Pune Vidyarthi Griha's

COLLEGE OF ENGINEERING, NASIK

COMPUTER ENGINEERING DEPT.

LAB MANUAL

SYSTEM PROGRAMMING & OPERATING SYSTEM(310257)

LABORATORY

Prepared by

PROF. ANAND GHARU

2017 - 18

PUNE VIDYARTHI GRIHA'S

COLLEGE OF ENGINEERING,NASHIK.

INDEX Batch : -

Sr. No	Title	Page No	Date of Conduction	Date of Submission	Signre of Staff
	GROUP - A				
1	Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature. Implementation should consist of a few instructions from each category and few assembler directives				
2	Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment.				
3	Design suitable data structures and implement pass-I of a two-pass macro-processor using OOP features in Java				
4	Write a Java program for pass-II of a two-pass macro-processor. The output of assignment-3 (MNT, MDT and file without any macro definitions) should be input for this assignment				
	GROUP - B				
5	Write a program to create Dynamic Link Library for any mathematical operation and write an application program to test it. (Java Native Interface / Use VB or VC++).				
6	Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of 'Java' program.				
7	Write a program using Lex specifications to implement lexical analysis phase of compiler to count no. of words, lines and characters of given input file.				
8	Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java.				
9	Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file				

PUNE VIDYARTHI GRIHA'S

COLLEGE OF ENGINEERING,NASHIK.

INDEX Batch : -

GROUP - C					
10	Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive)				
11	Write a Java program to implement Banker's Algorithm				
12	Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming).				
13	Study assignment on process scheduling algorithms in And and Tizen.				
GROUP - D					
14	Write a Java Program (using OOP features) to implement paging simulation using 1. Least Recently Used (LRU) 2. Optimal algorithm				

`

Certified that Mr/Miss _____ of

class _____ Sem ____ Roll no. ____ has completed the term work satisfactorily in the

subject _____of the Department _____of

PVG's College of Engineering Nashik. During academic year _____ .

Staff Member	**Head of Dept.**	**Principal**
Prof. Gharu A. N.	_____	_____

GROUP - A

GROUP - A

GROUP - A

EXPERIMENT NO : 01

1. Title:

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature. Implementation should consist of a few instructions from each category and few assembler directives.

2. Objectives :

- To understand Data structure of Pass-1 assembler
- To understand Pass-1 assembler concept
- To understand Advanced Assembler Directives

3. Problem Statement :

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature.

4. Outcomes:

After completion of this assignment students will be able to:

- Implemented Pass – 1 assebmler
- Implemented Symbol table, Literal table & Pool table.
- Understood concept Advanced Assembler Directive.

5. Software Requirements:

Latest jdk., Eclipse

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Introduction :-

There are two main classes of programming languages: *high level* (e.g., C, Pascal) and *low level*. *Assembly Language* is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.

An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.

Figure 1. Executable program generation from an assembly source code

Advantages of coding in assembly language are:

- Provides more control over handling particular hardware components
- May generate smaller, more compact executable modules
- Often results in faster execution
-

Disadvantages:

- Not portable
- More complex
- Requires understanding of hardware details (interfaces)

Pass – 1 Assembler:

An assembler does the following:

1. Generate machine instructions
   - evaluate the mnemonics to produce their machine code
   - evaluate the symbols, literals, addresses to produce their equivalent machine addresses
   - convert the data constants into their machine representations
2. Process pseudo operations

Pass – 2 Assembler:

A two-pass assembler performs two sequential scans over the source code:

Pass 1: symbols and literals are defined
Pass 2: object program is generated

Parsing: moving in program lines to pull out op-codes and operands

Data Structures:

- **Location counter (LC):** points to the next location where the code will be placed
- **Op-code translation table**: contains symbolic instructions, their lengths and their op-codes (or subroutine to use for translation)
- **Symbol table (ST):** contains labels and their values
- **String storage buffer (SSB):** contains ASCII characters for the strings
- **Forward references table (FRT):** contains pointer to the string in SSB and offset where its value will be inserted in the object code

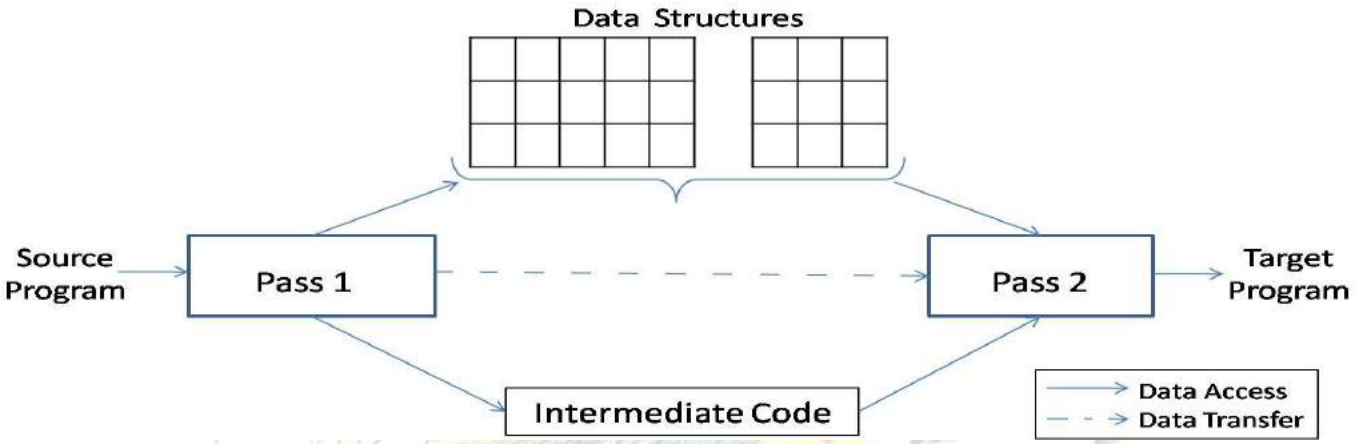


Figure 2. ***A simple two pass assembler*****.**

Elements of Assembly Language :

An assembly language programming provides three basic features which simplify programming when compared to machine language.

1. **Mnemonic Operation Codes :**

Mnemonic operation code / Mnemonic Opcodes for machine instruction eliminates the need to memorize numeric operation codes. It enables assembler to provide helpful error diagnostics. Such as indication of misspelt operation codes.

2. **Symbolic Operands :**

Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements. The assembeler performes memory bindinding to these names; the programmer need not know any details of the memory bindings performed by the assembler.

3. **Data declarations :**

Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constants into their internal machine representation, for example -5 into (11111010)2 or 10.5 into (41A80000)16

Statement format :

An assembly language statement has the following format :

[Label] <Opcode> <operand Spec> [, operand Spec> ..]

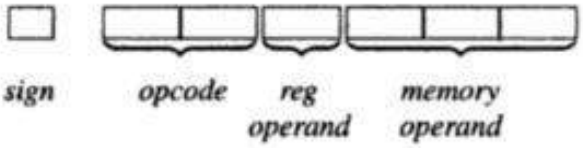
Where the notation [..] indicates that the enclosed specification is optional.

Label associated as a symbolic name with the memory word(s) generated for the statement

Mnemonic Operation Codes :

Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	*First operand is modified* *Condition code is set*
02	SUB	
03	MULT	
04	MOVER	Register ← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	*First operand is not used*
10	PRINT	

Instruction Format :



Sign is not a part of Instruction

An Assembly and equivalent machine language program :(solve it properly)

```
          START   101
          READ    N                 101)  + 09 0 113
          MOVER   BREG, ONE         102)  + 04 2 115
          MOVEM   BREG, TERM        103)  + 05 2 116
AGAIN     MULT    BREG, TERM        104)  + 03 2 116
          MOVER   CREG, TERM        105)  + 04 3 116
          ADD     CREG, ONE         106)  + 01 3 115
          MOVEM   CREG, TERM        107)  + 05 3 116
          COMP    CREG, N           108)  + 06 3 113
          PRINT   RESULT            111)  + 10 0 114
          STOP                      112)  + 00 0 000
N         DS      1                 113)
RESULT    DS      1                 114)
ONE       DC      '1'               115)  + 00 0 001
TERM      DS      1                 116)
          END
```

Note : you can also take other example with solution

Assembly Language Statements :

Three Kinds of Statements

1. Imperative Statements
2. Declaration Statements
3. Assembler Directives

a) **Imperative Statements :** It indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.

b) **Declaration Statements :** Two types of declaration statements is as follows

[Label] DS <constant>

[Label] DC '<Value>'

The DS (Declare Storage) statement reserves areas of memory and associates names with them.

Eg)**A DS 1**

B DS 150

First statement reserves a memory of 1 word and associates the name of the memory as A.
Second statement reserves a memory of 150 word and associates the name of the memory as B.

The DC (Declare Constant) Statement constructs memory word containing constants.

Eg) **ONE DC '1'**

Associates the name ONE with a memory word containing the value '1' . The programmer can declare constants in decimal,binary, hexadecimal forms etc., These values are not protected by the assembler. In the above assembly language program the value of ONE Can be changed by executing an instruction MOVEM BREG,ONE

c. Assembler Directives :

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some Assembler directives are described in the following

START <Constant>

Indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <Constant>

END [<operand spec>]

It Indicates the end of the source program

Pass Structure of Assembler :

One complete scan of the source program is known as a pass of a Language Processor.

Two types 1) Single Pass Assembler 2) Two Pass Assembler.

Single Pass Assembler :

First type to be developed Most Primitive Source code is processed only once.

The operand field of an instruction containing forward reference is left blank intially

Eg) MOVER BREG,ONE

Can be only partially synthesized since ONE is a forward reference

During the scan of the source program, all the symbols will be stored in a table called **SYMBOL TABLE.** Symbol table consists of two important fields, they are symbol name and address.

All the statements describing forward references will be stored in a table called Table of Incompleted Instructions (TII)

TII (Table of Incomplete instructions)

Instruction Address	Symbol
101	ONE

By the time the END statement is processed the symbol table would contain the address of all symbols defined in the source program.

Two Pass Assembler :

Can handle forward reference problem easily.

First Phase : (Analysis)

- Symbols are entered in the table called Symbol table
- Mnemonics and the corresponding opcodes are stored in a table called Mnemonic table
- LC Processing

Second Phase : (Synthesis)

- Synthesis the target form using the address information found in Symbol table.
- First pass constructs an Intermediated Representation (IR) of the source program for use by the second pass.

Data Structure used during Synthesis Phase :

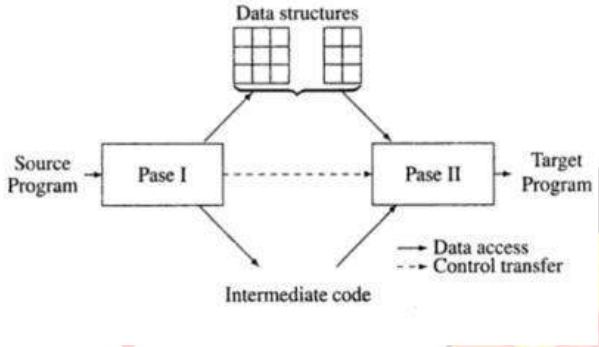
1. Symbol table
2. Mnemonics table

symbol	address
AGAIN	104
N	113

mnem-onic	op code	length
ADD	01	1
SUB	02	1

Processed form of the source program called Intermediate Code (IC)

mnem-onic	op code	length



ADVANCED ASSEMBLER DIRECTIVES

1. ORIGIN
2. EQU
3. LTROG

ORIGIN :

Syntax : ORIGIN < address spec>

< address spec>can be an <operand spec> or constant

Indicates that Location counter should be set to the address given by < address spec>

This statement is useful when the target program does not consist of consecutive memory words.

Eg) ORIGIN Loop + 2

EQU : *Syntax*

<symbol> EQU <address spec>

<address spec>operand spec (or) constant

``

Simply associates the name symbol with address specification No

Location counter processing is implied

Eg) Back EQU Loop

LTORG : (Literal Origin)

Where should the assembler place literals ?

It should be placed such that the control never reaches it during the execution of a program.

By default, the assembler places the literals after the END statement.

LTROG statement permits a programmer to specify where literals should be placed.

Note :(you can also write your own theory for this practical)

Solve the below example.for Pass-1 Assembler.

```
          START   101
          READ    X
          READ    Y
          MOVER   AREG, X
          ADD     AREG, Y
          MOVEM   AREG, RESULT
          PRINT   RESULT
          STOP
X         DS  1
Y         DS  1
RESULT    DS  1
          END
```

Fig. 1.7.4 : A sample program for finding X + Y

Algorithms :

Flowchart :

8. Conclusion :

Thus , I have implemented pass-1 assembler with symbol table, literal table and pool table.

References :

https://en.wikipedia.org/wiki/Assembly_language
http://freestudy9.com/one-pass-two-pass-assembler/
https://en.wikipedia.org/wiki/One-pass_compiler

http://enggedu.com/tamilnadu/university_questions/question_answer/be_am_2008/3th_sem/cse/CS1203/part_b/12_a.html

NOTE : don't write references.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is assembler.
2. Define system programming and its component.
3. State pass-1 and pass-2 assembler
4. What is forward reference & reverse references.
5. What is symbol table, literal table and pool table.
6. How pass-1 assembler works.
7. What is use of LTORG assembler directive.

GROUP - A

EXPERIMENT NO : 02

1. Title:

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment..

2. Objectives :

- To understand Data structure of Pass-1 & Pass-2 assembler
- To understand Pass-1 & Pass-2 assembler concept
- To understand Advanced Assembler Directives

3. Problem Statement :

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment..

4. Outcomes:

After completion of this assignment students will be able to:

- Implemented Pass – 2 assebmler
- Implemented machine code from intermediate code.
- Understood concept Pass-2 Assembler.

5. Software Requirements:

Latest jdk., Eclipse

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Introduction :-

There are two main classes of programming languages: *high level* (e.g., C, Pascal) and *low level*. *Assembly Language* is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.

An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.

Figure 1. Executable program generation from an assembly source code

Advantages of coding in assembly language are:

- Provides more control over handling particular hardware components
- May generate smaller, more compact executable modules
- Often results in faster execution
-

Disadvantages:

- Not portable
- More complex
- Requires understanding of hardware details (interfaces)

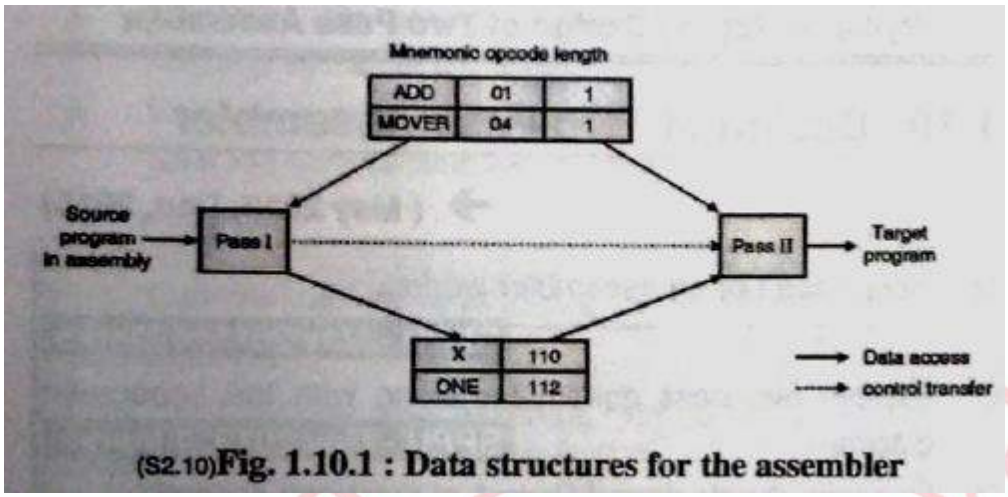
DESIGN OF TWO PASS ASSEMBLER:

Pass I : (Analysis of Source Program)

1) Separate the symbol, mnemonic opcode and operand fields
2) Build the symbol table.
3) Perform LC processing.
4) Construct intermediate representation

PASS 2:-

Processes the intermediate representation (IR) to synthesize the target program.



