and there is no change, one position is as good as another, the surface remains in canonic form.

There is a range of acceptable directions for the x-axis (and the y-axis). This same thing will occur with any surface of revolution. With any surface of revolution there will exist a plane of acceptable directions for two of the axes corresponding to a rotation of the system about the third axis, the axis of symmetry. Now consider the case of a sphere located in space. In this case the canonic system can have any orientation. No matter how it is oriented the sphere is still in canonic form. Here we can choose any direction for the x-axis and then rotate the system about the x-axis to point the y-axis in any direction we wish to give the system an orientation.

Did u know? How do we determine the directions of the x_c, y_c, and z_c axes of the canonic system?

We compute a set of eigenvectors associated with the quadric surface. The directions of the eigenvectors give the directions of the canonic system axes. In the case of a surface of revolution where there is a plane of acceptable directions for two of the axes there will be a corresponding plane of eigenvectors. We arbitrarily pick one eigenvector in the plane for one axis and one perpendicular to it for the other axis (the eigenvectors radiate out in all directions from a point).

Notes

Self Assessment

Fill in the blanks:

8. In some cases there is a range of orientations that we can rotate to instead of just a …….. .
9. If there is more than one center, ……... to any center will do.
10. Three surfaces, the elliptic and hyperbolic paraboloids and the parabolic cylinder, do not have …….. .

14.3 Canonic Recognition

Canonic Recognition is a non open-source Open CASCADE component to convert, if possible, a NURBS geometry (B-Spline/Bezier curve or surface) to its analytical form (primitive curve or surface). The component is crucial for algorithms implemented or best fitted to work on canonic geometry. This is especially important in CAM domain for CMM (Coordinate Measuring Machines) and other tooling applications.

The Canonic Recognition component is based on the Open CASCADE Shape Healing module and thereby takes advantage of its powerful capabilities.

The Canonic Recognition component includes:

1. Surfaces and curves simplification for a better reusability in downstream applications. The NURBS conversion depends on the original geometry definition.

   NURBS 3D curve can be translated to:

   - Line,
   - Circle or arc of circle,
   - Ellipse or arc of ellipse.

   NURBS surface can be translated to:

   - Plane,
   - Sphere,
   - Cylinder,
   - Cone,
   - Torus.

2. The capability to merge several adjacent topological objects (edges or faces) built on NURBS geometry into one single object based on an analytical geometry. This lowers complexity of the topological model and makes it more usable for other algorithms and, moreover, decreases usage of memory.
3. Tolerance management of the algorithms. The process of Canonic Recognition is handled by user-defined tolerance which is applied for recognition and approximation with canonic geometry.

   The conversion preserves topological structure of original shape and thereby retains connectivity of adjacent faces, edges with help of memorizing the history of modifications. Within applications Canonic Recognition can be optimally used after importing neutral or proprietary files (IGES, STEP, ACIS or others) or in other preparatory modules before using special algorithms.

Task What are the basic components of canonic recognition?

Notes

14.3.1 Benefits

Canonic Recognition brings added-value to:

- Open CASCADE based applications dealing mainly with analytical geometry such as CAM or Metrology applications.
- Performance and robustness of numeric algorithms (intersection, projection, etc.).
- All information appropriate for canonic geometry (radii, focuses, centers, equations, etc.) can be used in applications for display, quick search, statistics and so on.
- The easy-to-use API provides easy integration into existing or new Open CASCADE based applications as a part of CAD data import process or as a new functionality, which can be easily accessible via GUI.
- The demo application visually demonstrating the power of the component is available for Windows NT users.
- No third-party license is required to use this component.

Self Assessment

Fill in the blanks:

