SURESH GYAN VIHAR UNIVERSITY
Accredited by NAAC with 'A+' Grade

Master of Computer Application

(M.C.A.)

Introduction to Software

Semester-I

Author - Sonika Katta

SURESH GYAN VIHAR UNIVERSITY

Centre for Distance and Online Education
Mahal, Jagatpura, Jaipur-302025

EDITORIAL BOARD (CDOE, SGVU)

Dr (Prof.) T.K. Jain
Director, CDOE, SGVU

Dr. Dev Brat Gupta
Associate Professor (SILS) & Academic Head, CDOE, SGVU

Ms. Hemlalata Dharendra
Assistant Professor, CDOE, SGVU

Ms. Kapila Bishnoi
Assistant Professor, CDOE, SGVU

Dr. Manish Dwivedi
Associate Professor & Dy, Director, CDOE, SGVU

Mr. Manvendra Narayan Mishra
Assistant Professor (Deptt. of Mathematics) SGVU

Mr. Ashphaq Ahmad
Assistant Professor, CDOE, SGVU

Published by:

S. B. Prakashan Pvt. Ltd.

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com

© SGVU

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means (graphic, electronic or mechanical, including photocopying, recording, taping, or information retrieval system) or reproduced on any disc, tape, perforated media or other information storage device, etc., without the written permission of the publishers.

Every effort has been made to avoid errors or omissions in the publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice and it shall be taken care of in the next edition. It is notified that neither the publishers nor the author or seller will be responsible for any damage or loss of any kind, in any manner, therefrom.

For binding mistakes, misprints or for missing pages, etc., the publishers' liability is limited to replacement within one month of purchase by similar edition. All expenses in this connection are to be borne by the purchaser.

Designed & Graphic by : S. B. Prakashan Pvt. Ltd.

Printed at :

SYLLABUS
MCA (Semester - I)
Introduction to Software (MCA-102)

Learning Outcomes

- Distinguish between Operating Systems software and Application Systems software.
- Describe commonly used operating systems.
- Identify the primary functions of an Operating System.
- Describe the "boot" process.
- Identify Desktop and Windows features.
- Use Utility programs.

Unit-I:

Introduction to Software Engineering, Software Components, Software Characteristics, Software Crisis, Software Engineering Processes, Similarity and Differences from Conventional
Engineering Processes, Software Quality Attributes. Software Development Life Cycle (SDLC)
Models: Water Fall Model, Prototype Model, Spiral Model, Evolutionary Development Models,
Iterative Enhancement Models.

Unit-II:

Requirement Engineering Process: Elicitation, Analysis, Documentation, Review and Management of User Needs, Feasibility Study, Information Modeling, Data Flow Diagrams, Entity Relationship Diagrams, Decision Tables, SRS Document, IEEE Standards for SRS. Software Quality Assurance (SQA): Verification and Validation, SQA Plans, Software Quality
Frameworks, ISO 9000 Models, SEI-CMM Model.

Unit-III:

Basic Concept of Software Design, Architectural Design, Low Level Design: Modularization, Design Structure Charts, Pseudo Codes, Flow Charts, Coupling and Cohesion Measures, Design
Strategies: Function Oriented Design, Object Oriented Design, Top-Down and Bottom-Up Design. Software Measurement and Metrics: Various Size Oriented Measures: Halestead's Software Science, Function Point (FP) Based Measures, Cyclomatic Complexity Measures: Control Flow Graphs.

Unit-IV:

Testing Objectives, Unit Testing, Integration Testing, Acceptance Testing, Regression Testing,
Testing for Functionality and Testing for Performance, Top-Down and Bottom-Up Testing Strategies: Test Drivers and Test Stubs, Structural Testing (White Box Testing), Functional Testing (Black Box Testing), Test Data Suit Preparation, Alpha and Beta Testing of Products. Static Testing Strategies: Formal Technical Reviews (Peer Reviews), Walk Through, Code Inspection, Compliance with Design and Coding Standards.

Unit-V:

Software Maintenance and Software Project Management,Software as an Evolutionary Entity, Need for Maintenance, Categories of Maintenance: Preventive, Corrective and Perfective Maintenance, Cost of Maintenance, Software ReEngineering, Reverse Engineering. Software Configuration Management Activities, Change Control Process, Software Version Control, An Overview of CASE Tools. Estimation of Various Parameters such as Cost, Efforts, Schedule/Duration, Constructive Cost Models (COCOMO), Resource Allocation Models, Software Risk Analysis and Management.

References:

1. R. S. Pressman, Software Engineering: A Practitioners Approach, McGraw Hill.
2. Rajib Mall, Fundamentals of Software Engineering, PHI Publication.
3. K. K. Aggarwal and Yogesh Singh, Software Engineering, New Age International Publishers.
4. Pankaj Jalote, Software Engineering, Wiley.
5. Carlo Ghezzi, M. Jarayeri, D. Manodrioli, Fundamentals of Software Engineering, PHI Publication.
6. Ian Sommerville, Software Engineering, Addison Wesley.
7. Kassem Saleh,"Software Engineering", Cengage Learning.
8. Pfleeger, Software Engineering, Macmillan Publication.

MCA 02 Introduction to Software

		Page No
Block 1	**Basic Concepts**	**4**
Unit 1	Problem Solving Stages	5
Unit 2	Procedural Programming	30
Unit 3	Operating Systems	46
Unit 4	File Organization	91
Block 2	**Unix Operating System**	**131**
Unit 5	Unix Operating System	132
Unit 6	VI Editor	155
Unit 7	Input and Output	201
Unit 8	Pipes & Filters	209
Block 3	**Programming in Unix**	**221**
Unit 9	Shell Script	222
Unit 10	Control Statements	233
Unit 11	File System	243
Block 4	**Software Engineering**	**262**
Unit 12	Software Engineering	263

COURSE INTRODUCTION

System Software is a generic term referring to any computer software that is an essential part of the computer system. An Operating system is an obvious example, while an OpenGL or database library are less obvious examples. System Software contrasts with application software, which are programs that help the end-user to perform specific, productive tasks, such as word processing or image manipulation. System Software provides (user) interfaces for developing and running operating systems, administration and command interpreters ('shell'), communication systems, internet browsers, compiler / interpreters for programming language database management systems (DBMS). The most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs. Operating systems provide a software platform on top of which other programs, called application programs, can run. The application programs must be written to run on top of a particular operating system. Designing software is an exercise in managing complexity. The complexity exits within the software design itself, within the software organization of the company, and within the industry as a whole. A software design is very similar to systems design. It can span multiple technologies and often involves multiple sub-disciplines. Software specifications tend to be fluid, and change rapidly and often, usually while the design process is still going on. Introduction to Software is divided into Four Blocks. Block 1 describes the Basic Concepts of System Software. Block 2 explains the Unix Operating System. Block 3 discusses the Programming in Unix. Finally Block 4 is about Software Engineering.

BLOCK 1 INTRODUCTION

At the end of this block you will know the Basic concepts of System Software that describe the problem solving stages. The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer system. The most fundamental system program is the Operating System, which controls all the computer's resources and provides the base upon which the application programs can be written. Most computer users have had some experience with an operating system, but it is difficult to pin down precisely what an operating system is. Part of the problem is that operating systems perform two basically unrelated functions, and depending on who is doing the talking, you hear mostly about one function or the other.

Introduction to System Software is divided into Four Blocks. Block 1 consists of four Units.

Unit 1: deals with basic concepts of System Software, four stages of Problem solving, Types of Algorithm and Flow chart.

Unit 2: deals with the Procedural Programming, Loaders, Linkers and Graphical User Interface.

Unit 3: deals with the Operating System Concepts, Process Management and conditions for Deadlock.

Unit 4: deals with the File Organization, I/O device Management and Memory Management.

UNIT - 1

PROBLEM SOLVING STAGES

Structure

Overview

Learning Objectives

1.1 Introduction

  1.1.1 Polya's Four Stages of Problem Solving

1.2 Pseudo code

1.3 Algorithm

  1.3.1 Types of Algorithm

  1.3.2 Sorting Algorithms

1.4 Flowchart

  1.4.1 Overview

  1.4.2 Steps for Creating a Flowchart

Let us sum up

Answer to Learning Activities

References

OVERVIEW

Before we get too far into the discussion of problem solving, it is worth pointing out that we find it useful to distinguish between the three words "method", "answer" and "solution". By "method" we mean the means used to get an answer. This will generally involve one or more problem solving strategies. On the other hand, we use "answer" to mean a number, quantity or some other entity that the problem is asking for. Finally, a "solution" is the whole process of solving a problem, including the method of obtaining an answer and the answer itself.

method + answer = solution

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Understand the four stages of Problem Solving
- Familiar with Pseudo code
- Know the type of Algorithms
- Understand the Flow Chart

1.1 INTRODUCTION

How do we do Problem Solving?

There appear to be four basic steps. Polya enunciated these in 1945 but all of them were known and used well before then. And we mean well before then. The Ancient Greek mathematicians like Euclid and Pythagoras certainly knew how it was done.

1.1.1 Polya's Four Stages of Problem Solving are listed below.

1. Understand and explore the problem;
2. Find a strategy;
3. Use the strategy to solve the problem;
4. Look back and reflect on the solution.

Stage1: There is no chance of being able to solve a problem unless you first understand it. This process requires knowing not only what you have to find but also the key pieces of information that somehow need to be put together to obtain the answer. People will often not be able to absorb all the important information of a problem in one go. It will almost always be necessary to read a problem several times, both at the start and during working on it. During the solution process, people may find that they have to look back at the original

question from time to time to make sure that they are on the right track.

Stage 2: Here finding a strategy tends to suggest that it is a fairly simple matter to think of an appropriate strategy. However, there are certain problems where people may find it necessary to play around with the information before they are able to think of a strategy that might produce a solution. This exploratory phase will also help them to understand the problem better and may make them aware of some piece of information that they might have neglected after the first reading.

Having explored the problem and decided on a plan of attack, the **third problem-solving step**, solve the problem, can be taken. Hopefully now the problem will be solved and an answer obtained. During this phase it is important to keep a track of what they are doing. This is useful to show others what they have done and it is also helpful in finding errors should the right answer not be found.

At this point many children, especially mathematically able ones, will stop. But it is worth getting them into the habit of looking back over what they have done. There are several good reasons for this. First of all it is good practice for them to check their working and make sure that they have not made any errors. Second, it is vital to make sure that the answer they obtained is in fact the answer to the problem and not to the problem that they thought was being asked. Third, in looking back and thinking a little more about the problem leads to see another way of solving the problem. This new solution may be a nicer solution than the original and may give more insight into what is really going on. Finally, the better students especially, may be able to generalize or extend the problem.

Generalizing a problem means creating a problem that has the original problem as a special case. So a problem about three pigs may be changed into one which has any number of pigs.

1.2 PSEUDOCODE

What is pseudo code?

Pseudo code consists of short, English phrases used to explain specific tasks within a program's algorithm. Pseudo code should not include keywords in any specific computer languages. It should be written as a list of consecutive phrases. You should not use flowcharting symbols but you can draw arrows to show looping processes. Indentation can be used to show the logic in pseudo code as well. For example, a first-year, 9th grade Visual Basic programmer should be able to read and understand the pseudo code written by a 12th grade AP Data Structures student. In fact, the VB programmer could take the other student's pseudo code and generate a VB program based on that pseudo code.

Why is pseudo code necessary?

The programming process is a complicated one. You must first understand the program specifications, of course, and then you need to organize your thoughts and create the program. This is a difficult task when the program is not trivial (i.e. easy). You must break the main tasks that must be accomplished into smaller ones in order to be able to eventually write fully developed code. Writing pseudo code will save you time later during the construction & testing phase of a program's development.

How do I write pseudo code?

First you may want to make a list of the main tasks that must be accomplished on a piece of scratch paper. Then, focus on each of those tasks. Generally, you should try to break each main task down into very small tasks that can each be explained with a short phrase. There may eventually be a one-to-one correlation between the lines of pseudo code and the lines of the code that you write after you have finished pseudo coding.

It is not necessary in pseudo code to mention the need to declare variables. It is wise however to show the initialization of variables. You can use variable names in pseudo code but it is not necessary to be that specific. The word "Display" is used in some of the examples. This is usually general enough but if the task of printing to a printer, for example, is algorithmically different from printing to the screen, you may make mention of this in the pseudo code. You may show functions and procedures within pseudo code but this is not always necessary either. Overall, remember that the purpose of pseudo code is to help the programmer efficiently write code. Therefore, you must honestly attempt to add enough detail and analysis to the pseudo code. In the professional programming world, workers who write pseudo code are often not the same people that write the actual code for a program. In fact, sometimes the person who writes the pseudo code does not know beforehand what programming language will be used to eventually write the program.

Example:

Original Program Specification:

Write a program that obtains two integer numbers from the user. It will print out the sum of those numbers.

Pseudo code:

Prompt the user to enter the first integer
Prompt the user to enter a second integer
Compute the sum of the two user inputs
Display an output prompt that explains the answer as the sum
Display the result

Pseudo code is a written statement of an algorithm using a restricted and well-defined vocabulary. In the next section you look at the vocabulary in detail.

Input, output, iteration, decision and processing in pseudo code. It is useful to separate the pseudo code into several groups as shown here:

Group	Key words	Example
Input/Output	INPUT, READ Used to get values from a data source, a keyboard for instance DISPLAY Used to output values to a data sink, a screen or printer for instance	INPUT counter DISPLAY new_value

Iteration	REPEAT
	statement
	UNTIL <condition>
	
	The Repeat ... Until loop which was introduced in the introductory lesson on algorithms. The REPEAT loop executes the statement until the condition becomes true.
	
	DOWHILE <condition>
	statement
	END DOWHILE
	
	This is the While loop also introduced in the introductory lesson on algorithms. The WHILE loop executes the statement while the <condition> is true.
	
	FOR <var> = <start value> to <stop value>
	ENDFOR
	
	You haven't seen this loop before but it is a special case of the While loop. The FOR loop iterates for a fixed number of steps.
		SET count _ value TO 0
		REPEAT
		DISPLAY count_value
		ADD 1 TO count_value;
		UNTIL count_value >10
		
		DOWHILE count_value <10
		DISPLAY count_value
		count_value = count_value + 1
		END DOWHILE
		
		FOR count = 1 to 10
		DISPLAY count + count
		ENDFOR
		
		The statements inside the loops are indented to aid the readability of the pseudo code.

	The While and Repeat loops can be for a variable number of steps.	

Decision	IF <condition> THEN
	statement
	ENDIF
	IF <condition>
	THEN statement
	ELSE statement
	ENDIF
	Both of these decision forms were first introduced in the algorithms introductory lesson.
		IF count > 10
		THEN DISPLAY count
		ENDIF
		IF count > 10
		THEN DISPLAY 'count > 10'
		ADD 4 to sum
		ELSE DISPLAY 'count <= 10'
		ADD 3 to sum
		ENDIF

Processing	ADD, SUBTRACT, COMPUTE, SET	ADD 3 TO count SUBTRACT 5 FROM count SET count TO 12 COMPUTE 10 + count GIVING new_count

1.3 ALGORITHM

An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibn Musu Al-Khowarizmi, is defined as follows: Roughly speaking:

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

1.3.1 Types of Algorithm

1. **Linear:** a linear algorithm does something, once, to every object in turn in a collection. Examples: looking for a word in the dictionary that fits into a crossword. Adding up all the numbers in an array.
2. **Divide and Conquer:** the algorithms divide the problem's data into pieces and work on the pieces. For example conquering Gaul by dividing it into three pieces and then beating rebellious pieces into submission. Another example is the Stroud-Warnock for displaying 3D scenes: the view is divided into four quadrants, if a quadrant is simple, a simple process displays it, but if complex, the same Stroud-Warnock algorithm is

reapplied to it. Divide-and-conquer algorithms come in two forms. The division can be calculated to be precisely down the middle. Merge-sort splits and array into halves, sorts each of them and then merges the two into a single form. On the other hand, Tony Hoare's Quick sort and Tree sort make a rough split into two parts that can be sorted and rapidly joined together. On average these algorithms are faster than the precise divisions, but perform very badly on some data.

3. **Binary:** When we divide the data in two we call the algorithm a binary algorithm. The classic is binary search algorithm for hunting lions: divide the area into two and pick a piece that contains a lion.... repeat. This leads to an elegant way to find roots.

4. **Greedy algorithms** try to solve problems by selecting a best piece first and then working on the other pieces later. For example, to pack a knapsack, try putting in the biggest objects first and add the smaller one later. Or to find the shortest path through a maze, always take the shortest next step that you can see. Greedy algorithms don't always produce optimal solutions, but often give acceptable approximations to good ones.

5. **Iterative algorithms** start with a value and repeatedly change it in the direction of the solution. We get a series of approximations to the answer. The algorithm stops when two successive values get close enough. For example:

```
ALGORITHM square_root(a, epsilon)
 oldv=a
```

```
newv=a/w
WHILE | oldv - newv | < epsilon
   oldv =newv
   newv =(a+oldv * oldv)/(2*oldv)
END WHILE
END ALGORITHM
```

1.3.2 Sorting Algorithms

One of the fundamental problems of computer science is ordering a list of items. There's a plethora of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightening-fast results.

Sorting Algorithms

Bubble sort

Insertion sort

Merge sort

Quick sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant $O(n)$ level of complexity. General-case is an abysmal $O(n^2)$.

```
void bubbleSort(int numbers[], int array_size)
{
 int i, j, temp;

 for (i = (array_size - 1); i >= 0; i--)
 {
  for (j = 1; j <= i; j++)
  {
   if (numbers[j-1] > numbers[j])
   {
    temp = numbers[j-1];
    numbers[j-1] = numbers[j];
    numbers[j] = temp;
   }
  }
 }
}
```

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of $O(n \log n)$.

Elementary implementations of the merge sort make use of three arrays - one for each half of the data set and one to store the sorted list in. The below algorithm merges the arrays in-place, so only two arrays are required. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines.

```
void mergeSort(int numbers[ ], int temp[ ], int array_size)
{
  m_sort(numbers, temp, 0, array_size - 1);
}
void m_sort(int numbers[ ], int temp[ ], int left, int right)
{
  int mid;

  if (right > left)
  {
    mid = (right + left) / 2;
    m_sort(numbers, temp, left, mid);
    m_sort(numbers, temp, mid+1, right);
    merge(numbers, temp, left, mid+1, right);
```

```
 }
}
void merge(int numbers[ ], int temp[ ], int left, int mid, int
right)
{
 int i, left_end, num_elements, tmp_pos;
 left_end = mid - 1;
 tmp_pos = left;
 num_elements = right - left + 1;
 while ((left <= left_end) && (mid <= right))
 {
  if (numbers[left] <= numbers[mid])
  {
   temp[tmp_pos] = numbers[left];
   tmp_pos = tmp_pos + 1;
   left = left +1;
  }
  else
  {
   temp[tmp_pos] = numbers[mid];
   tmp_pos = tmp_pos + 1;
   mid = mid + 1;
  }
 }
```

```
  while (left <= left_end)
  {
    temp[tmp_pos] = numbers[left];
    left = left + 1;
    tmp_pos = tmp_pos + 1;
  }
  while (mid <= right)
  {
    temp[tmp_pos] = numbers[mid];
    mid = mid + 1;
    tmp_pos = tmp_pos + 1;
  }
  for (i=0; i <= num_elements; i++)
  {
    numbers[right] = temp[right];
    right = right - 1;
  }
}
```

The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

The recursive algorithm consists of four steps (which closely resemble the merge sort):

1. If there are one or less elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

```
void quickSort(int numbers[ ], int array_size)
{
 q_sort(numbers, 0, array_size - 1);
}
void q_sort(int numbers[ ], int left, int right)
{
 int pivot, l_hold, r_hold;
 l_hold = left;
 r_hold = right;
 pivot = numbers[left];
 while (left < right)
 {
  while ((numbers[right] >= pivot) && (left < right))
   right--;
```

```
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}
```

1.4 FLOWCHART

A flow chart is a pictorial representation showing all of the steps of a process.

A Flowchart is used for:

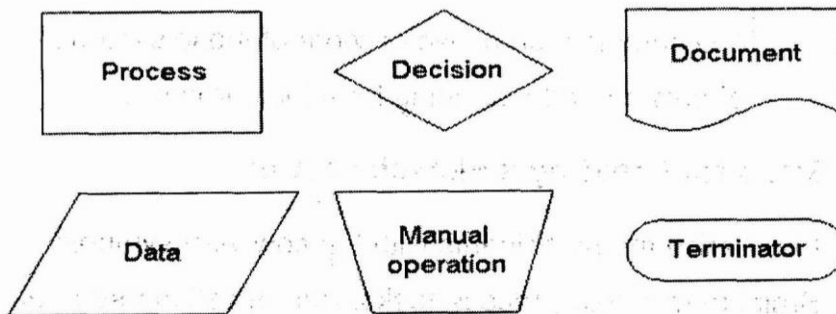
1. Defining and analyzing processes (example: What is the registration process for entering freshmen students?)
2. Building a step-by-step picture of the process for analysis, discussion, or communication purposes (example: Is it possible to shorten the length of time it takes for a student to complete the program?)
3. Defining, standardizing, or finding areas for improvement in a process.

1.4.1 Overview

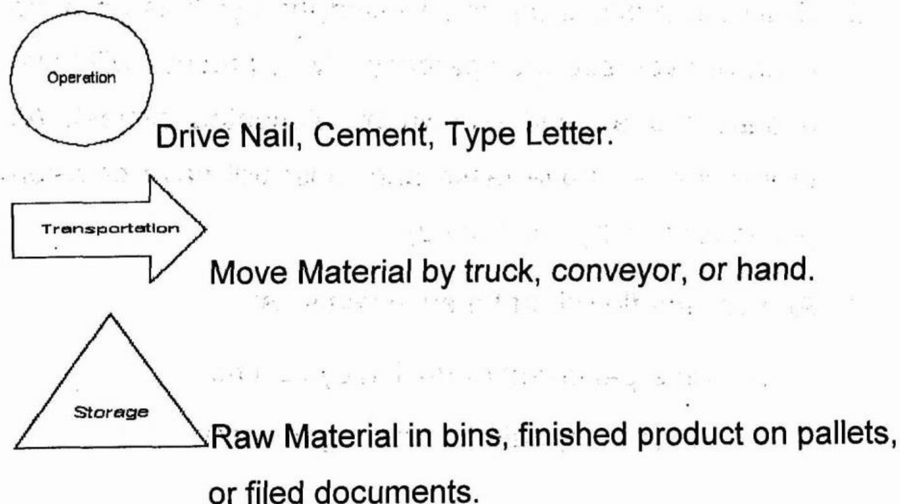
- **Quality Improvement Tool:** Flow charts are used specifically for a process.
- A flow chart is defined as a pictorial representation describing a process being studied or even used to plan stages of a project. Flow charts tend to provide people with a common language or reference point when dealing with a project or process.
- Four particular types of flow charts have proven useful when dealing with a process analysis: top-down flow chart, detailed flow chart, work flow diagrams, and a deployment chart. Each of the different types of flow charts tends to provide a different aspect to a process or a task. Flow charts provide an excellent form of documentation for a process, and quite often are useful

when examining how various steps in a process work together.

When dealing with a process flow chart, two separate stages of the process should be considered: the finished product and the making of the product. In order to analyze the finished product or how to operate the process, flow charts tend to use simple and easily recognizable symbols. The basic flow chart symbols below are used when analyzing how to operate a process.



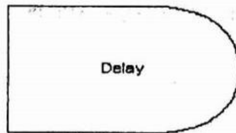
In order to analyze the second condition for a flow process chart, one should use the ANSI standard symbols. The ANSI standard symbols used most often include the following:



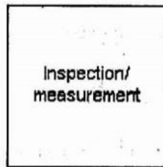
Drive Nail, Cement, Type Letter.

Move Material by truck, conveyor, or hand.

Raw Material in bins, finished product on pallets, or filed documents.



Wait for elevator, papers waiting, material waiting



Read gages, read papers for information, or check quality of goods.

Any combination of two or more of these symbols shows an understanding for a joint process.

1.4.2 Steps for Creating a Flowchart Are:

1. Familiarize the participants with the flowchart symbols
2. Brainstorm major process tasks. Ask questions such as "What really happens next in the process?", "Does a decision need to be made before the next step?", or What approvals are required before moving on to the next task?"
3. Draw the process flowchart using the symbols on a flip chart or overhead transparency. Every process will have a start and an end (shown by elongated circles). All processes will have tasks and most will have decision points (shown by a diamond).
4. Analyze the flowchart for such items as:
    - Time-per-event (reducing cycle time)
    - Process repeats (preventing rework)

- Duplication of effort (identifying and eliminating duplicated tasks)
- Unnecessary tasks (eliminating tasks that are in the process for no apparent reason)
- Value-added versus non-value-added tasks

CONSTRUCTION / INTERPRETATION tip for a flow chart.

- Define the boundaries of the process clearly.
- Use the simplest symbols possible.
- Make sure every feedback loop has an escape.
- There is usually only one output arrow out of a process box. Otherwise, it may require a decision diamond.

INTERPRETATION

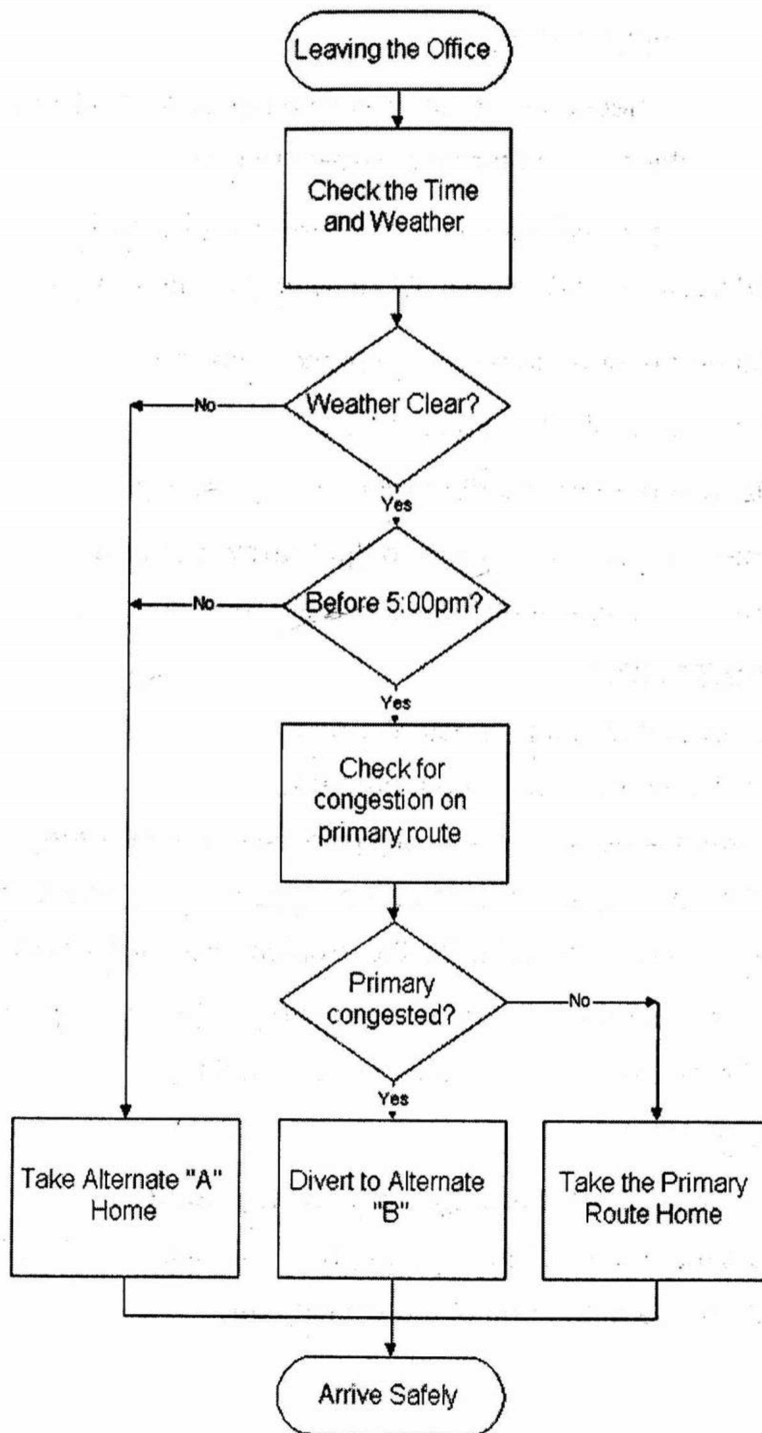
- Analyze flow chart of actual process.
- Analyze flow chart of best process.
- Compare both charts, looking for areas where they are different. Most of the time, the stages where differences occur is considered to be the problem area or process.

  Take appropriate in-house steps to correct the differences between the two separate flows.

EXAMPLE

Process Flow Chart- Finding the best way home

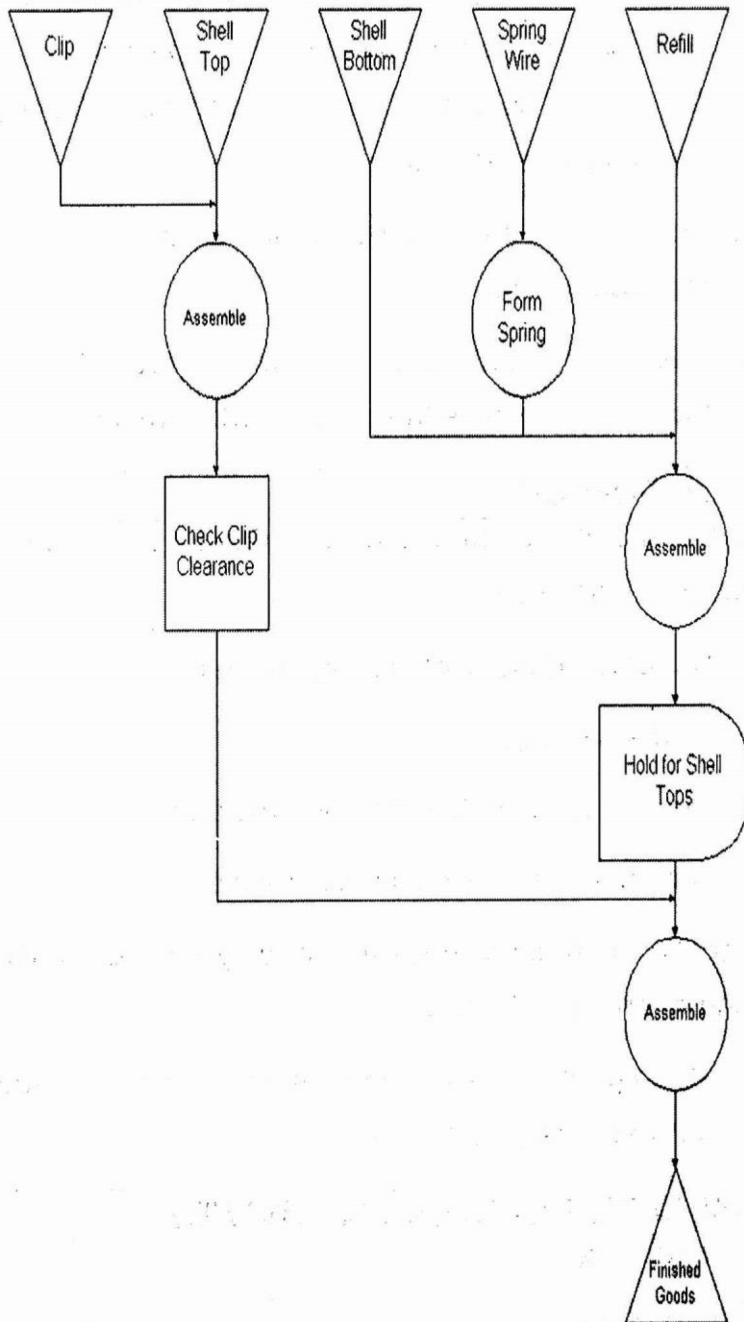
This is a simple case of processes and decisions in finding the best route home at the end of the working day.