(S2.10)**Fig. 1.10.1 : Data structures for the assembler**

Table 1.10.1 : An enhanced machine opcode table (MOT)

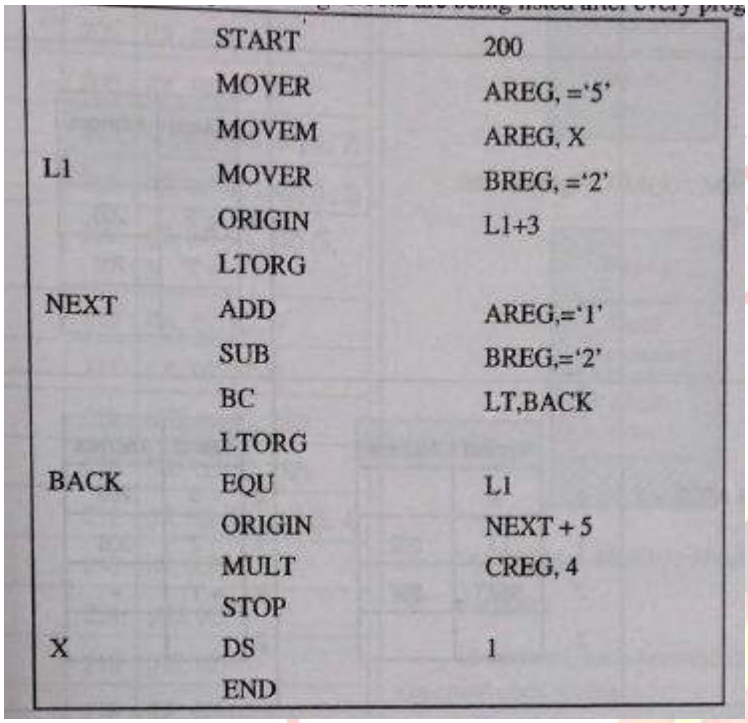
	Mnemonic opcode	Class	Opcode	Length
0	STOP	IS	00	1
1	ADD	IS	01	1
2	SUB	IS	02	1
3	MULT	IS	03	1
4	MOVER	IS	04	1
5	MOVEM	IS	05	1
6	COMP	IS	06	1
7	BC	IS	07	1
8	DIV	IS	08	1
9	READ	IS	09	1
10	PRINT	IS	10	1
11	START	AD	01	–
12	END	AD	02	–
13	ORIGIN	AD	03	–
14	EQU	AD	04	–
15	LTORG	AD	05	–
16	DS	DL	01	–
17	DC	DL	02	1
18	AREG	RG	01	–
19	BREG	RG	02	–
20	CREG	RG	03	–
21		CC	01	–

IS : Imperative Statement
AD : Assembler Directive
DL : Declaration Statement
RG : Register
CC : Comparison Condition

	Mnemonic opcode	Class	Opcode	Length
22	LT	CC	02	–
23	GT	CC	03	–
24	LE	CC	04	–
25	GE	CC	05	–
26	NE	CC	06	–
27	ANY	CC	07	–

Contents of various tables f...

Note : solve below example in details



	START	200
	MOVER	AREG, ='5'
	MOVEM	AREG, X
L1	MOVER	BREG, ='2'
	ORIGIN	L1+3
	LTORG	
NEXT	ADD	AREG,='1'
	SUB	BREG,='2'
	BC	LT,BACK
	LTORG	
BACK	EQU	L1
	ORIGIN	NEXT + 5
	MULT	CREG, 4
	STOP	
X	DS	1
	END	

Algorithms :

Flowchart : .

Note : you can also write your own theory respect to practical.

8. **Conclusion :**

Thus , I have implemented Pass-2 assembler by taking input as output of assignment-1

References :

NOTE : don't write references.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

``

1. What is Pass-2 assembler
2. How does Pass-2 works.,
3. What is intermediate code.
4. What is assembler directives.
5. What are the types of statement in assebler.
6. How to generate intermediate code.

GROUP - A

EXPERIMENT NO : 03

1. **Title:**

   Design suitable data structures and implement pass-I of a two-pass macro-processor using OOP features in Java

2. **Objectives :**

   - To understand Data structure of Pass-1 macroprocessor
   - To understand Pass-1 macroprocessor concept
   - To understand macro facility.

3. **Problem Statement :**

   Design suitable data structures and implement pass-I of a two-pass macro-processor using OOP features in Java.

4. **Outcomes:**

After completion of this assignment students will be able to:

- Implemented Pass – 1 macroprocessor
- Implemented MNT, MDT table.
- Understood concept Pass-1 macroprocessor.

5. **Software Requirements:**

   Latest jdk., Eclipse

6. **Hardware Requirement:**

   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. **Theory Concepts:**

   Macroprocessor

Macro pre-processor takes a source program containing macro definitions and macro calls and translates into an assembly language program without any macro definitions or calls. This program can now be handed over to a conventional assembler to obtain the target language (as shown in Fig. 2.5.1).

(S3.6)**Fig. 2.5.1 : A scheme for a macro pre-processor**

Macro :

Macro allows a sequence of source language code to be defined once & then referred to by name each time it is referred. Each time this name occurs in a program , the sequence of codes is substituted at that point.

Macro has following parts:-

(1) Name of macro
(2) Parameters in macro
(3) Macro Definition

Parameters are optional.

How To Define a Macro :-

Macro can be formatted in following order :-

Start of definition	MACRO
macro name	mymacro
macro body	ADD AREG, X ADD BREG, X
End of macro definition	MEND

'MACRO' pseudo-op is the first line of definition & identifies the following line as macro instruction name.

Following the name line is sequence of instructions being abbreviated the instructions comprising the 'MACRO' instruction.

The definition is terminated by a line with MEND pseudo-op.

Example of Macro:-

(1) Macro without parameters

```
MACRO
        mymacro
                ADD AREG,X
                ADD BREG,X
MEND
```

(2) Macro with parameters

```
MACRO
        addmacro  &A
                ADD AREG,&A
                ADD BREG,&A
MEND
```

The macro parameters (Formal Parameters) are initialized with '&' . used as it is in operation..Formal Parameters are those which are in definition of macro.

Whereas while calling macros we use Actual Parameters.

How To Call a Macro?

A macro is called by writing macro name with actual parameters in an Assembly Program.

Macro call leads to Macro Expansion.

Syntax: <macro-name> [<list of parameters>]

Example:- for above definitions of macro…

(1) mymcro

(2) addmacro X

Macro Expansion:-

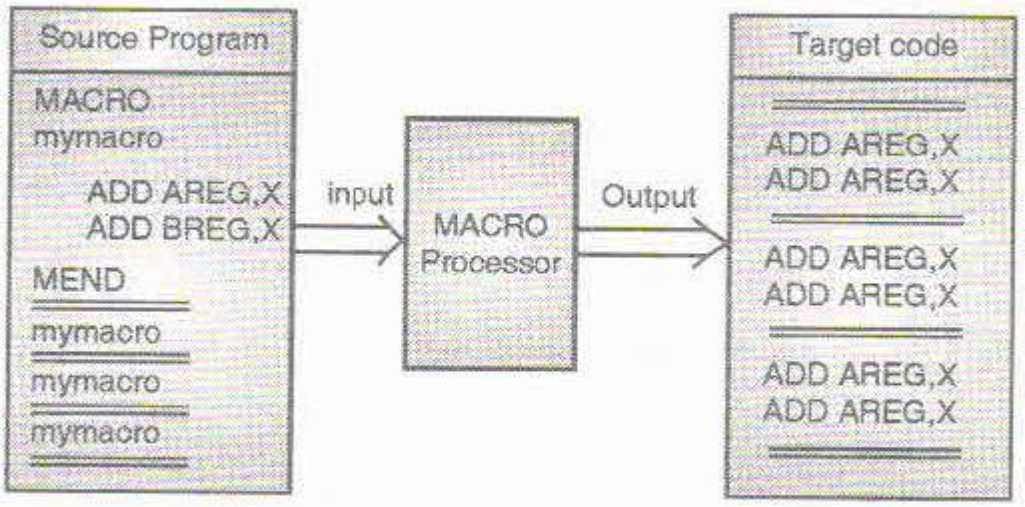
Each Call to macro is replaced by its body.

During Replacement, actual parameter is used in place of formal parameter.

- During Macro expansion, each statement forming the body of macro is picked up one bye one sequentially.
- Each Statement inside the macro may have:

  (1) An ordinary string, which is copied as it is during expansion.

  (2) The name of a formal parameter which is proceeded by character '&'.
- During macro expansion an ordinary string is retained without any modification. Formal Parameters(Strings starting with &) is replaced by the actual parameter value.



(S3.2)**Fig. 2.1.2 : Macro expansion**

Macro with Keyword Parameters :-

These are the methods to call macros with formal parameters. These formal parameters are of two types

(1) Positional Parameters :

Initiated with '&'.

Ex:- mymacro &X

(2) Keyword Parameters :

Initiated with '&' . but has some default value.

During a call to macro , a keyword parameter is specified by its name.

Ex:- mymacro &X=A

Nested Macro Calls :-

Nested Macro Calls are just like nested function calls in our normal calls. Only the transfer of control from one macro to other Is done.

Consider this example :-

```
MACRO
  Innermacro
      ADD AREG,X
MEND

MACRO
  outermacro
      innermacro
```

``

```
        ADD AREG,Y
    MEND

    outermacro
```

In this example, firstly the MACRO outermacro gets executed & then innermacro. So Output will be Adding X & Y values in AREG register.

Algorithm:

Scan all MACRO definition one by one.

(a) Enter its name in macro name table (MNT).

(b) Store the entire macro definition in the macro definition table (MDT).

(c) Add the information in the MNT indicates where definition of macro can be found in MDT.

(d) Prepare argument list array (ALA).

Data Structures of Two Pass Macros:

1] Macro Name Table Pointer (MNTP) :

2] Macro Definition Table Pointer (MDTP) :

3] Macro Name Table :

- macro number(i.e pointer referenced from MNTP)
- Name of macros
- MDTP (i.e points to start position to MDT)

4] Macro Definition Table :

- Location Counter(where MDTP points to start position of macro)
- Opcode
- Rest (i.e it will contain the other part than opcodes used in macro).

5] Argument List Array :

- Index given to parameter
- Name of parameter

Example for pass I of macro processor : -

Assembly Code Segment:-

```
MACRO
INCR &X,&Y,&REG=AREG
MOVER &REG,&Y
ADD &REG,&Y
MOVEM &REG,&X
MEND
MACRO
DECR &A,&B,&REG=BREG
MOVER &REG,&A
ADD &REG,&B
MOVEM &REG,&A
MEND
START
READ N1
READ N2
INCR N1,N2,REG=CREG
DECR N1,N2
STOP
N1 DS 1
N2 DS 2
END
```

Output of Pass-I of Macro Assmebler :-

MNT

	NAME	ADDRESS IN MDT
0	INCR	0
1	DECR	5

MDT

0	INCR &X,&Y,&REG=AREG
1	MOVER #3,#1
2	ADD #3,#2
3	MOVEM #3,#1
4	MEND
5	DECR &A,&B,&REG=BREG
6	MOVER #3,#1
7	ADD #3,#2
8	MOVEM #3,#1
9	MEND

#1 – First Parameter

#2 – Second Parameter

#3 – Thirs Parameter

``

.

Note : you can take other example also.

Algorithms :

Flowchart : .

Note : you can also write your own theory respect to practical.

8. **Conclusion :**

Thus , I have implemented Pass-1 macroprocessor by producing MNT and MDT table.

References :

NOTE : don't write references.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is Pass-2 macroprocessor
2. Compare assembler and macroprocessor
3. What is macro, macro expansion.
4. What is MNT and MDT.
5. How pass-1 macroprocessor works?
6. How to expand macro.
7. What nested macro.

GROUP - A

EXPERIMENT NO : 04

1. Title:

Write a Java program for pass-II of a two-pass macro-processor. The output of assignment-3 (MNT, MDT and file without any macro definitions) should be input for this assignment

2. Objectives :

- To understand Data structure Pass-2 macroprocessor
- To understand Pass-1 & Pass-2 macroprocessor concept
- To understand Advanced macro facility

3. Problem Statement :

Write a Java program for pass-II of a two-pass macro-processor. The output of assignment-3 (MNT, MDT and file without any macro definitions) should be input for this assignment

4. Outcomes:

After completion of this assignment students will be able to:

- Implemented Pass – 2 macroprocessor
- Implemented machine code from MDT and MNT table.
- Understood concept Pass-2 macroprocessor.

5. Software Requirements:

Latest jdk., Eclipse

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Advanced Macro Facilities:

(1) AIF
(2) AGO
(3) Sequential Symbol
(4) Expansion time variable

(1) AIF

Use the AIF instruction to branch according to the results of a condition test. You can thus alter the sequence in which source program statements or macro definition statements are processed by the assembler.

The AIF instruction also provides loop control for conditional assembly processing, which lets you control the sequence of statements to be generated.

It also lets you check for error conditions and thereby to branch to the appropriate MNOTE instruction to issue an error message.

(2) AGO

The AGO instruction branches unconditionally. You can thus alter the sequence in which your assembler language statements are processed. This provides you with final exits from conditional assembly loops.

3) Sequence Symbols

You can use a sequence symbol in the name field of a statement to branch to that statement during conditional assembly processing, thus altering the sequence in which the assembler processes your conditional assembly and macro instructions. You can select the model statements from which the assembler generates assembler language statements for processing at assembly time.

A sequence symbol consists of a period (.) followed by an alphabetic character, followed by 0 to 61 alphanumeric characters.

Examples:

```
.BRANCHING_LABEL#1
.A
```

Sequence symbols can be specified in the name field of assembler language statements and model statements; however, sequence symbols must not be used as name entries in the following assembler instructions:

ALIAS	EQU	OPSYN	SETC
AREAD	ICTL	SETA	SETAF
CATTR	LOCTR	SETB	SETCF
DXD			

Also, sequence symbols cannot be used as name entries in macro prototype instructions, or in any instruction that already contains an ordinary or a variable symbol in the name field.

Sequence symbols can be specified in the operand field of an AIF or AGO instruction to branch to a statement with the same sequence symbol as a label

4) **Expansion Time Variables**

Note :- write theory from book or notes.

- Data Structures of Two Pass Macros:

1] Input Source Program for pass- II . It is produced by pass – I .

2] Macro Definition Table : (MDT) produced by pass - I

- Location Counter(where MDTP points to start position of macro)
- Opcode
- Rest (i.e it will contain the other part than opcodes used in macro).

3] Macro Name Table : (MNT) produced by pass - I

- macro number(i.e pointer referenced from MNTP)
- Name of macros
- MDTP (i.e points to start position to MDT)
-

4] MNTP (macro name table pointer) gives number of entries in MNT.

5] Argument List Array :

- Index given to parameter
- Name of parameter

Which gives association between integer indices & actual parameters.

6] Source Program with macro-calls expanded. This is output of pass- II.

7] MDTP (macro definition table pointer) gives the address of macro definition in macro definition table.

Algorithm:

Take Input from Pass - I

Examine all statements in the assembly source program to detect macro calls. For Each Macro call:

(a) Locate the macro name in MNT.

(b) Establish correspondence between formal parameters & actual parameters.

(c) Obtain information from MNT regarding position of the macro definition in MDT.

(d) Expand the macro call by picking up model statements from MDT.

Example for pass I of macro processor: -

Assembly Code Segment:-

```
MACRO
INCR &X,&Y,&REG=AREG
MOVER &REG,&Y
ADD &REG,&Y
MOVEM &REG,&X
MEND
MACRO
DECR &A,&B,&REG=BREG
MOVER &REG,&A
ADD &REG,&B
MOVEM &REG,&A
MEND
START 200
READ N1
READ N2
INCR N1,N2,REG=CREG
DECR N1,N2
STOP
N1 DS 1
N2 DS 2
END
```

Output of Pass-I of Macro Macro Processor:-

MNT

	NAME	ADDRESS IN MDT
0	INCR	0
1	DECR	5

MDT

0	INCR &X,&Y,&REG=AREG
1	MOVER #3,#1
2	ADD #3,#2
3	MOVEM #3,#1
4	MEND
5	DECR &A,&B,&REG=BREG
6	MOVER #3,#1
7	ADD #3,#2
8	MOVEM #3,#1
9	MEND

#1 – First Parameter

#2 – Second Parameter

#3 – Third Parameter

Pass – II of macro pre-processor will create the argument list array , every time there is call to macro ,& expand macro.

(1) Macro Call:-

INCR &X,&Y,&REG=AREG

N1
N2
AREG

Attach expansion code of this macro in output as following:-

MOVER AREG,N1

ADD AREG,N2

MOVEM AREG,N1

(2) Macro Call:-

INCR &A,&B,&REG=BREG

N1
N2
BREG

Attach expansion code of this macro in output as following:-

MOVER BREG,N1

ADD BREG,N2

MOVEM BREG,N1

Expanded Source file at the end of pass – II :

START 200

READ N1

READ N2

MOVER AREG,N1

ADD AREG,N2 Expansion of INCR N1,N2

MOVEM AREG,N1

```
MOVER BREG,N1
ADD BREG,N2                Expansion of DECR N1,N2
MOVEM BREG,N1

STOP
N1 DS 1
N2 DS 2
END
```

Note : you can take other example also.