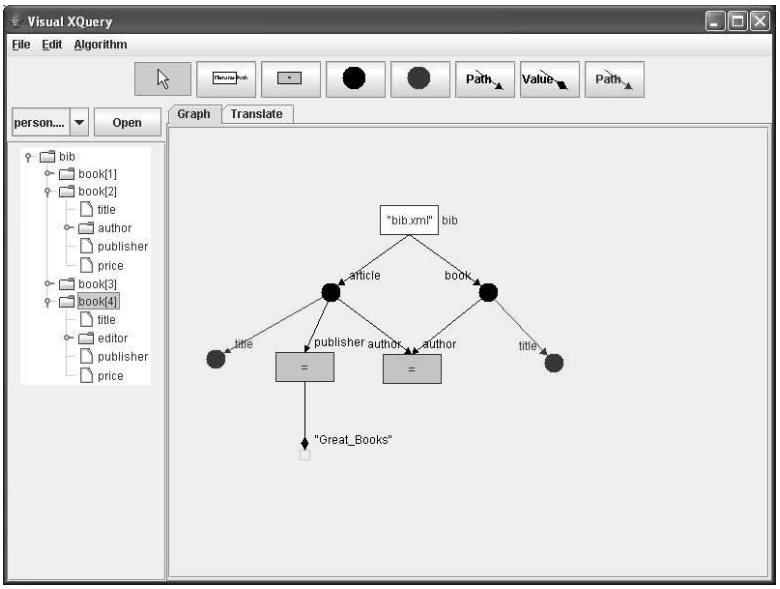
11. The process of Canonic Recognition is handled by user-defined tolerance which is applied for recognition and …….. with canonic geometry.
12. The …….. preserves topological structure of original shape.

14.4 Canonic Translation Algorithm

Main Window

Upon program activation, the user will be presented with the main window, depicted in Figure 14.3. The main window presents the toolbar, drawing panel and xml viewer. All the operations can be accessed via the menu.

Figure 14.3: Main Window



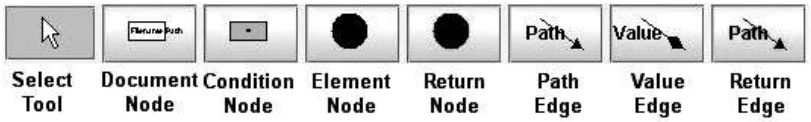
Notes

Toolbar

The toolbar shown in Figure 14.4 provides all the tools for drawing the query diagram. A node is drawn by selecting the node tool and clicking on the graph panel.

Notes An edge is drawn by selecting the edge tool and clicking on a start node and an end node.

Figure 14.4: Toolbar



Select Tool: Used to select and move graph components in the draw panel for editing.

Document Node: Used to create document nodes which accesses XML files specified in filename attribute and select elements using the pathname attribute to locate nodes.

Condition Node: Used to specify the general and value comparison operator through its condition attribute. A condition Node is only valid when it is connected to at least two edges.

Element Node: Correspond to element nodes in XML file.

Return Node: Used to specify query output node. A return edge is only valid when it is connected to a return edge.

Path Edge: Used to create a parent-child relationship between two nodes. The elements is selected using the pathname attribute to locate nodes.

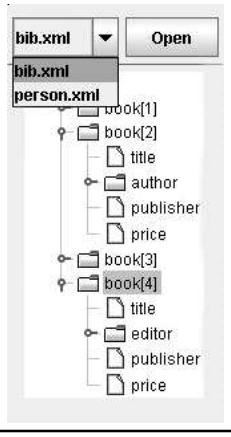
Value Edge: Valid only when connected to a condition node. Used to specify an absolute value for comparison with element.

Return Edge: Valid only when connected to a return node and an element node. Select elements using the pathname attribute to locate nodes for output.

XML Viewer

The XML Viewer shown in Figure 14.5 displays the hierarchy structure in a xml file. The user can use the combo box to select file for viewing and only one file can be viewed at one time.