Process Flow Chart- How a process works

(Assembling a ballpoint pen)

Ball-Point Pen Assembly



LEARNING ACTIVITIES

Fill in the Blanks:

1. **……………** a problem means creating a problem that has the original problem as a special case.
2. A ………… is a pictorial representation showing all of the steps of a process.
3. An ……… is a finite step-by-step procedure to achieve a required result.
4. The **………** splits the list to be sorted into two equal halves, and places them in separate arrays.

LET US SUM UP

At the end of this unit you have understood the Four Stages of Problem Solving are

- ➢ Understand and explore the problem;
- ➢ Find a strategy;
- ➢ Use the strategy to solve the problem;
- ➢ Look back and reflect on the solution.

An algorithm is a set of rules for carrying out calculation either by hand or on a machine.

- ✓ An algorithm is a finite step-by-step procedure to achieve a required result.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Generalizing
2. Flow chart
3. Algorithm
4. Merge sort

MODEL QUESTIONS

1. What is an Input device? Give 3 examples.
2. What is an Output device? Give 3 examples
3. Where do we save the results of the computers processing for later use?
4. What are the 2 main types of storage?
5. Where does the computer store information as it is working?
6. Write the four stages of Problem Solving.

REFERENCES

T.W. Pratt – Programming Languages, Design and Implementation – PHI

R.G. Dromey – How to solve it by Computer - PHI

UNIT - 2

PROCEDURAL PROGRAMMING

Structure

Overview

Learning Objectives

2.1 Procedural programming

    2.1.1 *Procedures* and Modularity

2.2 Linkers and Loaders

2.3 Elements of Real Programming Languages

2.4 Graphical User Interface

    2.4.1 GUI vs. CLI

Let us sum up

Answer to Learning Activities

References

OVERVIEW

Procedural programming is a programming paradigm based upon the concept of the *procedure call.* Procedures, also known as routines, subroutines, methods, or functions (not to be confused with mathematical functions, but similar to those used in functional programming) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or it.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Understand the Procedural programming
- Understand the Linkers and Loaders & DeadLock

❖ Familiar with Elements of Real Programming Languages

❖ Know the Graphical User Interface

2.1 PROCEDURAL PROGRAMMING

Procedural programming is often a better choice than simple sequential or unstructured programming in many situations which involve moderate complexity or which require significant ease of maintainability. Possible benefits:

- The ability to re-use the same code at different places in the program without copying it.
- An easier way to keep track of program flow than a collection of "GOTO" or "JUMP" statements. (Which can turn a large, complicated program into so-called "spaghetti code".)
- The ability to be strongly modular or structured.

2.1.1 Procedures and Modularity

Especially in large, complicated programs, modularity is often a desirable property. It can be implemented using procedures that have strictly defined channels for input and output, and usually also clear rules about what types of input and output are allowed or expected. Inputs are usually specified syntactically in the form of *arguments* and the outputs delivered as *return values.*

Scoping is another technique that helps keep procedures strongly modular. It prevents the procedure from accessing the variables of other procedures (and vice-versa), including previous instances of itself, without explicit authorization. This helps prevent confusion between variables

with the same name being used in different places, and prevents procedures from stepping on each other's feet.

Less modular procedures, often used in small or quickly written programs tend to interact with a large number of variables in the execution environment, which other procedures might also modify. The fact that lots of variables act as points of contact between various parts of the program is what makes it less modular. Because of the ability to specify a simple interface, to be self-contained, and to be reused, procedures are a convenient vehicle for making pieces of code written by different people or different groups, including through programming libraries.

(See Module (programming) and Software package.)

Comparison with imperative programming

Most or all extent procedural programming languages are also imperative languages, because they make explicit references to the state of the execution environment. This could be anything from *variables* (which may correspond to processor registers) to something like the position of the "turtle" in the Logo programming language (which could be anything from a cursor on the screen to an actual device which moves around on the floor of a room).

Some imperative programming forms, such as object-oriented programming, are not necessarily procedural.

Comparison with object-oriented programming

More sophisticated forms of modularity are possible with object-oriented programming, which is a more recent invention. Instead of dealing with procedures, inputs, and outputs, object-oriented programs pass around *objects*. Computation is

accomplished by asking an object to execute one of its internal procedures (or one it has inherited), possibly drawing on some of its internal state.

Procedural programming languages

Procedural programming languages facilitate the programmer's task in following a procedural programming approach. The canonical example of a procedural programming language is ALGOL. Others include Fortran, PL/I, Modula-2, and Ada. This list includes some languages that aren't exclusively procedural, such as Java, which was designed specifically for object-oriented programming.

- Ada
- BASIC
- C
- COBOL
- Fortran

2.2 LINKERS AND LOADERS

- The job of a **Linker** is to combine more than one separately assembled object files into one executable file.
- Difference between an object file and an executable program: Whereas a single object file might contain machine code for only one procedure or a set of procedures, an executable file must contain all the machine code needed for a particular program; it must contain the address of the first instruction to be executed.
- The job of a **loader** then is to copy an executable file into memory and initialize the PC register to the address of the first instruction.

- When the program finishes, control must somehow be returned to the operating system.

Definition:

Linker:

A program that takes as input the object code files of one or more separately compiled program modules, and links them together into a complete executable program, resolving references from one module to another.

Loader:

A program that takes as input an executable program, loads it into main memory, and causes execution to begin by loading the correct starting address into the PC register.

2.3 ELEMENTS OF REAL PROGRAMMING LANGUAGES

There are several elements which programming languages, and programs written in them, typically contain. These elements are found in all languages, not just C. If you understand these elements and what they're for, not only will you understand C better, but you'll also find learning other programming languages, and moving between different programming languages, much easier.

1. There are ***variables* or *objects***, in which you can store the pieces of data that a program is working on. Variables are the way we talk about memory locations (data), and are analogous to the ``registers" in our pocket calculator example. Variables may be *global* (that

is, accessible anywhere in a program) or *local* (that is, private to certain parts of a program).

2. There are ***expressions,*** which compute new values from old ones.

3. There are ***assignments*** which store values (of expressions, or other variables) into variables. In many languages, assignment is indicated by an equals sign; thus, we might have

   b = 3

   or

   c = d + e + 1

   The first sets the variable b to 3; the second sets the variable c to the sum of the variables d plus e plus 1. The use of an equals sign can be mildly confusing at first. In mathematics, an equals sign indicates equality: two things are stated to be inherently equal, for all time. In programming, there's a time element, and a notion of cause-and-effect: after the assignment, the thing on the left-hand side of the assignment statement is equal to what the stuff on the right-hand side was before. To remind yourself of this meaning, you might want to read the equals sign in an assignment as ``gets'' or ``receives'': a = 3 means ``a gets 3'' or ``a receives 3.''
   (A few programming languages use a left arrow for assignment

   a <-- 3

   to make the ``receives'' relation obvious, but this notation is not too popular, if for no other reason than that few character sets have left arrows in them, and the left

arrow key on the keyboard usually moves the cursor rather than typing a left arrow). If assignment seems natural and un confusing so far, consider the line

i = i + 1

What can this mean? In algebra, we'd subtract i from both sides and end up with

0 = 1

This doesn't make much sense. In programming, however, lines like

i = i + 1

are extremely common, and as long as we remember how assignment works, they're not too hard to understand: the variable i receives (its new value is), as always, what we get when we evaluate the expression on the right-hand side. The expression says to fetch i's (old) value, and add 1 to it, and this new value is what will get stored into i. So i = i + 1 adds 1 to i; we say that it *increments* i. (We'll eventually see that, in C, assignments are just another kind of expression.)

4. There are ***conditionals*** which can be used to determine whether some condition is true, such as whether one number is greater than another. (In some languages, including C, conditionals are actually expressions which compare two values and compute a ``true'' or ``false'' value.)

5. Variables and expressions may have ***types,*** indicating the nature of the expected values. For instance, you might declare that one variable is expected to hold a

number, and that another is expected to hold a piece of text. In many languages (including C), your ***declarations*** of the names of the variables you plan to use and what types you expect them to hold must be explicit. There are all sorts of data types handled by various computer languages. There are single characters, integers, and ``real'' (floating point) numbers. There are text strings (i.e. strings of several characters), and there are *arrays* of integers, reals, or other types. There are types which reference (point at) values of other types. Finally, there may be user-defined data types, such as *structures* or *records*, which allow the programmer to build a more complicated data structure, describing a more complicated object, by accreting together several simpler types (or even other user-defined types).

6. There are ***statements*** which contain instructions describing what a program actually does. Statements may compute expressions, perform assignments, or call functions (see below).

7. There are ***control flow constructs*** which determine what order statements are performed in. A certain statement might be performed only if a condition is true. A sequence of several statements might be repeated over and over, until some condition is met; this is called a *loop*.

8. An entire set of statements, declarations, and control flow constructs can be lumped together into a ***function*** (also called *routine*, *subroutine*, or *procedure*) which another piece of code can then *call* as a unit. When you call a function, you transfer control to it and wait for it to

do its job, after which it *returns* to you; it may also return a value as a result of what it has done. You may also pass values to the function on which it will operate or which otherwise direct its work.

Placing code into functions not only avoids repetition if the same sequence of actions must be performed at several places within a program, but it also makes programs easy to understand, because you can see that some function is being called, and performing some (presumably) well-defined subtask, without always concerning yourself with the details of how that function does its job. (If you've ever done any knitting, you know that knitting instructions are often written with little sub-instructions or patterns which describe a sequence of stitches which is to be performed multiple times during the course of the main piece. These sub-instructions are very much like function calls in programming.)

9. A set of functions, global variables, and other elements makes up a *program*. An additional wrinkle is that the source code for a program may be distributed among one or more ***source files.*** (In the other direction, it is also common for a suite of related programs to work closely together to perform some larger task, but we'll not worry about that ``large scale integration'' for now.)

10. In the process of specifying a program in a form suitable for a compiler, there are usually a few logistical details to keep track of. These details may involve the specification of compiler parameters or interdependencies between different functions and other parts of the program. Specifying these details often

involves miscellaneous syntax which doesn't fall into any of the other categories listed here, and which we might lump together as ``boilerplate.''

Many of these elements exist in a hierarchy. A program typically consists of functions and global variables; a function is made up of statements; statements usually contain expressions; expressions operate on objects. (It is also possible to extend the hierarchy in the other direction; for instance, sometimes several interrelated but distinct programs are assembled into a suite, and used in concert to perform complex tasks. The various ``office'' packages--integrated word processor, spreadsheet, etc.--are an example.)

As we mentioned, many of the concepts in programming are somewhat arbitrary. This is particularly so for the terms *expression*, *statement*, and *function*. All of these could be defined as ``an element of a program that actually does something.'' The differences are mainly in the level at which the ``something'' is done, and it's not necessary at this point to define those ``levels.'' We'll come to understand them as we begin to write programs.

An analogy may help: Just as a book is composed of chapters which are composed of sections which are composed of paragraphs which are composed of sentences which are composed of words (which are composed of letters), so is a program composed of functions which are composed of statements which are composed of expressions (which are in fact composed of smaller elements which we won't bother to define). Analogies are never perfect, though, and this one is weaker than most; it still doesn't tell us anything about what

expressions, statements, and functions really *are*. If ``expression'' and ``statement'' and ``function'' seem like totally arbitrary words to you, use the analogy to understand that what they are arbitrary words describing arbitrary levels in the hierarchical composition of a program, just as ``sentence,'' ``paragraph,'' and ``chapter'' are different levels of structure within a book.

The preceding discussion has been in very general terms, describing features common to most ``conventional'' computer languages. If you understand these elements at a relatively abstract level, then learning a new computer language becomes a relatively simple matter of finding out how that language implements each of the elements. (Of course, you can't understand these abstract elements in isolation; it helps to have concrete examples to map them to. If you've never programmed before, most of this section has probably seemed like words without meaning. Don't spend too much time trying to glean all the meaning, but do come back and reread this handout after you've started to learn the details of a particular programming language such as C.)

Finally, there's no need to overdo the abstraction. For the simple programs we'll be writing, in a language like C, the series of calculations and other operations that actually takes place as our program runs is a simpleminded translation (into terms the computer can understand) of the expressions, statements, functions, and other elements of the program. Expressions are evaluated and their results assigned to variables. Statements are executed one after the other, except when the control flow is modified by if/then conditionals and

loops. Functions are called to perform subtasks, and return values to their callers, which have been waiting for them.

2.4 GRAPHICAL USER INTERFACE

Abbreviated *GUI* (pronounced *GOO-ee*). A program interface that takes advantage of the computer's graphics capabilities to make the program easier to use. Well-designed graphical user interfaces can free the user from learning complex command languages. On the other hand, many users find that they work more effectively with a command-driven interface, especially if they already know the command language.

Graphical user interfaces, such as Microsoft Windows and the one used by the Apple Macintosh, feature the following basic components:

- **Pointer:** A symbol that appears on the display screen and that you move to select objects and commands. Usually, the pointer appears as a small angled arrow. Text -processing applications, however, use an *I-beam pointer* that is shaped like a capital *I*.
- **Pointing device**: A device, such as a mouse or trackball that enables you to select objects on the display screen.
- **Icons:** Small pictures that represent commands, files, or windows. By moving the pointer to the icon and pressing a mouse button, you can execute a command or convert the icon into a window. You can also move the icons around the display screen as if they were real objects on your desk.

- **Desktop:** The area on the display screen where icons are grouped is often referred to as the desktop because the icons are intended to represent real objects on a real desktop.
- **Windows:** You can divide the screen into different areas. In each window, you can run a different program or display a different file. You can move windows around the display screen, and change their shape and size at will.
- **Menus:** Most graphical user interfaces let you execute commands by selecting a choice from a menu.

The first graphical user interface was designed by Xerox Corporation's Palo Alto Research Center in the 1970s, but it was not until the 1980s and the emergence of the Apple Macintosh that graphical user interfaces became popular. One reason for their slow acceptance was the fact that they require considerable CPU power and a high-quality monitor, which until recently were prohibitively expensive.

In addition to their visual components, graphical user interfaces also make it easier to move data from one application to another. A true GUI includes standard formats for representing text and graphics. Because the formats are well-defined, different programs that run under a common GUI can share data. This makes it possible, for example, to copy a graph created by a spreadsheet program into a document created by a word processor.

Many DOS programs include some features of GUIs, such as menus, but are not *graphics based.* Such interfaces

are sometimes called *graphical character-based user interfaces* to distinguish them from true GUIs.

Types of GUIs

GUIs that are not PUIs are most notably found in computer games, and advanced GUIs based on virtual reality are now frequently found in research. Many research groups in North America and Europe are currently working on the Zooming User Interface or ZUI, which is a logical advancement on the GUI, blending some 3D movement with 2D or "2 and a half D" vectorial objects.

Some GUIs are designed for the rigorous requirements of vertical markets. These are known as "application specific GUIs." One example of such an application specific GUI is the now familiar touchscreen point of sale software found in restaurants worldwide and being introduced into self-service retail checkouts. First pioneered by Gene Mosher on the Atari ST computer in 1986, the application specific touchscreen GUI has spearheaded a worldwide revolution in the use of computers throughout the food & beverage industry and in general retail.

Other examples of application specific touchscreen GUIs include the most recent automatic teller machines, airline self-ticketing, information kiosks and the monitor/control screens in embedded industrial applications which employ a real time operating system (RTOS). The latest cell phones and handheld game systems also employ application specific touchscreen GUI.

2.4.1 GUI vs. CLI

GUIs were introduced in reaction to the steep learning curve of ***Command Line Interfaces*** (CLI), text-based user

interfaces requiring commands to be typed on the keyboard. Since the command words in CLIs are usually numerous and composable, very complicated operations can be invoked using a relatively short sequence of words and symbols. This leads to high levels of efficiency once the many commands are learned, but reaching this level can take a while because the command words aren't easily discoverable. WIMPs, on the other hand, present the user with numerous widgets that represent and can trigger some of the system's available commands.

WIMPs extensively use modes as the meaning of all keys and clicks on specific positions on the screen are redefined all the time. CLIs use modes only in the form of a current directory.

Most modern operating systems provide both a GUI and a CLI, although the GUIs usually receive more attention. The GUI is usually WIMP based, although occasionally other metaphors surface, such as Microsoft Bob, 3dwm or (partially) FSV. Applications may also provide both interfaces, and when they do the GUI is usually a WIMP wrapper around the CLI version. The latter used to be implemented first because it allowed the developers to focus exclusively on their product's functionality without bothering about interface details such as designing icons and placing buttons. Nowadays, the GUI is no longer an optional part of a successful application because users have grown accustomed to the ease of use provided by their familiar GUIs.

LEARNING ACTIVITIES

Fill in the Blanks :

1. .... is a programming paradigm based upon the concept of the **procedure call.**
2. .... is another technique that helps keep procedures strongly modular.
3. The job of a ....is to combine more than one separately assembled object files into one executable file.

LET US SUM UP

At the end of this unit you have understood the Procedural programming, the job of a Linker, the job of a Loader, Graphical User Interface,

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Procedural programming
2. Scoping
3. Linker

MODEL QUESTION

1. Distinction between two types of scheduling:

REFERENCES

T.W. Pratt – Programming Languages, Design and Implementation – PHI

R.G. Dromey – How to solve it by Computer - PHI

UNIT - 3

OPERATING SYSTEMS

Structure

Overview

Learning Objectives

3.1 Introduction to Operating Systems

3.1.1 Operating System Concepts

3.2 Process Management

3.2.1 Multiprogramming

3.2.2 Multi-tasking

3.2.3 Time Sharing

3.2.4 CPU Scheduling

3.3 Introduction to Deadlock

3.3.1 Conditions for Deadlock

3.3.2 Deadlock Modeling

3.3.3 Deadlock Avoidance

Let us sum up

Answer to Learning Activities

References

OVERVIEW

Computer software can be divided roughly into two kinds. System programs, which manages the operation of the computer itself, and application programs, which perform the actual work the user wants. The most fundamental system program is the Operating System. The interface between the Operating system and the user programs is defined by the set of "extended instructions" that the Operating system provides.

LEARNING OBJECTIVES

After completing this unit, you would be competent enough to:

- Understand the Operating Systems and Process Management
- Know the Conditions for Deadlock, Modeling and Avoidance

3.1 INTRODUCTION TO OPERATING SYSTEMS

This part presents the principal operation of an operating system as well as the usage of modern operating systems. It briefly describes the history, the basic components and utilities to support program development.

An operating system is a program that acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute programs. An operating system is an important part of almost every computer system. A computer system can roughly be divided into three components:

- The hardware (memory, CPU, arithmetic-logic unit, various bulk storage, I/O, peripheral devices...)
- Systems programs (operating system, compilers, editors, loaders, utilities...)
- Application programs (database systems, business programs...)

A computer system can be described or shown in Fig. 3.1

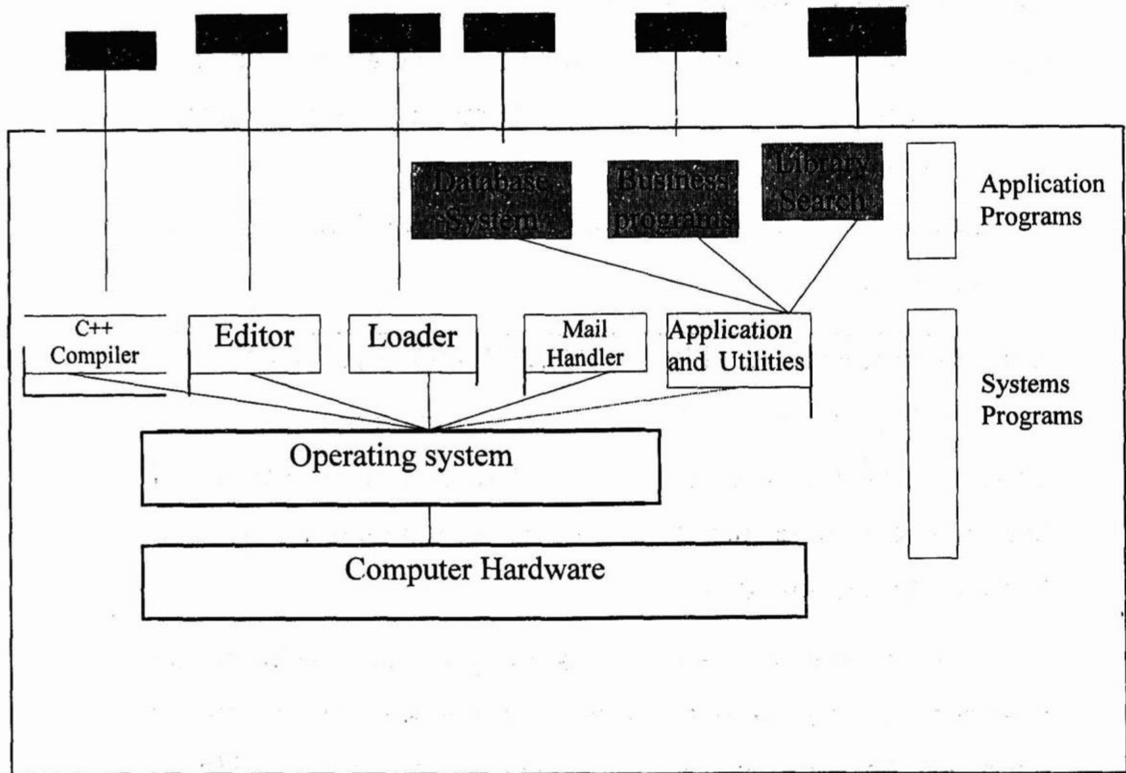


Fig. 3.1 Conceptual view of a computer system

The central processing unit (CPU) is located on chips inside the system unit. The CPU is the brain of the computer. This is the place where the computer interprets and processes information.

The operating system is the first component of the systems programs that interests us here. Systems programs are programs written for direct execution on computer hardware in order to make the power of the computer fully and efficiently accessible to applications programmers and other computer users. Systems programming is different from application programming because it requires an intimate knowledge of the computer hardware as well as the end users' needs. Moreover, systems programs are often large and more complex than

application programs, although that is not always the case. Since systems programs provide the foundation upon which application programs are built, it is most important that systems programs are reliable, efficient and correct.

In a computer system the hardware provides the basic computing resources. The applications programs define the way in which these resources are used to solve the computing problems of the users. The operating system controls and coordinates the use of the hardware among the various systems programs and application programs for the various users.

The basic resources of a computer system are provided by its hardware, software and data. The operating system provides the means for the proper use of these resources in the operation of the computer system. It simply provides an environment within which other programs can do useful work.

We can view an operating system as a resource Allocator. A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, files storage space, input/output devices etc.

The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system fairly and efficiently. An operating system is a control program. This program controls the execution of user programs to prevent errors and improper use of the computer.

Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of a computer system is to execute user programs and solve user problems.

The primary goal of an operating system is a convenience for the user. Operating systems exit because they are supposed to make it easier to compute with an operating system than without an operating system. This is particularly clear when you look at operating system for small personal computers.

A secondary goal is the efficient operation of a computer system. This goal is particularly important for large, shared multi-user systems. Operating systems can solve this goal. It is known that sometimes these two goals, convenience and efficiency, are contradictory.

While there is no universally agreed upon definition of the concept of an operating system, we offer the following as a reasonable starting point:

A computer's operating system (OS) is a group of programs designed to serve two basic purposes:

1. To control the allocation and use of the computing system's resources among the various users and tasks, and.
2. To provide an interface between the computer hardware and the programmer that simplifies and makes feasible the creation, coding, debugging, and maintenance of application programs.

Specifically, we can imagine that an effective operating system should accomplish all of the following:

- Facilitate creation and modification of program and data files through an editor program,
- Provide access to compilers to translate programs from high-level languages to machine language,
- Provide a loader program to move the compiled program code to the computer's memory for execution,
- Provide routines that handle the intricate details of I/O programming,
- Assure that when there are several active processes in the computer, each will get fair and no interfering access to the central processing unit for execution,
- Take care of storage and device allocation,
- Provide for long term storage of user information in the form of files, and
- Permit system resources to be shared among users when appropriate, and be protected from unauthorized or mischievous intervention as necessary.

Though systems programs such as editor and translators and the various utility programs (such as sort and file transfer program) are not usually considered part of the operating system, the operating system is responsible for providing access to these system resources.

3.1.1 Operating System Concepts

An operating system provides the environment within which programs are executed. To construct such an environment, the system is partitioned into small modules with

a well-defined interface. The design of a new operating system is a major task. It is very important that the goals of the system be will defined before the design begins. The type of system desired is the foundation for choices between various algorithms and strategies that will be necessary.

A system as large and complex as an operating system can only be created by partitioning it into smaller pieces. Each of these pieces should be a well defined portion of the system with carefully defined inputs, outputs, and function. Obviously, not all systems have the same structure. However, many modern operating systems share the system components outlined below.

1. Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when it's created, some initialization data (input) may be passed along. For example, a process whose function is to display on the screen of a terminal the status of a file, say F1, will get as an input the name of the

file F1 and execute the appropriate program to obtain the desired information.

We emphasize that a program by itself is not a process; a program is a passive entity, while a process is an active entity. It is known that two processes may be associated with the same program; they are nevertheless considered two separate execution sequences.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code. All of those processes can potentially execute concurrently.

The operating system is responsible for the following activities in connection with processes managed.

- The creation and deletion of both user and system processes
- The suspension is resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling.

2. Memory Management

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.

In order for a program to be executed it must be mapped to absolute addresses and loaded in to memory. As the

program executes, it accesses program instructions and data from memory by generating these absolute is declared available, and the next program may be loaded and executed.

In order to improve both the utilization of CPU and the speed of the compote's response to its users, several processes must be kept in memory. There are many different algorithms depends on the particular situation. Selection of a memory management scheme for a specific system depends upon many factor, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management.

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and reallocate memory space as needed.

Memory management techniques will be discussed in great detail in section 8.

3. Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modem computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers,

sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence the proper management of disk storage is of central importance to a computer system.

There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern.

The operating system is responsible for the following activities in connection with disk management

- Free space management
- Storage allocation
- Disk scheduling.

4. I/O System

One of the purposes of an operating system is to hide the peculiarities OS specific hardware devices form the user. For example, in Unix, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The I/O system consists of:

- A buffer caching system
- A general device driver code
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of a specific device.

We will discuss the I/O system in great length in unit 7.

5. File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; magnetic tape, disk, and drum are the most common forms. Each of these devices has it own characteristics and physical organization.

For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.

A file is a collection of related information defined by its creator. Commonly, files 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. It is a very general concept.

The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- The creation and deletion of files

- The creation and deletion of directory
- The support of primitives for manipulating files and directories
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.

6. Protection System

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorization from the operating system.

For example, memory addressing hardware ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices.

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An

unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

7. Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicated with each other through various communication lines, such as high speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies, and the problems of connection and security.

A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

8. Command Interpreter System

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system.

Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a

program which reads and interprets control statements is automatically executed. This program is variously called

(1) the control card interpreter, (2) the command line interpreter, (3) the shell (in Unix), and so on. Its function is quite simple: get the next command statement, and execute it.

The command statement themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

In the following sections of this Chapter we show four important components of the operating system. There are process management, file organization, input/output, and memory management,

3.2 PROCESS MANAGEMENT

The most central concept in any operating system is the concept of process: an abstraction of a running program. Everything else hinges on this concept, and it is important that the operating system designer know what a process is as early as possible.

3.2.1 Multi Programming

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and printing on a terminal or a printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of one second, it may work on several programs, thus giving the users the

illusion of parallelism. Sometimes people speak of pseudo parallelism to mean this rapid switching back and forth of the CPU between programs, to contrast it with the true hardware parallelism of the CPU computing while one or more I/O devices are running. Keeping track of multiple, parallel activities is not easy. Therefore, over the years operating system designers developed a model that makes parallelism easier to deal with. This model is the subject of the following paragraphs.

MCA 02 Introduction to Software

Block 1 : Programming Concepts: Introduction – Problem solving Stages – Pseudo code – Algorithm – Flowchart – Translators – Machine, Assembly and Procedural Languages – Linkers – Loaders –Elements of a programming language – Graphical User Interface (GUI) – Operating system concepts - Process Management – Multiprogramming – Multitasking – Timesharing – CPU Scheduling – Deadlock avoidance - I/O Device Management – Memory management – Partition – Partition – Page management – Swapping - File Management

Block 2 : UNIX Operating System : Foundations of UNIX operating system – Features of UNIX – Structure of UNIX operating system – File System – Different types of files – Command format – Text Manipulation commands – Text Editor – Line editors : ed,ex line editors – Vi Screen editor – Sed – File permissions – Super user, owner and other user categories and their privileges – Communication between users – Super user privileges

Block 3 : Programming in Unix : Shell Programming – Command Interpreter – Environment variables – Parameter passing – Shell programming language constructs – operators – Expression evaluation – Support for C programming Code – read, echo, if, case – Loops: do, for loops – System Administration – Adding user accounts – Changing privileges – File system mounting and un mounting – Running background processes

Block 4 : Software Engineering : Software Life Cycle – Role of software engineer – Qualities of a software product – Principles of software engineering – Trends in Software Development – 4GL and Natural Languages – System Investigations – Control of System Investigations – Case Tools

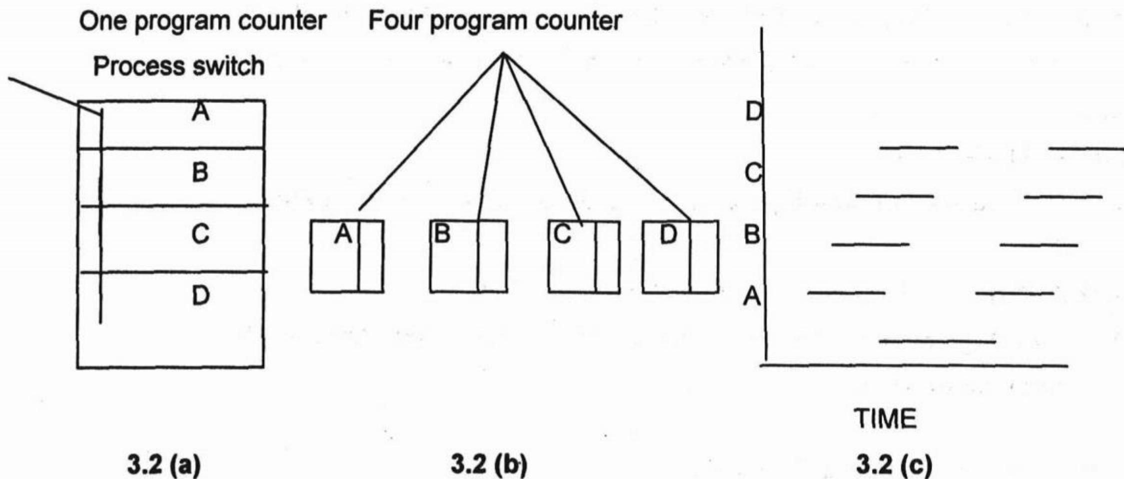
Books of Reference:

1. **T.W. Pratt – Programming Languages, Design and Implementation – PHI.**
2. **R.G. Dromey – How to solve it by Computer – PHI.**
3. **Operating system Design and Implementation by Andrew S. Tanenbaum – PHI**
4. **Software Engineering, Pressman**

In this model, all the runnable this program, often including the operating system is organized into a number of sequential processes, or just processes for short. A process is an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process, but to understand the system, it is much easier to think about a collection of process running in (pseudo) parallel, than to try to keep track of how the CPU switches form program to program. This rapid switching back and forth called multiprogramming, as we saw in the previous section. In Fig. 3.2(a) an exampled multiprogramming with four programs in given.