Algorithms :

Flowchart : .

Note : you can also write your own theory respect to practical.

8. Conclusion :

Thus , I have implemented Pass-2 macroprocessor by taking input as output of assignment-3 (i.e. MDT and MNT table)

References :

NOTE : don't write references.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is Pass-2 macroprocessor
2. How does Pass-2 macroprocessor works.
3. What is MDT and MNT.
4. What is Argument List Array.
5. Working of Pass-2 macroprocessor
6. Compare Pass-1 and Pass-2 macroprocessor.

GROUP - B

GROUP - B

GROUP - B

EXPERIMENT NO : 05

1. Title:

Write a program to create Dynamic Link Library for any mathematical operation and write an application program to test it. (Java Native Interface / Use VB or VC++).

2. Objectives :

- To understand Dynamic Link Libraries Concepts
- To implement dynamic link library concepts
- To study about Visual Basic

3. Problem Statement :

Write a program to create Dynamic Link Library for Arithmetic Operation in VB.net

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the concept of Dynamic Link Library
- Understand the Programming language of Visual basic

5. Software Requirements:

- Visual Studio 2010

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Dynamic Link Library :

A dynamic link library (DLL) is a collection of small programs that can be loaded when needed by larger programs and used at the same time. The small program lets the larger program communicate with a specific device, such as a printer or scanner. It is often packaged as a DLL program, which is usually referred to as a DLL file. DLL files that support specific device operation are known as device drivers.

A DLL file is often given a ".dll" file name suffix. DLL files are dynamically linked with the program that uses them during program execution rather than being compiled into the main program.

The advantage of DLL files is space is saved in random access memory (RAM) because the files don't get loaded into RAM together with the main program. When a DLL file is needed, it is loaded and run. For example, as long as a user is editing a document in Microsoft Word, the printer DLL file does not need to be loaded into RAM. If the user decides to print the document, the Word application causes the printer DLL file to be loaded and run.

A program is separated into modules when using a DLL. With modularized components, a program can be sold by module, have faster load times and be updated without altering other parts of the program. DLLs help operating systems and programs run faster, use memory efficiently and take up less disk space.

Feature of DLL :

- DLLs are essentially the same as EXEs, the choice of which to produce as part of the linking process is for clarity, since it is possible to export functions and data from either.
- It is not possible to directly execute a DLL, since it requires an EXE for the operating system to load it through an entry point, hence the existence of utilities like RUNDLL.EXE or RUNDLL32.EXE which provide the entry point and minimal framework for DLLs that contain enough functionality to execute without much support.
- DLLs provide a mechanism for shared code and data, allowing a developer of shared code/data to upgrade functionality without requiring applications to be re-linked or re-compiled. From the application development point of view Windows and OS/2 can be thought of as a collection of DLLs that are upgraded, allowing applications for one version of the OS to work in a later one, provided that the OS vendor has ensured that the interfaces and functionality are compatible.
- DLLs execute in the memory space of the calling process and with the same access permissions which means there is little overhead in their use but also that there is no protection for the calling EXE if the DLL has any sort of bug.

Difference between the Application & DLL :

- An application can have multiple instances of itself running in the system simultaneously, whereas a DLL can have only one instance.
- An application can own things such as a stack, global memory, file handles, and a message queue, but a DLL cannot.

Executable file links to DLL :

An executable file links to (or loads) a DLL in one of two ways:

- Implicit linking
- Explicit linking

Implicit linking is sometimes referred to as static load or load-time dynamic linking. Explicit linking is sometimes referred to as dynamic load or run-time dynamic linking.

With implicit linking, the executable using the DLL links to an import library (.lib file) provided by the maker of the DLL. The operating system loads the DLL when the executable using it is loaded. The client executable calls the DLL's exported functions just as if the functions were contained within the executable.

With explicit linking, the executable using the DLL must make function calls to explicitly load and unload the DLL and to access the DLL's exported functions. The client executable must call the exported functions through a function pointer.

An executable can use the same DLL with either linking method. Furthermore, these mechanisms are not mutually exclusive, as one executable can implicitly link to a DLL and another can attach to it explicitly.

Calling DLL function from Visual Basic Application :

For Visual Basic applications (or applications in other languages such as Pascal or Fortran) to call functions in a C/C++ DLL, the functions must be exported using the correct calling convention without any name decoration done by the compiler.

__stdcall creates the correct calling convention for the function (the called function cleans up the stack and parameters are passed from right to left) but decorates the function name differently. So, when **__declspec(dllexport)** is used on an exported function in a DLL, the decorated name is exported.

The __stdcall name decoration prefixes the symbol name with an underscore (_) and appends the symbol with an at sign (@) character followed by the number of bytes in the argument list (the required stack space). As a result, the function when declared as:

```
int __stdcall func (int a, double b)
```

is decorated as:

```
_func@12
```

The C calling convention (__cdecl) decorates the name as _func.
To get the decorated name, use /MAP. Use of **__declspec(dllexport)** does the following:

- If the function is exported with the C calling convention (**_cdecl**), it strips the leading underscore (_) when the name is exported.
- If the function being exported does not use the C calling convention (for example, __stdcall), it exports the decorated name.

Because there is no way to override where the stack cleanup occurs, you must use __stdcall. To undecorate names with __stdcall, you must specify them by using aliases in the EXPORTS section of the .def file. This is shown as follows for the following function declaration:

```
int  __stdcall MyFunc (int a, double b);
void __stdcall InitCode (void);
```

In the .DEF file:

```
EXPORTS
MYFUNC=_MyFunc@12
 INITCODE=_InitCode@0
```

For DLLs to be called by programs written in Visual Basic, the alias technique shown in this topic is needed in the .def file. If the alias is done in the Visual Basic program, use of aliasing in the .def file is not necessary. It can be done in the Visual Basic program by adding an alias clause to the Declare statement.

DLL's Advantages :

- Saves memory and reduces swapping. Many processes can use a single DLL simultaneously, sharing a single copy of the DLL in memory. In contrast, Windows must load a copy of the library code into memory for each application that is built with a static link library.
- Saves disk space. Many applications can share a single copy of the DLL on disk. In contrast, each application built with a static link library has the library code linked into its executable image as a separate copy.
- Upgrades to the DLL are easier. When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- Provides after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was shipped.
- Supports multi language programs. Programs written in different programming languages can call the same DLL function as long as the programs follow the function's calling convention. The programs and the DLL function must be compatible in the following ways: the order in which the function expects its arguments to be pushed onto the stack, whether the function or the application is responsible for cleaning up the stack, and whether any arguments are passed in registers.
- Provides a mechanism to extend the MFC library classes. You can derive classes from the existing MFC classes and place them in an MFC extension DLL for use by MFC applications.
- Eases the creation of international versions. By placing resources in a DLL, it is much easier to create international versions of an application. You can place the strings for each language version of your application in a separate resource DLL and have the different language versions load the appropriate resources.

Disadvantage :

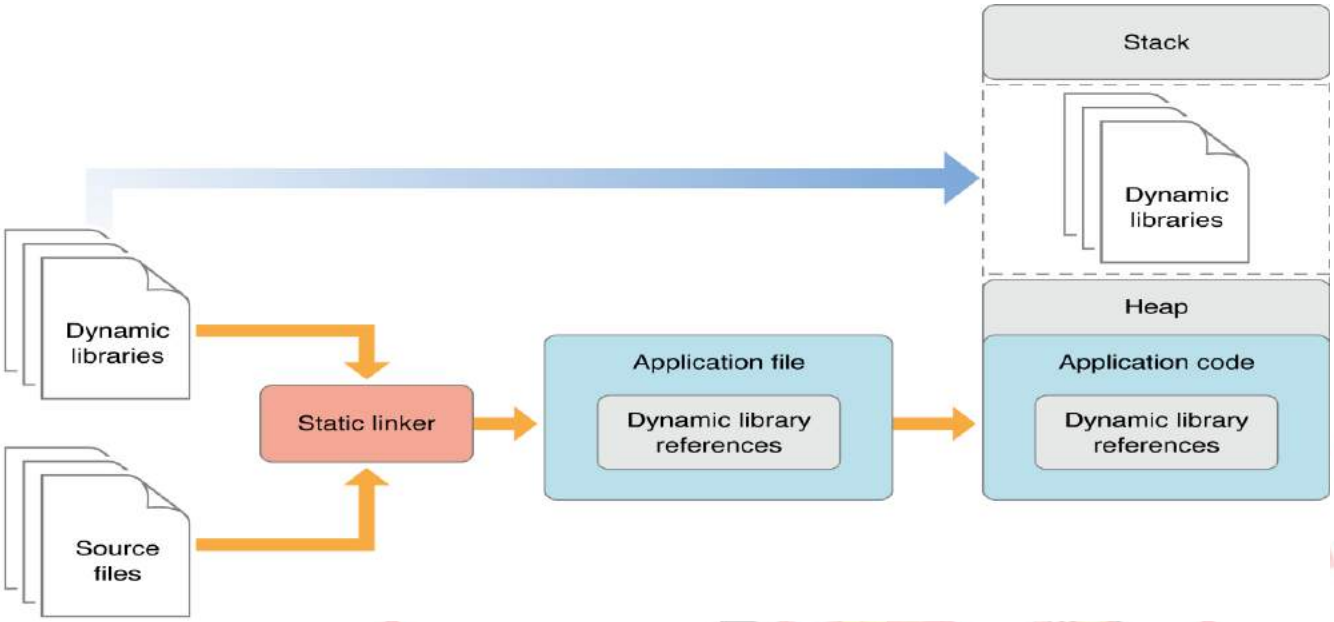
- A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module.

Visual Basic :

Visual Basic is a third-generation event-driven programming language first released by Microsoft in 1991. It evolved from the earlier DOS version called BASIC. **BASIC** means **B**eginners' **A**ll-purpose **S**ymbolic **I**nstruction **C**ode. Since then Microsoft has released many versions of Visual Basic, from Visual Basic 1.0 to the final version Visual Basic 6.0. Visual Basic is a user-friendly programming language designed for beginners, and it enables anyone to develop GUI window applications easily.

In 2002, Microsoft released Visual Basic.NET(VB.NET) to replace Visual Basic 6. Thereafter, Microsoft declared VB6 a legacy programming language in 2008. Fortunately, Microsoft still provides some form of support for VB6. VB.NET is a fully object-oriented programming language implemented in the .NET Framework. It was created to cater for the development of the web as well as mobile applications. However, many developers still favor Visual Basic 6.0 over its successor Visual Basic.NET.

8. Design (architecture) :



9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

Thus, I have studied visual programming and implemented dynamic link library application for arithmetic operation

References :

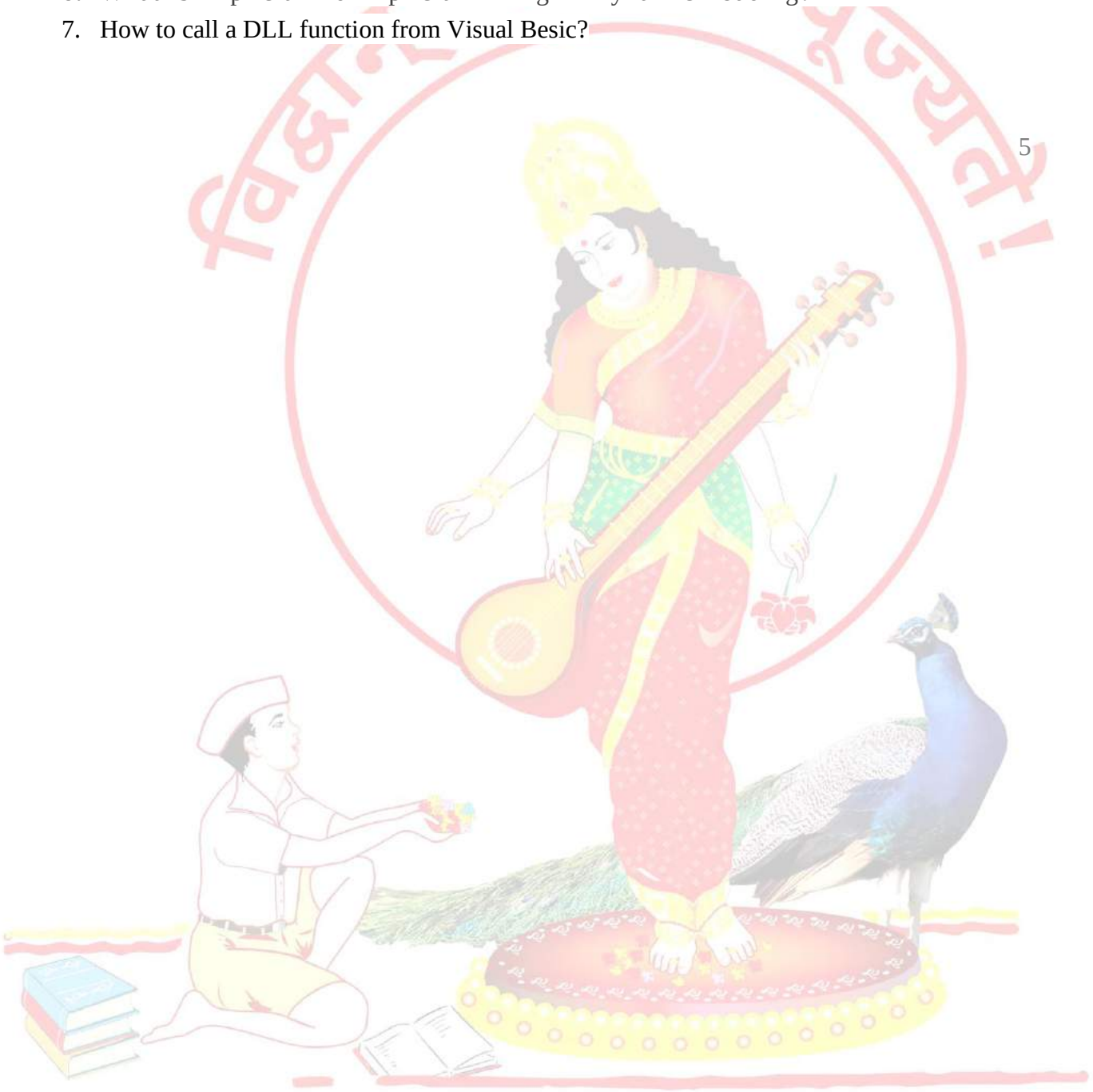
https://en.wikipedia.org/wiki/Dynamic-link_library
https://en.wikipedia.org/wiki/Visual_Basic
https://www.google.co.in/search?q=dynamic+link+library+architecture&dcr=0&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjqubTAuJvZAhWHQ48KHRZbD7sQ_AUICigB&biw=1366&bih=651#imgrc=LU8YqljE8-afxM
https://msdn.microsoft.com/en-us/library/9yd93633.aspx

A	P	C/W	Sign
(3)	(4)	(3)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What Is Dll And What Are Their Usages And Advantages?
2. What Are The Sections In A Dll Executable/binary?
3. Where Should We Store Dlls ?
4. Who Loads And Links The Dlls?
5. How Many Types Of Linking Are There?
6. What Is Implicit And Explicit Linking In Dynamic Loading?
7. How to call a DLL function from Visual Besic?



GROUP - B

EXPERIMENT NO : 06

1. **Title:**
   Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of 'Java' program

2. **Objectives :**
   - To understand LEX Concepts
   - To implement LEX Program
   - To study about Lex & Java
   - To know important about Lexical analyzer

3. **Problem Statement :**
   Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of 'Java' program

4. **Outcomes:**

After completion of this assignment students will be able to:
   - Understand the concept of LEX Tool
   - Understand the lexical analysis part
   - It can be used for data mining concepts.

5. **Software Requirements:**
   - LEX Tool (flex)

6. **Hardware Requirement:**

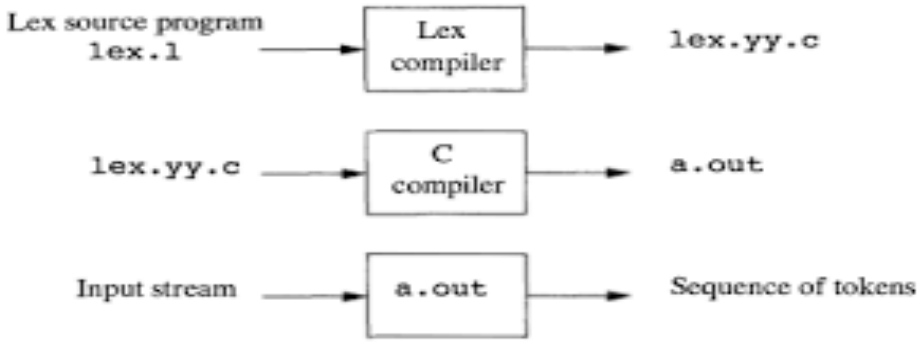
   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. **Theory Concepts:**

Lex stands for Lexical Analyzer. Lex is a tool for generating Scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular Expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it takes input one character at a time and continues until a pattern is matched, then lex performs the associated action (Which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message.