Figure 14.5: XML Viewer



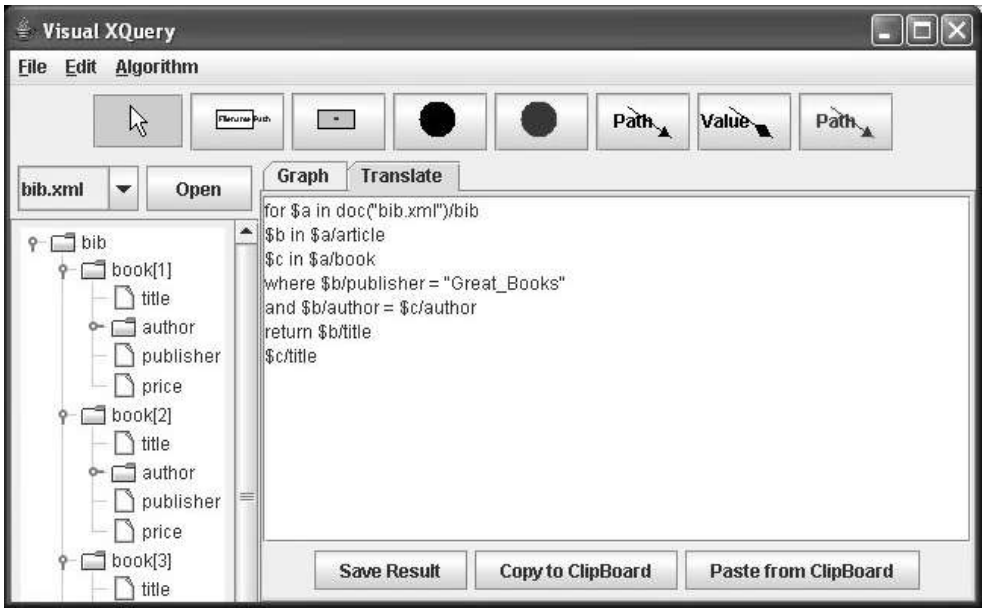
Notes

Translation

On selection of the translation tab, the program will translate the query graph into the relevant XML Query Language as shown in Figure 14.6. Users can edit, paste from clipboard, copy to clipboard or save the result into an xquery file (.xq). *

Notes Changes edited will not be reflected in graph.

Figure 14.6: Translation



File Menu

File > New: Clear the draw panel and create a new draw panel.

File > Open: Open a Visual XQuery graph file (.vxq) for editing.

File > Save: Save current graph on drawpanel into a Visual XQuery graph file (.vxq).

File >Import: Import a xquery file (.xq) into program for display into a Visual XQuery graph.

Edit Menu

Edit > Delete: Delete selected graph node or edge from graph.

Edit > Properties: Edit selected graph node or edge properties. Also can be performed through right-click on selected node or edge.

Algorithm Menu

Offers four translation algorithms:

Simple Translation: provides basic graph to XQuery translation.

Notes

Forward Translation: offers translation of reverse-axis path. Limited to 8 types of reverse-axis path listed in documentation.

Canonic Forward Translation: offers translation of reverse-axis path rewriting into canonic form. Supports path with multiple axes.

Enhanced Canonic Forward Translation: enhancement of canonic forward translation algorithm. More optimized query with reduction in joins, loops and document accesses.

Self Assessment

Fill in the blanks:

13. …….. is used to specify the general and value comparison operator through its condition attribute.
14. The main window presents the toolbar, drawing panel and …….. .

14.5 Canonic Systems and Formal Systems

A formal system is like a game in which tokens are manipulated according to rules in order to see what configurations can be obtained.

For examples: chess, checkers, go, tic-tac-toe. For examples: marbles, billiards, baseball.

All formal games have three essential features:

- They are token manipulation games;
- They are digital; and
- They are finitely playable.

Explanation

1. The tokens ... are just the "pieces" (E.g., markers or counters) with which the game is played. (E.g., chessmen, go stones, bits, 0s and Xs of tic-tac-toe).
2. Manipulating tokens means one or more of the following:
   - relocating them (e.g., moving them around on some board)
   - altering them (or replacing them with different ones)
   - adding new ones to the position; and/or
   - taking some away.
3. A digital system is a set of positive and reliable techniques (methods, devices) for producing and reidentifying tokens, or configurations of tokens, from some prespecified set of types.

   A positive technique is one that can succeed absolutely, totally and without qualification ... has the possibility of succeeding perfectly.

   We can substitute "writing" and "reading" for "producing" and "reidentifying", but only with two warnings: (1) writing isn't just making pen or pencil marks, but rather any kind of token manipulation that changes the formal position ... and (2) "reading" implies nothing about understanding (or even recognition) but only differentiation by type and position.
4. Finitely playable ... no infinite or magical powers required. A finite player is assumed to have some finite repertoire of specific primitive operations (e.g., move a chess piece, write a bit, read a bit) any finite number of times.

Notes

Is a Language a Real Formal System?

Chomsky and Halle and many other linguists assume that language IS such a system. Thus they typically assume that:

1. Phonetic segments are discrete formal tokens. And since they are, it follows that phonological segments (or phonemes) as well as words, phrases, sentences, etc. are formal objects as well since they are discrete combination of the a priori phonetic tokens.
2. The tokens are manipulated by formal rules or formally specified constraints, etc.<>
3. These tokens can be stored, read and manipulated essentially without error (thus, how many rules or constraints are involved is not an issue).