In Fig. 3.2(b) we see how this is abstracted into four processes, each with its own flow of control (i.e., its own program counter), and each one running independently of the other ones. In Fig. 3.2(c) we see that over a long enough time

interval, all the processes have made progress, but at any given instant only one process is actually running.



3.2 (a) 3.2 (b) 3.2 (c)

Fig:3.2 Multiprogramming of four programs.

Conceptual model of four independent, sequential processes. Only one program is active at any instant.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform, and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an I/O process that starts to move a magnetic tape, executes an idle loop 1000 times to let the tape get up to speed, and then issues a command to read the first record. If the CPU decides to switch to another process during the idle loop, the tape process might not run correctly. When a process has critical real-time requirements like this, that is, certain events absolutely must occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most

processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

The difference between a process and a program is subtle, but crucial. An analogy may help make this point clearer. Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well-stocked with the necessary input: flour, eggs, sugar, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher priority process (administering medical care), each having a different program (recipe vs. first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

Process Hierarchies

Operating systems that support the process concept must provide some way to create all the processes needed. In very simple systems, or in systems designed for running only a single application, it may be possible to have all the processes that will ever be needed be present when the system comes up. In most systems, however, some way is needed to create and destroy processes as needed during operation. In operating systems, system calls exist to create a process, load into memory, and start it running. Whatever the exact nature of the system call, processes need a way to create other processes. Note that each process has one parent but zero, one, two, or more children.

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input.

In Fig.3.3 we see a state diagram showing the three states a process may be in:

1. Running (actually using the CPU at the instant).
2. Blocked (unable to run until some external event happens).
3. Ready (runnable; temporarily stopped to let another process run).

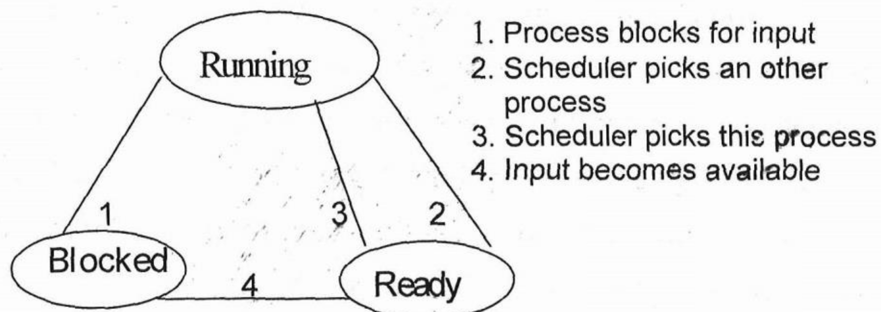


Fig. 3.3 A process can be in running, blocked or ready (also called runnable) state

Four transitions are possible among these states, as shown. Transition 1 occurs when a process discovers that it cannot continue. In some systems the process must execute a system call, BLOCK, to get into blocked state. In other systems, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their share and it is time for the first process to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one; we will look at it later in this unit. Many algorithms have been devised to try to balance the competing demands of efficiency for the systems as a whole and fairness to individual processes.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available.

Using the process model, it become much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When for example, a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. This entry contains information about the process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready state so that it can be restated later as if it had never been stopped.

In operating systems the process management, memory management, and file management are each handled by separate modules within the system, so the process table is partitioned, with each module maintaining the fields that is needs.

Standard Utilities

An operating system provides the appropriate environment for the programs to be executed. In this section, we consider what services an operating system provided, and how these are provided. The collection of services provided by the operating system is collect standard utilities. In the rest of this section we present some standard utilities. Two of them are system calls and systems programs.

The services provided for program and users differ from one operating system to another, but there are some common classes of services which can be identified. These operating system functions are provided for the convenience of the programmer to make the programming task easier.

- **Program Execution:** The system should be able indicated that to load a program into memory and run it. The program must be able to indicate that its execution ended, either normally or abnormally.
- **Input/Output Operations:** A running program may require input and output (I/O). This I/O may involve a file or an I/O device. For the specific devices, special functions may be desired (such as, rewind a tape drive, and so on). Since a user program cannot execute I/O operations directly, the operating system must provide some means to do so.

- **File System Manipulation:** The file system is of particular interest. Programs need to read and write files. User need to create and delete files by name.
- **Error Detection:** The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error OS power failure), in I/O devices (such as a parity error on tape, a card jam in the card reader, or the printer out of paper), or in the user program (such as an arithmetic overflow, an attempt to access illegal memory location, or using too much CPU time). For each type of error, the operating system should take the appropriate action.

In addition, another set of operating system functions exist not for the user but for the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

- **Resource Allocation:** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, while others (such as I/O devices) may have much more general request and release code.
- **Accounting:** We want to keep track of are used by which user how much and what kinds of computer resources. This record-keeping may be for the purpose of paying for the system and its operation, or simply for

accumulating usage static's. Usage static's may be a valuable tool in trying to configure the system to improve computing services.

- **Protection:** The owners of information stored in a multi-user computer system may want to control its use. When several disjoint jobs read being executed simultaneously in order to increase utilization, conflicting demands for various resources need to be reconciled fairly and scheduled reasonably.

System Calls

Since the actual mechanics of issuing a system call are highly machine-dependent, and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from high level language program. System calls provide the interface between a running program and the operating system.

Several languages, as C, PL/360, have been defined to replace assembly language for systems programming. These languages allow system calls to be made directly. Some Pascal systems also provide an ability to make system calls directly from Pascal program to the operating system. Most Fortran system provide similar capabilities, often by a set of library routines.

As an example of how system calls are used, consider writing a simple program to read data from one file and copy it to another file. The fist think the program will need is the name of the two files: the input file and output file. These can be specified in two ways. One approach is for the program to ask the user for the names of the two files. In an interactive system,

this will requite a sequence of system calls to first write a prompting message on the terminal, and then read from the terminal the characters which define the two files. Another approach, particularly used to batch system, is to specify the names of the files with control card. In this case, there must be a mechanism for passing these parameters from the control cards to the executing program.

Once the two file names are obtained, the program must open the input file and create the output file. Each of the operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name of that the file is protected against access. In these cases, the program should print a message on the console (another system call). If the input file exists, then we must create a new out file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (another system call). Another option, in an interactive system, is to ask the user (a sequence of system calls to output the prompting message and read the response from the terminal) whether to replace the existing file or abort.

Now that both files are set up, we enter a loop that reads from the input files (s system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached, or that there was a hardware failure in the read (such

as a parity error). The write operation may encounter various errors, depending upon the output device (no more disk space, physical end of tape, printer out of paper, and so on).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console (more system calls) and finally terminal normally (the last system call). As we can see, programs may make heavy use of the operating system. All interactions between the program and its environment must occur as the result of requests from the program to the operating system.

Most users never see this level of detail, however. The run-time supper system for most programming language provides a much simpler interface. For example, a write statement in Pascal or Fortran most likely is compiled into a call to a run-time support routine that issues the necessary system calls, check for errors, and finally returns to the user program. Thus most of the detail of the operating system interface is hidden from the user programmer by the compiler and its run-time support package.

System calls occur in different ways, depending upon the computer in use. Often more information is required than simply the identity of the desired system call. The exact type and amount of information varies according to the particular operating system and call. For example, to read a carve image, we may need to specify the file or device to use, and the address and length of the memory buffer into which it should be read. Of course, the device or file may be implicit in the call and, if the card images are always 80 characters, we may not need to specify the length.

Two general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in register. However, in some access there may be more parameters than registers. In this case the parameters stored in a block or t able in memory, and the address of the block is passed as a parameter in a register (Figure 3.4). Some operating system prefers this uniform interface, even when there are enough registers for all of the parameter for most cases.

System calls can be roughly grouped into three major categories, such as process OS job control, device and the file manipulation, and information maintenance. In the following, we summarize the types of system calls that may be provided by an operating system.

Register

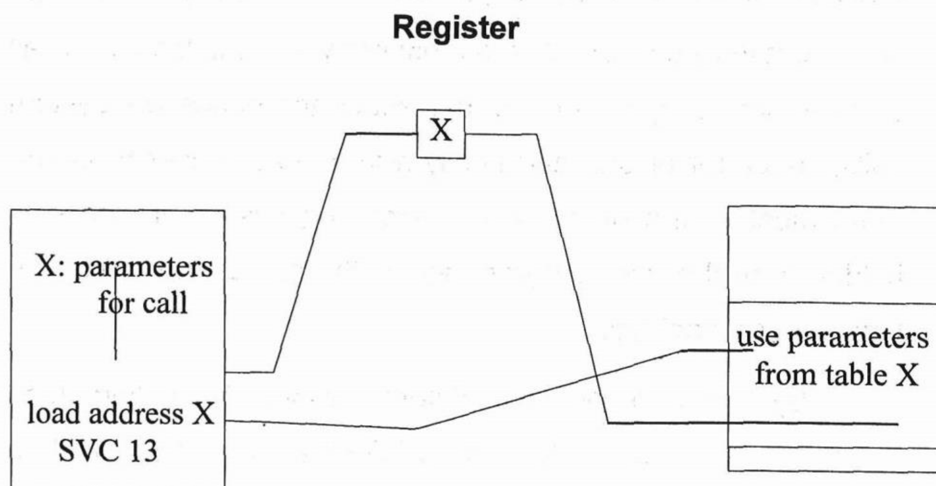


Fig. 3.4 Passing parameter as a table

Fig. Summarizes

- Process Control
- End, Abort
- Load, Execute

- Create Process, Terminate Process
- Get Process Attributes, Set Process Attribute
- Wait for Time
- Wait Event, Signal Event

- File Manipulation
  - Create File, Delete File
  - Open/Close
  - Read, Write, Reposition
  - Get File Attributes, Set File Attribute
- Device Manipulation
  - Request Device, Release Device
  - Read, Write, Reposition
  - Get Device Attributes, Set Device Attribute
- Information Maintenance
  - Get Time or Date, Set Time or Date
  - Get System Data, Set System Data
  - Get Process, File, or Device Attributes, Set Process, File, or Device Attributes.

Systems Programs

Another aspect of a modern system is its collection of systems programs. While we could write a program to copy one file to another, as shown above, it is unlikely that we would want to. In addition to the actual operating system monitor code, most system supplies a large collection of systems programs to solve

common problems and provide a more convenient environment for program development and execution.

Systems programs can be divided into several categories.

- **File Manipulation:** These program create, delete, copy, rename, print, dump, list, and generally manipulates files and directories.
- **Status Information:** Some program simply asks the system for the date, time, amount of available memory or disk space, number of users, or similar status information. That information is then formatted and printed to the terminal or other output device or file.
- **File Modification:** Several text editors may be available to create and modify the content of files stored on disk.
- **Programming Language Support:** Compliers, assemblers, and interpreters for common programming language (such as for Pascal, Basic and so on) are often provided with the operating system. Recently many of these programs are being priced and provided separately.
- **Program Loading and Execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher level language or machine language are also needed.

3.2.2 Multi-Tasking

To make a multi-tasking OS we need loosely to reproduce all of the features discussed in the last chapter for each task or process which runs. It is not necessary for each task to have its own set of devices. The basic hardware resources of the system are shared between the tasks. The operating system must therefore have a `manager' which shares resources at all times. This manager is called the `kernel' and it constitutes the main difference between single and multitasking operating systems.

COMPETITION FOR RESOURCES

Users - authentication

If a system supports several users, then each user must have his or her own place on the system disk, where files can be stored. Since each user's files may be private, the file system should record the *owner* of each file. For this to be possible, all users must have a *user identity* or *login name* and must supply a *password* which prevents others from impersonating them. Passwords are stored in a cryptographic (coded) form. When a user logs in, the OS encrypts the typed password and compares it to the stored version. Stored passwords are never decrypted for comparison.

Privileges and Security

On a multi-user system it is important that one user should not be able to interfere with another user's activities, either purposefully or accidentally. Certain commands and system calls are therefore not available to normal users directly. The *super-user* is a *privileged user* (normally the

system operator) who has permission to do anything, but normal users have restrictions placed on them in the interest of system safety.

For example: normal users should never be able to halt the system; nor should they be able to control the devices connected to the computer, or write directly into memory without making a formal request of the OS. One of the tasks of the OS is to prevent collisions between users.

Tasks – two mode operation

It is crucial for the security of the system that different tasks, working side by side, should not be allowed to interfere with one another (although this occasionally happens in microcomputer operating systems, like the Macintosh, which allow several programs to be resident in memory simultaneously). *Protection* mechanisms are needed to deal with this problem. The way this is normally done is to make the operating system all-powerful and allow no user to access the system resources without going via the OS.

To prevent users from tricking the OS, multi-user systems are based on hardware which supports *two-mode* operation: *privileged mode* for executing OS instructions and *user mode* for working on user programs. When running in *user mode* a task has no special privileges and must ask the OS for resources through system calls. When I/O or resource management is performed, the OS takes over and switches to *privileged* mode. The OS switches between these modes personally, so provided it starts off in control of the system, it will always remain in control.

At boot-time, the system starts in privileged mode.

During user execution, it is switched to user mode.

When interrupts occur, the OS takes over and it is switched back to privileged mode.

Other names for privileged mode are monitor mode *or* supervisor mode.

I/O and Memory protection

To prevent users from gaining control of devices, by tricking the OS, a mechanism is required to prevent them from writing to an arbitrary address in the memory. For example, if the user could modify the OS program, then it would clearly be possible to gain control of the entire system in privileged mode. All a user would have to do would be to change the addresses in the interrupt vector to point to a routine of their own making. This routine would then be executed when an interrupt was received in privileged mode.

The solution to this problem is to let the OS define a segment of memory for each user process and to check, when running in user mode, *every* address that the user program refers to. If the user attempts to read or write outside this allowed segment, a *segmentation fault* is generated and control returns to the OS. This checking is normally hard-wired into the hardware of the computer so that it cannot be switched off. No checking is required in privileged mode.

```
//*******************************************************************
//
// Example of a segmentation fault in user mode
//
//*******************************************************************

main()        // When we start, we are by definition in user mode.

{ int *ptr;

ptr = 0;      // An address guaranteed to NOT be in our segment.

cout << *ptr;

}
```

3.2.3 Time Sharing

There is always the problem in a multi-tasking system that a user program will go into an infinite loop, so that control never returns to the OS and the whole system stops. We have to make sure that the OS always remains in control by some method. Here are two possibilities:

The operating system fetches each instruction from the user program and executes it personally, never giving it directly to the CPU. The OS software switches between different processes by fetching the instructions it decides to execute. This is a kind of *software emulation.* This method works, but it is extremely inefficient because the OS and the user program are always running together. The full speed of the CPU is not realized. This method is often used to make simulators and debuggers.

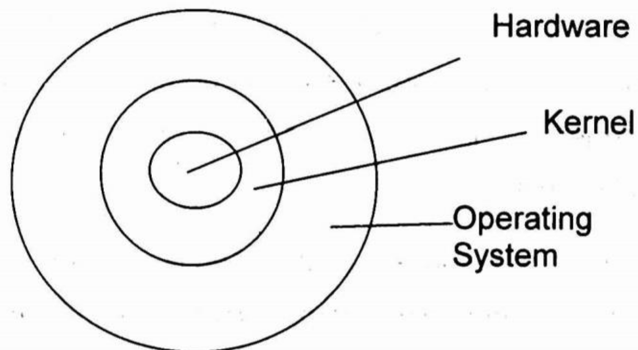


Fig. 3.5 Structure of kernel-based operating system

A more common method is to switch off the OS while the user program is executing and switch off the user process while the OS is executing. The switching is achieved by hardware rather than software, as follows. When handing control to a user program, the OS uses a hardware timer to ensure that control will return after a certain time. The OS loads a fixed time interval into the timer's control registers and gives control to the user process. The timer then counts down to zero and when it reaches zero it generates a *non-mask able interrupt*, whereupon control returns to the OS.

3.2.4 CPU Scheduling

Scheduling is a fundamental operating system function, since almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Consequently, its scheduling is often performed in the operating system.

The kernel-based design often is used for designing of the operating system. The kernel (more appropriately called the nucleus) is a collection of primitive facilities over which the rest of the operating system is built, using the functions provided by the kernel (see Fig. 3.5). Thus, a kernel provides an environment to build an operating system in which the designer has considerable flexibility because policy and optimization decisions are not made at the kernel level. An operating system is an orderly growth of software over the kernel, where all decisions regarding process scheduling, resource allocation, execution environment, file system, and resource protection etc. are made.

Consequently, a kernel is a fundamental set of primitives that allows the dynamic creation and control of process, as well as communication among them. Thus, the kernel only supports the notion of processes and does not include the concept of a resource. However, as operating systems have matured in functionality and complexity, more functionality has been relegated to the kernel. A kernel should contain a minimal set of functionality that is adequate to build an operating system with a given set of objectives.

We shall make a broad distinction between two types of scheduling:

Queuing: This is appropriate for serial or batch jobs like print spooling and requests from a server. There are two main ways of giving priority to the jobs in a queue. One is a *first-come first-served* (FCFS) basis, also referred to as *first-in first-out* (FIFO); the other is to process the *shortest job first* (SJF).

Round-robin: This is the time-sharing approach in which several tasks can coexist. The scheduler gives a short time-slice to each job, before moving on to the next job, polling each task round and round. This way, all the tasks advance, little by little, on a controlled basis.

These two categories are also referred to as *non-preemptive* and *preemptive* respectively, but there is a grey area.

Strictly non-preemptive: Each program continues executing *until* it has finished, or until it must wait for an event (e.g. I/O or another task). This is like Windows 95 and MacIntosh system 7.

Strictly preemptive: The system decides how time is to be shared between the tasks, and interrupts each process after its time-slice *whether it likes it or not*. It then executes another program for a fixed time and stops, then the next...etc.

Politely-preemptive? The system decides how time is to be shared, but it will not interrupt a program if it is in a *critical section*. Certain sections of a program may be so important that they must be allowed to execute from start to finish without being interrupted. This is like UNIX and Windows NT.

To choose an algorithm for scheduling tasks we have to understand what it is we are trying to achieve. i.e. What are the criteria for scheduling?

We want to maximize the efficiency of the machine. i.e. we would like all the resources of the machine to be doing useful work all of the time - i.e. not be idling during one process, when another process could be using them. The key to organizing the resources is to get the CPU time-sharing right, since this is the central `organ' in any computer, through which almost everything must happen. But this cannot be achieved without also thinking about how the I/O devices must be shared, since the I/O devices communicate by interrupting the CPU from what it is doing. (Most workstations spend most of their time idling. There are enormous amounts of untapped CPU power going to waste all over the world each day.)

We would like as many jobs to get finished as quickly as possible.

Interactive users get irritated if the performance of the machine seems slow. We would like the machine to appear fast for interactive users - or have a fast *response time*.

3.3 INTRODUCTION TO DEADLOCK

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software.

3.3.1 Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. **Mutual exclusion condition:** Each resource is either currently assigned to exactly one process or is available.

2. **Hold and wait condition:** Processes currently holding resources granted earlier can request new resources.

3. **No preemption condition:** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

4. Circular wait condition: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once?

Can a process hold a resource and ask for another? Can resources be preempted?

Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions.

3.3.2 Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 3-6(a), resource *R* is currently assigned to process *A*.

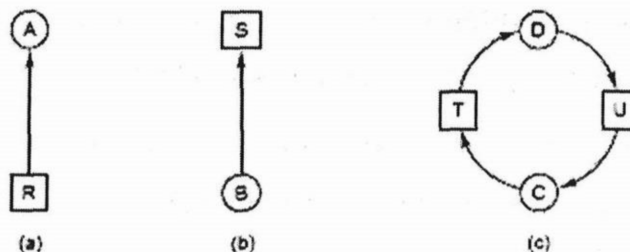


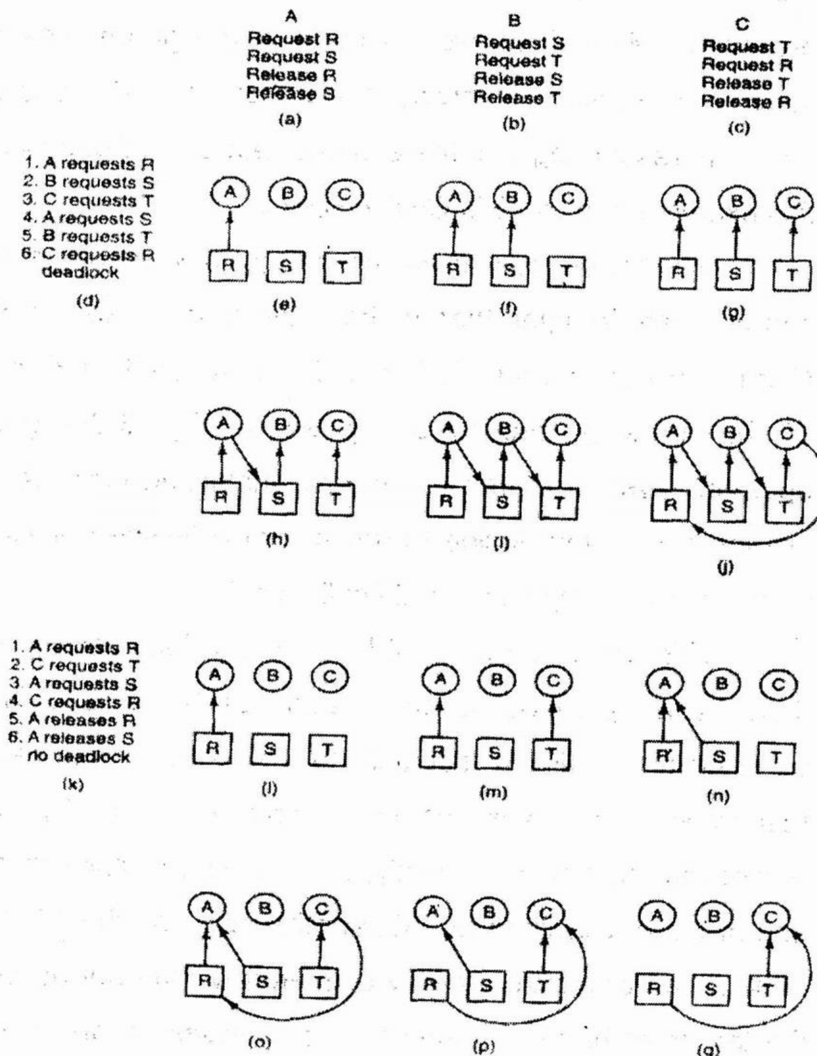
Figure 3-6. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig.3-6(b), process *B* is waiting for resource *S*. In Fig. 3-6(c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by process *D*. Process *D* is not about to release resource *T* because it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is *C–T–D–U–C*.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, *A*, *B*, and *C*, and three resources, *R*, *S*, and *T*. The requests and releases of the three processes are given in Fig. 3-6(a)-(c). The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run *C*.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 3-6(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 3-6(e)-(j). After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 3-6(h).



An example of how deadlock occurs and how it can be avoided.

In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 3-6(j). However, as we have

already mentioned, the operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe.

In Fig. 3-6, if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig. 3-6(k) instead of Fig. 3-6(d). This sequence leads to the resource graphs of Fig. 3-6(l)-(q), which do not lead to deadlock. After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* should eventually block when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished. For the moment, the point to understand is that resource graphs are a tool that let us see if a given request/release sequence leads to deadlock. We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock. Although our treatment of resource graphs has been for the case of a single resource of each type, resource graphs can also be generalized to handle multiple resources of the same type (Holt, 1972).

In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.

2. Detection and recovery. Let deadlocks occur, detect them, and take action.

3. Dynamic avoidance by careful resource allocation.

4. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

3.3.3 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra [1965], is the Banker's algorithm. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

Banker's Algorithm

In this analogy

Customers≡Processes

Units≡ resources, say, tape drive

Banker≡ Operating System

Customers	Used	Max
A	0	6
B	0	5
C	0	4
D	0	7

Available Units = 10

Fig.3.7(a)

In the above figure, we see four customers each of whom has been granted a number of credit nits. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	Used	Max	
A	1	6	Availa
B	1	5	ble
C	2	4	Units =
D	4	7	2

Fig. 3.7(b)

Safe State The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of figure 3.7(b) is safe because with 2 units left, the banker can delay any request except *C's*, thus letting *C* finish and release all four resources. With four units in hand, the banker can let either *D* or *B* have the necessary units and so on.

Unsafe State Consider what would happen if a request from B for one more unit were granted in above figure 3.7(b). We would have following situation

Customers	Used	Max	
A	1	6	
B	2	5	Available
C	2	4	Units = 1
D	4	7	

Fig. 3.7(c)

This is an unsafe state.

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later.

LEARNING ACTIVITIES

Fill in the Blanks:

1. The collection of services provided by the operating system is collect ....
2. An ....is a program that acts as an interface between a user of a computer and the computer hardware.
3. The ....is located on chips inside the system unit.

LET US SUM UP

At the end of this unit you have understood the Operating System. Operating Systems can viewed from two view points: resource managers and extended machines. Operating Systems have a long history, starting from the days when they replaced the operator, to modern multiprogramming systems.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Standard utilities.
2. operating system
3. central processing unit (CPU)

MODEL QUESTION

1. What are the two main functions of an Operating System?
2. What is multiprogramming?

UNIT - 4

FILE ORGANIZATION

Structure

Overview

Learning Objectives

4.1 File Organization

- 4.1.1 File Concept
- 4.1.2 File Operations
- 4.1.3 Access Methods
- 4.1.4 Directory Systems
- 4.1.5 Directory Structure Organization
- 4.1.6 File Protection

4.2 I/O Device Management

- 4.2.1 Device Controllers
- 4.2.2 Direct Memory Access (DMA)
- 4.2.3 Principles of I/ O software

4.3 Memory Management

- 4.3.1 Partitions
- 4.3.2 Swapping
- 4.3.3 Paging

Let us sum up

Answer to Learning Activities

References

OVERVIEW

For most users, the file system is the most visible aspect of an operating system. Files store data and programs. The operating system implements the abstract concept of a file by

managing mass storage devices, such as tapes and disks. Also files are normally organized into directories to ease their use, so we look at a variety of directory structures. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. This control is known as file protection.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Understand the File Concept, File Operations & Access Methods
- Understand the I/O Device Management
- Familiar with Memory Management
- Know the Swapping & Paging

4.1 FILE ORGANIZATION

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; magnetic tape, disk, and are the most common forms. Each of these devices has its own characteristics and physical organization.

4.1.1 File Concept

For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped by the operating system onto physical devices.

Consequently, a file is a collection of related information defined by its creator. Commonly, files 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. It is a very general concept.

A file is named and is referred to by its name. It has certain other properties such as its type, the time of its creation, the name (or account number) of its creator, its length, and so on.

The information in a file is defined by its creator. Many different types of information may be stored in a file: source programs, object programs, numeric data, text, payroll records, and so on. A file has a certain defined *structure* according to its use. A text file is a sequence of characters organized into lines (and possibly pages); a source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statement; an object file is a sequence of words organized into loader record blocks.

One major consideration is how much of this structure should be known and supported by the operating system. If an operating system knows the structure of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a use tries to print the binary object form of a program. This attempt normally produces garbage, but can be prevented if the operating system has been told that the file is a binary object program.

Often when the user attempts to execute an object program whose source file has been modified (edited) since the object file was produced; the source file will be recompiled automatically. This function ensures that the user always runs

an up-to-date object file. Otherwise, the user could waste a significant amount of time executing the old object file. Notice that in order for this function to be possible, the operating system must be able to identify the source file from the object file, check the time that each file was last modified or created, and determine the language of the source program (in order to use the correct compiler).

There are disadvantages to having the operating system know the structure of a file. One problem is the resulting size of the operating system. If the operating system defined fourteen different file structures, it must then contain the code to support these file structures correctly. In addition, every file must be definable as one of the file types supported by the operating system. Severe problems may result from new applications that require information structured in ways not supported by the operating system.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now if we (as a user) want to define an encrypted file to protect our files from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines but (apparently) random bits. Built though it may appear to be a binary file, it is not executable. As a result we may have misuse the operating system's file types mechanism, or modify or to impose (and support) no file type in the operating system. This approach has been adopted in Unix, among others. Unix considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the

operating system. This scheme provides maximum flexibility, but minimal support. Each application program must include its own code to interpret an input file into the appropriate structure.

Files are usually kept on disks. Disk systems typically have a well defined block size determined by the size of a sector. All disk I/O is in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length Packing a number of logical records into physical blocks is a common solution to this problem.

The operating system often defined all files to be simply a stream of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record is one byte. The file system automatically packs and unpacks bytes into physical disk blocks as necessary. Knowledge of the logical record size, physical block size and packing technique determine how many logical records are packed into each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered to be a sequence of blocks. All of the basic I/O function operates in terms of blocks. The conversion from logical records to physical block is a relatively simple software problem.

Notice that allocating disk space in blocks means that, in general, some portion of the last block of each file may be wasted. If each block is 512 bytes, then a file of 1949 bytes would be allocated 4 blocks (2048 bytes); the last 99 bytes would be wasted. The wasted bytes allocated to keep

everything in units of blocks (instead of bytes) are internal fragmentation. All file systems suffer from internal fragmentation. In general, large block sizes cause more internal fragmentation.

4.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations which can be performed on files. System calls are provided to create, write, read, rewind, and delete files. To understand how file systems are supported, let us look at these five file operations in more detail.

For convenience, assume the file system is disk-based. Let us consider what the operating system must do for each of the five basic file operations. It should then be easy to see how similar operations, such as renaming a file, would be implemented.

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and its location in the file system.
- **Writing a file:** To write a file, a system call is made specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The directory entry will need to store a pointer to the current end of the file. Using this pointer, the address of the next block can be computed and the information can be written. The

write pointer must be updated. In this way successive writes can be used to write a sequence of block to the file.