Lex and C are tightly coupled. A .lex file (Files in lex have the extension .lex) is passed through the lex utility, and produces output files in C. These file(s) are coupled to produce an executable version of the lexical analyzer.

Lex turns the user's expressions and actions into the host general –purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.



Overview of Lex Tool

- **Regular Expression in Lex:-**

A Regular expression is a pattern description using a meta language. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

- **Defining regular expression in Lex:-**

Character	**Meaning**
A-Z, 0-9,a-z	Character and numbers that form of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[]	A character class. Matches any character in the brackets. If character is ^ then it indicates a negation pattern. Example: [abc] matches either of a,b and c.
*	Matches zero or more occurences of the preceiding pattern.
+	Matches one or more occurences of the preceiding pattern.
?	Matches zero or one occurences of the preceiding pattern.
$	Matches end of line as the last character of the pattern.
{}	Indicates how many times a pattern can be present. Example: A {1, 3} implies one or three occurences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation
|	Logical OR between expressions.
"<some symbols>"	Literal meaning of characters. Meta characters hold.

/	Look ahead matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
()	Groups a series of regular expressions.

- **Examples of regular expressions**

Regular expression	**Meaning**
Joke[rs]	Matches either jokes or joker
A {1,2}shis+	Matches Aashis, Ashis, Aashi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.

Tokens in Lex are declared like variable names in C.Every token has an associated expression.(Examples of tokens and expression are given in the following table). Using the examples in our tables, we'll build a word-counting program. Our first task will be to show how tokens are declared.

- **Examples of token declaration**

Token	**Associated expression**	**Meaning**
Number	([0-9])+	1 or more occurences of a digit
Chars	[A-Za-z]	Any character
Blank	""	A blank space
Word	(chars)+	1 or more occurences of chars
Variable	(chars)+(number)*(chars)*(number)*	

- **Programming in Lex:-**

Programming in Lex can be divided into three steps:

1. Specify the pattern-associated actions in a form that Lex can understand.
2. Run Lex over this file to generate C code for the scanner.
3. Compile and link the C code to produce the executable scanner.

Note: If the scanner is part of a parser developed using Yacc, only steps 1 and 2 should be performed.

A Lex program is divided into three sections: the first section has global C and Lex declaration, the second section has the patterns (coded in C), and the third section has supplement C functions. Main (), for example, would typically be founding the third section. These sections are delimited by %%.so, to get back to the word to the word-counting Lex program; let's look at the composition of the various program sections.

Table 1: Special Characters	
Pattern	**Matches**
`.`	any character except newline
`\.`	literal .
`\n`	newline
`\t`	tab
`^`	beginning of line
`$`	end of line

Table 2: Operators	
Pattern	**Matches**
`?`	zero or one copy of the preceding expression
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`a|b`	`a` or `b` (alternating)
`(ab)+`	one or more copies of `ab` (grouping)
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`"abc*"`	literal `abc*`
`abc+`	`abc abcc abccc abcccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`

Table 3: Character Class	
Pattern	**Matches**
`[abc]`	one of: `a b c`
`[a-z]`	any letter a through z
`[a\-z]`	one of: `a - z`
`[-az]`	one of: `- a z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a b`
`[a^b]`	one of: `a ^ b`
[a|b]	one of: **a | b**

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators supported in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... definitions ...

%%

.

.. rules ...

%%

... subroutines ...

Input to Lex is divided into three sections with **%%** dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first **%%** is always required as there must always be a rules section. However if we don't specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example with defaults explicitly coded:

```
%%
/* match everything except newline */
. ECHO;
/* match newline */
\n ECHO;

%%

int yywrap(void) {
return 1;
}

int main(void) {
yylex();
return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by *whitespace* (space, tab or newline) and an optional action associated with the pattern. The action may be a single Cstatement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable **yytext** is a pointer to the matched string (NULL-terminated) and **yyleng** is the length of the matched string. Variable **yyout** is the output file and defaults to **stdout**. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a **main** function. In this case we simply call **yylex** that is the main entry-point for lex . Some implementations of lex include copies of **main** and**yywrap** in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lexprogram, functioned properly.

Table 4: Lex Predefined Variables

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

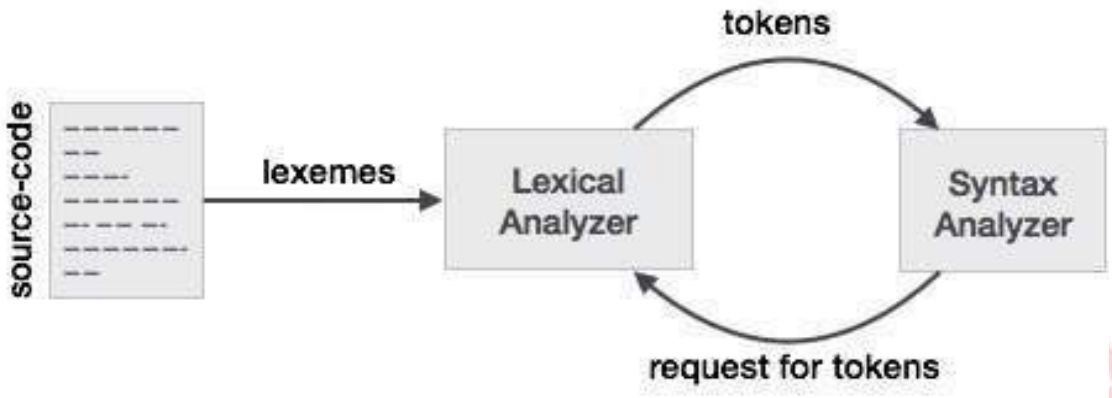
```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
   int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
   yyin = fopen(argv[1], "r");
   yylex();
```

```
        fclose(yyin);
    }
```

8. Design (architecture) :



9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

Thus, I have studied lexical analyzer and implemented an application for lexical analyzer to perform scan the program and generates token of subset of java.

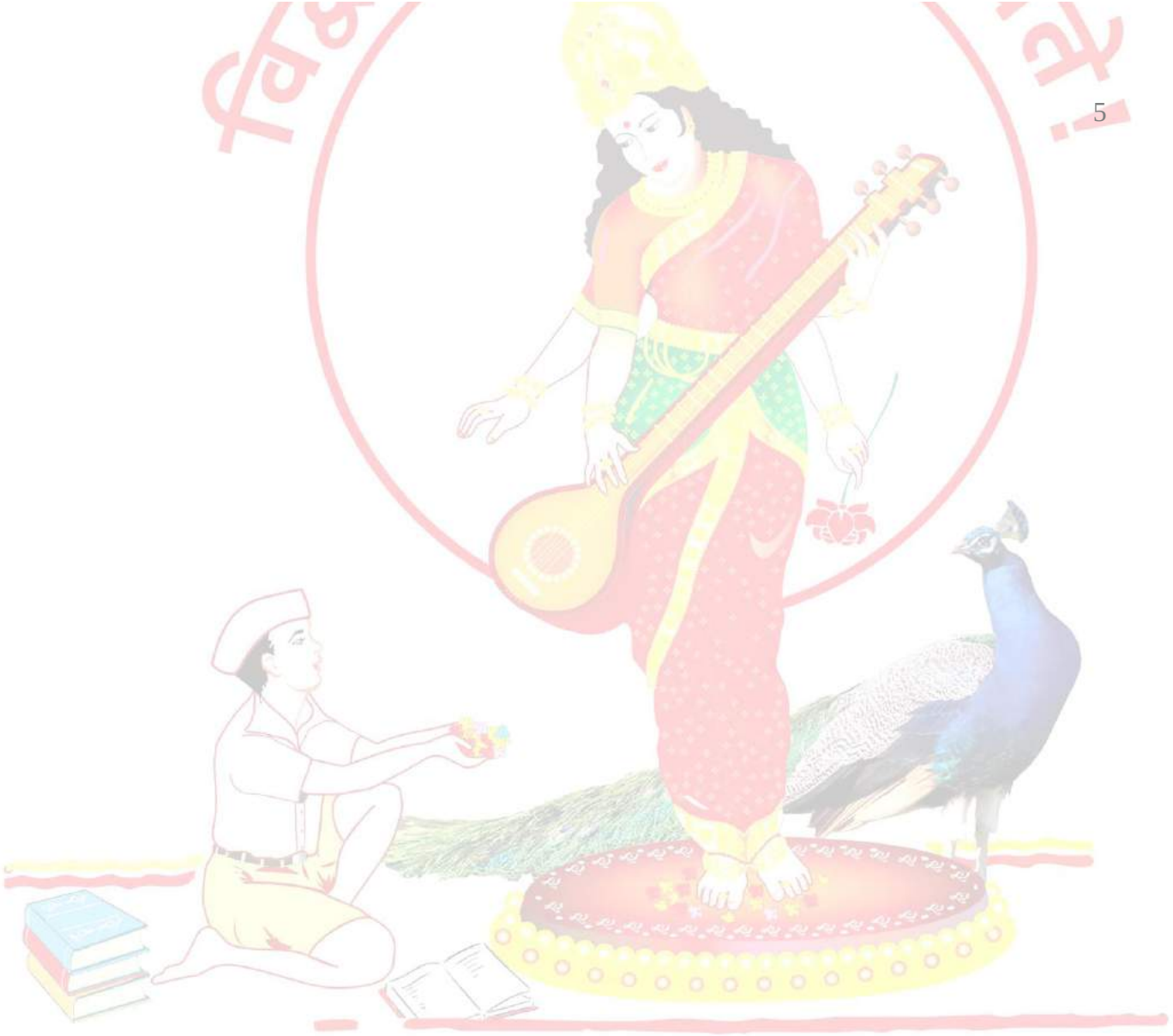
References :

A	P	C/W	Sign
(3)	(4)	(3)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is Lex.
2. What is Compiler and phases of compiler.
3. What is Lex specification.
4. What is the difference between Lex and YACC.
5. What is Regular Expression.
6. How to run a Lex program.
7. What is yytext, yyin, yyout.
8. What is yywrap().
9. What is yylex().
10. token, lexemes, pattern?



GROUP - B

EXPERIMENT NO : 07

1. **Title:**
   Write a program using Lex specifications to implement lexical analysis phase of compiler to total nos of words, chars and line etc of given file.

2. **Objectives :**
   - To understand LEX Concepts
   - To implement LEX Program for nos of count
   - To study about Lex & Java
   - To know important about Lexical analyzer

3. **Problem Statement :**
   Write a program using Lex specifications to implement lexical analysis phase of compiler to count nos. of words, chars and line of the given program/file.

4. **Outcomes:**

After completion of this assignment students will be able to:
   - Understand the concept of LEX Tool
   - Understand the lexical analysis part
   - It can be used for data mining concepts.

5. **Software Requirements:**
   - LEX Tool (flex)

6. **Hardware Requirement:**

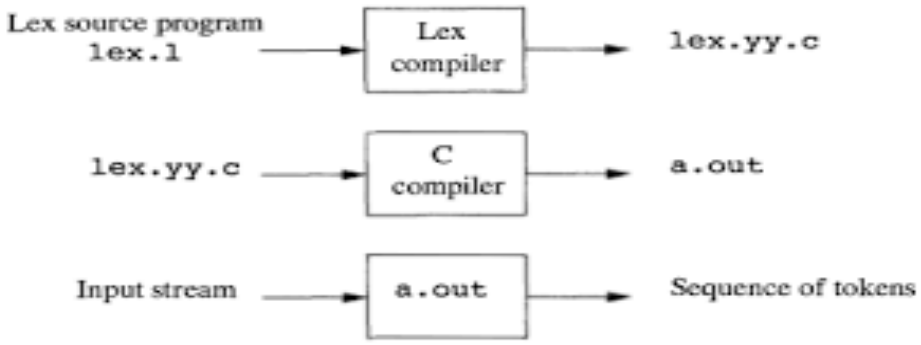
   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. **Theory Concepts:**

Lex stands for Lexical Analyzer. Lex is a tool for generating Scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular Expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it takes input one character at a time and continues until a pattern is matched, then lex performs the associated action (Which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message.

Lex and C are tightly coupled. A .lex file (Files in lex have the extension .lex) is passed through the lex utility, and produces output files in C. These file(s) are coupled to produce an executable version of the lexical analyzer.

Lex turns the user's expressions and actions into the host general –purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.



Overview of Lex Tool

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*****" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

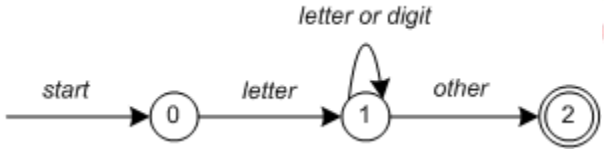


Figure 2: Finite State Automaton

In Figure 3 state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. *Any* FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character and *current state* the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "**(**" we push it on the stack. When a "**)**" is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

- **Regular Expression in Lex:-**

A Regular expression is a pattern description using a meta language. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

- **Defining regular expression in Lex:-**

Character	Meaning
A-Z, 0-9,a-z	Character and numbers that form of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.

[]	A character class. Matches any character in the brackets. If character is ^ then it indicates a negation pattern. Example: [abc] matches either of a,b and c.
*	Matches zero or more occurences of the preceiding pattern.
+	Matches one or more occurences of the preceiding pattern.
?	Matches zero or one occurences of the preceiding pattern.
$	Matches end of line as the last character of the pattern.
{}	Indicates how many times a pattern can be present. Example: A {1, 3} implies one or three occurences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation
|	Logical OR between expressions.
"<some symbols>"	Literal meaning of characters. Meta characters hold.
/	Look ahead matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
()	Groups a series of regular expressions.

- **Examples of regular expressions**

Regular expression	**Meaning**
Joke[rs]	Matches either jokes or joker
A {1,2}shis+	Matches Aashis, Ashis, Aashi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.

Tokens in Lex are declared like variable names in C.Every token has an associated expression.(Examples of tokens and expression are given in the following table). Using the examples in our tables, we'll build a word-counting program. Our first task will be to show how tokens are declared.

- **Examples of token declaration**

Token	**Associated expression**	**Meaning**
Number	([0-9])+	1 or more occurences of a digit
Chars	[A-Za-z]	Any character
Blank	""	A blank space
Word	(chars)+	1 or more occurences of chars
Variable	(chars)+(number)*(chars)*(number)*	

- **Programming in Lex:-**

Programming in Lex can be divided into three steps:

1. Specify the pattern-associated actions in a form that Lex can understand.
2. Run Lex over this file to generate C code for the scanner.
3. Compile and link the C code to produce the executable scanner.

Note: If the scanner is part of a parser developed using Yacc, only steps 1 and 2 should be performed.

A Lex program is divided into three sections: the first section has global C and Lex declaration, the second section has the patterns (coded in C), and the third section has supplement C functions. Main (), for example, would typically be founding the third section. These sections are delimited by %%.so, to get back to the word to the word-counting Lex program; let's look at the composition of the various program sections.

Table 1: Special Characters

Pattern	Matches
`.`	any character except newline
`\.`	literal .
`\n`	newline
`\t`	tab
`^`	beginning of line
`$`	end of line

Table 2: Operators

Pattern	Matches
`?`	zero or one copy of the preceding expression
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`a|b`	`a` or `b` (alternating)
`(ab)+`	one or more copies of `ab` (grouping)
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`"abc*"`	literal `abc*`
`abc+`	`abc abcc abccc abcccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`

Table 3: Character Class	
Pattern	**Matches**
`[abc]`	one of: `a b c`
`[a-z]`	any letter a through z
`[a\-z]`	one of: `a - z`
`[-az]`	one of: `- a z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a b`
`[a^b]`	one of: `a ^ b`
[a|b]	one of: **a | b**

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators supported in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... definitions ...

%%

.

.. rules ...

%%

... subroutines ...

Input to Lex is divided into three sections with **%%** dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first **%%** is always required as there must always be a rules section. However if we don't specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example with defaults explicitly coded:

```
%%
/* match everything except newline */
. ECHO;
/* match newline */
\n ECHO;

%%
```

```
int yywrap(void) {
return 1;
}

int main(void) {
yylex();
return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by *whitespace* (space, tab or newline) and an optional action associated with the pattern. The action may be a single Cstatement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable **yytext** is a pointer to the matched string (NULL-terminated) and **yyleng** is the length of the matched string. Variable **yyout** is the output file and defaults to **stdout**. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a **main** function. In this case we simply call **yylex** that is the main entry-point for lex . Some implementations of lex include copies of **main** and**yywrap** in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lexprogram, functioned properly.