Caution These are powerful assumptions that not only are not fully defended, they appear to have no evidence at all except the intuitions of linguists.

Self Assessment

Fill in the blank:

15. A digital system is a set of positive and reliable techniques (methods, devices) for producing and …… .

Case Study

A Window to open Source (OS)

Ubuntu is an ancient African word meaning 'humanity to others,' informs www.ubuntu.com. The word also means 'I am what I am because of who we all are', adds the site.

"The vision for Ubuntu is part social and part economic: free software, available free of charge to everybody on the same terms, and funded through a portfolio of services provided by Canonical."

It was, therefore, a pleasant meeting that I recently had with Prakash Advani, Regional Manager - Asia Pacific, Canonical, Mumbai (http://bit.ly/F4TAdvaniP). "I have been using the latest Ubuntu 11.04 on a laptop that's almost three years now and I don't see any degradation of performance," says Prakash. "It performs the same that it used to three years back when I installed Ubuntu 9.10. After that we released three more versions and they have all worked beautifully on the same hardware." Our conversation continues over email.

Excerpts from the Interview:

How has been the growth of the open source operating system market in recent times?

Today, the open source operating systems market is growing much faster than proprietary offerings. Most of the growth is coming from dissatisfied Windows users who are finding their computers slowing down, leading to poor performance, or not being able to run the latest Windows 7. These users are replacing their current Windows system with Ubuntu.

Contd.....

Notes

What Ubuntu 11.04 brings is the opportunity to disrupt the status quo, where the majority of users simply stick with Windows. Rather than a loveless decision to replace the machine with another Windows one because that's all there is - there is a chance to make a really individual choice that has big advantages in terms of look, feel, and how the system operates.

In my experience, once a person settles down with an open source operating system, he/she starts enjoying the benefits and would rarely ever switch back to a proprietary system. They find that they are out of the endless cycle of planned obsolescence in the proprietary world, where newer versions are released.

The new version has, for example, a new format which is not actually needed. Now they have to get the new version of the software because everyone is sending them newer files. To get the new version they need to upgrade their operating system for which they need to buy new hardware. With open source operating system, they are out of this endless loop. In the open source world, LibreOffice supports the oldest and the newest file formats.

What are the main drivers of the growth in the open source OS market?

Open source operating system provides a completely legal alternative to pirating proprietary software. It works well, is virus-free, and can be installed on as many computers as you like without worrying about legal issues.

Also, open source software, such as Ubuntu, is becoming much more appealing to wider consumer audiences, as it is easier to use. This means that, today, a user doesn't have to understand code, or be technical to use open source operating systems to receive all the benefits.

At the same time, open source operating system continues to provide a solid platform for power users who want to download an open source operating system because they want to write code, customise it, develop new applications and functionality, and generally get under the hood.

Can you tell us about your experience in the government IT space? What are the challenges, expectations and savings?

Many governments are deploying open source technologies such as operating systems because they directly lead to significant cost savings of tax payers' money. Part of the government's challenges has been deployment on a large-scale.

Canonical has helped it address this by working closely with OEMs and providing a customised image, which has all the necessary drivers and applications required for their users. In India, we have engagements with Dell, Wipro, Lenovo, HCL, and eSys. Many of the states expect good support for local language, and open source projects can ensure quick and accurate translation through local community involvement, which makes it easy for anyone to modify an open source operating system and localise it to their requirements. In India, for example, Kerala, West Bengal, Assam and Gujarat are some of the states that have large Ubuntu deployments.

Do students benefit from an early exposure to open source operating system? How?

We have found that students with early exposure to open source have a deeper understanding of technology. The nature of open source allows students to take a peek at the code and see - and get excited about - how things work.

Today, students are not just users of technology, but they also want to have an understanding of how things work internally. Many of them end up developing their own

Contd...

Notes

solutions and contributing to technology. These students also do very interesting projects; for instance, a student whom I met recently was doing a project on a grid search engine.