- **Reading a File:** To read from file, a system call specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated directory entry. And again, the directory will need a pointer to the next block to be read. Once that block is read, the pointer is updated.

In general, a file is either being read or written, thus although it would be possible to have two pointers, a read pointer and a write pointer, most systems have only one, a current file position. Both the read and write operations use this same pointer, saving space in the directory entry, and reducing the system complexity.

- **Rewind a file:** Rewinding a file need not involve any actual I/O rather the directory is searched for the appropriate entry, and the current file position is simply reset to the beginning of the file.

- **Delete a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space (so it can be reused by other files) and invalidate the directory entry.

It is known that all of the operations mentioned involve searching the directory for the entry associated with the named file. The directory entry contains all of the important information needed to operate on the file. To avoid this constant searching, many systems will open a file when it first becomes actively used. The operating system keeps a small table containing

information about all open files. When a file operation is requested, only this small table is seated, not the entire directory. When the file is no longer actively used, it is closed and removed from the table of open files.

Some systems implicitly open a file when the first reference is made to it. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that a file be opened explicitly by the programmer with a system call (open) before it can be used. The open operation takes a file name and searches the directory, copying the directory entry into the table of open files. The (open) system call will typically return a pointer to the entry in the table of open files. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching.

The five operations described above are certainly the minimal required file operations. More commonly, we will also want to edit the file and modify its contents. A common modification is appending new information to the end of an existing file. We may want to create a copy of a file, or copy it to an I/O device, such as a printer or a display. Since files are named objects, we may want to rename an existing file.

4.1.3 Access Methods

Files store information. This information must be accessed and read into computer memory before it is used. There are several ways the information in the file can be accessed. Some systems provide only one access method for files, and so the concepts are less important. On other systems, such as those of IBM, many different access methods are

supported, and choosing the right one for a particular application is a major design problem.

Sequential Access

Information in the file is processed in order, one record after the other. This is by far the most common mode of access of files. For example, editors programs usually access files in this fashion.

The read operation on a file automatically advances the file pointer. Similarly a write appends the new information to the end of the file, and advances the file pointer to the new end. Such a file can be rewound, and on some systems, a program may be able to skip forwarded or back n record, for some integer n (perhaps only for n = 1). This scheme is known as sequential access to a file. Sequential access is based upon a tape model of a file.

Direct Access

An alternative access method is direct access, which is based upon a disk model of a file. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct access file allows arbitrary blocks to be read of written. Thus we may read block 14, then red block 53, and then write block 7. There are no retractions on the order of reading or writing for a direct access file.

Direct access files are of great use for immediate access to large amounts of information. They are often used in accessing large data bases. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

The file operations must be modified to include the block number as a parameter. Thus we have to read block n, where n is the block number, rather than read the next, and write block n rather than write the next. An alternative approach is to retain read next and write next, as with sequential access, and to add an operation, position file to n, where n is the block number. Then to perform a read block n, we would position to block n and then read next.

The block number provided by the user to the operating system is normally a relative block number. A relative block number is an index relative to the beginning of the file. Thus the first relative block of the file is 0, the next...is 1, and so on, even through the actual absolute disk address of the block may be 14703 for the first block, and 14704 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed, and prevents the user from accessing portions of the file system which may not be part of his file. Some systems start their relative block number at 0; others start at 1.

Not all operating systems support both sequential and direct access for files. Some systems allow only that a file is defined as sequential or direct when it is created; such a file can only be accessed in a manner consistent with its declaration.

Other Access Methods

Other access method can be built on top of a direct access method. These additional methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find an entry in the file the index is consulted first.

With large files the index file itself may become too large to be kept in memory. One solution is then to create an index for the index file. The primary index file would contain pointers to secondary index files which then point to the actual data items.

4.1.4 Directory Systems

The prior discussion allows us to create files, read, write and reposition them, and, finally, to delete them. The files are represented by entries in a device directory or volume table of contents. The device directory records information, such as name, location, size, and type, for all files on that device.

A device directory may be sufficient for a single-user system with limited storage space. As the amount of storage and the number of users increase, however, it becomes increasingly difficult for the users to organize and keep track of all of the files on the file system. A directory structure provides a mechanism for organizing the many files in the file system. It may span device boundaries and include several different disk units. In this way, the user need be concerned only with the logical directory and file structure, and can completely ignore the problems of physically allocating space for files.

In fact, many systems actually have two separate directory structures: the device directory and the file directories. The device directory is stored on each physical device and describes all files on that device. The device directory entry mainly concentrates on describing the physical properties of the files: where it is, how long it is, how it is allocated, and so on. The file directories are a logical organization of the files on all devices. The file directory entry concentrates on logical properties of each name: file, file type, owing user, accounting

information, protection access code, and so on. A file directory entry may simply point to the device directory entry to provide physical properties or may duplicate this information. Our main interest now is with the file directory structure; device directories should be well understood.

The particular information kept for each file in the directory varies from operating system to operating system. The following is a list of some of the information which may be kept in a directory entry. Not all systems keep all this information, of course.

- File name: The symbolic file name.
- File type: For those systems that support different types.
- Location: A pointer to the device and location on that device of the file.
- Size: The current size of the file (in bytes, words of blocks) and the maximum allowed size.
- Current Position: A pointer to the current read or writes position in the file.
- Protection: Access control information the number of process that are currently using (have opened) this file.
- Time, date and process identification: This information may be kept for (a) creation, (b) last modification, and (c) last use. These can be useful for protection and usage monitoring.

It may take from 16 to over 100 bytes to record this information for the each file. In a system with a large number of files, the size of the directory itself may be hundreds of thousands of bytes. Thus the device directory may need to be stored on the device and brought into memory piecemeal, as needed. More specifically, when a file is open, the directory information about this file is brought into main memory. This information remains there until the file is closed.

If we think of the directory as a symbol table that translates file names into their directory entries, it becomes apparent that the directory itself can be organized in many ways. We want to be able to insert entries, delete entries, search for a named entry and list all the entries in the directory. Next we consider what data structure is used for the directory.

A linear list of directory entries requires a linear search to find a particular entry. This is simple to program but time consuming in execution. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then we can add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark it unused (a special name such as an all-blank name, or a used/unused bit in each entry), or attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory in the freed location and decrease the length of the directory. A link list can also be used to decrease the time to delete a file.

The real disadvantage of a linear list of directory entries is the linear search to find a file. A sorted list allows a binary search, and decreases the average search time. However, the search algorithm is more complex to program. In addition, the list must be kept sorted. This requirement may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. (Notice, however, that if we want to be able to produce a list of all files in a directory sorted by file name, we do not have to sort before listing). A linked binary tree might help here.

4.1.5 Directory Structure Organization

Many different file directory structures have been proposed, and are in use. The directory is essentially a symbol table. The operating system takes the symbolic file name and finds the named file. We examine some directory structures here. When considering a particular directory structure, we need to keep in mind the operations which are to be performed on a directory.

- Search: We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship between the files, we may want to be able to find all files that match a particular pattern.
- Create File: New files need to be created and added to the directory.
- Delete File: When a file is no longer needed, we want to remove it from the directory.

- List Directory: We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- Backup: For reliability, it is generally a good idea to save the contents and structure of the file system at regular intervals. This often consists of copying all files to magnetic tape. This provides a backup copy in use. In this case, the file can be copied to tape and the disk space of that file released for reuse by another file.

4.1.6 File protection

When information is kept in a computer system, a major concern is its protection from both physical damage and improper access. Protection can be provided in many ways. In a multi-user system, appropriate mechanisms are needed. The need for protection files is a direct result of the ability to access file.

Protection mechanisms provide controlled access by limiting the types of file access which can be made. Several different types of operations may be controlled:

1. Read: Read from the file.
2. Write: Write or rewrite the file.
3. Execute: Load the file into memory and execute it.
4. Append: Write new information at the end of the file.
5. Delete: Delete the file and free its space for possible reuse.

Introduction to input/output

One of the main functions of operating systems is to control all the computer's input/output (I/O) devices. It must issue commands to the devices, catch interrupts (I/O), and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. In this section we will look briefly at some of the principles of I/O hardware, and then we will look at I/O software in general.

Principles of I/O hardware

Different people look at I/O hardware in different ways. Electrical engineers look at it in term of chips, wires, power supplies, motors and all the other physical components that make up the hardware. Programmers look at the interface presented to the software the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. In the next two parts we will provide a little general background on I/O hardware as it relates to programming.

4.2 I/O DEVICE MANAGEMENT

I/O devices can be roughly divided into two categories: Block devices and character devices. A block device is one that stores information in fixes-size blocks, each one with its own address. Common block sizes range from 128 bytes to 1024 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. In other words, at any instant, the program can read or write any of the blocks. Disks are block devices.

If you look closely, the boundary between device that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device

because no matter where the arm currently is, it is always possible to seek t another cylinder and then wait for the required block to rotate under the head. Now consider a magnetic tape containing blocks of 1K bytes. If the tape drive is given a command to read block N, it can always rewind the tape and go forward until it comes to block N. This operation is analogous to a disk doing a seek, except that it take much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if in were possible to use magnetic tapes as block devices, that is stretching the point somewhat: they are normally not used that way.

The other type of I/O device is the character device. A character device delivers or accepts a stream of character, without regard to any block structure. It is not addressable and does not have any seek operation. Terminals, line printers, paper tapes, punched cards, network interface, mice (for pointing), and most other devices that are not disk like can be seen as character devices.

This classification scheme is not perfect. Some devices just do not fit in. However, the model of block and character devices is general enough that it can be used as a basis for making the I/O system device independent.

4.2.1 Device Controllers

I/O units typically consist of a mechanical component and an electronic component. It is often possible to separate the two portions to provide a more modular and general design. The electronic component is called the device controller or adapter. On mini and microcomputer, it often takes the form of a printed circuit card that can be inserted into the parent bard. The mechanical component is the device itself.

The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controller can handle two, four, or even eight identical device. If the interface between the controller and device is a standard interface, either an official standard such as ANSI, IEEE or ISO, or de facto one, then companies can make controllers or devices that fit that interface. Many companies, for example, make disk drives that match the IBM disk controllers interface.

We mention this distinction between controller and device because the operating systems nearly always deal with the controller, not the device. Nearly all microcomputer and minicomputers use the single bus model of Fig.4.1 for communication between the CPU and the controllers. Large mainframes often use a different model, with multiple buses and specialized I/O computers called I/O channels taking some of the load off the main CPU.

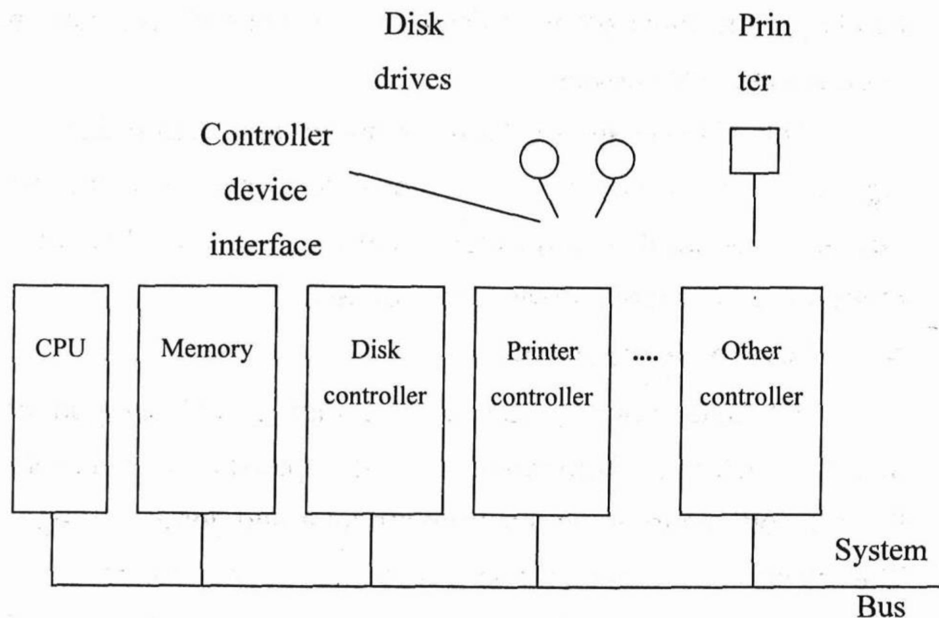


Fig. 4.1 A model for connecting the CPU, memory, controller, and I/O device.

The operating system performs I/O by writing commands into the controllers' registers. The IBM PC floppy disk controller, for example, accepts 15 different commands, such as READ, WRITE, SEEK, FORMAT, and RECALIBRATE. many of the commands have parameters, which are also loaded into the controller's registers. When a command has been accepted, the CPU can leave the controlled alone, and go off to do other work. When the command has been completed, the controller causes an interrupt in order to allow the operating system to again control of the CPU, and test the results of the operation. The CPU gets the results and the device status by reading one or more bytes of information form the controller's registers.

4.2.2 Direct Memory Access (DMA)

Many controllers, especially those for block devices, support direct memory access (DMA). To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the controller reads the block (one or more sectors) form the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it performs the checksum computation to verify that no read errors have occurred. (The checksum can be computed only after the entire block has been read). Then the controlled causes an interrupt. When the operating system starts processing the interrupt, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in memory.

Naturally, a programmed CPU loop to ear the bytes one at a time form the controller wastes CPU time. DMA was invented to free the CPU from this low-level work. When it is

used, the CPU gives the controller two items of information, in addition to the disk address of the block: the memory address where the block is to go, and the number of bytes to transfer, as shown in Fig. 4.2

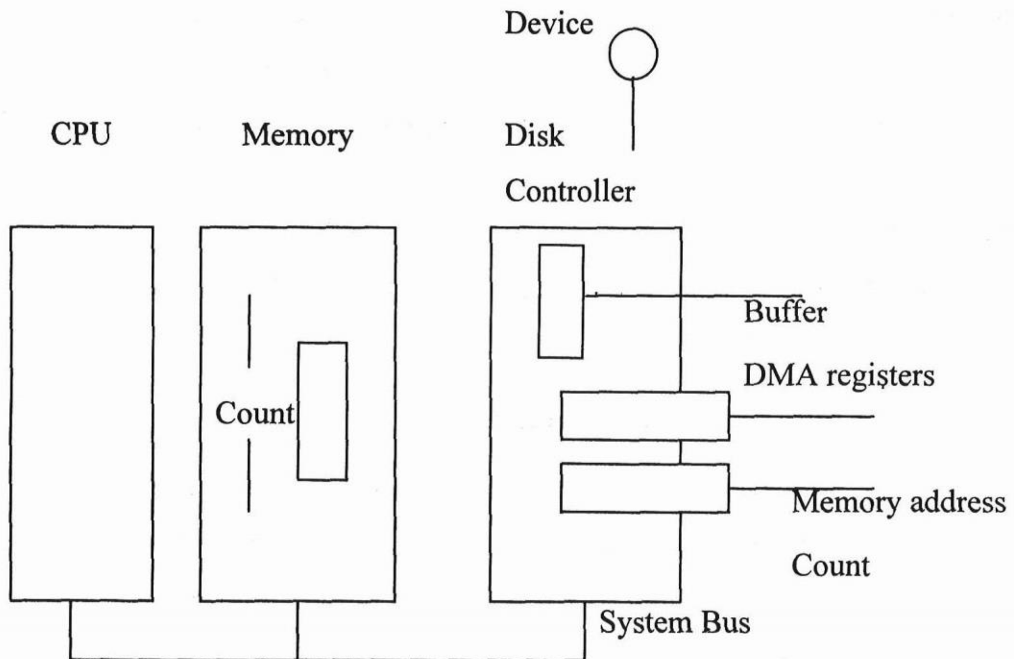


Fig. 4.2 A DMA transfer is done entirely by the controller

After the controller has read the entire block from the device into its buffer and verified the checksum, it copies the first byte or word into the main memory at the address specified by the DMA memory address. Then it increment the DMA address and decrement the DMA count by the number of bytes just transferred. This process is repeated until the DMA count becomes zero, at which time the controller causes an interrupt. When the operating system starts up, it does not have to copy the block to memory: it is already there.

4.2.3 Principles of I/ O software

Let's turn away from the hardware and now look at how the I/O software is structured. The general goals of the I/O software are easy to state. the basic idea is to organize the software as a series of layers, with the lower ones concerned with hiding the peculiarities of the hardware from the upper ones, and the upper ones concerned with presenting a nice, clean, regular interface to the users. In the following we will look at these goals.

A key concept in the design of I/O software is device independence. It should be possible to write programs that can be used with files on a floppy disk or a hard disk, without having to modify the programs for each device type. In fact, it should be possible to move the program without recompiling it.

Closely related to device independence is the goal of uniform naming. The name of a file or a device should simply be a string or a integer and not depend on the device in any way. In some operating systems, floppy disks, hard disks and all other block devices can be mounted in the file system hierarchy in arbitrary places, so the user need not be aware of which name corresponds to which device. All files and devices are addressed the same way: by a path name.

Another important issue for I/O software is error handling. In general, errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it can not, then the device driver should handle it, perhaps by just trying to read the block again. Many errors are transient, such as read errors caused by specks of dust on the read head, and will go away if the operation is

repeated. Only if the lower layers are not able to deal with the problem should the upper layers be told about it.

Still another key issue is synchronous (blocking) versus asynchronous (interrupt-driven) transfers. Most physical I/O is asynchronous the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking after a READ command the program is automatically suspended until the data are avail-able in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

The final concept that we will deal with here is sharable versus dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished. Having five users printing lines intermixed at random on the printer just would not work. Introducing dedicated devices also introduces a variety of problems, including deadlock. Again, the operating system must handle both shared and dedicated devices in a way that avoids problems.

These goals can be achieved in a comprehensible and efficient way by structuring the I/O software in four layers:

1. Interrupt handlers.
2. Device drivers.
3. Device- independent operating system software.
4. User level software.

decremented by 1 with every input count pulse. The count of a 4-bit down counter starts from binary 15 and continues to binary counts 14, 13, 12...0 and then back to 15. The circuit of figure 1.32 will function as a binary down counter if the outputs are taken from the complement terminals Q' of all flip-flops. A list of the count sequence of a count down binary counter shows that the lowest-order bit must be complemented with every count pulse. Any other bit in the sequence is complemented if its previous lower order bit goes from 0 to 1.

Synchronous Counters

Synchronous counters are distinguished from ripple counters in that clock pulses are applied to the CP inputs of all flip-flops. The common pulse triggers all the flip-flops simultaneously, rather than one at a time in succession as in a ripple counter. The decision whether a flip-flop is to be complemented or not is determined from the values of the J and K inputs at the time of the pulse. If J = K = 0, the flip-flop remains unchanged. If J = K = 1, the flip-flop complements.

Binary Counter

The design of synchronous binary counters is so simple. Ina synchronous binary counter, the flip-flop in the lowest order position is complemented with every pulse. A flip-flop in any other position is complemented with a pulse provided all the bits in the lower order positions are equal to 1, because the lower order bits will change to 0's on the next count pulse. Synchronous binary counters have a regular pattern and easily be constructed with complementing flip-flops and gates. With a regular pattern, the CP terminals of all flip-flops are connected to a common clock-pulse source. The first stage A_1 has its J and K equal to 1 if the counter is enabled. The other J and K inputs are

equal to 1 if all previous low order bits are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the J and K inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip-flop outputs are 1's.

Binary Up-Down counter

In a synchronous count-down binary counter, the lip-flop in the lowest order position is complemented with every pulse. A flip-flop in any other position is complemented with a pulse provided all the lower order bits are equal to 0. For example, if the present states of a 4-bit count down binary counter is $A_4A_3A_2A_1$ = 1100, the next count will be 1011. A_1 is always complemented. A_2 is complemented because the present state of A_1 = 0. A_3 is complemented because the present state of A_2A_1 = 00. But A_4 is not complemented because the present state of $A_3A_2A_1$ = 100, which is not an all 0's condition. The two operations can be combined in one circuit. A binary counter capable of counting either up or down also exists. The T flip-flops employed in this circuit may be considered as JK flip-flops with the J and K terminals tied together. When the up input control I 1, the circuit counts up, since the T inputs are determined from the previous values of the normal outputs in Q. When the down input control is 1, the circuit counts down, since the complement outputs Q' determine the states of the T inputs. When both the up and down signals are 0's the register does not change state but remains in the same count.

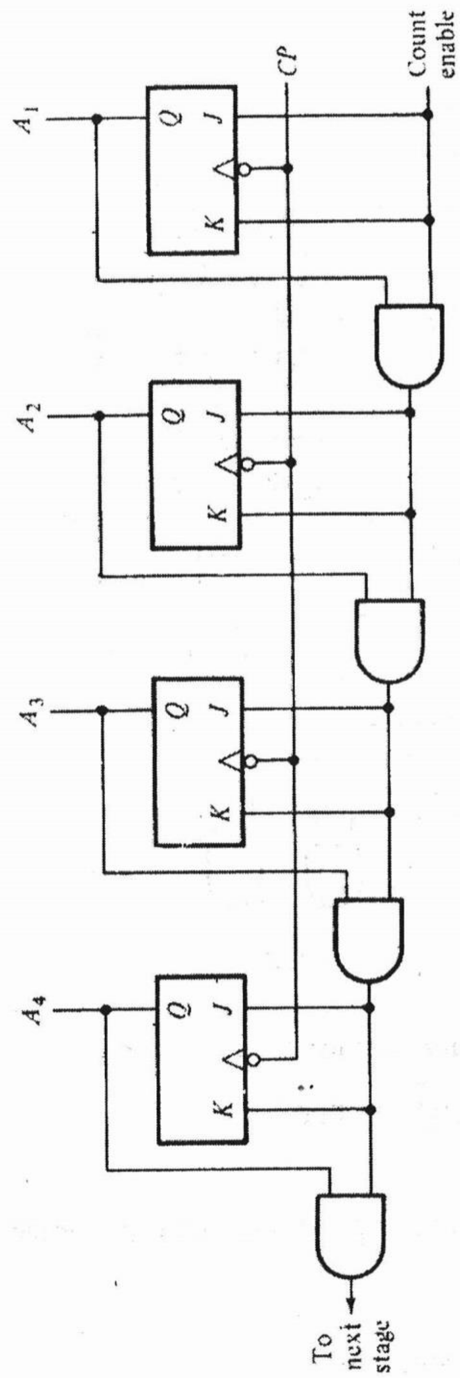


Fig1.33 4-bit synchronous binary counter

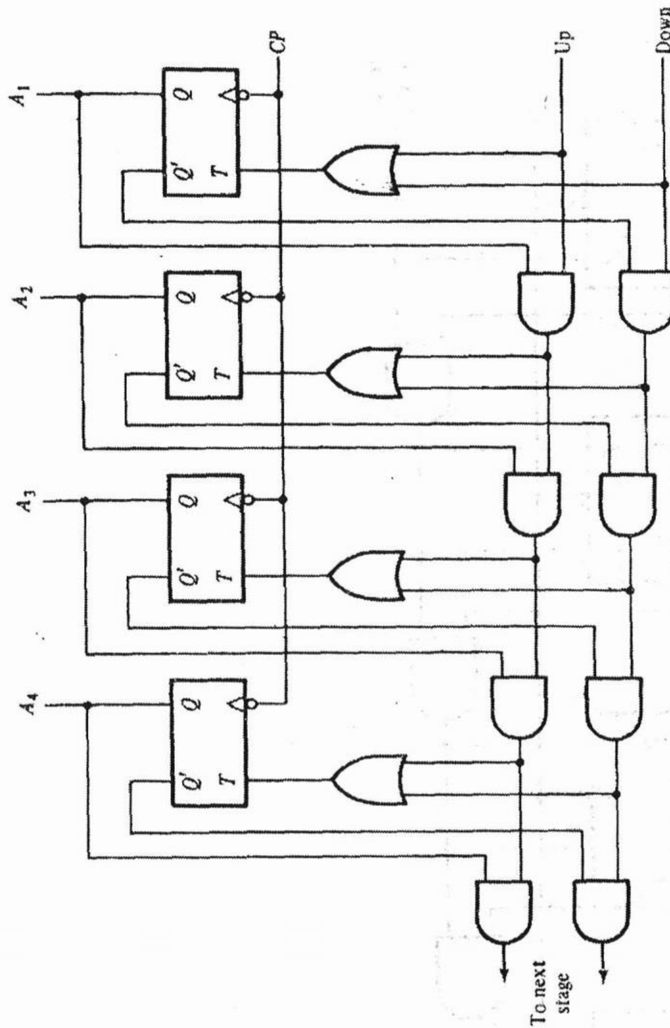


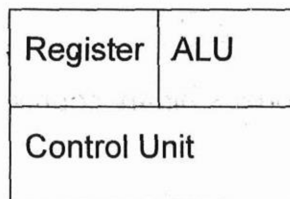
Figure 7-18 4-bit up-down binary counter

Fig1.34 4-bit up-down binary counter

3.8 Interconnection Structures

CENTRAL PROCESSOR

The block diagram of a typical processor is shown in the following figure 1.35;



CPU

Fig 1.35 block diagram of a processor

Here ALU called the arithmetic and logic unit performs all the processing operation like arithmetic and logic operations. The control unit is responsible for generating the control signals for the performance of operations of ALU and other input output devices, and for synchronization. The registers are the memory part of the CPU, which hold the data for the processing. Some registers are used for general operations and are called general-purpose registers. A collection of registers is called *memory unit.*

REGISTERS:

The CPU of a computer includes a set of high-speed registers. These registers may be classified as general purpose registers.

The general-purpose register, as the name implies is usually used for general purpose operations. These registers may hold data, temporary results or memory address when a computation is in progress. These versa or register to register, using a program. A program is a asset of instructions written in a machine understandable language and are stored in a collection of general purpose registers called memory. The processor reads the instructions and data, transfers them to other registers, store the temporary results and permanent results after the processing in these registers. During complex arithmetic operations like multiplication, division, etc., It is necessary to store intermediate results temporally. For this purpose there are usually in or more *scratchpad registers or a scratch pad memory.* These are purely internal; hardware resources and selective registers are addressable by program.

Apart from these general-purpose registers, there are some registers called special purpose registers. These performs only the special tasks assigned to them and are incapable of dong any other operation.

Some of the special purpose registers and their purpose mentioned below.

Register Name	Purpose
Program Counter (PC)	Holds address of the next instruction to be executed.
Instruction Register (IR)	Holds the instruction currently being executed
Effective Address Register (EAR)	Holds the address of the data to be retrieved from the memory

These are also called dedicated registers because they are available for the exclusive use of the control unit and cannot be accessed by a user program.

MEMORY UNIT

The major and probably the most important advantage of digital systems, over their analog counterpart is that large quantities of data and information can be stored for short or large periods. This improves the versatility and adaptability of the digital systems.

The basic memory device of digital system is flip flops. Already the shift registers are the high speed memory elements and used extensively in the internal operations of a digital computer have been discussed. Let us have a detailed discussion on some more memory devices.

MEMORY TERMINOLOGY

Before knowing the different types of memories, it is essential to be familiar with the different terms associated with memories.

Memory Cell: A device or a circuit, which can store a single bit a FF or magnetic spot.

Memory Word: A group of bits, typically 8, 16, or 32 bits depending on the capacity of the memory registers.

Byte: A group of 8 bits.

Capacity: Number of bits that can be stored in the memory. This is represented as multiple of 2. Usually a memory capacity is represented as the number of bytes. Typically a 1K X 8 memory contains 1024 words of 8 bits or 1024 X 8 = 8,192 bits.

Density: Another term for capacity. Dense memories can store more bits in the same amount of space.

Address: An identification, which shows the location of a word in memory. Each byte or word stored will have a unique address used to refer to it.

Access Time: The time taken to read the data from memory. This is the time difference between the address being given to the memory and the data being available at the output of memory.

Access Time: Speak Time + Transfer Time

Seek Time: Time required to position the read - write head to a location. This is a unique factor of electromechanical memory devices.

Transfer Time: Time required to transfer the data to or from the devices. Transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

GENERAL MEMORY OPERATION:

Though the internal operation of the memory differs for the different types, the basic operating principles are the same for all memories. The input and output lines and the control lines are the same. Though the name may differ, they perform the same operation.

The operation performed on a memory can be listed as follows:

1. Selection of address being accessed.
2. Loading the data into the memory through a buffer register, for write operation.
3. Holding the output data in the buffer register from memory, for read operation.
4. Selection of the memory chip by enabling the enable input.
5. Selection of any one of the read o write operations.

The general block diagram of memory with the above control and data inputs and data outputs is given in the following figure 1.36.