Table 4: Lex Predefined Variables

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
   int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
   yyin = fopen(argv[1], "r");
   yylex();
   fclose(yyin);
}
```

- **Global C and Lex declaration**

In this section we can add C variable declaration. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We'll also perform token declaration of Lex.

- **Declaration for the word-counting program**

```
%{
int wordcount=0;
%}
Chars [A-za-z\_\'\.\"]
Number ([0-9]) +
Delim [" "\n\t]
Whitespace {delim} +
Words {chars} +
%%
```

The double percent sign implies the end of this section and the beginning of the three sections in Lex programming.

- **Lex rules for matching patterns:**

Let's look at the lex rules for describing the token that we want to match.(well use c to define to do when a token is matched).continuing with word counting program,here are the rules for matching token.

- **Lex rules for word counting program:**

```
{words}{word count++; /*
Increase the word count by one*/}
{whitespace}{/*do
Nothing*/}
{Number}{/*one may want to add some processing here*/}
%%
```

- **C code**

The third and final section of programming in lex cover c function declaration (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has set the function and variable that are available to the user. One of them is yywrap.Typically, yywrap () is define as shown in the example below.

C code section for word counting program

```
void main()
{
Ylex();/*start the analysis*/
printf("No of words:%d\n",wordCount);
}
int yywrap()
{
return1;
}
```

In the processing section we have the basic element of lex programming.which should help in the writing simple lexical analysis programs.

- **Putting it all together**

The lex file is Lex scanner. It is represented to lex program as

$ lex <file name.lex>

This produce the lex.yy.c file which can be compiled using a C compile. It can also be used with parser to produce executable or you can include the Lex library in the link step with the option A-11.

- **Here some of Lex's flags:**
  - -c Indicate C action and is the default.
  - -t causes the lex.yy.c program to be written instead to standard output.
  - -v Provide a two-line summary of statistic.
  - -n will not print out the –v summary.

➢ **Lex variable and Function**

Lex has several functions and variable that provides different information and can be used to build programs that can perform complex function. Some of these variable and function along with their uses are listed in the following table.

Yyleng	Give the length of the match pattern

Yylineno	Provide the current line number information. (May or may not b supported by the lexer.)

- **Lex variables:**

Yyin	Of the type FILE*.This point to the current file being parsed by the lexer
Yyout	Of the type FILE*.This points location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
Yytext	The text of the matched pattern is stored in this variable (char*).

- **Lex functions:**

yylex()	The function that start the analysis. It is automatically generated by Lex.
Yywrap	This function is called when the file is encountered. If this function returned 1.The parsing stops. So, this can be used to parse multiple files .code can be written in the third section, which will allow multiple file to be parsed. The strategy is to make yyin file pointer (see the preceding table)point to different file until all file are parsed .At the end, yywrap() can return 1 to indicate end of the parsing
yyless(int n)	This function can be used to push back all but first 'n' character of the read token.
yymore()	This function tells the lexer to append the next token to the current token

Examples:

The following example prepend line number to n each line in the file. Some implementations of the lex predefine &calculate yylineno. The input file for lex is yyin, and default to stdin.

```
%{
Int yylineno;
%}
%%
 ^ (.*)\n printf ("%s",++yylineno, yytext);
%%
I nt main (int argc,char *argv[]) {
 yyin=fopen (argv [1],"r");
 yylex ();
 fclose (yyin);
}
```

Here is a scanner that the number of character, words, and lines in a file

```
%{
```

```
int nchar,nword,nline;
%}
%%
\n{nline++;nchar++ ;}
[^\t\n]+ {nword++, nchar+=yyleng ;}
. {nchar++ ;}
%%
int main(void){
yylex ();
Printf ("%d\t%d\t%d\n", nchar, nword,nline);
Return 0;
}
```

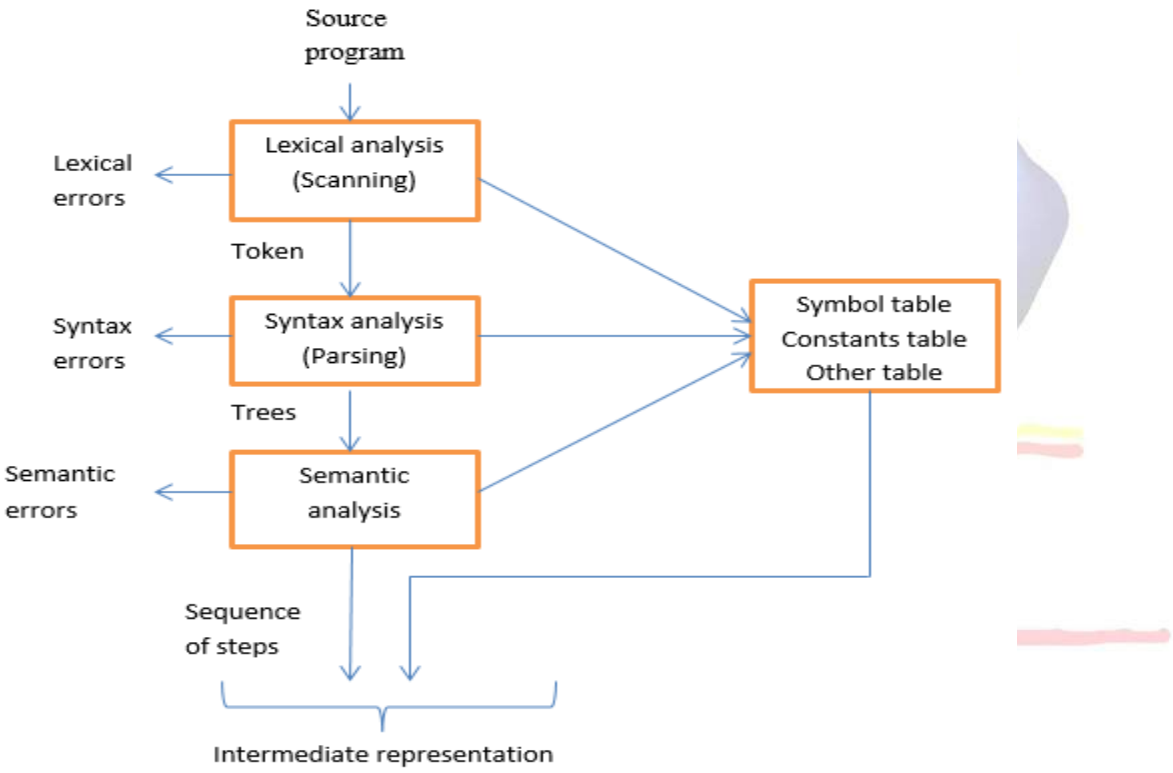
HOW THE INPUT IS MATCHED

When the generated scanner is run, it analyzes its input looking for strings, which match any of its patterns. If it finds more than one match, it takes the one matching the most text. If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match(called the *token*) is made available in the global character pointer 'yytext',and its length in the global integer 'yyleng'.The *action* corresponding to the matched pattern is then executed,and then the remaining input is scanned for another match.

If no match is found, then the *default rule* is executed: the next character in the input is considered matched and copied to the standard output.

8. Design (architecture) :



Front end of toy compiler

9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

Thus, I have studied lexical analyzer and implemented an application for lexical analyzer to count total number of words, chard and line etc

References :

https://en.wikipedia.org/wiki/Lex_(software)
http://epaperpress.com/lexandyacc/prl.html
https://www.ibm.com/developerworks/library/l-lexyac/index.html

A	P	C/W	Sign
(3)	(4)	(3)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is Lex.
2. What is Compiler and phases of compiler.
3. What is Lex specification.
4. What is the difference between Lex and YACC.
5. What is Regular Expression.
6. How to run a Lex program.
7. What is yytext, yyin, yyout.
8. What is yywrap().
9. What is yylex().
10. token, lexemes, pattern?

GROUP - B

EXPERIMENT NO : 08

1. **Title:**
   Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java.

2. **Objectives :**
   - To understand LEX & YACC Concepts
   - To implement LEX Program & YACC program
   - To study about Lex & Yaac specification
   - To know important about Lexical analyzer and Syntax analysis

3. **Problem Statement :**
   Write a program using YACC specifications to implement syntax analysis phase of compiler to validate infix expression & arithmetic expression in Java.

4. **Outcomes:**

After completion of this assignment students will be able to:
   - Understand the concept of LEX & YACC Tool
   - Understand the lexical analysis & Syntax analysis part
   - It can be used for data mining and checking(validation) concepts.

5. **Software Requirements:**
   FLEX, YACC (LEX & YACC)

6. **Hardware Requirement:**

   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. **Theory Concepts:**

Yacc (**Yet Another Compiler-Compiler**) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.

Yacc is one of the automatic tools for generating the parser program. Basically Yacc is a LALR parser generator. The Yacc can report conflicts or ambiguities (if at all) in the form of error messages. LEX and Yacc work together to analyse the program syntactically.

Yacc is officially known as a “parser”. Its job is to analyze the structure of the input stream, and operate of the “big picture”. In the course of it’s normal work, the parser also verifies that the input is syntactically sound.

YACC stands for “**Yet Another Compiler Compiler**” which is a utility available from Unix.

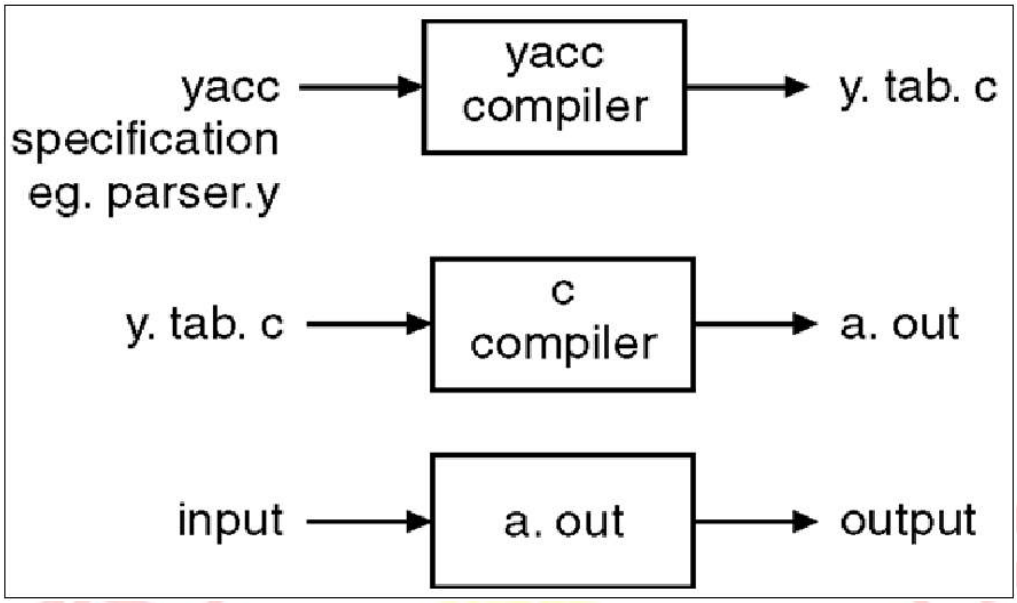


Fig:-YACC: Parser Generator Model

Structure of a yacc file:

A yacc file looks much like a lex file:

...definitions..

%%

...rules...

%%

...code...

Definitions As with lex, all code between %{ and %} is copied to the beginning of the resulting C file. Rules As with lex, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the 3 case with lex code. This can be very Elaborate, but the main ingredient is the call to yyparse, the grammatical parse.

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.
This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backusand Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

1 E -> E + E
2 E -> E * E
3 E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as **E**, are nonterminals. Terms such as **id** (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E * E (r2)
-> E * z (r3)
-> E + E * z (r1)
-> E + y * z (r3)
-> x + y * z (r3)

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we a need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to *reduce* an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

1 . x + y * z shift
2 x . + y * z reduce(r3)
3 E . + y * z shift
4 E + . y * z shift
5 E + y . * z reduce(r3)
6 E + E . * z shift
7 E + E * . z shift
8 E + E * z . reduce(r3)
9 E + E * E . reduce(r2) emit multiply
10 E + E . reduce(r1) emit add
11 E . accept

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle* and we are *reducing* the handle to the lhsof the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the **x** to the stack. Step 2 applies rule **r3** to the stack to

change **x** to **E**. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule **r2**, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule **r1**. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous*because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an **id** on the stack we may reduce to **T** or **E**.

E -> T
E -> id
T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to yacc is divided into three sections. The *definitions* section consists of token declarations and C code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the *rules* section and user subroutines are added in the *subroutines*section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

%token INTEGER

This definition declares an **INTEGER** token. Yacc generates a parser in file **y.tab.c** and an include file **y.tab.h**:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls **yylex**. Function **yylex** has a return type of **int** that returns a token. Values associated with the token are returned by lex in variable **yylval**. For example,

```
[0-9]+      {
              yylval = atoi(yytext);
              return INTEGER;
            }
```

would store the value of the integer in **yylval**, and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]        return *yytext;    /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258 because lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
  #include "y.tab.h"
  #include <stdlib.h>
  void yyerror(char *);
%}

%%

[0-9]+      {
              yylval = atoi(yytext);
              return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");
```

```
%%

int yywrap(void) {
   return 1;
}
```

Internally yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of **YYSTYPE** elements and associates a value with each element in the parse stack. For example when lex returns an **INTEGER** token yacc shifts this token to the parse stack. At the same time the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

```
%{
   #include <stdio.h>
   int yylex(void);
   void yyerror(char *);
%}


%token INTEGER

%%

program:
      program expr '\n'       { printf("%d\n", $2); }
      |
      ;

expr:
      INTEGER               { $$ = $1; }
      | expr '+' expr       { $$ = $1 + $3; }
      | expr '-' expr       { $$ = $1 - $3; }
      ;

%%

void yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
}

int main(void) {
   yyparse();
   return 0;
}
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

With left-recursion we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

expr: expr '+' expr { $$ = $1 + $3; }

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop "**expr '+' expr**" and push "**expr**". We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying "**$1**" for the first term on the right-hand side of the production, "**$2**" for the second, and so on. "**$$**" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

Numeric values are initially entered on the stack when we reduce from **INTEGER** to **expr**. After **INTEGER** is shifted to the stack we apply the rule

expr: INTEGER { $$ = $1; }

The **INTEGER** token is popped off the parse stack followed by a push of **expr**. For the value stack we pop the integer value off the stack and then push it back on again. In other words we do nothing. In fact this is the default action and need not be specified. Finally, when a newline is encountered, the value associated with **expr** is printed.

In the event of syntax errors yacc calls the user-supplied function **yyerror**. If you need to modify the interface to**yyerror** then alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is**main** … in case you were wondering where it was. This example still has an ambiguous grammar. Although yacc will issue shift-reduce warnings it will still process the grammar using shift as the default operation.

In this section we will extend the calculator from the previous section to incorporate some new functionality. New features include arithmetic operators multiply and divide. Parentheses may be used to over-ride operator precedence, and single-character variables may be specified in assignment statements. The following illustrates sample input and calculator output:

user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5)
user: y = 5
user: x
calc: 27
user: y

```
calc:  5
user:  x + 2*y
calc:  37
```

The lexical analyzer returns **VARIABLE** and **INTEGER** tokens. For variables **yylval** specifies an index to the symbol table **sym**. For this program **sym** merely holds the value of the associated variable. When **INTEGER** tokens are returned, **yylval** contains the number scanned. Here is the input specification for lex:

```
%{
   #include <stdlib.h>
   #include "y.tab.h"
   void yyerror(char *);
%}

%%

   /* variables */
[a-z]      {
              yylval = *yytext - 'a';
              return VARIABLE;
           }

   /* integers */
[0-9]+     {
              yylval = atoi(yytext);
              return INTEGER;
           }

   /* operators */
[-+()=/*\n] { return *yytext; }

   /* skip whitespace */
[ \t]      ;

   /* anything else is an error */
.             yyerror("invalid character");