Students can participate in the numerous open source projects that are around, or even start their own project. In a traditional software development company, cost of development is only 5-10 per cent of the total cost; the rest of the costs are in testing and marketing of the software. In open source, a lot of the testing and marketing costs get reduced as that is done by the community.

On the benefits to the academia

The benefits to educational institutions deploying open source is that their students and teachers can copy the same software and deploy them to as many home computers that they like. This is completely legal and we encourage people to do that. Students can also share the software with their friends. That way, with open source, students learn about sharing and caring for others.

Many projects done in a proprietary way never end up being actually used. Institutes should, therefore, encourage students doing their project in open source. This ensures that the code that they write is reusable by someone else. Even if the code is not perfect, someone else can fix the code. This ensures that open source projects continue to be active. Institutes can work with the industry to propose projects based on business needs.

With large enterprises now migrating to open source, this will open up lots of job opportunities for students trained in open source. Education institutions should get their students trained on open source so that they are ready to provide services to these enterprises.

What factors do CTOs and CIOs consider before choosing an open source operating system?

Before choosing an open source operating system, the organisation needs to start getting ready for adoption. The IT team will need to ensure that all of their software works with the operating system they plan to deploy. Alternatively, if new software is required, staff will need to be trained to familiarise them with new ways of working.

For example, if they want to use Linux and LibreOffice/OpenOffice they can first deploy LibreOffice on their Windows installation so users can become familiar. They should also ensure that any new applications being purchased or developed should be cross-platform. This will ensure that they are able to migrate to another operating system easily.

While CIOs can reap substantial savings by deploying open source, they should also budget for training and support. This ensures that there is professional support available to help them and ensure they have a smooth deployment.

14.6 Summary

- When using autoconf, there are three system definitions (or machine definitions) that are used to identify the "actors" in the build process; each definition relates to a similarly-named variable.
- Advanced Query Syntax (AQS) is the default query syntax used by Windows Search to query the index and to refine and narrow search parameters.
- AQS is employed by developers to build queries programmatically (and by users to narrow their search parameters).

Notes

- Canonic Recognition is a non open-source Open CASCADE component to convert, if possible, a NURBS geometry (B-Spline/Bezier curve or surface) to its analytical form (primitive curve or surface).
- The component is crucial for algorithms implemented or best fitted to work on canonic geometry.
- This is especially important in CAM domain for CMM (Coordinate Measuring Machines) and other tooling applications.

14.7 Keywords

AQS: Advanced Query Syntax

XML: Extensible Markup Language

14.8 Review Questions

1. The canonic syntax for the Boolean operators consists of the keywords AND, OR, and NOT, in all uppercase. Explain.
2. The most basic autoconf based build systems won't need to know any of these values, at least directly. Discuss.
3. If the system definition variables are used in a macro instead, we should use the AC_REQUIRE macro to ensure that they are executed before entering. Comment.
4. Explain how the operators <, >, =, and so forth, are not localized and are thus also part of the canonic syntax.
5. Canonic AQS was introduced in Windows 7 and must be used in Windows 7 and later to programmatically generate AQS queries. Explain.
6. If you do not use canonic syntax and your application is deployed in a locale. Scrutinize.
7. A quadric surface may have a single center, a line of centers, or a plane of centers. Substantiate.
8. The elliptic and hyperbolic paraboloids have a single vertex and the parabolic cylinder has a line of vertices. Comment.
9. The component is crucial for algorithms implemented or best fitted to work on canonic geometry. Explain.
10. The user can use the combo box to select file for viewing and only one file can be viewed at one time. Discuss.

Answers: Self Assessment

Fill in the blanks:

1. target (CTARGET
2. floating point
3. current
4. libtool
5. AC_CANONIC_TARGET
6. keywords
7. AQS queries
8. single orientation

9. translation
10. centers
11. approximation
12. conversion
13. Condition Node
14. xml viewer
15. reidentifying tokens

Notes

14.9 Further Readings

Books

Chattopadhyay, *System Software,* PHI Learning Pvt. Ltd.

Donovan, *Systems Programming,* Tata McGraw-Hill Education

I.A. Dhotre A.A. Puntambekar, *Systems Programming,* Technical Publications

M. Joseph, *System Software,* Firewall Media

Online links

http://en.wikipedia.org/wiki/Formal_system

http://www.cmpe.boun.edu.tr/~yurdakul/papers/CISS01.pdf

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in