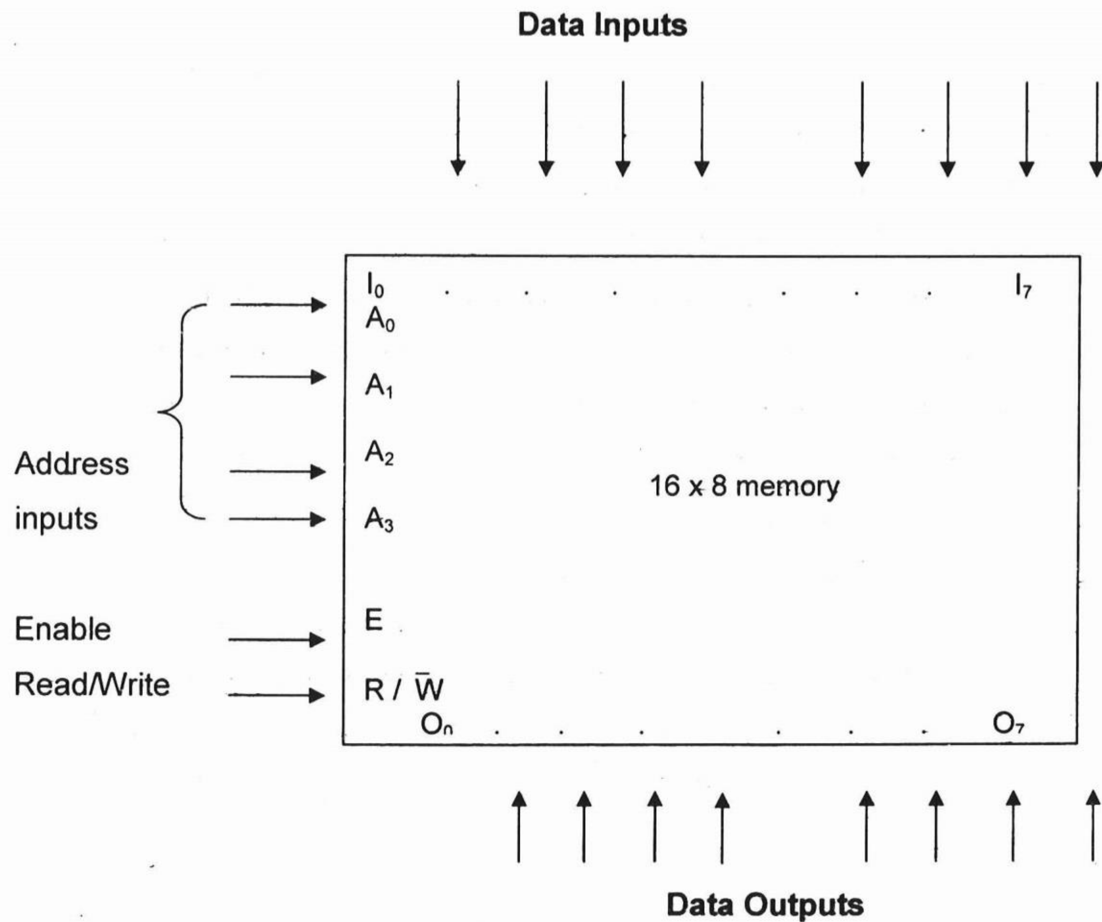


Fig1.36: (a) Block diagram of a 16 x 8 memory

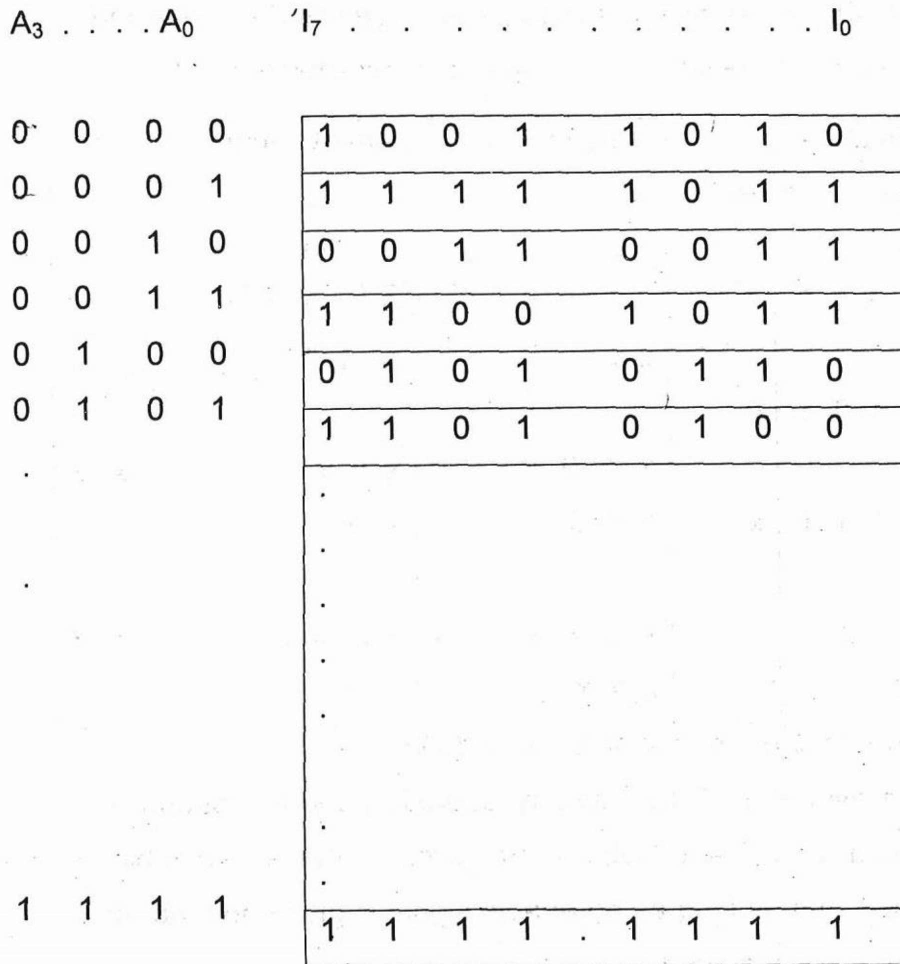


Fig1.36 (b) Virtual arrangement of memory registers.

The explanation for the different input, output lines and the control can be made simultaneously by explaining the operation of the memory.

The memory chip is selected by a "1" to enable. The memory capacity is 16 X 8 bits. Since the total number of memory locations are 16, the address bits needed are $2^n = 16$ i.e., n = 4. So the address commences from 0000 and extends up to 1111. Just like a PLA, decoding them enables the address lines and the read/write line gives the necessary enable signal. Since there is a bar over write, the same line is enabled for if the signal given is "*high*" and for write if the signal given is "*low*".

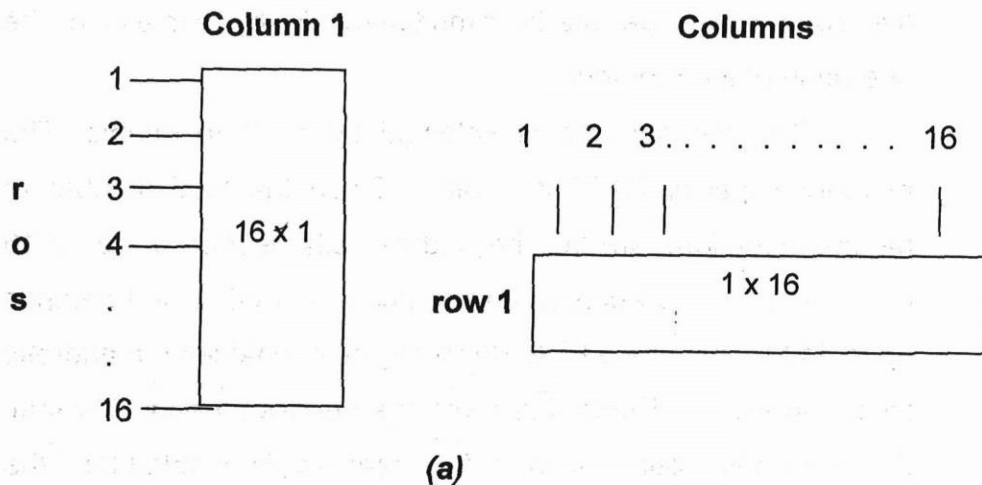
The *data inputs* are given, if the operation selected is *write* and *data outputs* can be taken, if the operation selected is *read.*

E	Address A_3 A_0	R/ $\bar{W}$	Data Inputs I_7 . . . . I_0	Data Outputs O_7 . . . . O_0
1	0 1 0 0	1	x x x x x x x x	0 1 0 1 0 1 1 0
1	0 1 0 1	0	1 1 0 1 0 1 0 0	x x x x x x x x
0	1 1 1 1	x	x x x x x x x x	x x x x x x x x

Table: Status of memory inputs and outputs.

An overview of the memory operation and the status of the different inputs and outputs are given in the above table. This shows that as long as no enable signal is given no input or output lines are activated.

This is a rectangular configuration. There are three different configurations possible as shown in the figure.



(a)

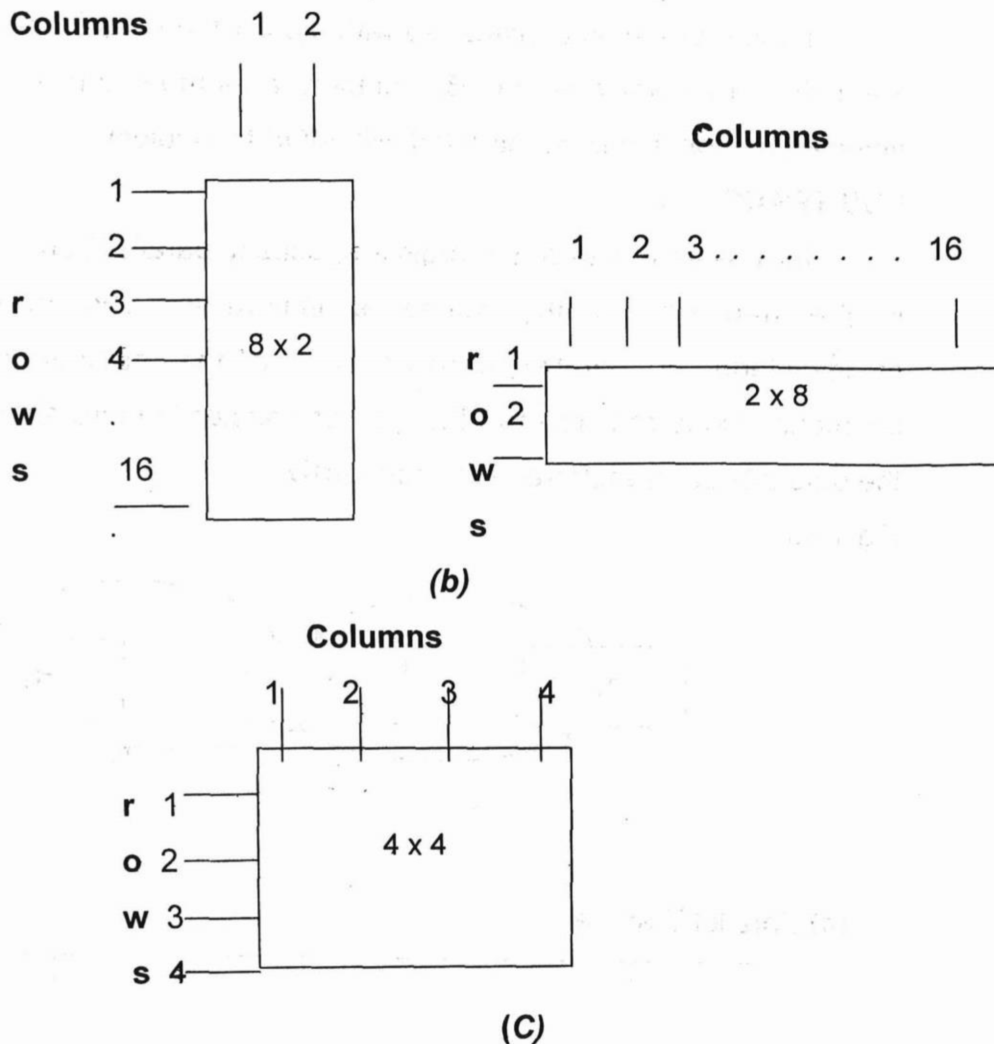


Fig1.37 Block diagram of a Central Processing Unit

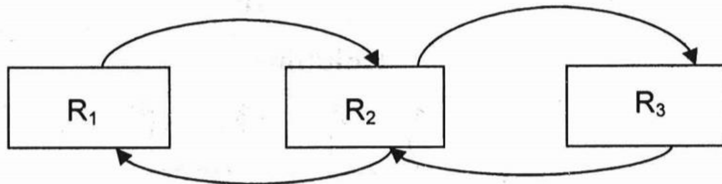
Here the capacity of all memories shown is the same though the number of rows and columns differ. Out of the three configurations (a) require 16 + 1 address lines, 16 for the rows and 1 for the columns, (b) requires 8 + 2 address lines and (c) require 4 + 4 address lines. The configuration in (c) requires the *lowest number of address line.* It is for this reason, that the square configuration is so widely used in industry. This arrangement of n rows and n columns is often referred to as *matrix addressing.* The arrangement in figure where a selection of a cell means selection of row is called *linear addressing.*

if the memory is constructed with bipolar transistors, then it is called a *bipolar memory*. By contrast, a memory can be a *unipolar memory* if it is constructed with MOS technology.

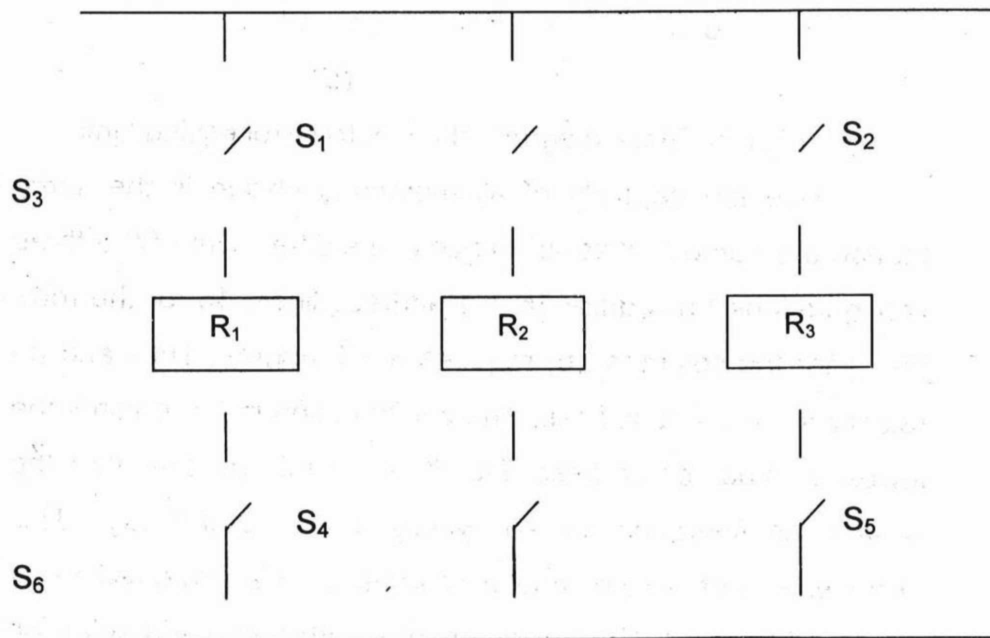
BUS TRANSFER:

In a system with many registers, usually parallel transfer is preferred only if the speed is imperative. The main disadvantage of parallel transfer is that the number of interconnections required for this type of transfer is more since the data bits are transferred simultaneously

Fig 1.38



(a) Parallel Transfer



(b). Transfer through a common line

Also as the number of registers become more and more, the interconnections also increase. Figure1.38 shows the connections between 3 registers. For this 6 set of lines are required for interconnections.

Instead if common lines can be used for transfer among registers and the registers can be chosen using switches, this provides a compromise between the amount of hardware required and the speed. This is called *bus transfer.*

A group of wires through which binary information is transferred among registers is called a bus. For a parallel transfer, the number of wires in the bus is equal to the number of flip-flops in the register. But here the numbers of lines are reduced using multiplexers.

The following figure(a) shows a bus system with 4 registers. There are 4 multiplexers whose outputs are connected together to form the data output. Each one of the multiplexers has at least one input from each of the 4 registers. The multiplexer input can be selected using X, Y selection lines.

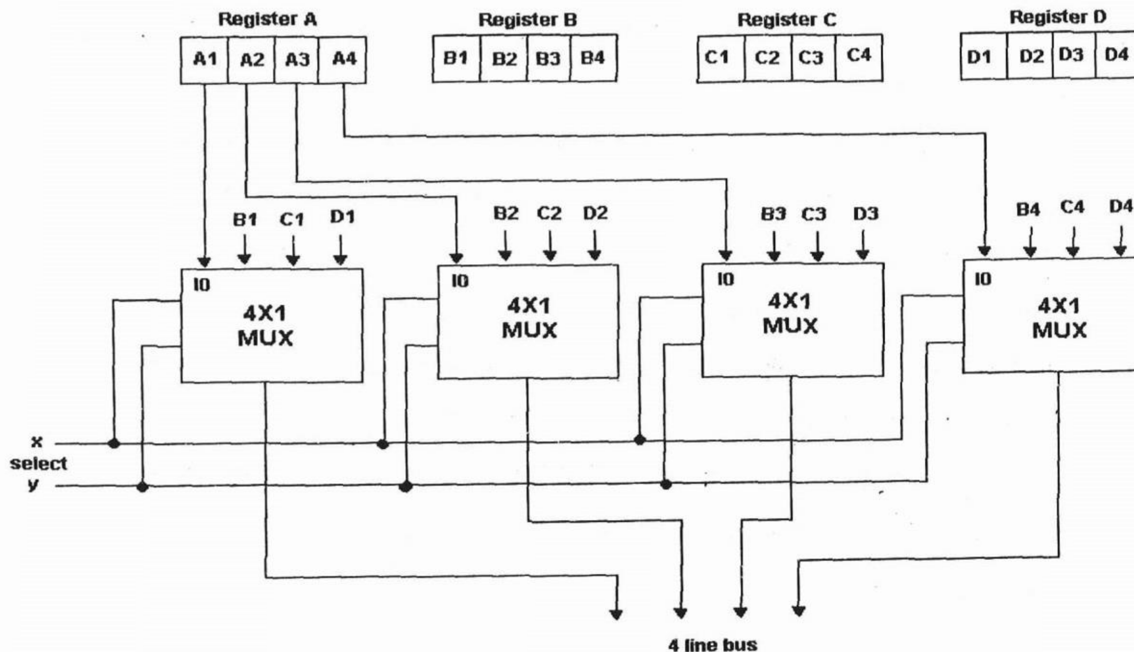


Fig1.39 (a): Transfer of data from register to bus

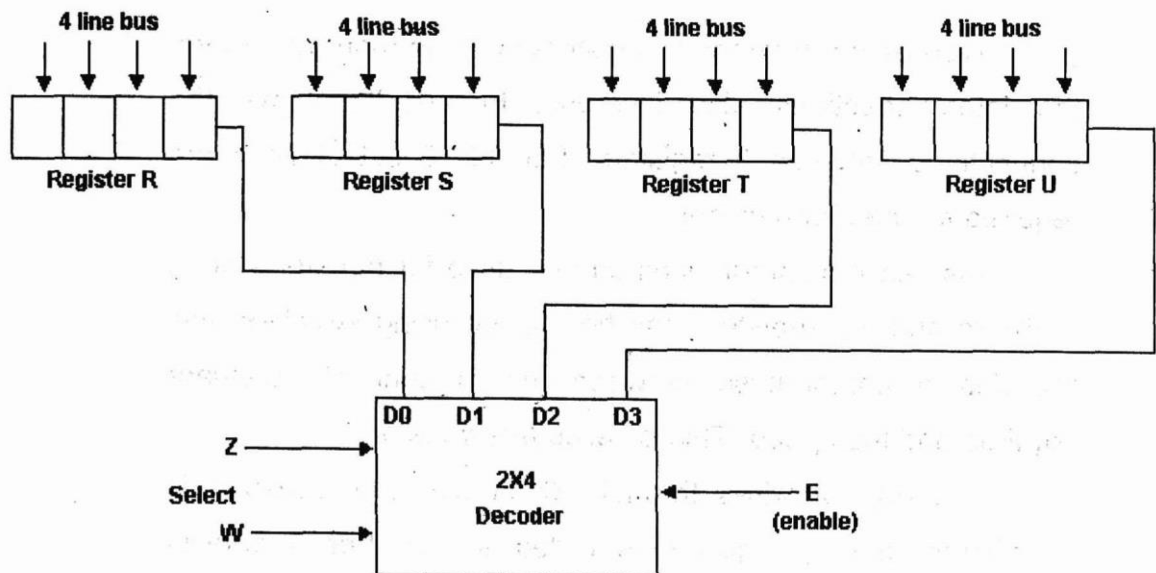


Fig1.39 (b): Transfer of data from bus to register

When X, Y lines are 0,0 the first inputs of all multiplexers are selected. Since the first I_0 are connected to the outputs of Register A, the data in register A appears at the multiplexer output lines, similarly the other register contents can be brought to the outputs by using selection lines accordingly.

After bringing the data to the bus the data has to be moved into the destination register. This is also a similar operation, where any one of the registers can be chosen using a decoder as shown in the figure.

Usually a buffer register is preferred in between the two operations shown by figure1.39 (a) and (b). The data fro source register is first transferred to the buffer register during one clock pulse and this is again transferred to the destination register during the 2nd clock pulse. A quicker way will be to use the two edges of the clock pulses to synchronize them.

Transfer through a bus is limited to one transmission at a time. If two transfers are required at the same time, two buses must be used.

The micro operations can be given as follows;

$$xy' : Bus \leftarrow C$$

$$z'wE : R_1 \leftarrow Bus$$

Also since buses are known to exist in the system, it may also be given as,

$$xy'z'wE : R_1 \leftarrow C$$

In a typical computer there are three types of buses – Address Bus, Data Bus and Control Bus. The address bus is used to access the memory or peripheral devices using the address placed on it. The data bus transfers the data through and outside the system. The control bus is for carrying the control signals for various parts of the system.

This is illustrated in the following figure. The connections of a typical CPU – memory show how the different buses are connected.

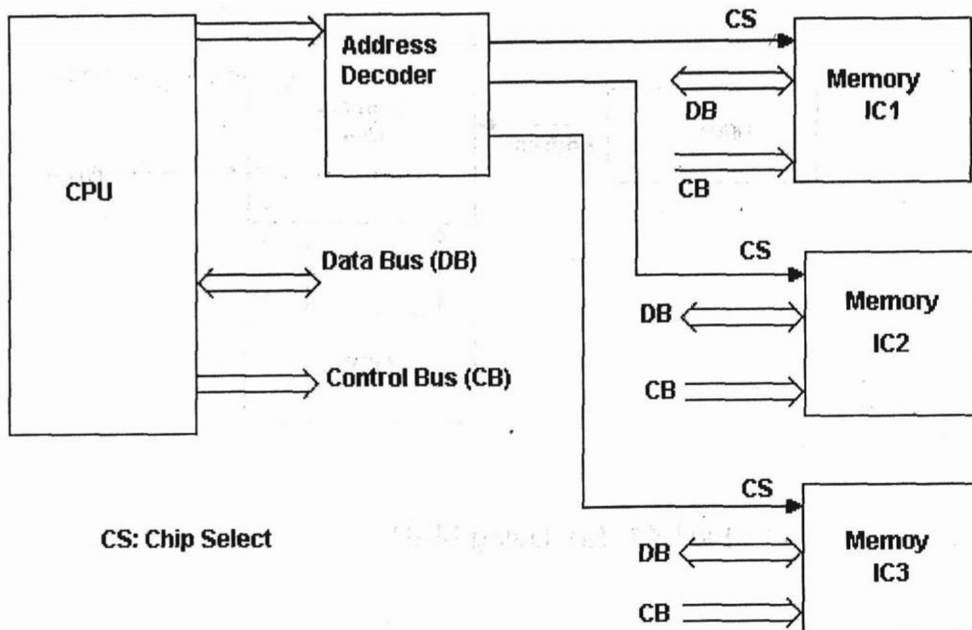


Fig1.40: CPU - Memory Connections through bus

MEMORY TRANSFER:

The operation of a CPU – memory data transfer can be explained as two different operations – read operation and writes operation. The transfer of a data from memory to the external environment is called a read operation. The transfer of a new data into the memory is called the write operation. In both operations, the particular memory word selected must be specified by an address.

The letter M will symbolize a memory register or word. When a word is to be selected the selection is done, by specifying the address through a special purpose register called *Memory Address Register (MAR).* Similarly another register is used for holding the data before being transferred into and out of memory. This is called *Memory Buffer Register (MBR).*

The following figure shows the connection of address to a memory unit using a specialized MAP and using a multiplexer.

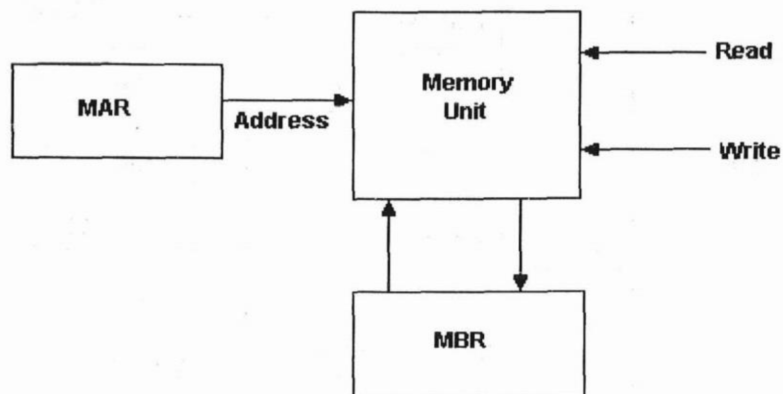


Fig1.41: (a). Using MAR

These approaches would argue that perhaps swapping does too much. Both segmentation and paging move part of programs back and forth between secondary storage and main memory as needed. Segmentation and paging differ from one another primarily in the way the code for a particular process is divided. In segmentation, a program code is divided into number OS variable sized blocks corresponding to the logical structure of the program, such as procedures, functions and data segments. Paging, on the other hand, divides the program code into fixed blocks, called pages. It is evident that the more logical subdivision of segmentation makes program linking easier, while the fixed blocks of paging, being each interchangeable with the other, makes memory management easier. In either case, since portions of program's code are being moved around during a program's execution, something like a hardware relocation register will be needed to compute actual addresses in order to avoid unacceptable slowdown in program execution times.

LEARNING ACTIVITIES

Fill in the Blanks:

1. .... is one of the most visible services of an operating system.
2. .... access is based upon a tape model of a file.
3. ....mechanisms provides controlled access by limiting the types of file access which can be made.
4. Linux uses a .... paging technique to fairly choose pages which might be removed from the system.

LET US SUM UP

At the end of this unit you have understood the concept of File System. Files are managed by the Operating System. How they are structured, named, accessed, used, protected and implemented are major topics in Operating System Design. As a whole, that part of the operating system dealing with files is known as the file system.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. File management
2. Sequential
3. Protection
4. Least Recently Used (LRU)

MODEL QUESTIONS

1. Write short notes on Swapping.
2. Write short notes on Paging.

REFERENCES

T.W. Pratt – Programming Languages, Design and Implementation – PHI

R.G. Dromey – How to solve it by Computer – PHI

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

BLOCK 2 INTRODUCTION

At the end of this block you will know the Unix Operating System. Unix Operating System prospered at Bell Labs, finding their way into laboratories, software development projects, word processing centers and operations support systems in telephone companies. Since, then, it has spread world-wide, with tens of thousands of systems installed, from microcomputers to the largest mainframes. Unix systems run on a range of computers from microprocessors to the largest mainframes; this is a strong commercial advantage. Second, the source code is available and written in a high-level language, which makes the system easy to adapt to particular requirements. Finally, and most important, it is a good operating system, especially for programmers. The Unix programming environment is unusually rich and productive. Even though the Unix system introduces a innovative programs and techniques, no single program or idea makes it work well.

Introduction to System Software is divided into Four Blocks. Block 2 consists of four Units.

Unit 5 : is a discussion of the Unix Operating System, Unix File System. The file system is central to the operation and use of the system. The command interpreter, or shell is a fundamental tool, not only for running programs, but also for writing them.

Unit 6: deals with a text editor VI is used to create and manage text files and documents.

Unit 7: talks about writing new programs using the standard I/O library. The programs are written in C, which the reader is assumed to know, or at least be learning concurrently.

Unit 8: is about Pipes and Filters. Programs that perform some simple transformation on data as it flows through them.

UNIT - 5

UNIX OPERATING SYSTEM

Structure

Overview

Learning Objectives

5.1 Introduction to Operating System

5.1.1 Different types of Operating System

5.1.2 Multi Processor Operating System

5.2 Foundations of Unix Operating System

5.2.1 Evolution of Unix

5.2.2 Versions of Unix

5.3 Structure of Unix

5.3.1 The Kernel

5.3.2 *Shell*

5.3.3 File system

5.4 Command Format

5.5 Communication between Users

5.6 Text Manipulation Commands

Let us sum up

Answer to Learning Activities

References

OVERVIEW

An Operating System is a program that acts as an interface between the user and the computer. Operating Systems have earned the reputation for being the most critical software in a computer system. Operating system primarily provides a convenient interface to its users and at the same

time manages the computer's resources processor, memory, and I/O devices. In a nutshell operating system can be defined as a resource manager.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Understand the Operating System
- Familiar with Structure of Unix
- Know the Command Format
- Understand the Communication between Users

5.1 INTRODUCTION TO OPERATING SYSTEM

A general organization of an operating system is shown below:

Without an operating system, the most powerful computer in the world would be useless. No matter how powerful and elegant your programs are, they can't function without the assistance of an operating system.

5.1.1 Different types of Operating System

The operating system is categorized into the following types.

Single User System

This type of operating system is popularly known as personal computer operating system. Their job is to provide a good interface to a single user. The two popular operating systems under this category are DOS and Windows.

DOS is an example for a single user operating system that is single user with no multitasking. Multitasking means running more than one program concurrently. Windows is an example for a single user operating system with multitasking

capability. There are various versions of Windows operating systems like, Windows 95, Windows 98 and Windows XP.

These operating systems are widely used for small applications like word processing, Internet access, managing small databases etc.

Multi User System

Operating systems, which can serve for more than one user at a time, are known as multi user operating systems. Each user can run his own program. The operating system allots a quantum of time for each user for processing his tasks. The most popular multi user operating system is UNIX Operating system.

5.1.2 Multi Processor Operating System

Some computer systems involve more than one CPU. Depending on precisely how these CPU's are connected and what is shared, these computers are called parallel computers or multiprocessor computers. They need a special type of operating system to manage the additional resources.

5.2 FOUNDATIONS OF UNIX OPERATING SYSTEM

UNIX is a time-sharing operating system: a program that controls the resources of a computer and allocates among users in addition to controlling the peripheral devices and managing a file system.

5.2.1 Evolution of Unix

UNIX was developed in 1969 at AT&T Bell laboratories. It was developed by designers, who were involved in the

development of less popular MULTICS operating system. UNIX is the brainchild of two persons Ken Thompson and Dennis Ritchie. Their first venture was a modest multitasking system to support two users. This operating system supported an efficient file system, a command interpreter and a set of utilities.

Earlier versions of operating system did not support machine compatibility. But UNIX changed the operating system world scenario entirely by breaking through this seemingly difficult demerit by running on different systems.

UNIX was not written in assembly language as most operating systems were. It was written in C to aid the machine compatibility feature by making it compatible to different hardware platforms.

5.2.2 Versions of Unix

The AT&T Bell laboratory where the UNIX was primarily developed was not able to commercialize its products due to a judgement imposing a ban passed by the government. This judgement forced the AT&T Bell laboratories to sell its product to various academic institutions. Later business establishments joined in the development of the UNIX operating systems. As the result of all these efforts different versions have emerged. These versions have been discussed briefly in the forth-coming sections.

Berkeley Unix

Of all the versions of Unix this deserves a special mention due to its larger contribution to the development of the operating system. Most of the new features in the Unix operating system were developed at the university of California, Berkeley.

We could almost say they created a Unix version of their own. This version was named as BSD UNIX where BSD is the acronym for Berkeley software distribution. Berkeley UNIX had a more efficient file system than the AT&T original version and is also equipped with better linking facilities.

Other versions

AT&T Bell laboratories embarked on commercialization of its Unix after the government's ban on it was removed. Their earlier versions previously know as "editions" were then changed to "systems". The first to come in the "system" series was the System3, which later became system V Release 3.0. Release 3.2 followed this version.

There are many other versions of UNIX. These include the Microsoft and Sun Microsystems versions. Microsoft developed its own version of UNIX and named it 'ZENIX' which was later sold off to Santa Cruz operation(SCO). The sun Microsystems version of Unix is called as 'SOLARIS'

LINUX

Linux was the first non-commercial version of UNIX. LINUX was developed by Linus Torwalds as his final year project while doing under graduation at Helsinki University in Finland. The licensing made the source code public. Linux is strong in networking and Internet features. Linux can run on all PENTIUM PC's apart from Apple's Power PC and Sun's Sparc Computers.

5.3 STRUCTURE OF UNIX

The following gives the detail high-level architecture of the Unix operating system.