%%

int yywrap(void) {
   return 1;
}
```

The input specification for yacc follows. The tokens for **INTEGER** and **VARIABLE** are utilized by yacc to create **#defines** in **y.tab.h** for use in lex. This is followed by definitions for the arithmetic operators. We may specify **%left**, for left-associative or **%right** for right associative. The last

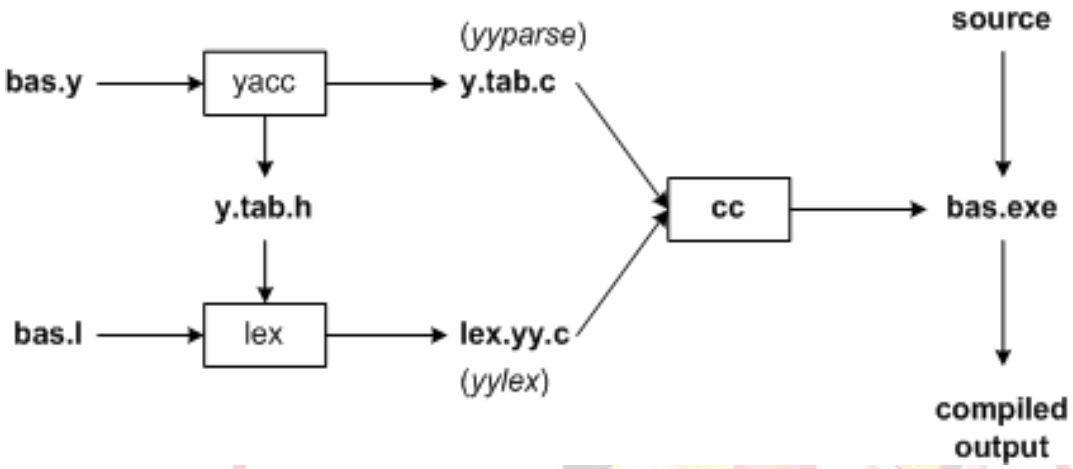
definition listed has the highest precedence. Consequently multiplication and division have higher precedence than addition and subtraction. All four operators are left-associative. Using this simple technique we are able to disambiguate our grammar.

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%{
   void yyerror(char *);
   int yylex(void);
   int sym[26];
%}

%%

program:
        program statement '\n'
        |
        ;

statement:
        expr                    { printf("%d\n", $1); }
        | VARIABLE '=' expr     { sym[$1] = $3; }
        ;

expr:
        INTEGER
        | VARIABLE              { $$ = sym[$1]; }
        | expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        | expr '*' expr         { $$ = $1 * $3; }
        | expr '/' expr         { $$ = $1 / $3; }
        | '(' expr ')'          { $$ = $2; }
        ;

%%

void yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
}

int main(void) {
   yyparse();
   return 0;
}
```

Application:

· YACC is used to generate parsers, which are an integral part of compiler.

8. Design (architecture) :



9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

Thus, I have studied lexical analyzer, syntax analysis and implemented Lex & Yacc application for Syntax analyzer to validate the given infix expression.

References :

https://en.wikipedia.org/wiki/Lex_(software)
http://epaperpress.com/lexandyacc/prl.html
https://www.ibm.com/developerworks/library/l-lexyac/index.html
http://epaperpress.com/lexandyacc/pry2.html

A	P	C/W	Sign
(3)	(4)	(3)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is Lex & Yacc .
2. What is Compiler and phases of compiler.
3. What is Lex & Yacc specification.
4. What is the difference between Lex and YACC.
5. What is Regular Expression & grammer.
6. How to run a Lex & Yacc program.
7. What is yytext, yyin, yyout.
8. What is yywrap().
9. What is yylex().
10. What is yyparse().
11. Define token, lexemes, pattern & symbol error?
12. What is left, right & no associativity.
13. What is use of $$?
14. What is yylval.

GROUP - B

EXPERIMENT NO : 09

1. **Title:**
   Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file.

2. **Objectives :**
   - To understand LEX & YACC Concepts
   - To implement LEX Program & YACC program
   - To study about Lex & Yaac specification
   - To know important about Lexical analyzer and Syntax analysis

3. **Problem Statement :**
   Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file.

4. **Outcomes:**

After completion of this assignment students will be able to:
   - Understand the concept of LEX & YACC Tool
   - Understand the lexical analysis & Syntax analysis part
   - Understand the Simple and Compound sentense.

5. **Software Requirements:**
   FLEX, YACC (LEX & YACC)

6. **Hardware Requirement:**

   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. **Theory Concepts:**

Yacc (**Yet Another Compiler-Compiler**) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.

Yacc is one of the automatic tools for generating the parser program. Basically Yacc is a LALR parser generator. The Yacc can report conflicts or ambiguities (if at all) in the form of error messages. LEX and Yacc work together to analyse the program syntactically.

Yacc is officially known as a “parser”. Its job is to analyze the structure of the input stream, and operate of the “big picture”. In the course of it’s normal work, the parser also verifies that the input is syntactically sound.

YACC stands for “**Yet Another Compiler Compiler**” which is a utility available from Unix.

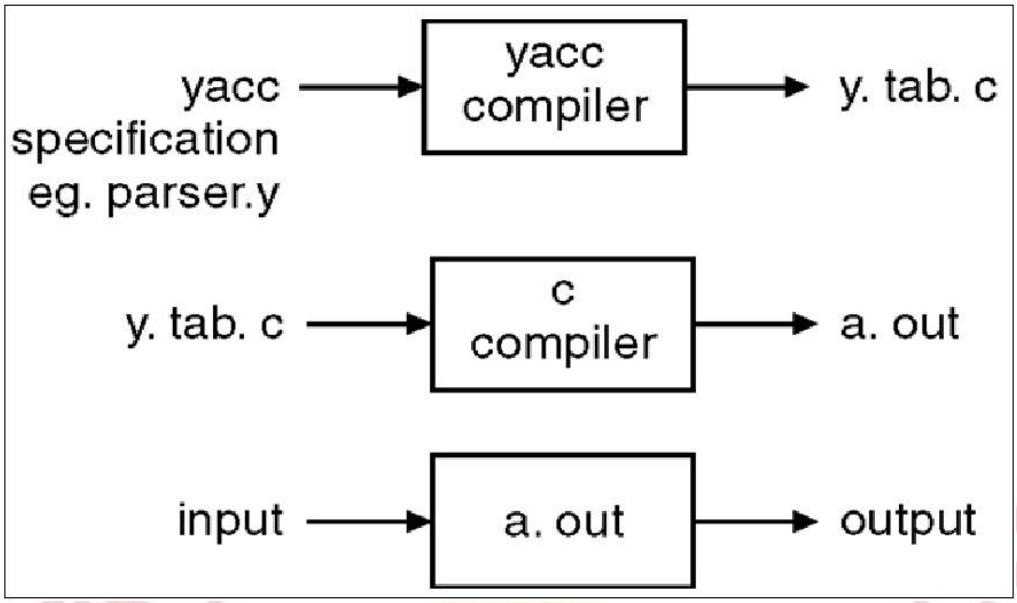


Fig:-YACC: Parser Generator Model

Structure of a yacc file:

A yacc file looks much like a lex file:

...definitions..

%%

...rules...

%%

...code...

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month_name day ',' year ;

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon

merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input July 4, 1776 might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

        …….

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters such as ``," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be ``slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can

often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules.

Basic Specifications :

Names refer to either tokens or non-terminal symbols. Yacc requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well.

Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

declarations
%%
rules
%%
programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also;

thus, the smallest legal Yacc specification is

%%
rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : BODY ;

A represents a non-terminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.", underscore ``_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or non-terminal symbols.

A literal consists of a character enclosed in single quotes ``'". As in C, the backslash ``\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n' newline
'\r' return
'\" single quote ``'"

'\\' backslash ``\"
'\t' tab
'\b' backspace
'\f' form feed
'\xxx' ``xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to Yacc as

```
A : B C D
  | E F
  | G
  ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a non-terminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 . . .
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Of all the non-terminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. If the tokens up to, but not including, the end-marker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the end-marker is seen in any other context, it is an error.

Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A     :     '('  B  ')'
                {     hello( 1, "abc" );  }
```

and

```
XXX   :     YYY  ZZZ
                {     printf("a message\n");
                      flag = 25;   }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign" ``$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B  C  D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :     '('  expr  ')'  ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr  :     '('  expr  ')'      {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B   ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by

the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A     :     B
                  {  $$ = 1;  }
            C
                  {   x = $2;   y = $3;  }
      ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new non-terminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT   :     /* empty */
                  {  $$ = 1;  }
      ;

A     :     B  $ACT  C
                  {   x = $2;   y = $3;  }
      ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr   :     expr  '+'  expr
                  {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

Translating, Compiling and Executing A Yacc Program

The Lex program file consists of Lex specification and should be named <file name>.l and the Yacc program consists of Yacc sepecification and should be named <file name>.y. following command may be issued to generate the parser

Lex <file name>.l

Yacc –d <file name>.y

cc lex.yy.c y.tab.c –ll

./a.out

Yacc reads the grammar description in <file name>.yand generates a parser, function yyparse, in file y.tab.c . the –d option causes yacc to generate the definitions for tokens that are declared in the <file name>.y and palce them in file y.tab.h. Lex reads the pattern descriptions in <file name>.l, includes file y.tab.h, and generates a lexical analyzer, function yylex, in the file lex.yy.c

Finally, the lexer and the parser are compiled and linked (-ll) together to form the output file, a.out(by default).

The execution of the parser begins from the main function, which will be ultimately call yyparse() to run the parser. Function yyparse() automatically calls yylex() whenever it is in need of token .

Lexical Analyzer for YACC

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
              . . .
```

```
        case '0':
        case '1':
          . . .
        case '9':
              yylval = c-'0';
              return( DIGIT );
              . . .
              }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names if or while will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling, and should not be used naively.

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

When Yacc generates, the parser(by default y.tab.c, which is C file), it will assign token numbers for all the tokens defined in Yacc program.Token numbers will be assigned using"#define"and will be copied, by default, to y.tab.h file. The lexical analyzer will reasd from this file or any furthe use.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
  expr : expr OP expr
```

and

```
  expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. You specify as disambiguating rules the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with the **yacc** keywords **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. + and - are left associative and have lower precedence than *and /, which are also left associative. The keyword **%right** is used to describe right associative operators. The keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. That is, because

```
A .LT. B .LT. C
```

is invalid in FORTRAN, **.LT.** would be described with the keyword **%nonassoc** in **yacc**.

As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | NAME
     ;
```

might be used to structure the input

```
a = b = c * d - e - f * g
```

as follows

```
a = ( b = ( ((c * d) - e) - (f * g) ) )
```

in order to achieve the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator

have the same symbolic representation but different precedences. An example is unary and binary minus.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword **%prec** changes the precedence level associated with a particular grammar rule. **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```
%left '+' '-'
%left '*' '/'


%%


expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr %prec '*'
     | NAME
     ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not, but may, be declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a **reduce-reduce** or **shift-reduce** conflict, and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given in the preceding section are used, and the conflicts are reported.
4. If there is a **shift-reduce** conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action -- **shift** or **reduce** -- associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; nonassociating implies **error**.

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar.

The yyerror() Function

The yyerror function is called when Yacc encounters an invalid synatx. Whenver an invalid syntax finds error, it will move to already predefined error state. Moving to error state maens shifting (shift/reduce) to error, which is areserved token name for error handling.that is, any move to error state will cause to call function yyerror.the yyerror() is passed a single string of type char* as argument. The basic yyerror() function is like this:

```
yyerror(char* err)

{

  fprintf(stderr,"%s\n",err);

}
```

The above function just prints the error message when we call the function by passsing the argument.

A **compound sentence** is a sentence that has at least two independent clauses joined by a comma, semicolon or conjunction. An **independent clause** is a clause that has a subject and verb and forms a complete thought.

An example of a compound sentence is, 'This house is too expensive, and that house is too small.' This sentence is a compound sentence because it has two independent clauses, 'This house is too expensive' and 'that house is too small' separated by a comma and the conjunction 'and.'

Compound Sentences and Meaning

When independent clauses are joined with **coordinators** (also called coordinating conjunctions) commas and semicolons, they do more than just join the clauses. They add meaning and flow to your writing. First let's look at the coordinators you can use to join independent clauses. They are:

- For
- And
- Nor
- But
- Or
- Yet
- So

Note that they form the handy mnemonic FANBOYS. The three you will use most often are 'and,' 'but' and 'or.'

Here's an example of how coordinating conjunctions add meaning:

'I think you'd enjoy the party, but I don't mind if you stay home.'

In this sentence, the coordinator 'but' shows a clear relationship between the two independent clauses, in this case, that the speaker is making a suggestion that the person being addressed isn't expected to follow it. Without the coordinator 'but,' the relationship isn't apparent, making the writing choppy and the meaning less clear:

'I think you'd enjoy the party. I don't mind if you stay home.'

You can also join independent clauses with a **semicolon (;)** , which looks something like a cross between a colon and a comma. If you join clauses with a semicolon, you add an abrupt pause, creating a different kind of effect, as shown in the sentence below:

'He said he didn't mind if I stayed home; it soon became clear he wasn't being honest.'

You should use a semicolon when the independent clauses are related, but contrast in a way that you want to stand out. In the sentence above, the contrast is that the person being talked about in the first clause sounded honest when he said he didn't mind if the speaker stayed home, but in the second clause, the speaker is telling you that the person being talked about was not honest. You could just as easily have written the sentence using a coordinating conjunction:

'He said he didn't mind if I stayed home, but it soon became clear he wasn't being honest.'

The sentence still means the same as before, but using the coordinator 'but' softens the impact of the second clause.

Comparing Sentence Types

Sentences give structure to language, and in English, they come in four types: simple, compound, complex and compound-complex. When you use several types together, your writing is more interesting. Combining sentences effectively takes practice, but you'll be happy with the result.

1. The **simple sentence** is an independent clause with one subject and one verb.

   For example: we are the indian.

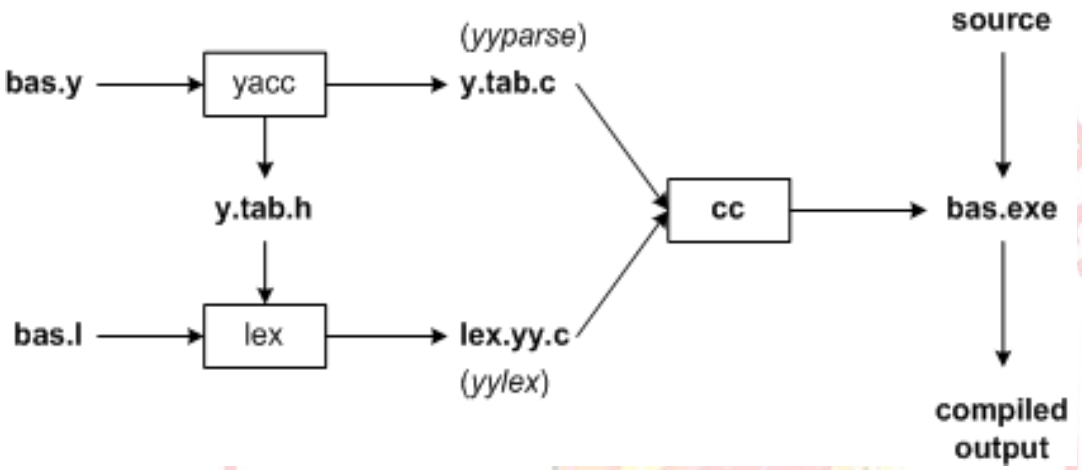
2. The Compound sentence is two or more independent clause, joined with comma, semicolon & conjuction.

For example: I am student and indian

Application:

· YACC is used to generate parsers, which are an integral part of compiler.

8. Design (architecture) :



9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

Thus, I have studied lexical analyzer, syntax analysis and implemented Lex & Yacc application for Syntax analyzer to validate the given infix expression.

References :

https://en.wikipedia.org/wiki/Lex_(software)
http://epaperpress.com/lexandyacc/prl.html
https://www.ibm.com/developerworks/library/l-lexyac/index.html
http://epaperpress.com/lexandyacc/pry2.html

A	P	C/W	Sign
(3)	(4)	(3)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is Lex & Yacc .
2. What is Compiler and phases of compiler.
3. What is Lex & Yacc specification.
4. What is the difference between Lex and YACC.
5. What is Regular Expression & grammer.
6. How to run a Lex & Yacc program.
7. What is yytext, yyin, yyout.
8. What is yywrap().
9. What is yylex().
10. What is yyparse().
11. Define token, lexemes, pattern & symbol error?
12. What is left, right & no associativity.
13. What is use of $$?
14. What is yylval.

GROUP - C

GROUP - C

GROUP - C

EXPERIMENT NO : 10

1. **Title:**
   Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive).