5.3.1 The Kernel

The heart of a UNIX operating system is a collection of programs termed as Kernel. The kernel controls the computer's resources. When we log in, it is the kernel that checks if we are an authorized user and has the correct password. The kernel keeps track of all the various programs being run, allotting time to each, deciding when one stops and another starts. The kernel assigns storage for yours files. The kernel runs the shell programs. The kernel handles the transfer of information between the computers and terminals, type drivers and printers. In other words, the kernel is what we call an operating system. It's the heart of the UNIX system, which is why it is called the kernel. The functionalities of the kernel can be summarized as follows:

- ◊ Memory management
- ◊ File System management
- ◊ I/O management
- ◊ Process scheduling
- ◊ Process dispatching

5.3.2 Shell

Another important part of the UNIX operating system is the shell. It is an interface between the user and the kernel. It has a program capability of its own. The main merit of the shell

programming is hiding the implementation of the kernel functioning from the user thus reducing much overhead of the user.

5.3.3 File system

UNIX operating system manages its documents through a powerful system called the File system. Unix File System is a hierarchical or tree structured.

Different types of files

UNIX File System has the following files

1) Ordinary Files (text or binary)
2) Directory Files
3) Special Files
4) Standard Files

UNIX Operating System treats directories and devices as files.

Directories

Directories are collection of files or other directories.

Special Files

Devices like printers or terminals are treated as special files.

Standard Files

Standard Files are used to display information on the standard input / output devices.

5.4 COMMAND FORMAT

Basic commands in Unix

UNIX is the most secured operating system. A user can enter into UNIX operating system, if he maintains an account. When we enter the UNIX operating system we are prompted for login name and password. Creating a new account is possible only by System Administrator.

Login Command

Login is a process by which a user identifies himself to the system. This command requires the person to type in the username of the account maintained for him in the operating system. After entering the username the user is asked for the password.

Password is a secret code known only to the user. This code should be type in. The typed characters are not echoed in the monitor screen as a safety measure.

NOTE: Even the administrator does not know the password.

Example

$ LOGIN: you	# type you name then press RETURN
password: xxxxx	# password typed will not be # echoed as you type it
You have mail	# there is an unread mail to be # read after you log in.
$	# the system is ready for # accepting commands from the # user. This is a dollar prompt.

Date & Time Command

The date command is used for displaying both the system date & time.

Example

$ date

Wed Sep 11 23:02:36 IST 1998

There are several optional formats for the date command.

a) date +%m

This command displays only the month in number

Example

$ date +%m

05

b) date +%h

This date command displays the name of the current month of the year.

Example

$ date +%h

May

c) date"+%h%m"

This command combines the functions of the above two commands i.e. it displays both the name and numeric representation of the month.

Example

$ date +"%h%m"

May 05

d) other format specifiers

d	day of the month(1 to 31)
y	last two digits of the year
H, M and S	hour, minute and second

Echo Command

The main function of this command is to display its arguments on the screen. It is an internal command

Example

$ echo Hello   World

Hello World

(Spaces are not considered here as characters but ignored)

ECHO may also be used to evaluate the value of the variable.

Example

$ x=23   #value assigned to variable 'x'

$

$ echo $x

23

Man Command

This command is used to obtain online help on any command.

Example

$ man wc

The following is the help text available for wc command.

wc (C)

wc - count words, lines, and characters or bytes

Syntax

wc [-lw][-c][-m][file....]

Description

The wc command counts newline characters, words and characters in the named files. It reads from the standard input if no files are named. wc also keeps a total count for all named files. A word is a maximal string of characters delimited by white space as defined by the current locale.

The option -l, -w, and -c or -m may be used in any combination to specify that a subset of newline characters, words, and bytes or characters (respectively) are to be reported. The default options are -lwm.

The order and number of output columns are not affected by the order and number of options. There is always at most one column in the following order: number of newline characters, words, bytes, and file name. The filename is not present if one or no filename is given on the command line. If more than one filename is given on the command line, the final line contains that total number of newline characters, words, and bytes in all files, and is labeled with the word total in the filename column.

Limitations

The -c option formerly stood for character count. This can be misleading as it actually counts bytes; this may not be the same as the number of characters for some locales. Use the -m option to count characters.

Standard conformance

wc is conformant with:

ISO/IEC DIS 9945-2:1992, Information technology - portable Operating System

Interface (POSIX) - part 2: Shell and Utilities (IEEE Std 1003.2-1992);

AT&T SVID Issue 2

X/Open CAE Specification, Commands and Utilities, Issue 4, 1992.

1_May_1995

man page for wc (SCO_UNIX)

(courtesy SCO-Open Server)

Options of man command

a) -e option

$ man -e grep

grep, egrep, fgrep (C) - search files for a pattern.

Here -e gives an one-line introduction to the command and displays the other related commands equivalent to that of grep command.

b) -k option

if we do not know the exact command but we know some key words associated with the command, then we can use -k option to find list of commands that uses the key word.

Example

$ man -k inode

clri (ADM)	-clear inode
inode (FP)	-format of an inode

ncheck (ADM) -generate names from inode numbers

For the key word inode there are three commands.

Who Command

The 'who' command is used to know the detail of all the users working on various terminals.

Example

```
$ who
root    console  sep11   10:32
ray     tty05    sep11   14:09
ram     tty04    sep11   13:17
```

Who am i Command

This command gives the details about us (who is currently logged on). This command is derived from who command. Instead of printing information about all the users, it displays the information about the particular user alone. (It is the arguments to the command, which decides something different from that of the regular process.)

Example

```
$ who am i
ray tty05   sep11   14:09
```

cal command

CAL is a command to view the calendar of any specific month of the year or a complete year. Calendar can be printed from 1 to 9999 year.

Example

To display the calendar for the entire year 2002 the command should be

$ cal 2002

All the month of the year 2002 are displayed in a formatted manner.

To print the calendar for a particular month the following format has to be used.

$ cal month year

Example

$ cal 1 2003

S	M	Tu	W	Th	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Banner Command

This command is used for creating fancy objects or posters on the screen. It displays the arguments we type in large size and in multiple lines.

Example

$ banner Hello

Hello

tput clear Command

This command is used in UNIX to clear the screen. The first command is

Clear

This is similar to CLS in DOS

```
$ tput clear
```

clear is given as a argument to the tput command.

Example

```
$ tput clear
```

The screen is cleared and the cursor is placed at the top left corner.

ls (a simple file command)

When we log in to UNIX system, we are automatically placed inside a directory called as HOME directory. This directory is created at the time of creating an account for us on the system. To see the content of the directory, a simple file command ls is used.

Example

```
$ ls
name.txt
prime.c
perfect.cpp
fact.c
```

Wildcard patterns

In general, file names differ only by a few characters in the prefix or suffix. These files are accessed and actions can be performed on them collectively. Special characters are used to represent character patterns, such as '?' , '*' and [..].

'*' is known as the meta character it represents any number of characters

Example

$ ls shu* (prefix usage)

Output

shunt

shum.c

shun.c

Example.2

$ ls *ing (suffix usage)

Output

reading

seeing

skewing

another meta character is -?. The ? meta character differs from * due to the fact that it represents only one character.

Example

$ ls sha?.?

Output

Shal.c

Shat.c

"[ ]" is used to access a subset of related files

Example

bas[1-4]

This command displays the file names bas1, bas2, bas3 and bas4 i.e., '-' gives the range of characters within the brackets

Creating a file command

To create a new file on the system cat command is used.

Syntax

```
$ cat > filename
```

Example

```
$ cat > names
balaji prabu
srinivasan
sundararaman
sriram
CTRL + d #CTRL and d key must be pressed simultaneously
$
```

To check the availability of the above file, we can use ls command.

Example

```
$ ls
names
$
```

Command to display the content of the file

The same cat command which is used for creating a file is also used to display the content of the file.

Syntax

```
$ cat filename
```

Example

```
$ cat names
balaji prabu
srinivasan
sundararaman
sriram
$
```

grep command

grep command allows the user to search for the particular pattern in a single file or in a group of files.

Syntax

```
$ grep pattern files
```

Example

```
$ grep balaji names
balaji prabu
```

$

If the specified pattern does not exist in the file then grep command displays a new prompt.

Example

$ grep shaiksulaiman names

$

5.5 COMMUNICATION BETWEEN USERS

Write Command (two way communication)

write command allows us to have two way communication with any persons who is currently logged in. We can write a message to a user and wait for the reply from the user.

This process can be continued until both of the user decides to terminate it.

Syntax

$ write srinivasan

Hello srinivasan how are you, did u attend UNIX practical or not- balajiprabu

CTRL + d

$

If the user by name srinivasan has logged on, the following message will be displayed on his terminal.

Message from balajiprabu tty15....... along with beep sound.

If the user is not logged on, then the following message will be displayed on our terminal.

write: srinivasan is not logged in

On the other side, user srinivasan can reply to the message sent by balajiprabu, by using the same write command.

$ write balajiprabu

hi i am fine, yes i have attended the UNIX Pratical. Bye - srinivasan

CTRL + d

$

Mail Command

'Mail' is the most well known command in UNIX. It is similar to electronic mail.

Unlike write command, mail command can be used to send messages, even when the user is not logged on.

Example

$ mail balaji

subject: About Industrial Visit

We are going for a industrial visit to banglore on 25th jan 2003.We are planning to visit the following places,

HCL Infotech

Satyam info way

Infosys

Mysore

If you are interested please enroll your name to our class incharge on or before 15th jan 2003. Bye Shaiksulaiman

<ctrl+d> #toend the mail

eot #system indication for end of text

Here the UNIX operating system requires the user, to type the subject before the actual mail text. The above mail sent from the user shaiksulaiman to balaji will not disturb the user balaji if he is busy in some program. Once he completes the program, the following message will be displayed on the screen.

you have new mail

Logout command

This command is used to sign out or terminate the currently running session and make it available to next user.

Example

$ logout

Login: # the terminal is available for a next user

5.6 TEXT MANIPULATION COMMANDS

In all the above discussions, we have not considered any mistakes done by the user. The user can commit mistakes while entering the command or sometimes a wrong entry of commands may result in to abnormal output for the command. In order to handle the error situations, the list of keyboard commands are widely used.

The <enter> key is used to complete the command line. If this key doesn't work, we can use either <ctrl-j> or <ctrl-m>.

Backspace key is used to move the cursor left and remove all characters it encounters on the way. If the backspace key doesn't work, use <ctrl-h>. If the line contains so many mistakes beyond correction we can kill the line altogether without executing it by <Ctrl-u>. To terminate the command prematurely, press the <delete> key (the default on

SCO UNIX). On other machines, including Linux, you may need to press <ctrl-c>

If the display from a command is scrolling too fast we can halt the output temporarily by pressing <ctrl-s>. To resume scrolling, press<ctrl-q>.

Summary of keyboard commands

Keystroke	Function
<ctrl-s>	Stops scrolling of screen output
<ctrl-q>	Resumes scrolling of screen output
<delete>	Interrupts a command (Use <ctrl-c>if this fails)
<ctrl-d>	Terminates login session (use exit also)
<ctrl-h>	Erases text (if backspace key doesn't work)
<ctrl-u>	Kills command line without executing it
<ctrl-\>	Kills command line without executing it
<ctrl-j>	Alternative to <enter>
<ctrl-m>	Alternative to <enter>

LEARNING ACTIVITIES

Fill in the Blanks:

1. .... command allows us to have two way communication with any persons who is currently logged in.
2. The ....command is used to know the detail of all the users working on various terminals.
3. .... command allows the user to search for the particular pattern in a single file or in a group of files.

LET US SUM UP

At the end of this unit you have understood the concept of Unix Operating System. It is a time-sharing Operating System kernel: a program that controls the resources of a computer and allocates them among its users. Unix is often taken to include not only the Kernel, but also essential programs like compilers, editors, command languages, programs for copying and printing files and so on.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks :

1. write
2. 'who'
3. grep

MODEL QUESTIONS

1. Why is UNIX more popular than other operating systems?
2. Which flavours of UNIX run on the Pc?
3. Who owns the UNIX trademark today?
4. When you enter a command, whose clearance has to be obtained before it can executed?
5. What does multi-taking mean?

REFERENCES

R.G. Dromey – How to solve it by Computer – PHI

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

UNIT - 6

VI EDITOR

Structure

Overview

Learning Objectives

6.1 Introduction to VI Editor

  6.1.1 Starting with VI

  6.1.2 Modes in VI Editor

6.2 Some more Commands

  6.2.1 Scrolling commands (screen commands)

  6.2.2 The Undo Commands

6.3 Replace Commands

6.4 Control Mode

6.5 Summary of VI Command

Let us sum up

Answer to Learning Activities

References

OVERVIEW

A text editor is used to create and manage text files and documents. An editor is application software that is usually bundled with an operating system.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- ❖ Know the Text Editor
- ❖ Understand the Commands
- ❖ Understand the Control Mode

6.1 INTRODUCTION TO VI EDITOR

UNIX system supports two screen editor ed and vi. Of the two, we will discuss vi, which is more popular. The vi editor is a visual editor, used to create and edit text files and programs. vi offers cryptic and sometimes mnemonic internal commands for editing works. It makes complete use of the keyboard, where practically every key has a function.

6.1.1 Starting with VI

A vi session starts with vi command and a filename after the prompt $.

Syntax

$ vi filename<enter>

Example

```
$ vi students
 $ date
 Thu Dec 19 13:44:37 IST 2003
 $ cat students
 Priya
 Sneha
 Anandhi
 Sujitha
 $ vi students
 "students" 4 lines 24 char
```

We can run vi like other Unix commands. When we start vi, it will printout the file name, number of lines and number of

characters at the bottom of our screen. Then it will clear the screen and fill it with the content of the file that we are going to edit

Priya
Sneha
Anandhi
Sujitha
~
~
~
~
~
~
~
~
~
"students" 4 lines 24 char

Now we are presented with a full screen, each line beginning with a ~(tilde) this is vi's way of indicating that they are non-existent lines. The bottom line is the message line. The filename appears in this line with the message "student" [newfile]. If the file exists, then it shows the contents of the file in command mode.

6.1.2 Modes in VI Editor

vi works in three different modes. The command mode, the input mode and the ex escape mode.

Command mode where keys are used as commands to act on text

Input Mode where key pressed is entered as text.

Ex escape mode Ex mode commands are entered in last line of screen to act on text

Command mode

Initially the vi editor is in command mode, to enter into input mode we have to press the insertion key 'i'. To return back to the command mode from the input mode, we need to press escape key <ESC>

The following operation cursor movement, scrolling (screen movement), editing, searching, saving and quitting -are performed in command mode.

Cursor Movement

The basic cursor movement commands are h, j, k & l are left, down, up and right respectively as shown below. Before using the above keys "ESC" key should be pressed. Here are a few examples for the above cursor movement commands. Note that the commands are not echoed on the screen. Assume a file by name students is already available with 4 names. When we call the vi editor the following screens will come.

$ vi studentsCR

"students" 4 lines 24 char

The only thing vi shows on the screen is the contents of the file. From the above figure, we consider the location of the cursor as the highlighted character. After the key j is pressed the cursor moves downwards by one character.

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

move down

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

move down

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

move right

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

move up

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

move right

Priya

Sneha

Anandhi

Sujitha

~

~

~

“students” 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

“students” 4 lines 24 char

move up

Priya

Sneha

Anandhi

Sujitha

~

~

~

“students” 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

“students” 4 lines 24 char

move left

Priya

Sneha

Anandhi

Sujitha

~

~

~

“students” 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

3j

down 3

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

3k

up 3

Priya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 24 char

All the other cursor movement commands work as shown above.

6.2 SOME MORE COMMANDS

In order to work faster in vi, the following commands are used.

W - moves forward by a word

e - moves to last character of the word.

b - moves backward by a word

^ - takes us to the beginning of the line

$ - takes us to end of the line

I - takes us to last line of the file

g - works like the GOTO command. For example to the fifth line in file we can say 5g

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

“sports” 29 lines 699 char

w

go to next word

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack’s spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

“sports” 29 lines 699 char

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack’s spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

“sports” 29 lines 699 char

go to end of word

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

"sports" 29 lines 699 char

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

"sports" 29 lines 699 char

go back to previous word

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

"sports" 29 lines 699 char

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

"sports" 29 lines 699 char

go forward
2 words

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

“sports” 29 lines 699 char

6.2.1 Scrolling commands (screen commands)

Here are some scrolling commands, which are frequently used in Vi editor.

Ctrl f - Moves forward by a screen

Ctrl b - Moves backward by a screen

Ctrl d - Moves the cursor half of the screen forward.

Ctrl u - Moves the cursor half of the screen backward

Ctrl l - It clears any system message that appears on the screen

Ctrl g - Displays the status on the status line.

Assume the following text is available in a file by name ‘sports’. The content of the file can be displayed by using cat command. Using vi editor the above scrolling commands are demonstrated.

$ cat sports

The world cup is the

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have really thought long and hard, before arriving at my squad. My side will have seven specialist batsmen. Virender Sehwag, Sourav Ganguly, Sachin Tendulkar, Rahul Dravid, V V S Laxman, Yuvaraj Singh and Mohammed Kaif. Pick themselves and this is bound to be one of the finest line-up's in the competition. Batting will remain India strength. However, it is important that we pick the right bowlers. Zaheer Khan will surely be the pace attack's spearhead, and the left armer has been surely impressive in recent times. - The Hindu

$ vi sports

The world cup is the

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

“sports” 29 lines 699 char

The world cup is the

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

“sports” 29 lines 699 char

6j

down 6 lines

The world cup is the

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

"sports" 29 lines 699 char

The world cup is the

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

"sports" 29 lines 699 char

down 1 line

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

really thought long and

"sports" 29 lines 699 char

biggest cricketing event,

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

really thought long and

“sports” 29 lines 699 char

j

down 1 line

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

really thought long and

hard, before arriving at

“sports” 29 lines 699 char

and picking a side for the

mega-tournament should

only be done after taking

all the factors into

consideration. I have

really thought long and

hard, before arriving at

“sports” 29 lines 699 char

4j

down 4 lines

consideration. I have

really thought long and

hard, before arriving at

my squad. My side will

have seven specialist

batsmen. Virender Sehwag,

Sourav Ganguly, Sachin

“sports” 29 lines 699 char

consideration. I have

really thought long and

hard, before arriving at

my squad. My side will

have seven specialist

batsmen. Virender Sehwag,

Sourav Ganguly, Sachin

“sports” 29 lines 699 char

^D

down 1/2 screen

have seven specialist

batsmen. Virender Sehwag,

Sourav Ganguly, Sachin

Tendulkar, Rahul Dravid, V

V S Laxman, Yuvaraj Singh

and Mohammed Kaif. Pick

themselves and this is

“sports” 29 lines 699 char

have seven specialist

batsmen. Virender Sehwag,

Sourav Ganguly, Sachin

Tendulkar, Rahul Dravid, V

V S Laxman, Yuvaraj Singh

and Mohammed Kaif. Pick

themselves and this is

“sports” 29 lines 699 char

^D

down 1/2 screen

V S Laxman, Yuvaraj Singh

and Mohammed Kaif. Pick

themselves and this is

bound to be one of the

finest line-up’s in the

competition. Batting will

remain India strength.

“sports” 29 lines 699 char

V S Laxman, Yuvaraj Singh

and Mohammed Kaif. Pick

themselves and this is

bound to be one of the

finest line-up’s in the

competition. Batting will

remain India strength.

“sports” 29 lines 699 char

^U

up 1/2 screen

have seven specialist

batsmen. Virender Sehwag,

Sourav Ganguly, Sachin

Tendulkar, Rahul Dravid, V

V S Laxman, Yuvaraj Singh

and Mohammed Kaif. Pick

themselves and this is

“sports” 29 lines 699 char

Input Mode

Initially the vi editor is in command mode, to enter into input mode we have to press the insertion key 'i'. To return back to the command mode from the input mode, we need to press escape key. <ESC>.

Editing Commands

Editing commands involve different operations such as insertion, deletion, copy and etc. these operations are discussed below.

Insertion commands

The simplest type of input is insertion of text. To insert a text at the cursor position, assuming you are in command mode, press the character 'i'. Pressing this character i, will change the mode from command to input mode. Further key press will result in text being entered and displayed on the screen. Enter key is used to go to the next line.

Insert command can also work as append command by placing the cursor to the required place and then by pressing character 'a'. Now everything we type is appended to the text after the character where the cursor was positioned over. The following examples illustrate this concept.

```
Priya
Sneha
Anandhi
Sujitha
~
~
```

~

"students" 4 lines 24 char

axxyyzz

add xxyyzz

Prxxyyzziya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 30 char

When we complete adding text we have to press ESC key. By doing so, the cursor will move back to the last character that we have entered. After this we are not in insert mode. We will be in command mode.

Prxxyyzziya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 30 char

ESC

Prxxyyzziya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 30 char

We can even put RETURN in the added text. By doing so a new line will be appended.

Prxxyyzziya

Sneha

Anandhi

Sujitha

~

~

~

"students" 4 lines 30 char

aoneCR

twoESC

embedded CR

Prxxyyzzone

twoiya

Sneha

Anandhi

Sujitha

~

~

“students” 5 lines 36 char

Prxxyyzzone

twoiya

Sneha

Anandhi

Sujitha

~

~

“students” 5 lines 36 char

3j

down 3

Prxxyyzzone

twoiya

Sneha

Anandhi

Sujitha

~

~

“students” 5 lines 36 char

Prxxyyzzone

twoiya

Sneha

Anandhi

Sujitha

~

~

"students" 5 lines 36 char

iabcESC

insert abc

Prxxyyzzone

twoiya

Sneha

Anandhi

Suabcjitha

~

~

"students" 5 lines 39 char

Note: Pressing the ESC key after typing the text ends the insertion mode and enters into command mode.

Some more commands

O - allows insertion by creating a blank line above the current line

o -allows insertion by creating a blank line below the current line

A -used for appending the text. The text is appended at the end the line

Delete Command (works in command mode)

Basically there are two commands that delete text in vi :X and dd. To delete one character at the current cursor position, we use x command. And also, a number to indicate

how many characters to delete, can precede the x command, which is shown as below:

Prxxyyzzone

twoiya

Sneha

Anandhi

Suabcjitha

~

~

"students" 5 lines 39 char

x

delete "c"

Prxxyyzzone

twoiya

Sneha

Anandhi

Suabjitha

~

~

"students" 5 lines 38 char

Prxxyyzzone

twoiya

Sneha

Anandhi

Suabjitha

~

~

“students” 5 lines 38 char

x

delete “j”

Prxxyyzzone

twoiya

Sneha

Anandhi

Suabitha

~

~

“students” 5 lines 37 char

Prxxyyzzone

twoiya

Sneha

Anandhi

Suabitha

~

~

“students” 5 lines 37 char

4x

delete “itha”

Prxxyyzzone

twoiya

Sneha

Anandhi

Suab

~

~

"students" 5 lines 33 char

Prxxyyzzone

twoiya

Sneha

Anandhi

Suab

~

~

"students" 5 lines 33 char

xxxx

delete "suab"

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

"students" 5 lines 29 char

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

"students" 5 lines 29 char

x

beep

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

"students" 5 lines 29 char

To delete a line, you can use the dd command. A number to indicate the number of lines to delete can also precede it.

x - deletes the characters before the cursor position

d - deletes line from the current position to the end of the line.

dd- deletes the entire line in the cursor position

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

"students" 5 lines 29 char

dd

delete line

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

3k

move up 3

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

3dd

delete 3 lines

Prxxyyzzone

~

~

~

~

~

~

"students" 1 lines 11 char

Prxxyyzzone

~

~

~

~

~

~

"students" 1 lines 11 char

u

undo last delete

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

"students" 5 lines 29 char

dd

delete line

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

3k

move up 3

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

“students” 4 lines 29 char

3dd

delete 3 lines

Prxxyyzzone

~

~

~

~

~

~

“students” 1 lines 11 char

6.2.2 The Undo Commands (works in command mode)

vi has powerful undo features. The commands used for undo operation is the key 'u'. suppose we want to undo the last delete operation. It can be done as shown below

The other undo command is 'U' which will undo all the changes in the current line, where as 'u' undo's only the recent change.

Prxxyyzzone

~

~

~

~

~

~

"students" 1 lines 11 char

u

undo last delete

Prxxyyzzone

twoiya

Sneha

Anandhi

~

~

~

"students" 4 lines 29 char

6.3 REPLACE COMMANDS

Text can be replaced with any of the following commands r R, s and S. To replace a single character, by

another r can be used. This can be explained with the following example. The r command works in command mode.

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

new text

the left armer has been

surely impressive in

"sports" 30 lines 697 char

r2

replace "t" with "2"

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

new tex2

the left armer has been

surely impressive in

"sports" 30 lines 697 char

To replace more than one character, we can make use of other replace command 'R', which replaces text as the cursor moves right. Note: R command works in **Input Mode.**

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

new tex2

the left armer has been

surely impressive in

"sports" 30 lines 697 char

R6789ESC

replace text

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

6789tex2

the left armer has been

surely impressive in

"sports" 30 lines 697 char

Some more commands

S	-	replace single character under cursor with any number of characters
Cc	-	replace the entire line
Cw	-	replace a word

Search Commands (works in command mode)

Lines containing a string, can also be located by prefixing the string with the/(front slash) to locate the first occurrence of the string "muthu", simply enter

/muthu<eneter> #search forward for the string "muthu" likewise the sequence

?anand<center> #search backward for the most previous instance of the anand

Note: The / searches in the forward direction while the ? searches in the reverse direction.

Some more commands

fx - finds the character, x on the current line after the current cursor position.

Fx- finds the character , x before the cursor position in the current line.

6.4 CONTROL MODE

Search and Replace Commands

vi offers yet another powerful feature, the substitution or the replacement. This is achieved with the ex escape mode's s (substitute) commands.

The address % represents all lines in the file, g makes it truly global. If the pattern cannot be found, vi responds with the following message.

Saving and Quitting Commands

When we edit a file using vi, the original file is not disturbed as such, a copy of the file is placed in buffer. From

time to time we should save our work by writing the buffer contents to disk.

Saving Text

To save a file and remain in the editing mode, use the w(write) command preceded by colon ':'

Example

:w<enter>

$

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

:wCR

:wCR

write file

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

"sports" 29 lines 689 char

with the w command you can optionally specify a filename as well. In that case, the contents are separately written to another file.

The above command keeps you in the command mode so that you can continue editing. However, to save and quit the editor, use the 'x'(exit) command instead.

Example

:x<enter>

$

Note: Instead of using :x command it is better to use :wq command or :z command

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

:wCR

:wCR

write file

that we pick the right

bowlers. Zaheer Khan will

surely be the pace

attack's spearhead, and

the left armer has been

surely impressive in

recent times. - The Hindu

"sports" 29 lines 689 char

6.5 SUMMARY OF VI COMMAND

Using vi from UNIX

$ vi file	edit file
$ vi -r file	recover file from crash

Note: most of the following vi commands may be preceded by a number for repetition.

Basic cursor motions

h j k l	□□□□
CR	Down line to first non-blank
0[zero]	Beginning of line

Screen control

^U ^D	Up or down half page
^B ^F	Up or down whole page
^L	Reprint page

Character input modes

†a	Append after cursor

†A	Append at end of line
†i	Insert before cursor
†I	Insert before first non-blank
†o	Add lines after current line
†O	Add lines before current line

Delete and change

dd	Delete line
†cc	Change line
D	Delete from cursor to EOL
†C	Change from cursor to EOL
x	Delete character
†s	Change character
†S	Change line
r*chr*	Replace current chr with *chr*
†R	Overprint change

Word commands

w	Next word
b	Back word
e	End of word
dw	Delete word
†cw	Change word

Search

/string/	Search for *string*

?*string*?	Reverse search for *string*
n	Repeat last / or ?
N	Reverse of n

Generic commands

object is any cursor motion: w for word; b for back word; h, j, k, l for left, down, up, right; /*string* for upto *string* etc.

d*object*	Delete *object*
†c*object*	Change *object*

Miscellaneous

u	Undo previous command
U	Restore entire line
y*object*	Save *object* in temp buffer
Y	Save line(s) in temp buffer
p	Put saved buffer after cursor
P	Put saved buffer before cursor

Control commands

:w	Write file
:wq	Write file and quit
:q	Quit
:q!	Quit (override checks)
:*ed-num*	Run the ed command ed-cmd
:*num*	Goto line *num*
zz	Same as :q

†Note: all commands marked with † enter input mode and are exited with the escape (ESC) character.

LEARNING ACTIVITIES

Fill in the Blanks:

1. A .... is used to create and manage text files and documents.
2. The .... is a visual editor, used to create and edit text files and programs.
3. .... involve different operations such as insertion, deletion, copy and etc.

LET US SUM UP

At the end of this unit you have understood the concept of Text Editor, the most popular text editor is VI. UNIX system supports two screen editor ed and vi.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Text editor
2. vi editor
3. Editing commands

REFERENCES

R.G. Dromey – How to solve it by Computer – PHI

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

UNIT - 7

INPUT AND OUTPUT

Structure

Overview

Learning Objectives

7.1 Standard Output

7.2 Standard Input

7.3 Redirection

    7.3.1 Output Redirection

    7.3.2 Input Redirection

7.4 Standard Error

Let us sum up

Answer to Learning Activities

References

OVERVIEW

Almost all command in UNIX system takes input from the keyboard and send the resulting output to the display unit. Internally UNIX operating system reads it's input from a place called as standard input which happens to be the terminal. Similarly output are written on to a place called as standard output which is also our terminal by default.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Know the Standard Output
- Understand the Standard Input
- Understand the Redirection
- Know the Standard Error

7.1 STANDARD OUTPUT

To discuss the standard output let us take the 'who' command. 'who' command displace the currently logged in users. In technical terms, 'who' command writes a list of logged in users to standard output (video display unit).

Example

```
$who
vel     tty03    Jan 5  09:45
anand  tty06  Jan 5  09:57
raja    tty01    Jan 5  10:02
raj  tty03    Jan 5  10:40
```

7.2 STANDARD INPUT

Each input command considers it's own input as a stream. The streams can come from different sources. The following is the list of streams for input commands.

1) Keyboard (i.e. keyed in by the user. This is the default source)

2) A file (using redirection)

3) Another program (using the concept of pipeline)

In our previous examples, the following input commands 'cat' and 'wc' are used with filename as arguments. Both the commands have a built in mechanism, to take the input from standard input device also.

Example1

```
$ wc
```

Now I, am learning UNIX operating system

I find it is useful.

<ctrl-d>

2 12 48

In the above example 'wc' command gets the input from the standard input device(keyboard)

Example2

when a sort command is executed without a filename argument, then the command will take input from the standard input(keyboard terminal)Example1

$ sort

Tamil

Raju

Ragu

Divya

<ctrl-d>

Divya

Ragu

Raju

Tamil

$

7.3 REDIRECTION

Apart from the standard input available in Unix, input can be taken from other sources and can be passed to any

destination other than the standard outputs. This process is called redirection. There are two types of redirection:

1. Input redirection
2. Output redirection

The shell is completely responsible for the redirection.

7.3.1 Output Redirection

When output from a process is redirected to any destination other than the standard output i.e. terminal, it is called output redirection

Syntax

$command>file

Here the output is redirected to the file specified instead of VDU.