2. **Objectives :**
   - To understand OS & SCHEDULLING Concepts
   - To implement Scheduling FCFS, SJF, RR & Priority algorithms
   - To study about Scheduling and scheduler

3. **Problem Statement :**
   Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF, Priority and Round Robin .

4. **Outcomes:**

After completion of this assignment students will be able to:
   - Knowledge Scheduling policies
   - Compare different scheduling algorithms

5. **Software Requirements:**
   JDK/Eclipse

6. **Hardware Requirement:**

   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. **Theory Concepts:**

CPU Scheduling:

• CPU scheduling refers to a set of policies and mechanisms built into the operating systems that govern the order in which the work to be done by a computer system is completed.

• Scheduler is an OS module that selects the next job to be admitted into the system and next process to run.

• The primary objective of scheduling is to optimize system performance in accordance with the criteria deemed most important by the system designers.

What is scheduling?

Scheduling is defined as the process that governs the order in which the work is to be done. Scheduling is done in the areas where more no. of jobs or works are to be performed. Then it requires some plan i.e. scheduling that means how the jobs are to be performed i.e. order. CPU scheduling is best example of scheduling.

What is scheduler?

1. Scheduler in an OS module that selects the next job to be admitted into the system and the next process to run.
2. Primary objective of the scheduler is to optimize system performance in accordance with the criteria deemed by the system designers. In short, scheduler is that module of OS which schedules the programs in an efficient manner.

Necessity of scheduling

• Scheduling is required when no. of jobs are to be performed by CPU.

• Scheduling provides mechanism to give order to each work to be done.

• Primary objective of scheduling is to optimize system performance.

• Scheduling provides the ease to CPU to execute the processes in efficient manner.

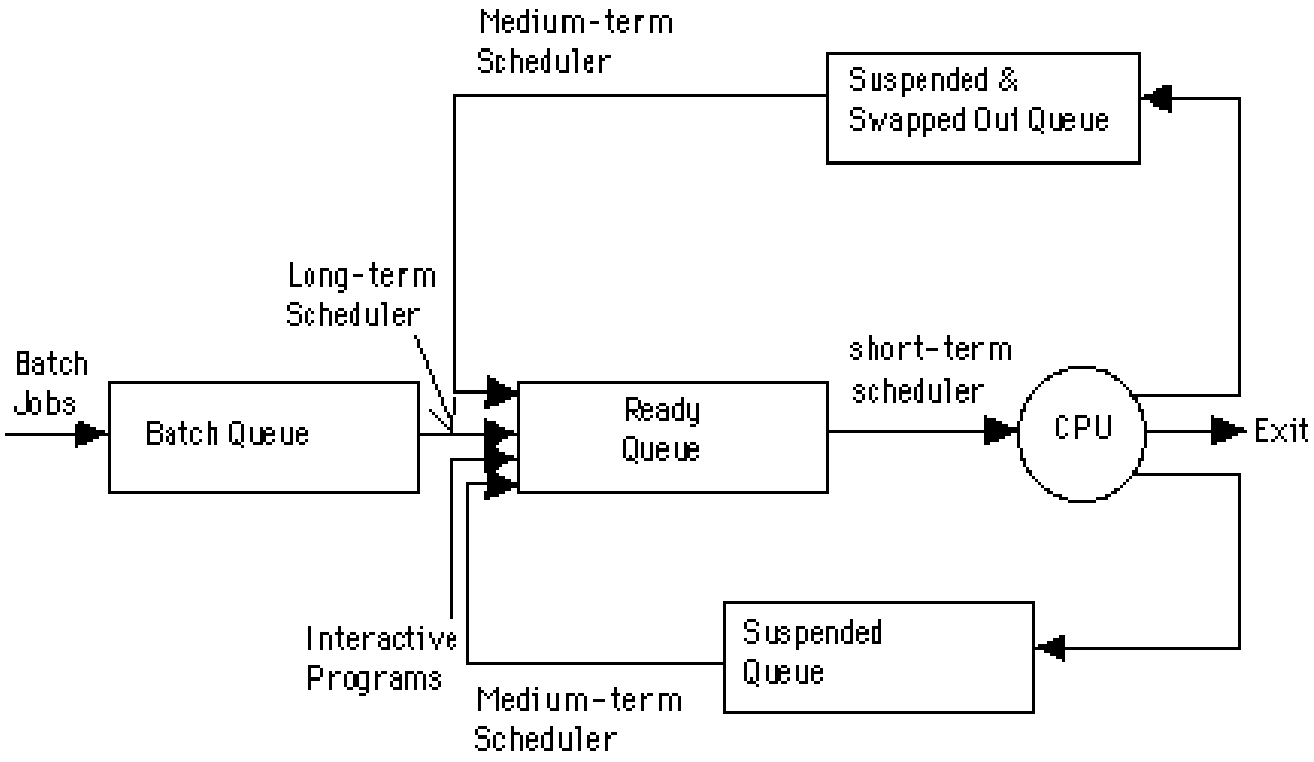
Types of schedulers

In general, there are three different types of schedulers which may co-exist in a complex operating system.

• Long term scheduler

• Medium term scheduler

• Short term scheduler.



Long Term Scheduler

• The long term scheduler, when present works with the batch queue and selects the next batch job to be executed.

• Batch is usually reserved for resource intensive (processor time, memory, special I/O devices) low priority programs that may be used fillers of low activity of interactive jobs.

• Batch jobs usually also contains programmer-assigned or system-assigned estimates of their resource needs such as memory size, expected execution time and device requirements.

• Primary goal of long term scheduler is to provide a balanced mix of jobs.

Medium Term Scheduler

• After executing for a while, a running process may because suspended by making an I/O request or by issuing a system call.

• When number of processes becomes suspended, the remaining supply of ready processes in systems where all suspended processes remains resident in memory may become reduced to a level that impairs functioning of schedulers.

• The medium term scheduler is in charge of handling the swapped out processes.

• It has little to do while a process is remained as suspended.

Short Term Scheduler

• The short term scheduler allocates the processor among the pool of ready processes resident in the memory.

• Its main objective is to maximize system performance in accordance with the chosen set of criteria.

• Some of the events introduced thus for that cause rescheduling by virtue of their ability to change the global system state are:

• Clock ticks

• Interrupt and I/O completions

• Most operational OS calls

• Sending and receiving of signals

• Activation of interactive programs.

• Whenever one of these events occurs ,the OS involves the short term scheduler.

Scheduling Criteria :

- **CPU Utilization:**

  Keep the CPU as busy as possible. It range from 0 to 100%. In practice, it range from 40 to 90%.

- **Throughput:**

  Throughput is the rate at which processes are completed per unit of time.

- **Turnaround time:**

  This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion.

- **Waiting time:**

  Waiting time is the sum of the time periods spent in waiting in the ready queue.

- **Response time:**

  Response time is the time it takes to start responding from submission time. It is calculated as the amount of time it takes from when a request was submitted until the first response is produced.

Non-preemptive Scheduling :

In non-preemptive mode, once if a process enters into running state, it continues to execute until it terminates or blocks itself to wait for Input/Output or by requesting some operating system service.

Preemptive Scheduling :

In preemptive mode, currently running process may be interrupted and moved to the ready State by the operating system.
When a new process arrives or when an interrupt occurs, preemptive policies may incur greater overhead than non-preemptive version but preemptive version may provide better service.

It is desirable to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time and response time.

Types of scheduling Algorithms

• In general, scheduling disciplines may be pre-emptive or non-pre-emptive .

• In batch, non-pre-emptive implies that once scheduled, a selected job turns to completion.

There are different types of scheduling algorithms such as:

- FCFS(First Come First Serve)
- SJF(Short Job First)
- Priority scheduling
- Round Robin Scheduling algorithm

First Come First Serve Algorithm

• FCFS is working on the simplest scheduling discipline.

• The workload is simply processed in an order of their arrival, with no pre-emption.

• FCFS scheduling may result into poor performance.

• Since there is no discrimination on the basis of required services, short jobs may considerable in turn around delay and waiting time.

Advantages

- Better for long processes
- Simple method (i.e., minimum overhead on processor)
- No starvation

Disadvantages

- Convoy effect occurs. Even very small process should wait for its turn to come to utilize the CPU. Short process behind long process results in lower CPU utilization.
- Throughput is not emphasized.

First Come, First Served

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: *P1* , *P2* , *P3*
- The Gantt Chart for the schedule is:

- Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Shortest Job First Algorithm :

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

Advantages

- It gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
- Throughput is high.

Disadvantages

- Elapsed time (i.e., execution-completed-time) must be recorded, it results an additional overhead on the processor.
- Starvation may be possible for the longer processes.

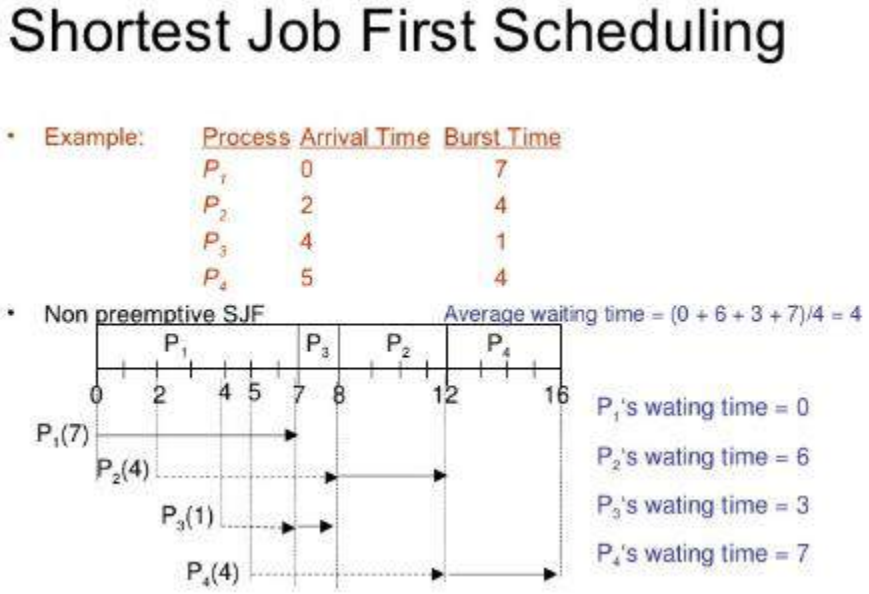
This algorithm is divided into two types:

• Pre-emptive SJF

• Non-pre-emptive SJF

• Pre-emptive SJF Algorithm:

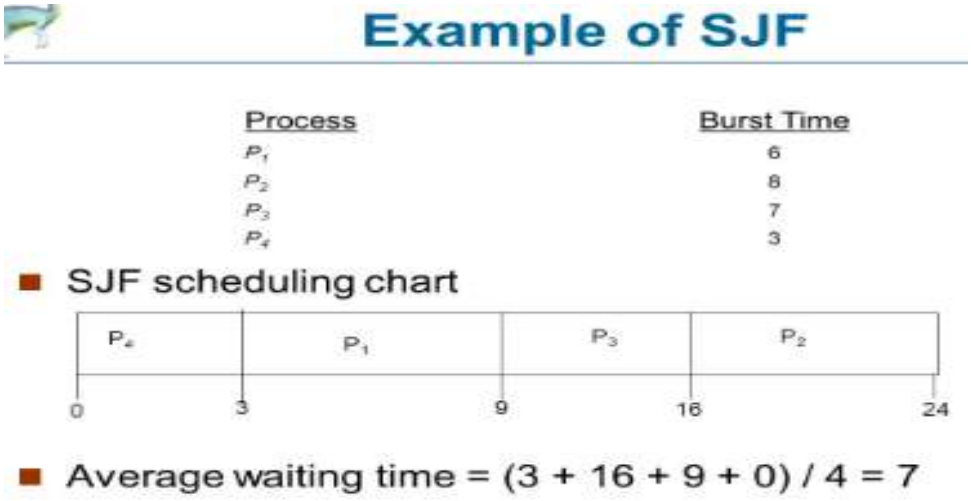
In this type of SJF, the shortest job is executed 1st. the job having least arrival time is taken first for execution. It is executed till the next job arrival is reached.



Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Non-pre-emptive SJF Algorithm:

In this algorithm, job having less burst time is selected 1st for execution. It is executed for its total burst time and then the next job having least burst time is selected.



Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Round Robin Scheduling :

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

Advantages

- Round-robin is effective in a general-purpose, times-sharing system or transaction-processing system.
- Fair treatment for all the processes.
- Overhead on processor is low.
- Overhead on processor is low.
- Good response time for short processes.

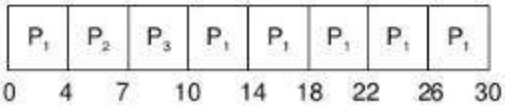
Disadvantages

- Care must be taken in choosing quantum value.
- Processing overhead is there in handling clock interrupt.
- Throughput is low if time quantum is too small.

Round Robin

Process	Burst Time
P1	24
P2	3
P3	3

- Quantum time = 4 milliseconds
- The Gantt chart is:



- Average waiting time = {[0+(10-4)]+4+7}/3 = 5.6

Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

Priority Scheduling :

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Advantage

- Good response for the highest priority processes.

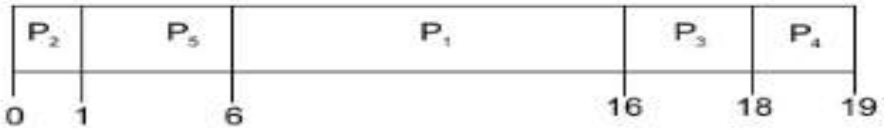
Disadvantage

- Starvation may be possible for the lowest priority processes.

Priority

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- Gantt Chart



- Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2

Note : solve complete e.g. as we studied in practical(above is just sample e.g.). you can take any e.g.

8. Algorithms(procedure) :

FCFS :

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

SJF :

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to

highest burst time.

Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

(c) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(d) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(c) Average waiting time = Total waiting Time / Number of process

(d) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

RR :

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

(a) Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)

(b) Turn around time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

(e) Average waiting time = Total waiting Time / Number of process

(f) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process.

Priority Scheduling :

Algorithms :

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time, priority

Step 4: Start the Ready Q according the priority by sorting according to lowest to highest burst time and process.

Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

(e) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(f) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(g) Average waiting time = Total waiting Time / Number of process

(h) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

Note: you can write algorithm & procedure as per your program/concepts

9. Flowchart :

Note: you should draw flowchart as per algorithm/procedure as above

10. Conclusion:

Hence we have studied that-

• CPU scheduling concepts like context switching, types of schedulers, different timing parameter like waiting time, turnaround time, burst time, etc.

• Different CPU scheduling algorithms like FIFO, SJF,Etc.

• FIFO is the simplest for implementation but produces large waiting times and reduces system performance.

• SJF allows the process having shortest burst time to execute first.

References :

https://www.studytonight.com/operating-system/cpu-scheduling
https://www.go4expert.com/articles/types-of-scheduling-t22307/
https://en.wikipedia.org/wiki/Scheduling_(computing)
https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. Scheduling? List types of scheduler & scheduling.
2. List and define scheduling criteria.
3. Define preemption & non-preemption.
4. State FCFS, SJF, Priority & Round Robin scheduling.
5. Compare FCFS, SJF, RR, Priority.

GROUP - C

EXPERIMENT NO : 11

1. Title:

Write a Java program to implement Banker's Algorithm

2. Objectives :

- To understand safe and unsafe state of a system
- To understand deadlock
- Implementation of banker's algorithm for deadlock detection and avoidance

3. Problem Statement :

Write a Java program to implement Banker's Algorithm
.

4. Outcomes:

After completion of this assignment students will be able to:

- Knowledge Bankers Algorithms
- Application of Bankers Algorithms

5. Software Requirements:

JDK/Eclipse

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

The **Banker's algorithm**, sometimes referred to as the **detection algorithm**, is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Everytime a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers. Whenever a new process is created, it must exactly specify the maximum instances of each resource type that it needs.

Let us assume that there are **n** processes and **m** resource types. Some data structures are used to implement the banker's algorithm. They are:

- Available: It is an array of length **m**. It represents the number of available resources of each type. If Available[j] = k, then there are **k** instances available, of resource type **Rj**.
- Max: It is an **n x m** matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process **Pi** can request atmost **k** instances of resource type **Rj**.
- Allocation: It is an **n x m** matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process **Pi** is currently allocated **k** instances of resource type **Rj**.
- Need: It is an **n x m** matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process **Pi** may need **k** more instances of resource type **Rj** to complete its task.

```
Need[i][j] = Max[i][j] - Allocation [i][j]
```

Resource Request Algorithm:

This describes the behavior of the system when a process makes a resource request in the form of a request matrix. The steps are:

1. If number of requested instances of each resource is less than the need (which was declared previously by the process), go to step 2.
2. If number of requested instances of each resource type is less than the available resources of each type, go to step 3. If not, the process has to wait because sufficient resources are not available yet.
3. Now, assume that the resources have been allocated. Accordingly do,

```
Available = Available - Requesti
Allocationi = Allocationi + Requesti
Needi = Needi - Requesti
```

This step is done because the system needs to assume that resources have been allocated. So there will be less resources available after allocation. The number of allocated instances will increase. The

need of the resources by the process will reduce. That's what is represented by the above three operations.