Example1

$date>today

This command sends the data to the file today. It concordinates the content with its file as follows

$cat today

Output

wed apr 2 11:33:31 pst 1992

Example2

$ who>usernames

The above command line makes the 'who' command to execute and write the output in to the file named "usernames". To see the content of the file we can use cat command. If the

output file doesn't exist, shell will create before executing the command. If the file exists, its content will be erased.

$cat usernames

vel	tty03	Jan 5	09:45
anand	tty06	Jan 5	09:57
raja	tty01	Jan 5	10:02
raj	tty03	Jan 5	10:40

We can also redirect the output of a command to a file in an append mode.

Syntax

$command >> filename

Example

$who >> students

Sometimes two or more commands can be combined and the aggregate output can be sent to a file. A pair of parentheses is used to group the command

Example

(ls -1 ; who) > students

Note: when the output of a command is redirected to a file, the output file is created by the shell before the command is executed.

7.3.2 Input Redirection

Similar to the output redirection, here the input is redirected from a source file rather than the standard input. This is called input redirection

Syntax

```
$command<filename
```

Here filename is the input source. If the specified file is not found, the shell will report an error.

Example1

```
$ cat<usernames
vel      tty03    Jan 5    09:45
anand    tty06    Jan 5    09:57
raja     tty01    Jan 5    10:02
raj      tty03    Jan 5    10:40
```

The input source file used is usernames, whose content is input to the cat command.

Example2

```
$wc - l < usernames
4
```

Combining standard input and output redirection

Both input and output redirection can be combined in a command. The < and > operators can be combined to use both forms of redirection in a single command line.

Example

```
$wc < usernames > usercount        # first input, then output
$wc > usernames < usercount        # first output, then input
$> usernames < usercount wc        # As above, but command
at end
```

7.4 STANDARD ERROR

It includes all the error messages written to the terminal. This output is generated either by command or by the shell. In both the cases the default destination is the terminal. Like standard output, this can also be redirected to a file. (Either using > or >> symbol).

Example

Assume a file "user" is not available. If we try to display the content of the file using cat command error will occur and we can redirect it to any file so that it will be useful for future references.

```
$cat user > errorfile
cat: cannot open user: No such file or directory (error 2)
```

Note: The standard input , standard output and standard error has a number called file descriptor. This is used for identification purposes.

- File descriptor 0 represents standard input
- File descriptor 1 represents standard output
- File descriptor 2 represents standard error

Normally it is not necessary to prefix the numbers 0 and 1.However; we need to use the descriptor 2 for the standard error

```
$cat username 2 >> errorfile
$cat errorfile
cat: cannot open user: No such file or directory (error 2)
cat: cannot open username:No such file or directory (error 2)
```

LEARNING ACTIVITIES

Fill in the Blanks:

1. Internally UNIX operating system reads it's input from a place called as ....
2. The .... is completely responsible for the redirection.
3. The standard input, standard output and standard error has a number called.................

LET US SUM UP

At the end of this unit you have understood all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices. The I/O code represents a significant fraction of the total Operating System.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Standard input.
2. Shell
3. File descriptor

REFERENCES

R.G. Dromey – How to solve it by Computer – PHI

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

UNIT - 8

PIPES & FILTERS

Structure

Overview

Learning Objectives

8.1 Pipes

8.2 Filters

8.2.1 Type of Filters

8.2.2 Sort Filter

8.3 File Permissions

Let us sum up

Answer to Learning Activities

References

OVERVIEW

In UNIX, commands are used to perform single tasks only. If at all we want to perform multiple tasks in one command, it is not possible. Redirection provides solution to this problem. But, it creates lot of temporary files, which are redundant and occupy more disk space. Pipes and filters provides the solution to this problem.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Know the Pipes
- Familiar with File Permissions
- Understand the Filters

8.1 PIPES

Pipes are a mechanism which takes the output of a command as its input for the next command.

Example1

```
$ who > usernames
$ wc -l usernames
4
```

The above example illustrates there are four users currently logged in. This command uses the files and redirection operators to get the number of users. The same result can be obtained with the help of pipes as shown below

```
$ who | wc - l
4
```

The 'who' command is used to show the details of all who have currently logged in the Unix system. The 'wc' command is used to count the number of lines, words or characters in a file. So the output of the 'who' command is the input for the 'wc-l command.

Example2

```
$ cat text | head -3
```

In this statement, 'head' displays the initial 3 lines of the named text. The 'cat' command displays the contents of the file we specify. The output of this statement will be the display of the first three lines of the file text.

Example3

```
$who > users
```

vel	tty03	Jan 5	09:45
anand	tty06	Jan 5	09:57
raja	tty01	Jan 5	10:02
raj	tty03	Jan 5	10:40

$sort users

anand	tty06	Jan 5	09:57
raj	tty03	Jan 5	10:40
raja	tty01	Jan 5	10:02
vel	tty03	Jan 5	09:45

Using pipes the above output can be obtained without using files

$ who | sort

anand	tty06	Jan 5	09:57
raj	tty03	Jan 5	10:40
raja	tty01	Jan 5	10:02
vel	tty03	Jan 5	09:45

8.2 FILTERS

A filter gets the input from the standard input, processes it, and sends to the standard output. Sometimes, the input is also taken from a file.

Uses of filters

- Extract lines with a particular pattern.
- Sort a file.
- Replacing existing characters with the new ones.

- Storing intermediate outputs of a long pipe.
- To get particular columns of a file.
- Merging two or more files together using filters.

8.2.1 Type of Filters

Various types of filters exist in Unix. Some of them are

Sort filter	Arrange input in alphabetical
grep filter	Global search for regular expression
uniq filter	Prevents multiple occurrences of files
pg filter	Output is shown page by page
More filter	Output is shown screenful by space bar

8.2.2 Sort Filter

It is used to customize the output with various options. The 'sort' filter arranges the input taken from the standard input in alphabetical order. It comes with various options like

r option	Sort in reverse alphabetical order
f option	Ignores case distinction
s option	Sorts using numerical values
t option	Specifies field separator (character)

Example

```
$ sort
Ram
Arul
Seetha
```

Output

Arul

Ram

Seetha

From this example, we see that a set of names is taken as input and the same is displayed in alphabetical order.

r OPTION

Sort command when used with this option will display the input stored in reverse alphabetical order

Input

```
$ sort -r
Raj
Geetha
Priya
```

Output

Raj

Priya

Geetha

From the example, it is seen that the names are sorted in reverse alphabetical order.

f OPTION

Sorting of digits, alphabets and other special characters is usually done by comparing their ASCII values. It should be noted that ASCII values for A to Z is lesser than that of a to z.

This case distinction is ignored using the sort command with f option.

Example

$ sort -f

Ram

Raj

Priya

Preetha

The output will be

Preetha

Priya

Raj

Ram

n Option

When sorting numbers, incorrect results are due to comparison of their ASCII values.

For example,

ASCII value of 10 is less than 2 and is placed above 2, which is an error. The n option will sort the input given using their numerical values and then display it.

Example

$ sort -n

28

8

39

Output

8

28

39

THE +pos1 -pos2 OPTION

Assume a file containing names in the following format

First name middle name last name

These are separated by single spaces. Each column is called a field. If we want to sort on any one field, and then sort is used with the option

"+pos1-pos2" option

Example

$ cat names

Raj Kumar Anand

John Mathew Thomas

Victoria Thomas Becker

Mohammed Ali Rafi

To sort on the middle name,

$ sort +1 -2 names.

Output

Mohammed ali rafi

Raj Kumar Anand

Jojn Victoria Thomas

Similarly, name can be sorted on the first and last names also

t option

Usually, the field separator is a blank space or a tab space. But if it is a character, these can be specified using the t-option.

Example

```
$ cat names
Raj:Kumar:Anand
John:Mathew:Thomas
Victoria:Thomas:becker
```

In this example, the field separator is a colon. To sort on any one field,

```
$ sort -t ":" +2 -3
Raja :Kumar :Anand
John: Mathew:Thomas
Vivtoria: Thomas:Becker
```

8.3 FILE PERMISSIONS

File Security using access permission

File security deals with who can access a file and what they can do with the file. For example you might want a file containing some sensitive information to be unreadable by other users or groups. At the same time you might want another file to be accessed by everyone. This is implemented in UNIX by giving suitable access permission for a file to users and/or groups. The following access permissions are implemented for a file in UNIX system.

1. Read access
2. Write access
3. Execute access

The above access permissions can be implemented for an individual owner, group, others. UNIX system uses the following access permission flags for every file.

Example

Assume a file by name priya.dat is available. This can be confirmed by running the following command

$ ls -l priya.dat

-rwxrwxrwx 1 raj csc 80 Jun 27 20:23 priya.dat

The first character is '-' which indicates that priya.dat is a file and not a directory. If it was a directory, it would have displayed'd' instead of '-'.

There are three set of 'rwx' which indicates access permission for owner, group, others

```
 -   r w x    r w x    r w x
    ------------ ------------ ------------
     owner    group    others
```

where

r = read

w = write

x = executable

By properly setting the flag patterns for each set, the user can grant / remove access permission for that particular owner, group and others by using chown or chmod or chgrp.

chmod

chmod is used to set all the three access permission for owner, group and others.

Syntax

$ chmod usertype operation access_permission file(s)

where

usertype	-	owner, group or others
operation	-	add/remove
access_permission	-	read, write and execute
file(s)	-	one or more files

Example

$ ls -l priya.dat

-rwxrwxrwx 1 raj csc 80 Jun 27 20:23 priya.dat

In the above example the file priya.dat has full access rights to all the category of users.

To deny read access to groups and for others the following command is issed

$ chmod g -r o -r priya.dat

$ ls -l priya.dat

-rwx-wx-wx 1 raj csc 80 Jun 27 20:23 priya.dat

UserType	Operation	Access_Permission
u - user	+ grant	r - read
g - group	- remove	w - write
o - others		x - execute
a - all		

umask

umask command can be used to set the default creation mode of users file and directories. This command is mostly used by the administrators.

Syntax

$ umask accessflags

Typical umask modes

Command	**Description**
umask 002	Create files without write permission for others
umask 002	Create files without write for group or others
umask 006	Create files without read or write for others
umask 026	Create files without read or write for others and without write for group
umask 007	Create files without read, write, or execute for others
umask 077	Create files without read, write or execute for anyone but for the owner

LEARNING ACTIVITIES

Fill in the Blanks:

1. ...... is a mechanism which takes the output of a command as its input for the next command.
2. A .... gets the input from the standard input, processes it, and sends to the standard output.
3. The .... filter arranges the input taken from the standard input in alphabetical order.

LET US SUM UP

At the end of this unit you have understood the Pipes and Filters. Pipes are a mechanism which takes the output of a command as its input for the next command. A filter gets the input from the standard input, processes it, and sends to the standard output. Sometimes, the input is also taken from a file.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Pipes

2. Filter

3. 'sort'

MODEL QUESTIONS

1. Define Pipe.
2. Define Filter.

REFERENCES

R.G. Dromey – How to solve it by Computer – PHI

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

BLOCK 3 INTRODUCTION

At the end of this block you will know the Programming in Unix. Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design. This block explains the basics of shell programming by showing the evolution of some useful shell programs.

Introduction to System Software is divided into Four Blocks. Block 3 consists of Three Units.

Unit 9: discusses how to use the shell for writing programs that will stand up to use by other people. Topics include more advanced control flow and variables.

Unit 10: deals with the Control Statements. The shell programming also has the facilities by using 'for' and the 'while' loops.

Unit 11: is a discussion of the UNIX file system. To talk comfortably about commands and their interrelationships, we need a good background in the structure and outer workings of the file system.

UNIT - 9

SHELL SCRIPT

Structure

Overview

Learning Objectives

9.1 Introduction

9.2 Functions of Shell

9.2.1 Command Line Structure

9.2.2 Execution of a Shell Script

9.3 Shell Variables

9.3.1 Assigning value to a shell variable

9.3.2 Using shell variables

9.3.3 Reading data from Terminal/User

9.3.4 Arithmetic operations with shell variables

Let us sum up

Answer to Learning Activities

References

OVERVIEW

The shell or the command interpreter is a program that interprets your request to run the UNIX commands. This can be stored in a file. The shell can read the file and execute the command in it. When the system prints the prompt $ and you type commands that get executed, it's not the kernel that is talking to you, but a go-between called the command interpreter or shell. The shell is just an ordinary program like data or who, although it can do some remarkable things.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Know the Shell Script
- Familiar with Functions of Shell
- Understand the Shell variable

9.1 INTRODUCTION

The shell or the command interpreter is a program that interprets your request to run the UNIX commands. This can be stored in a file. The shell can read the file and execute the command in it; such a file is called a SCRIPT FILE. The UNIX operating system is flexible in a way that it is not tied to any particular command interpreter.

These are currently three popular shells.

The "Bourne shell"	sh
The "C shell"	csh
The "Korn shell"	ksh

The korn shell and Bourne shell was prepared by Stephen Bourne and Davuid Korn respectively. The "c shell" was developed at the university of California at Berkeley which resembles the c programming language.

The Bourne shell is currently distributed with standard AT&T UNIX systems. The Korn shell is compatible with borne shell but shell is not. The user prompt for c shell is '%'. If the user prompt is '#' then the shell is administrator shell.

9.2 FUNCTIONS OF SHELL

Program Execution

When the shell is waiting for the response of the programmer to enter the command, it displays the $ prompt. Once you type in your command and press <return> the shell analyzes the line. This line is commonly known as command line.

9.2.1 Command Line Structure

The command line structure is as follows

$ command argument

The simplest command is single word, usually name of a file for execution.

$ who <enter>

The above command gives details of the currently logged user.

$ who am i<enter>

this command gives information about the currently working user.

A command usually ends with a new line, but a semicolon; is also a command terminator or delimiter.

$ date; <enter>

This command gives the current date along with time.

i.e., wed sep 28 09:07:15 EDT 1983

9.2.2 Execution of a Shell Script

The UNIX operating system does not grant permission to execute any of its files. A file canon be directly executed, it can

be executed using sh command. For instance, to execute a script called general, the command is

$ sh general<enter>

To execute a shell script directly at the $ prompt we have to change the File Access Permission (FAP) of the specified shell script by generating the execute permission. For instance, to execute the shell script called general,

$ chmod u+x general #change File Access Permission
#to Execute#

$ general #execute the shell script#

File name substitution

When file name substitution is specified on the command line with the wild characters such *, ?, [...], then the shell will perform substitution. This happens before the program gets executed.

I/O Redirection

If input and/or output redirection is specified on the command line then I/O redirection is handled by shell.

Pipeline hookup

If the command line contains two commands connected by a pipe then the shell takes responsibility of connecting the output of first program to the input of the second.

Example

$ who | wc –l

Environment control

Shell gives us flexibility in customizing our command line environment as per our needs. This environment includes a path name of our home directory, the directories those will be searched by the shell, whenever a name of a file is specified to be executed.

Interpretive programming languages

The shell provides a powerful programming language. The statements in this language can be typed in directly at the terminal for execution or can be entered into a file.

The echo command

The echo command is similar to the 'printf' statement in C language. This command is used to display messages on the screen. For example,

$ echo UNIX is an operating system<enter>

UNIX is an operating system

The echo command displays the text and then puts a new line character at the need of the next. The new line character causes the cursor to move to the next line after the text is displayed.

Example

$ echo UNIX is an operating system\007

Will display the cursor on the same line of the screen after displaying the argument UNIX is an operating system\n

Will display an additional blank line after the argument.

9.3 SHELL VARIABLES

All variables in UNIX are treated as character strings. A shell variable name begins with a letter (upper or lowercase) or underscore character and optionally is followed by a sequence of letters, underscore characters or numeric characters.

a5

total

output_file

ROOTDIR

_cflag

are examples of valid shell variable names, whereas the names

5a	#cannot begin with numeric character#
.home	#'.' Is not a valid character#
echo	#keyword are not allowed as variable#

are invalid shell variable names

9.3.1 Assigning value to a shell variable

In UNIX variables are not explicitly declared. They are created at any point of time, by a simple assignment of value. A variable can be created as follows,

<variable name>=<value>

Example

$area=100

$echo $ area

100

Here, the value of the shell variable 'area' is displayed.

Note: spaces are not allowed before and after the assignment operator.

If the value of the variable is a string of characters, in such a case it is advisory to enclose the string within double quotes.

Example

```
$ variable="top class"

$ echo Your performance is $variable

echo Your performance is top class
```

Whenever shell encounters a dollar sign followed by a variable name, the value of that variable gets substituted at that same point by the shell. This explains the output from the following sequence of commands.

```
$ height=180

$ echo the height of the cylinder is $height

#the output for the above shell script is as follows#

the height of the cylinder is 180

$
```

It can also be used as command arguments.

```
$ variable= "top"

$ wc-l $variable

150 top

$
```

This command gives the number of lines in the file named variable.

9.3.2 Using shell variables

The shell variables can be used as command line arguments.

Example

```
$ file=emp.txt #assume emp.txt has 100 lines
$ wc -l $file
100 emp.txt
$
```

Sometimes the value of one shell variable can be assigned to another shell variable.

```
$ file=emp.txt #assume emp.txt has 100 lines
$ file2=$file
$ wc -l $file2
100 emp.txt
$
```

9.3.3 Reading data from Terminal/User

The shell allows the user to enter a value into a variable during execution of a shell script. This is done using the read command. The general syntax is

```
read <file name>
```

The read command in UNIX is similar to 'scanf' statement in c language.

For example,

```
$ echo "Enter your name: \c"; read name
Enter your name: Mohanakrishnan
```

```
$ echo $name
Mohanakrishnan
```

In the above program \c is used to keep the cursor in the same line and wait for user's input.

Example,

```
$ read a b
onethousand two hundred and twenty rupees
$ echo :$a:$b
:onethousand:two hundred and twenty rupees
$
```

In the above program, values typed in are delimited by blanks or tabs. That's the reason, why variable 'a' has got one thousand and variable 'b' has got two thousand and twenty rupees.

9.3.4 Arithmetic operations with shell variables

As already said, the shell does not support numeric variable. All the variable are treated as character strings. The declaration,

```
A=96
```

Means that the variable A contains character 9 and 6 and not the number 96.

The "expr" command in UNIX is used to evaluate expressions.

For example,

```
$ expr 10 + 20
```

Will display 30 on the screen

Note: There must be space on either side of the operator + .

Consider the example

```
$a=20
$a='expr $a + 20'
```

would assign 40 to a.

LEARNING ACTIVITIES

Fill in the Blanks:

1. The .... is a program that interprets your request to run the UNIX commands.
2. The .... displays the text and then puts a new line character at the need of the next.
3. The .... can be used as command line arguments.

LET US SUM UP

The shell - the program that interprets your requests to run programs – is the most important program for most UNIX users; with the possible exception of your favorite text editor, you could spend more time working with the shell than any other program.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. shell or the command interpreter
2. echo command
3. shell variables

MODEL QUESTION

1. Describe the Functions of Shell.
2. Write a shell script to find smallest of two numbers.
3. Write a shell script to find sum of given numbers.
4. Write shell scripts to find the given number is add or even.
5. Write a shell script to find a factorial of a number.
6. Write a shell script to find the given number is positive or negative or zero.
7. Write a shell script to print odd numbers upto 'n'.

REFERENCES

R.G. Dromey – How to solve it by Computer – PHI

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

UNIT - 10

CONTROL STATEMENTS

Structure

Overview

Learning Objectives

10.1 Control Statements

10.1.1 Different types of if construct

10.2 Case Statement

10.3 Iterative Statements

10.3.1 for Loop

10.3.2 While Loop

Let us sum up

Answer to Learning Activities

References

OVERVIEW

All programming languages provide some statements that allows a programmer to do some actions based upon some test conditions.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Know the Control Statements
- Familiar with Case Statement
- Understand the Iterative Statements

10.1 CONTROL STATEMENTS

In shell, the if performs this functionality.

Normally in shell script, the test conditions are obtained using relational operators as follows:

Operator	Used to test if	Example
=	Two strings are equal	"$a" = yahoo
!=	Two strings are not equal	"$user"!= priya
-n	A string has nonzero length	-n "$a"
-z	A string has zero length	-z "$file"
-eq	Two integers are equal	"$sum" -eq 30
-ne	Two integers are not equal	"$count" -ne 3
-lt	One integer is less than another	"$a" - lt 334
-le	One integer is less than or equal to another	"$count1" -le 433
-gt	One integer is greater than another	"$a" - gt 334
-ge	One integer is greater than or equal to another	"$count1"-ge 433
-f	A file is an ordinary file	-f employee_file
-d	A file is a directory	-d mydir
-s	A file has nonzero length	-s grepout

In addition, relational expressions can be joined with either the and operator -a or the or operator -o.

For example

```
$ if [$a -gt 10 -a $b -le 15]
```

10.1.1 Different types of if construct

1. Simple if construct
2. if .. then .. else construct
3. Nested if construct

Simple if construct

The general format for if construct is

```
if condition
then
    command1
    command2
    ...
    commandn
fi
```

If the condition is TRUE then the commands enclosed in between then and fi will be executed. Otherwise, control will be transferred to the next statement in the script.

Example

```
$ if [$a -gt $b] #assume a=10, b=4
then
    echo A is greater than B
fi
A is greater than B
$
```

if .. then .. else construct

Like most of the high level programming languages, shell also provides the statement 'if' to implement the decision control structure

Syntax

if condition

then

Operator	Used to test if	Example
=	Two strings are equal	"$a" = yahoo
!=	Two strings are not equal	"$user" != priya
-n	A string has nonzero length	-n "$a"
-z	A string has zero length	-z "$file"
-eq	Two integers are equal	"$sum" -eq 30
-ne	Two integers are not equal	"$count" -ne 3
-lt	One integer is less than another	"$a" - lt 334
-le	One integer is less than or equal to another	"$count1" -le 433
-gt	One integer is greater than another	"$a" - gt 334

-ge	One integer is greater than or equal to another	"$count1" -ge 433
-f	A file is an ordinary file	-f employee_file
-d	A file is a directory	-d mydir
-s	A file has nonzero length	-s grepout

In addition, relational expressions can be joined with either the and operator -a or the or operator -o.

For example

$ if [$a -gt 10 -a $b -le 15]

elif - Nested if construct

The if...then...elif...else...fi statement is used to check multiple conditions in one statement.

Syntax

```
if condition
then
    command
    command
    ...
elif condition
then
    command
    command
```

```
fi
```

Example

```
echo enter a number
read a
if[a -gt 0]
then
   echo "a is positive"
elif [a -le 0]
then
   echo "a is negative"
else
   echo "a is zero"
fi
```

10.2 CASE STATEMENT

The shell case statement is very useful when we want to compare a value against a collection of values. It could also be implemented with if...then...elif...fi statement chain. But the case statement is more convenient for implementation.

Syntax

```
case value
in
pattern1     ) command
               command
```

```
                  command;;
   pattern2    )  command
                  command
                  ...
                  command;;
   pattern3    )  command
                  command
                  ...
                  command;;
   esac
```

<u>Example</u>

```
   case "$myval"
   in
      0 ) echo zero;;
      1 ) echo one;;
      2 ) echo two;;
      3 ) echo three;;
      4 ) echo four;;
      5 ) echo five;;
      6 ) echo six;;
      7 ) echo seven;;
      8 ) echo eight;;
```

```
    9 ) echo nine;;
    * ) echo invalid argument;;
esac
```

10.3 ITERATIVE STATEMENTS

Most of the programming languages have a mechanism to execute a group of statements repeatedly. The shell programming also has the same facilities by using 'for' and the 'while' loops.

10.3.1 for Loop

for loop is a repetitive control structure to execute statements repeatedly for a particular number of times.

Syntax

```
for variable in data1 data2.......datan
do
    body of the loop
done
```

The for loop will execute the statements which are written in the body of the loop for values mentioned in the list following keyword in.

Example

To print 3 numbers

```
for a in 1 2 3
do
    echo "the value of a is $a"
done
```

output

the value of a is 1

the value of a is 2

the value of a is 3

In the above program each listed item following the keyword, is assigned to variable name 'a' and echo command sends the value to the screen.

10.3.2 while Loop

This loop is used to perform the same task as long as the condition is true.

Syntax

```
while condition
do
    body of the loop
done
```

Example

To print three numbers

```
a=1
while [$a -le 3]
do
    echo $a
    $a='expr $a+1'
done
```

output

2 3

The above shell program prints numbers from 1 to 3. since shell script values are string, expr(expression) function converts string into its numeric equivalent.

LEARNING ACTIVITIES

Fill in the Blanks:

1. The .... statement is very useful when we want to compare a value against a collection of values.
2. .... loop is used to perform the same task as long as the condition is true.
3. .... loop is a repetitive control structure to execute statements repeatedly for a particular number of times.

LET US SUM UP

At the end of this unit you have understood the Control Statements. However, for general problem solving we need the ability to control which statements are executed and how often. The control how many times a statement list is executed. The for loop was used for a number of simple iteration programs. Usually, a for loops over a set of filenames, as in 'for i in *.c' or all the arguments to a shell program, as in 'for I in $*'. But shell loops are more general. The while and until use the exit status from a command to control the execution of the commands in the body of the loop.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. shell case
2. while
3. for

MODEL QUESTION

1. Explain the control statements.

UNIT - 11

FILE SYSTEM

Structure

Overview

Learning Objectives

11.1 File System

    11.1.1 Naming of files

    11.1.2 Parent - Child Relationship

    11.1.3 File Management Utilities

11.2 Directory management commands

11.3 File operation commands

11.4 File compression

    11.4.1 Copy, Move, Remove & Time

11.5 File Comparison

11.6 File Security

Let us sum up

Answer to Learning Activities

References

OVERVIEW

This unit will help you to understand the File System. It can contain text, commands, data and sometimes machine language codes. Also to learn Directory management commands.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Know the File System
- Familiar with Directory management commands
- Understand the File operation commands

11.1 FILE SYSTEM

File system in UNIX is a stream of bytes. It can contain text, commands, data and sometimes machine language codes. For example

Text Files	:Lines of ASCII characters separated by a new line
Commands	:Sequence of commands interpreted by UNIX text
Data	:File containing data as stream of bytes
Executable	:File containing machine language instructions

Although everything is treated as files by UNIX, files are categorized as follows

Ordinary Files - Contains only data

Directory Files - Contains other files and directories

Device Files - Represents all hardware devices

11.1.1 Naming of files

In most of UNIX systems a file name can consist of 255 characters. File name may or may not have extensions and can consist of any ASCII characters except /.

Sometimes a file can be identified by two different names (multiple links to the same file) as shown below.

Example

average.c largest.cpp

.profile my_text_file.txt - are valid file names in UNIX.

11.1.2 Parent - Child Relationship

Every file in UNIX system is related to one another.

This relationship is organized in a hierarchical structure as shown below :

/bin	Basic UNIX utilities
/dev	Special I/O device files
/etc	Administrative Programs
/lib	Libraries used by UNIX
/usr/bin	UNIX utilities
/usr/adm	Administrative commands and files
/tmp	Temporary files created on error conditions

11.1.3 File Management Utilities

In UNIX Operating System all types of files are managed by a set of file utility commands.

The following are the frequently used file management commands

1. Directory management commands
   a. cd
   b. pwd
   c. mkdir
   d. rmdir
   e. mvdir
2. File operation commands
   a. File content

i. cat
ii. ls
iii. wc
iv. find
v. file

b. File compression
   i. pack
   ii. unpack
c. File mount
   i. mount
   ii. unmount
d. Copy, move, remove & time
   i. cp
   ii. ln
   iii. mv
   iv. rm
   v. touch

3. File Comparison
   a. cmp
   b. comm
4. File Security
   a. passwd
   b. crypt
   c. chown
   d. chmod
   e. chgrp
   f. umask

11.2 DIRECTORY MANAGEMENT COMMANDS

cd - Change Directory

cd command is used to change the current working directory

Example

$ cd /usr/david/c # current directory is c #

$ cd .. # current directory becomes david

$ cd c # again current directory becomes c #

pwd - Print Working Directory

When user logs in to the system it provides a working directory for the user called as current working directory. pwd command is used to know the current working directory.

pwd is an abbreviation of Print Working Directory.

Example

$ cd /usr/david/c # current directory is c #

$ pwd # current working directory will be

as follows:

/usr/david/c

mkdir - Make Directory

mkdir command is used to create an empty directory.

Syntax

$ mkdir directoryname

Example

$ mkdir anand # creates a subdirectory called anand #

$ mkdir sudeer # creates a subdirectory called sudeer#

rmdir - Remove Directory

rmdir command is used to remove a directory provided that particular directory is empty, i.e, the directory which is to be removed should not have any files or other sub directories in it.

Syntax

$ rmdir directoryname

Example

$ rmdir anand # removes a subdirectory - anand #

$ rmdir sudeer # removes a subdirectory - sudeer#

mvdir - Move Directory

mvdir command is used to

Syntax

$ mvdir directoryname

Example

$ mvdir anand # removes a subdirectory - anand #

11.3 FILE OPERATION COMMANDS

File content

cat - Concatenate & Print : cat command is used to create a file or concatenate or print the content of a file on the standard output devices such as screen or printer.

Syntax : $ cat filename

Example

$ cat myfile.c #displays the content of myfile.c on screen#

Sometimes, cat command can also be used to create files as follows

Example

```
$ cat file1 > file2                # file2 is created with the
                                   # content of file1

$ cat file1 file2 > file3          # a single file by name
file3
                                   # is created with the
contents
                                   # of file1 merged with file2
```

ls - list

ls command is used to list all the files in the current directory or a specified directory.

Syntax

$ ls [options]

Options

-1 number one single column output

-l long format

-a all entries including dot files

-s gives no. of disk blocks

-i inode no.

-t ordered by modification time recent first

-R recursively display all directories, starting from specie or current directory

Example

Assume current directory as following files: test1.c, test2.c, test3.c

$ ls -1

Output

test1.c

test2.c

test3.c

Example

$ ls -l

Output

total 42

-rw-r--r-- 1 anand mech 4334 May 7 01:10 test1.c

-rw-r--r-- 1 anand mech 324 May 7 05:10 test2.c

-rw-r--r-- 1 anand mech 3234 May 7 03:10 test3.c

The first line of the output indicates the total of 42 blocks of disk space consisting of 512 bytes are occupied by the 3 files. The -l (long) option gives the information about access permissions, file size, ownership details, last modified date & time.

wc command

wc command is used to get the total number of lines, words and characters for a given file.

Syntax

$ wc [options] filename

Options

-l counts only the number of lines

-w counts only the number of words

-c counts only the number of characters

Example

Assume myfile.dat has 7 lines, 24 words and 104 characters in it.

$ wc myfile.dat

7 24 104

The find Command

This command locates the files both in the directories and subdirectories. It performs recursive searches for finding the files.

Syntax

find path_list selection_criteria action

The search always begins with a root directory when the path list '/' is specified. All find operators start with a -, and the path list can never contain one. The various options used are

The -name Option

This lists out the specific files in all directories beginning from directory name specified. Wild card options can also be used here.

The -type option

This option is useful in identifying ordinary and directory files.

-type d -> represents directory files.

-type f -> represents ordinary files.

The -mtime optionL

This option allows finding the files that has been modified before or after a specified time.

Example

Current date -15/11/02

-mtime +5

Output

Displays those files, which have been modified before 10/11/02.

The -exec option

The files, which have been located using the find command, can be executed using this option.

The -ok option

This option is similar to -exec option in executing the files located by the find command, but this option has its special interactive nature

file command

Since UNIX system categorize files into ordinary, directory and device files, this file command is used to get the type of the file.

Syntax

$ file filename

Example

Assume employee.dat has details about the employees of a company.

$ file employee.dat

employee.dat: ascii text

11.4 FILE COMPRESSION

pack - compress a file

Pack command is used to compress a file, so that it occupies less space [Normally 20-40%]. The compressed will be named as filename.z. In general executable files are packed.

Syntax

```
$ pack filename
```

Example

```
$ pack employee.dat     #employee.dat.z file will be created
```

unpack - Uncompress a file

unpack command is used to uncompress a packed file.

Syntax

```
$ unpack filename.z
```

Example

```
$ unpack employee.z    #will create employee.dat
```

11.4.1 Copy, Move, Remove & Time

cp - copy a file

cp command is used to copy the contents of one file into another file. And also from one directory to another directory.

Syntax

```
$ cp sourcefile destinationfile
```

The content of sourcefile is copied to the destinationfile. If the destinationfile exists already UNIX system overwrites without giving any warning.

Example

$ cp myfile.dat yourfile.txt

$ cp myfile.dat /usr/raj/yourfile.txt

$ cp /usr/csc/raj/myfile.dat /usr/mech/ravi/edata.txt

ln - link

UNIX allows a file to have more than one name, yet maintaining a single copy in the disk. When changes in file takes place, it reflects in the other. To implement such a link between two files, we use ln command.

Syntax

$ ln file1 file2

Example

$ ln employee.dat emplist.txt

mv - move/rename a file

mv command is used to move or rename files. When two arguments refer different directories, then the file is actually moved from the first directory to the second directory.

Syntax

$ mv file1 file2

$ mv dir1 newdir2

Example

$ mv employee.dat emplist.txt #rename file employee.dat #to emplist.txt within #the same directory

$ mv /usr/csc/raj/employee.dat /usr/mech/mahi

in the above example, the file employee.dat from /usr/csc/raj will be moved to the directory usr/mech/mahi with the same file name.

rm - remove a file

rm command is used to remove files. The argument to the rm command is a name of the file to be removed. Sometimes we can remove more than one file using multiple arguments and wildcards.

Syntax

$ rm filename

Example

$ rm employee.dat

touch - to change timestamps

touch command is used to change the timestamp of one or more files. Sometimes, it may be required to modify the access time of a file into a predefined value. The touch command does this.

Syntax

$ touch [options] [expressions] filename(s)

expression should be in mmddhhmm format.

Example

```
$ touch 11182305 file2.txt #file2.txt will have the date
                           #and time like 18th Nov 23:05
```

11.5 FILE COMPARISON

cmp - compare two files

cmp command is used to compare two files. When the user has doubt about having identical files, he can use this command to have a comparison of those files and if necessary he may delete either of the files.

Syntax

```
$ cmp file1 file2
```

Example

```
$ cmp emp1.txt employee.dat
```

Output

```
emp1.txt employee.dat differ: char 10, line 3
```

Options

The -l (list) option gives the detailed list of the byte number and the differing bytes in octal for each character that differs in both files.

Example

```
$ cmp -l emp1.txt employee.dat
4 145 147
6 135 144
8 132 136
9 130 142
```

There are four differences in between the two files at the character position 4, 6, 8 and 9.

The two files are compared byte by byte and location of the first mismatch is echoed to the screen.

Identical files

If both the files are identical, the cmp command displays no message but simply returns to the $ prompt

Example

```
$ cmp emp1.txt emp1.txt
$
```

comm - to find what is common

comm command is used to compare two sorted files. It compares each line of the first file with its corresponding line in the second.

Syntax

```
$ comm file1 file2
```

Example

```
$ cat file1
andal
babu
devi
hari
sony
$ cat file2
anand
babu
```

dayal

hari

sony

$ comm file1 file2

anand

andal

babu

dayal

devi

hari

sony

The first column contains 2 lines unique to the first file while the second column shows 2 lines unique only to the second file and 3rd column displays 3 lines common to both files.

11.6 FILE SECURITY

passwd - Password Command

passwd command is used to change the password of the current user whenever it's necessary.

Syntax

$ passwd

Example

$ passwd

Changing password for ram

Old password: xxxxxx [not printed]

New password: xxxxxx [not printed]

Re-enter new password: xxxxxx [not printed]

$

This command prompts the user to type the old password and then the new password. For security reasons the typed password won't be displayed on the screen.

Crypt

crypt command is used to hide the content of the file using encryption method. Secret files and passwords are normally encrypted to prevent from being misused.

<u>Syntax</u>

$ crypt key < filename

<u>Example</u>

$ cat students

ram 29/10/1975

raghu 11/10/1975

rajesh 14/12/1976

$ crypt mykey < students

#sfd.$55*fsd.$432s.fsdfsdfds_*%

The encrypted file will have file name as students.crypt.

To decrypt an encrypted file the same crypt command is used.

<u>Example</u>

$ crypt mykey < students.crypt

$ cat students

ram 29/10/1975

raghu 11/10/1975

rajesh 14/12/1976

LEARNING ACTIVITIES

Fill in the Blanks:

1. In most of UNIX systems a file name can consist of .... characters.
2. ....command is used to hide the content of the file using encryption method.
3. .... command is used to compare two sorted files.

LET US SUM UP

At the end of this unit you have understood the File System. Files are managed by the Operating System. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the file system and is the subject of this unit.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. 255
2. crypt
3. comm.

MODEL QUESTIONS

1. How will you list the ordinary files in your current directory that are not writable?
2. When do you use grep?
3. Which two commands in the UNIX system let two users "chat" with each other?
4. Why is mail preferable to write/
5. If you don't want to be distributed by others, what precautions should you take?
6. What is the default location of a user's mailbox?

REFERENCES

Andrew S. Tanenbaum – Operating System Design and Implementation - PHI

BLOCK 4 INTRODUCTION

At the end of this block you will know the Software Engineering. Software Engineering techniques have evolved over many years as a result of a series of innovations and accumulation of program writing experience of writing good quality programs. A software life cycle is the series of identifiable stages that a software product undergoes during its lifetime. Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. CASE tools promise reduction in software development and maintenance costs. Case tools help develop better quality products more efficiently. A Case tool is a generic term used to denote any form of automated support for software engineering.

Introduction to System Software is divided into Four Blocks. Block 4 is about Software Engineering.

Unit 12: deals with the Software life cycle processes, Engineer's responsibility, Software Quality and Case Tools

UNIT - 12

SOFTWARE ENGINEERING

Structure

Overview

Learning Objectives

12.1 Introduction to Software Engineering

12.2 Software Life Cycle Processes

12.3 The Engineer's Responsibility

12.4 The Software Quality

12.6 4GL and Natural Languages

12.7 Case Tools

Let us sum up

Answer to Learning Activities

References

OVERVIEW

Software design technique concerns how to effectively decompose a large problem into manageable parts. Handling complexity in a software development problem is a central theme of the software engineering discipline. You would also learn the techniques of software specification, user-interface development, testing project management and so forth. Even if you intend to write small programs, the software engineering principles could help you to achieve higher productivity and at the same time enable you to produce better quality programs.

LEARNING OBJECTIVES

After completing this unit, you should be able to:

- Know the Software Engineering

- ❖ Familiar with Software Life Cycle Processes
- ❖ Understand the Software Quality
- ❖ Understand the Case Tools

12.1 INTRODUCTION TO SOFTWARE ENGINEERING

What is Software Engineering?

Software engineering is a **modeling** activity **problem-solving** activity **knowledge acquisition** activity **rational-driven** activity.

Modeling

A *model* is an abstract representation of a system (real or imaginary) that enables us to answer questions about the system. Models are constructed of the problem domain in order to understand the problem solution domain in order to understand different solutions and trade-offs. Problem Solving

Knowledge Acquisition

Knowledge acquisition is non-linear - additional knowledge may invalidate previous knowledge.

Rational Management

The *context in which each design decision was made* is called the *rational* of the system.

The Nature of Software

Software is flexible. Software is an executable specification of a computation. *Software is expressive.* All computable functions may be expressed in software. Complex event driven systems may be expressed in software. *Software*

is huge. An operating system may consist of millions of lines of code. *Software is complex.* Software has little regularity recognizable components found in other complex systems and there are exponentially many paths through the code and changes in one part of the code may have unintended consequences in other equally remote sections of the code. *Software is cheap.* Manufacturing cost is zero, development cost is everything. Thus, the first copy is the engineering prototype, the production prototype and the finished product. *Software is never finished.* The changing requirements and ease of modification permits the maintenance phase to dominate a software product's life cycle, i.e., the maintenance phase consists of on going design and implementation cycles. *Software is easily modified.* It is natural to use an iterative development process combining requirements elicitation with design and implementation and use the emerging implementation to uncover errors in design and in the requirements. *Software is communication.* Communication with a machine but also communication between the client, the software architect, the software engineer, and the coder. Software must be readable in order to be evolvable.

12.2 SOFTWARE LIFE CYCLE PROCESSES

Software life cycle

The period of time that begins when a software product is conceived and ends when the software is no longer available for use. This cycle typically includes a

Traditional software life cycle phases	**Extreme programming**
concept phase software development life cycle requirements phase design phase implementation phase test phase installation and checkout phase operation and maintenance phase retirement phase These phases may overlap and/or be performed iteratively.	Listening/Planning Designing Testing Coding These phases overlap and are performed iteratively.

Software life cycle processes

<table>
<tr><th colspan="2">Primary life cycle processes</th><th>Supporting life cycle processes</th></tr>
<tr><td colspan="2">Acquisition</td><td>Documentation</td></tr>
<tr><td colspan="2">Supply</td><td>Configuration management</td></tr>
<tr><td>Development</td><td>Operation
Maintenance</td><td>Quality assurance
Verification
Validation
Joint review</td></tr>
</table>

	Audit Problem resolution
Organizational life cycle processes	
Management Improvement	Infrastructure Training

12.3 THE ENGINEER'S RESPONSIBILITY

The software engineer who writes a program is best able to find and fix its defects. It is thus important that software engineers take personal responsibility for the quality of the programs they produce. Writing defect-free programs, however, is challenging and it takes effective methods, skill, practice, and data. By using the Personal Software Process (PSP), engineers learn how to consistently produce essentially defect-free programs.

12.4 THE SOFTWARE QUALITY

The Software Quality Problem

Software quality is becoming increasingly important. Software is now used in many demanding applications and software defects have caused serious damage and even physical harm. While defects in financial or word processing programs are annoying and possibly costly, nobody is killed or injured. When software-intensive systems fly airplanes, drive automobiles, control air traffic, run factories, or operate power plants, defects can be dangerous. People have been killed by defective software.

While there have not been many fatalities so far, the numbers will almost certainly increase. In spite of all its problems, software is ideally suited for critical applications: it does not wear out or deteriorate. Computerized control systems are so versatile, economical, and reliable that they are now the common choice for almost all systems. Software engineers must thus consider that their work could impact the health, safety, and welfare of many people.

The Risks of Poor Quality

Any defect in a small part of a large program could potentially cause serious problems. While it may seem unlikely that a simple error in a remote part of a large system could be potentially disastrous, these are the most frequent sources of trouble. The problem is that systems are becoming faster, more complex, and automatic. Catastrophic failures thus are increasingly likely and potentially more damaging.

In the design of large systems, the difficult design issues are often carefully studied, reviewed, and tested. As a result, the most common causes of software problems are simple oversights and goofs. These are typically simple mistakes that were made by individual software engineers. While most of these simple mistakes will get caught in compiling and testing, engineers inject so many defects that large numbers still escape the entire testing process and are left to be found during product use.

The problem is that software engineers often confuse simple with easy. They feel that their frequent simple mistakes will be simple to find. They are often surprised to learn that such trivial errors as omitting a punctuation mark, misnaming a

parameter, incorrectly setting a condition, or disseminating a loop could escape testing and cause serious problems in actual use. These, however, are the kinds of things that cause a large proportion of the problems software suppliers spend millions of dollars finding and fixing. While most of these trivial mistakes will have trivial consequences, a few can cause unpredictable and possibly damaging problems.

The quality of large programs depends on the quality of the smaller programs of which they are built. Thus, to produce high quality large programs, every software engineer who develops one or more of the system's parts must do high-quality work. This means that all the engineers must be personally committed to quality. When they are so committed, they will track and manage their defects with such care that few if any defects will later be found in integration, system testing, or by the customers. The SEI has developed the Personal Software Process (PSP) and the Team Software Process^{SM} (TSP^{SM}) to help engineers work this way.

Measuring Software Quality

Software quality impacts development costs, delivery schedules, and user satisfaction. Because software quality is so important, we need to first discuss what we mean by the word ***quality***. The quality of a software product must be defined in terms that are meaningful to the product's users. What is most important to them and what do they need?

Defects and Quality

A software engineer's job is to deliver quality products for their planned costs, and on their committed schedules.

Software products must also meet the user's functional needs and reliably and consistently do the user's job. While the software functions are most important to the program's users, these functions are not usable unless the software runs. To get the software to run, however, engineers must remove almost all its defects. Thus, while there are many aspects to software quality, the first quality concern must necessarily be with its defects.

The reason defects are so important is that people make a lot of mistakes. In fact, even experienced programmers typically make a mistake for every seven to ten lines of code they develop. While they generally find and correct most of these defects when they compile and test their programs, they often still have a lot of defects in the finished product.

What Are Defects?

Some people mistakenly refer to software defects as bugs. When programs are widely used and are applied in ways that their designers did not anticipate, seemingly trivial mistakes can have unforeseeable consequences. As widely used software systems are enhanced to meet new needs, latent problems can be exposed and a trivial-seeming defect can truly become dangerous. While the vast majority of trivial defects have trivial consequences, a small percentage of seemingly silly mistakes can cause serious problems. Since there is no way to know which of these simple mistakes will have serious consequences, we must treat them all as potentially serious defects, not as trivial-seeming "bugs."

The term defect refers to something that is wrong with a program. It could be a misspelling, a punctuation mistake, or an

incorrect program statement. Defects can be in programs, in designs, or even in the requirements, specifications, or other documentation. Defects can be redundant or extra statements, incorrect statements, or omitted program sections. A defect, in fact, is anything that detracts from the program's ability to completely and effectively meet the user's needs. A defect is thus an objective thing. It is something you can identify, describe, and count.

Simple coding mistakes can produce very destructive or hard-to-find defects. Conversely, many sophisticated design defects are often easy to find. The sophistication of the design mistake and the impact of the resulting defect are thus largely independent. Even trivial implementation errors can cause serious system problems. This is particularly important since the source of most software defects is simple programmer oversights and mistakes. While design issues are always important, initially developed programs typically have few design defects compared to the number of simple oversights, typos, and goofs. To improve program quality, it is thus essential that engineers learn to manage all the defects they inject in their programs.

12.5 PRINCIPLES OF SOFTWARE ENGINEERING

Separation of Concerns

Separation of concerns is recognition of the need for human beings to work within a limited context. As described by G. A. Miller. The human mind is limited to dealing with approximately seven units of data at a time. A unit is something that a person has learned to deal with as a whole - a single abstraction or concept. Although human capacity for forming

abstractions appears to be unlimited, it takes time and repetitive use for an abstraction to become a useful tool; that is, to serve as a unit.

When specifying the behavior of a data structure component, there are often two concerns that need to be dealt with: basic functionality and support for data integrity. A data structure component is often easier to use if these two concerns are divided as much as possible into separate sets of client functions. It is certainly helpful to clients if the client documentation treats the two concerns separately. Further, implementation documentation and algorithm descriptions can profit from separate treatment of basic algorithms and modifications for data integrity and exception handling.

There is another reason for the importance of separation of concerns. Software engineers must deal with complex values in attempting to optimize the quality of a product. From the study of algorithmic complexity, we can learn an important lesson. There are often efficient algorithms for optimizing a single measurable quantity, but problems requiring optimization of a combination of quantities are almost always NP-complete. Although it is not a proven fact, most experts in complexity theory believe that algorithms that run in polynomial time cannot solve NP-complete problems.

In view of this, it makes sense to separate handling of different values. Dealing can do this either with different values at different times in the software development process, or by structuring the design so that responsibility for achieving different values is assigned to different components.

As an example of this, run-time efficiency is one value that frequently conflicts with other software values. For this reason, most software engineers recommend dealing with efficiency as a separate concern. After the software is design to meet other criteria, it's run time can be checked and analyzed to see where the time is being spent. If necessary, the portions of code that are using the greatest part of the runtime can be modified to improve the runtime. This idea is described in depth in Ken Auer and Kent Beck's article "Lazy optimization: patterns for efficient mall talk programming".

Modularity

The principle of modularity is a specialization of the principle of separation of concerns. Following the principle of modularity implies separating software into components according to functionality and responsibility. Parnas wrote one of the earliest papers discussing the considerations involved in modularization. A more recent work, describes a responsibility-driven methodology for modularization in an object-oriented context.

Abstraction

The principle of abstraction is another specialization of the principle of separation of concerns. Following the principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it.

Failure to separate behavior from implementation is a common cause of unnecessary coupling. For example, it is common in

recursive algorithms to introduce extra parameters to make the recursion work. When this is done, the recursion should be called through a non-recursive shell that provides the proper initial values for the extra parameters. Otherwise, the caller must deal with a more complex behavior that requires specifying the extra parameters. If the implementation is later converted to a non-recursive algorithm then the client code will also need to be changed.

Design by contract is an important methodology for dealing with abstraction. Fowler and Scott sketch the basic ideas of design by contract. Meyer gives the most complete treatment of the methodology.

Anticipation of Change

Computer software is an automated solution to a problem. The problem arises in some context, or *domain* that is familiar to the users of the software. The domain defines the types of data that the users need to work with and relationships between the types of data.

Software developers, on the other hand, are familiar with a technology that deals with data in an abstract way. They deal with structures and algorithms without regard for the meaning or importance of the data that is involved. A software developer can think in terms of graphs and graph algorithms without attaching concrete meaning to vertices and edges.

Working out an automated solution to a problem is thus a learning experience for both software developers and their clients. Software developers are learning the domain that the clients work in. They are also learning the values of the client:

what form of data presentation is most useful to the client, what kinds of data are crucial and require special protective measures.

The clients are learning to see the range of possible solutions that software technology can provide. They are also learning to evaluate the possible solutions with regard to their effectiveness in meeting the client's needs.

If the problem to be solved is complex then it is not reasonable to assume that the best solution will be worked out in a short period of time. The clients do, however, want a timely solution. In most cases, they are not willing to wait until the perfect solution is worked out. They want a reasonable solution soon; perfection can come later. To develop a timely solution, software developers need to know the requirements: how the software should behave. The principle of anticipation of change recognizes the complexity of the learning process for both software developers and their clients. Preliminary requirements need to be worked out early, but it should be possible to make changes in the requirements as learning progresses.

Coupling is a major obstacle to change. If two components are strongly coupled then it is likely that changing one will not work without changing the other.

Cohesiveness has a positive effect on ease of change. Cohesive components are easier to reuse when requirements change. If a component has several tasks rolled up into one package, it is likely that it will need to be split up when changes are made.

Generality

The principle of generality is closely related to the principle of anticipation of change. It is important in designing software that is free from unnatural restrictions and limitations. One excellent example of an unnatural restriction or limitation is the use of two digit year numbers, which has led to the "year 2000" problem: software that will garble record keeping at the turn of the century. Although the two-digit limitation appeared reasonable at the time, good software frequently survives beyond its expected lifetime.

For another example where the principle of generality applies, consider a customer who is converting business practices into automated software. They are often trying to satisfy general needs, but they understand and present their needs in terms of their current practices. As they become more familiar with the possibilities of automated solutions, they begin seeing what they need, rather than what they are currently doing to satisfy those needs. This distinction is similar to the distinction in the principle of abstraction, but its effects are felt earlier in the software development process.

Incremental Development

Fowler and Scott give a brief, but thoughtful, description of an incremental software development process. In this process, you build the software in small increments; for example, adding one use case at a time.

An incremental software development process simplifies verification. If you develop software by adding small increments of functionality then, for verification, you only need to deal with

the added portion. If there are any errors detected then they are already partly isolated so they are much easier to correct.

A carefully planned incremental development process can also ease the handling of changes in requirements. To do this, the planning must identify use cases that are most likely to be changed and put them towards the end of the development process.

Consistency

The principle of consistency is recognition of the fact that it is easier to do things in a familiar context. For example, coding style is a consistent manner of laying out code text. This serves two purposes. First, it makes reading the code easier. Second, it allows programmers to automate part of the skills required in code entry, freeing the programmer's mind to deal with more important issues.

At a higher level, consistency involves the development of idioms for dealing with common programming problems. Coplien gives an excellent presentation of the use of idioms for coding in C++.

Consistency serves two purposes in designing graphical user interfaces. First, a consistent look and feel makes it easier for users to learn to use software. Once the basic elements of dealing with an interface are learned, they do not have to be relearned for a different software application. Second, a consistent user interface promotes reuse of the interface components. Graphical user interface systems have a collection of frames, panes, and other view components that support the common look. They also have a collection of

controllers for responding to user input, supporting the common feel. Often, both look and feel are combined, as in pop-up menus and buttons. These components can be used by any program.

Meyer applies the principle of consistency to object-oriented class libraries. As the available libraries grow more and more complex it is essential that they be designed to present a consistent interface to the client. For example, most data collection structures support adding new data items. It is much easier to learn to use the collections if the name add is always used for this kind of operation.

12.6 4GL AND NATURAL LANGUAGES

Computers are designed to process, retrieve and store programmed information. Typically, they can only respond to electronic signals that correspond to binary numbers. Programming languages are designed to translate human language into commands that the computer can understand. There are currently five "generations" or levels of languages available [1]. Machine language, or First-Generation language, consists solely of binary numbers. Machine languages are computer dependent and therefore every type of computer has its own machine language. Assembly language, or Second-Generation language, which came into use in the mid-1950s, uses a shorthand notation of letters and numbers to communicate with the computer in place of binary groupings [2]. Assemblers are then utilized to translate the assembly language into machine language. Assembly language is still computer-dependent and therefore results in a unique assembly language for every type of machine. Higher Order

language (HOL), or Third-Generation language (3GL), which came into use in the 1960s, consists of statements which more closely resemble the spoken language, and therefore are easier to read and write since they require fewer statements per function. Special programs, or compilers, must then translate the HOL statements to assembly or machine language. Expanding upon this, Fourth-Generation languages (4GLs), or Very High Level Languages (VHLL), also use instructions, which resemble the spoken language, but they allow the programmers to define "what" they want the computer to do without necessarily telling the computer "how" to do it. Typically, the compilers, or interpreters, for 4GLs are not as efficient as HOL compilers in using available memory and processing speed. Finally, the highest level of programming, Fifth-Generation languages (5GLs), would involve a computer, which responds directly to spoken or written instructions, or English language commands. There exist only a few 5GLs or "natural languages", and they are typically used in artificial intelligence applications.

Eighty percent of weapon system and AIS applications are written in 3GLs. However, 3GLs require a special program, or compiler, to translate the HOL statements in to assembly or machine instructions. Compilers are not as efficient in terms of memory utilization or processing speed, so a tradeoff between performance and ease is often required when employing 3GLs. Additionally, 3GLs require a vast amount of code to provide the same functionality and are time consuming to debug. Hence, the necessity for the creation of 4GLs, which capitalize on advanced techniques of programming while simplifying the

man-machine interfaces [3]. Currently, there are many 4GLs available, such as Oracle, VisiCalc, FOCUS, RAMIS II, and DBase IV, and new ones are still appearing daily. The languages tend to fall into four functional areas: Query and Report Generators, Graphic Languages, Database management tools and Spreadsheets. As such, they typically have a very limited range of application. This issue paper will address the productivity differences, anticipated and experienced, between 3GLs and 4GLs.

Ease of use:

Because the syntax of 4GLs closely relates to the human-language syntax, it is easier to learn. Additionally, due to the non-procedural nature of many of the languages, the techniques for accomplishing things are also simple, while the results are fast and impressive. Fourth-Generation languages also aim to remove the use of unnecessary acronyms, thereby allowing users to expend their effort on the purpose of the application, vice being bogged down with extraneous requirements.

Limited range of functions:

4GLs are typically designed for a limited set of functions or specific applications. This is because the product then becomes easier to use than a full programming language. For example, Lotus 1-2-3 gained a large number of users quickly because it made it easy to manipulate spreadsheet data.

Restrictions of Options:

Higher level languages often restrict the options available to users of lower level languages, such as the

capability to modify themselves at execution time. To compensate for this, 4GLs permit automatic verification before testing.

Default Options:

A user of a 4GL is not required to specify everything. Instead, a compiler or interpreter is capable of making intelligent assumptions about what it thinks the user needs. Therefore, while 4GLs often require that many parameters be specified, they also provide a default option if the user does not make a selection. This saves time and debugging efforts.

Monologue and Dialogue:

With 4GLs, a dialogue occurs between the user and the computer. This allows for more opportunities to catch errors as they are being made. The software may ask the user questions, signal errors, and flag inconsistencies as the application is being created.

Summary

The objectives of 4GLs are: 1) to speed up the application-building process, 2) to make applications easy and quick to change, hence reducing maintenance cost, 3) to minimize debugging problems, 4) to be able to generate "bug-free" code from high expressions of requirements and 5) to make the language easy to use, so that the end users could solve their own problems [3]. Fourth-Generation languages require far fewer lines of code than would a 3GL, and they also employ a wide variety of other tools such as screen interaction, filling in forms or panels, and computer aided graphics. The goal is to allow the programmer to tell the computers "what-to-

do" and not have to worry about telling the computers "how-to-do-it" (i.e. non-procedural). There are currently no standards for 4GLs, primarily because new ideas in language syntax, dialogue and semantics are appearing daily. Some 4GLs have already disappeared, and many of the best languages now come from relatively small, new vendors. As such, their application to Mission Critical Computer Resources (MCCR) systems, where obsolescence and supportability are key issues, is limited. Due to their limited range of application and their slow processing speeds, they will tend to be more prevalently utilized in Management Information Systems (MIS) developments.

Fourth Generation Languages

1st Generation -- Machine Language

2nd Generation -- Assembly Languages

3rd Generation -- High-Level Languages

4th Generation -- Non-Procedural Languages

5th Generation -- Knowledge-based Natural Language

Where do Object-Oriented Languages fit

In the database environment these are used for creation of database applications To speed up the application building process To make applications easy and quick to change. To minimize debugging problems. To generate bug-free code from high-level expressions of requirement To make languages user-friendly so that "end-users" can solve their own problems and put computers to work.

Basic Principles of 4GLs

- The Principle of Minimum Work
- The Principle of Minimum Skill
- The Principle of avoiding alien syntax and mnemonics
- The Principle of Minimum Time
- The Principle of Minimum errors
- The Principle of Minimum Maintenance
- The Principle of Maximum Results

5GLs -- Natural Language

Advantages of using NL

- It encourages untrained users to start
- It encourages upper-management use of computers
- It reduces the time taken learning complex syntax
- It lessens the frustration, bewilderment and anger caused by BAD COMMAND responses
- It is likely to extend greatly the usage of computers

Disadvantages of using NL

- It lacks precision
- It is not good for expressing precise and complex logic
- It is not good for expressing neat structures
- It encourages semantic overshoot
- It takes substantial time to key in sentences
- Ambiguities are possible
- Substantial processing is needed

Appropriate response to the

Disadvantage

- It should be combined with other dialogue constructs that aid in the representation of precise logic and structures
- Sentences and words can be abbreviated
- Speech input as well as typed input will be used
- The computer should detect and resolve ambiguities
- The processing should be on PC workstations. Processing is dropping rapidly in cost. Computer-aided

software engineering

12.7 CASE TOOLS

Computer-aided software engineering (**CASE**) is the use of software tools to assist in the development and maintenance of software. Tools used to assist in this way are known as **CASE Tools**. A set of these tools are referred to as ICASE.

All aspects of the software development lifecycle can be supported by software tools, and so the use of tools from across the spectrum can, arguably, be described as CASE; from project management software through tools for business and functional analysis, system design, code storage, compilers, translation tools, test software, and so on.

However, it is the tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, that are most frequently thought of as CASE tools. Such tools arose out of developments such

as Jackson Structured Programming and the software modelling techniques promoted by researchers such as Ed Yourdon, Chris Gane and Trish Sarson (see structured programming, SSADM). In this narrower range, CASE applied, for instance, to a database software product, might normally involve:

- Modelling business / real world processes and data flow
- Development of data models in the form of entity-relationship diagrams
- Development of process and function descriptions
- Production of database creation SQL and stored procedures
- Some typical CASE tools are:
- Code generation tools
- UML editors and the like
- Refactoring tools
- Configuration management tools including revision control

CASE tools do not only output code. They also generate other output typical of various systems analysis and design methodologies such as SSADM. E.g.

- database schema
- data flow diagrams
- entity relationship diagrams
- program specifications
- user documentation

Sometimes **CASE** tools are separated in two groups:

Upper **CASE**: Tools for the analyse and design phase of the software development lifecycle (diagraming tools, report and form generators, analysis tools)

Lower **CASE**: Tools to support implementation, testing, configuration management

List of sample CASE tools

ArgoUML

BlueInk

CASEWise

DBDesigner 4 is a visual database design system that integrates database design, modeling, creation and maintenance into a single environment DMS Software Reengineering Toolkit Eclipse with plug-in.

LEARNING ACTIVITIES

Fill in the Blanks:

1. **............**is the use of software tools to assist in the development and maintenance of software.
2. A ......is an abstract representation of a system (real or imaginary) that enables us to answer questions about the system.
3. ....components are easier to reuse when requirements change.

LET US SUM UP

Software engineering as the engineering approach to develop software. From this point of view, we could say that the discipline of software engineering discusses systematic and cost-effective software development approaches which have

come about from past innovations and lessons learnt from mistakes. Software engineering techniques are essential for the development of large software products where a group of engineers are useful even when developing small programs.

ANSWER TO LEARNING ACTIVITIES

Fill in the Blanks:

1. Computer-aided software engineering (CASE)
2. Model
3. Cohesive

MODEL QUESTIONS

1. What is Software Engineering?
2. What's a CASE Tool?
3. What's a 'function point'?
4. What's the 'spiral model'?
5. Where can I find a public-domain tool to compute metrics?
6. What metrics are there for object-oriented systems?
7. How do I write good C style?
8. What is 'Hungarian Notation'?
9. Are lines-of-code (LOC) a useful productivity measure?
10. Should software professionals be licensed /certified?
11. How do I get in touch with the SEI?
12. What is the SEI maturity model?
13. Where can I get information on API?
14. What's a 'bug'?
15. Where can I get copies of standards??
16. What is 'clean room'?
17. What is the Personal Software Process?

REFERENCES

Pressman - Software Engineering