After completing the above three steps, check if the system is in safe state by applying the safety algorithm. If it is in safe state, proceed to allocate the requested resources. Else, the process has to wait longer.

Safety Algorithm:

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,

```
Work = Available
Finish[i] =false for i = 0, 1, ... , n - 1.
```

This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

4. Find an index **i** such that both

```
Finish[i] ==false
Needi <= Work
```

If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.

7. Perform the following:

```
Work = Work + Allocation;
Finish[i] = true;
```

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

10. If Finish[i] == true for all i, then the system is in a safe state.

That means if all processes are finished, then the system is in safe state.

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Note : above example just for reference for algorithm.

Safe state:

A **state** is **safe** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock **state**

The system is said to be in a **safe** state if there exists a sequence of other valid system states that leads to the successful completion of all processes.

- Processes request only 1 resource at a time.
- Request is granted **only** it results in a safe state.
- If request results in an unsafe state, the request is denied and the process continues to hold resources it has until such time as it's request can be met.
- All requests will be granted in a finite amount of time.
- Algorithm can be extended for multiple resource types.

Advantage: Avoids deadlock and it is less restrictive than deadlock prevention.

Disadvantage: Only works with fixed number of resources and processes.

- Guarantees finite time - **not** reasonable response time
- Needs advanced knowledge of maximum needs
- Not suitable for multi-access systems
- Unnecessary delays in avoiding unsafe states which may not lead to deadlock.

Limitation :

Like the other algorithms, the Banker's algorithm has some limitations when implemented. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making it impossible to implement the Banker's algorithm. Also, it is unrealistic to assume that the number of processes is static since in most systems the number of processes varies dynamically. Moreover, the requirement that a process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm, however it is not sufficient for a practical system. Waiting for hours (or even days) for resources to be released is usually not acceptable.

Example:

Determination of safe state

The concept of safe state is shown with the help of Fig. 6.19.2.

- There are four processes.
- There are three resources. The total amount of these resources are :
  1. R1 – 13 units 2. R2 – 7 units
  3. R3 – 10 units
- The maximum need of each process is shown in maximum claim matrix ().
- The initial allocation is shown in allocation matrix(R).

Out of 13 units of R1, 2, 7, 3 and 1 units are allocated to P1, P2, P3 and P4 respectively.

This leaves 13 – 2 – 7 – 3 – 1 = 0 unit of R1 for allocation.

- Initially, available resources are :

0	1	1
R1	R2	R3

- Process P1 needs $(R_1\ R_2\ R_3)$ $(4, 3, 3) - (2, 1, 1) = (2, 2, 2)$ $(R_1\ R_1\ R_2)$ units of resources.

Resources in existence (E) =

R1	R2	R3
13	7	10

(a) Initial state

	Maximum claim matrix (C)			Allocation matrix (R)			Available Resources (A)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	4	3	3	2	1	1	0	1	1
P2	7	2	4	7	2	3			
P3	4	2	5	3	2	2			
P4	5	3	3	1	1	3			

(b) P2 runs to completion

	R1	R2	R3
P1	4	3	3
P2	0	0	0
P3	4	2	5
P4	5	3	3

	R1	R2	R3
P1	2	1	1
P2	0	0	0
P3	3	2	2
P4	1	1	3

R1	R2	R3
7	3	4

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	4	2	5
P4	5	3	3

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	2	2
P4	1	1	3

R1	R2	R3
9	4	5

(d) P3 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	5	3	3

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	1	1	3

R1	R2	R3
12	6	7

(e) P4 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

R1	R2	R3
13	7	10

Fig. 6.19.2 : Determination of safe state

Available resources $(0, 1, 1)_{R_1\ R_2\ R_3}$ is not sufficient to execute.

– Process P2 needs $(7, 2, 4)_{R_1\ R_2\ R_3} - (7, 2, 3)_{R_1\ R_2\ R_3} = (0, 0, 1)_{R_1\ R_2\ R_3}$ units of resources. This requirement can be met with available resources. Hence P2 can run to its completion. Once, P2 completes, its resources (allocated) can be added to available resources.

After P2 completes,

$$\text{Available resources} = (0, 1, 1)_{R_1\ R_2\ R_3} + (7, 2, 3)_{R_1\ R_2\ R_3}$$
$$= (7, 3, 4)_{R_1\ R_2\ R_3}$$

– Now P1 can run to its completion, leaving allocated resources

$$\text{Available resources} = (7, 3, 4)_{R_1\ R_2\ R_3} + (2, 1, 1)_{R_1\ R_2\ R_3}$$
$$= (9, 4, 5)_{R_1\ R_2\ R_3}$$

– Now P3 can run to its completion, leaving allocated resources

$$\text{Available resources} = \underset{R_1\ R_2\ R_3}{(9, 4, 5)} + \underset{R_1\ R_2\ R_3}{(3, 2, 2)}$$
$$= \underset{R_1\ R_2\ R_3}{(12, 6, 7)}$$

– Now P4 can run to its completion, leaving allocated resources

$$\text{Available resources} = \underset{R_1\ R_2\ R_3}{(12, 6, 7)} + \underset{R_1\ R_2\ R_3}{(1, 1, 3)}$$
$$= \underset{R_1\ R_2\ R_3}{(13, 7, 10)}$$

Thus, these are a sequence through which all of the processes have been run to completion. Thus, the initial state defined in Fig. 6.19.2 is a safe state.

Note : you can write another example also if you wish… this is sample example. (atleast one e.g. is to be mentioned)

8. Conclusion :

Thus, I have implemented how resource allocation is done with the bankers algorithm to avoid the deadlocks.

References :

https://en.wikipedia.org/wiki/Banker%27s_algorithm
https://www.studytonight.com/operating-system/bankers-algorithm
https://www.geeksforgeeks.org/operating-system-bankers-algorithm/

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is bankers algorithms.
2. Inventor of bankers algorithms
3. State safe and unsafe state.
4. Define deadlock.
5. Define deadlock prevention, detection and avoidance.
6. Advantages & disadvantages.

.

GROUP - C

EXPERIMENT NO : 12

1. Title:

Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming)

2. Objectives :

- To understand UNIX system call
- To understand Concept of process management
- Implementation of some system call of OS

3. Problem Statement :

Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming)

4. Outcomes:

After completion of this assignment students will be able to:

- Knowledge of System call
- Compare system call and system function
- Application of System call

5. Software Requirements:

GCC or JDK/Eclipse

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

SYSTEM CALL :

- When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.
- When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.
- Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.

- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

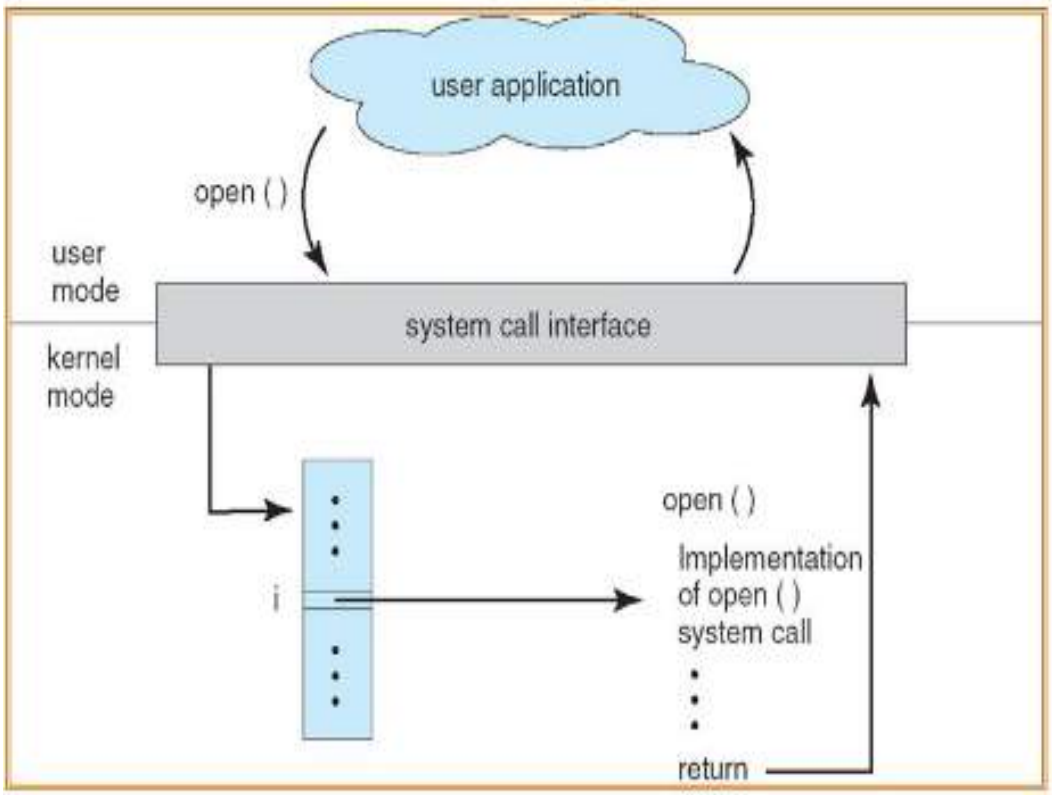
- To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.



Examples of Windows and Unix System Calls –

	WINDOWS	UNIX
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile(), ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleeo()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Call Basics

- Since system calls are functions, we need to include the proper header files
  - E.g., for getpid() we need
    - #include <sys/types.h>
    - #include <unistd.h>
- Most system calls have a meaningful return value
  - Usually, -1 or a negative value indicates an error
  - A specific error code is place in a global variable called
    - errno
  - To access errno you must declare it:
    - extern int errno;

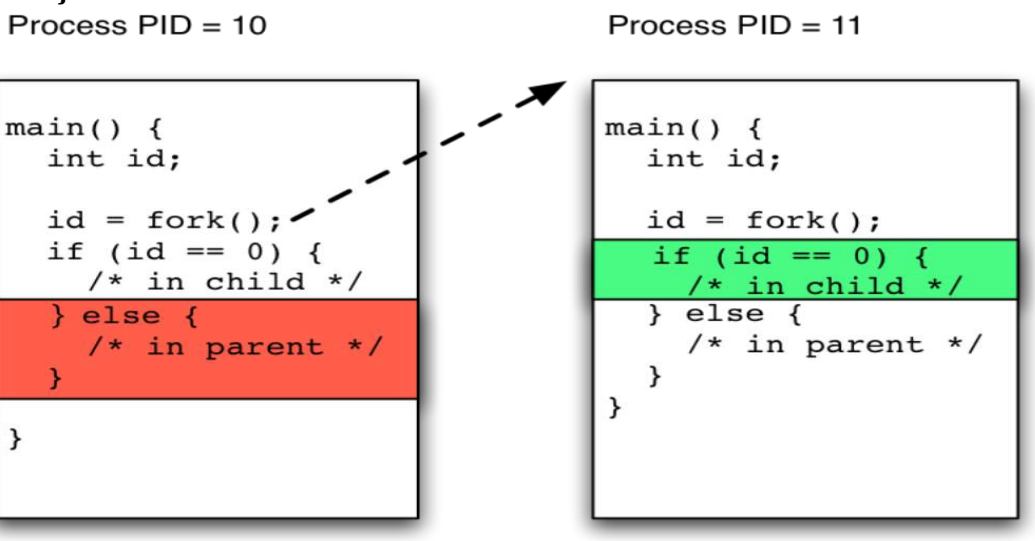
UNIX Processes

- Recall a process is a program in execution
- Processes create other processes with the fork() system call
- fork() creates an identical copy of the parent process
- We say the parent has cloned itself to create a child
- We can tell the two process apart use the return value of fork()

- In parent: fork() returns the PID of the new child
- In child: fork() returns 0

- fork() may seem strange at first, that's because it is a bit strange!
- Draw picture

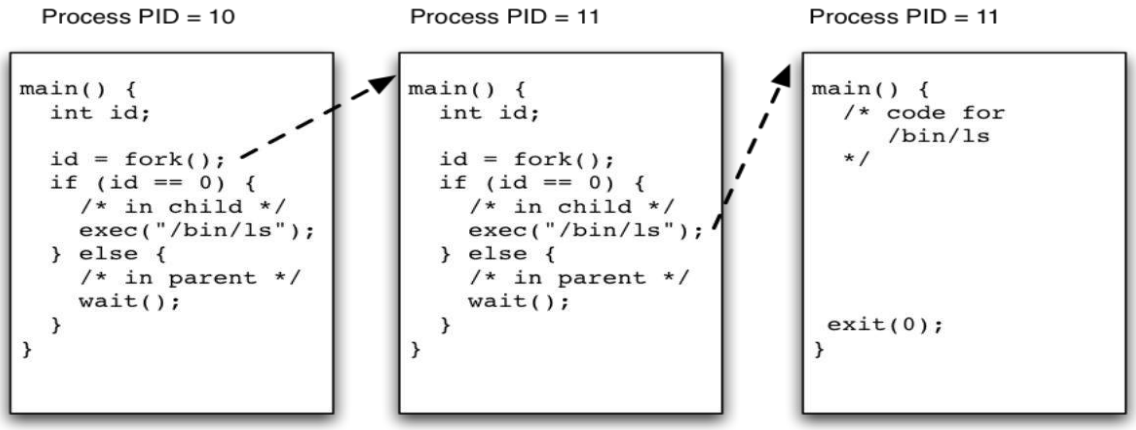
```
main() {
   int id;

   id = fork();
   if (id == 0) {
      /* in child */
   } else {
      /* in parent */
   }
}
```



Starting New Programs

- fork() only allows us to create a new process that is a duplicate of the parent
- The exec() system call is used to start a new program
- exec() replaces the memory image of the calling processes with the image of the new program
- We use fork() and exec() together to start a new program

```
main() {
   int id;

   id = fork();
   if (id == 0) {
      /* in child */
      exec("/bin/ls");
   } else {
      /* in parent */
      wait();
   }
}
```



Syscalls for Processes

- pid_t fork(void)
  - Create a new child process, which is a copy of the current process
  - Parent return value is the PID of the child proces
  - Child return value is 0

- int execl(char *name, char *arg0, ..., (char *) 0)
  - Change program image of current process
  - Reset stack and free memory
  - Start at main()
  - Also see other versions: execlp(), execv(), etc.

- pid_t wait(int *status)
  - Wait for a child process (any child) to complete
  - Also see waitpid() to wait for a specific process

- void exit(int status)
  - Terminate the calling process
  - Can also achieve with a return from main()

- int kill(pid_t pid, int sig)
  - Send a signal to a process
  - Send SIGKILL to force termination

➢ **UNIX SYSTEM CALLS :-**

- **Ps command :**

The *ps* (i.e., *process status*) command is used to provide information about the currently running *processes*, including their *process identification numbers* (PIDs).

A process, also referred to as a *task*, is an *executing* (i.e., running) instance of a program. Every process is assigned a unique PID by the system.

The basic syntax of ps is

ps [options]

When ps is used without any options, it sends to *standard output*, which is the display monitor by default, four items of information for at least two processes currently on the system: the *shell* and ps.

A shell is a program that provides the traditional, text-only user interface in Unix-like operating systems for issuing commands and interacting with the system, and it is *bash* by default on Linux. ps itself is a process and it *dies* (i.e., is terminated) as soon as its output is displayed.

The four items are labeled PID, TTY, TIME and CMD. TIME is the amount of CPU (central processing unit) time in minutes and seconds that the process has been running. CMD is the name of the command that launched the process.

- ***Fork()***
- The fork() system call is used to create processes. When a process (a program in execution) makes a fork() call, an exact copy of the process is created. Now there are two processes, one being the **parent** process and the other being the **child** process.
- The process which called the fork() call is the **parent** process and the process which is created newly is called the **child** process. The child process will be exactly the same as the parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.
- Both processes start execution from the next line of code i.e., the line after the fork() call. Let's look at an example:

```
//example.c
#include <stdio.h>
void main() {
   int val;
   val = fork();  // line A
   printf("%d",val);  // line B
}
```

- When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the fork() call.
- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.
- This means that according to the above program, the output of parent process will be the **process ID**of the child process and the output of the child process will be 0.

- **Join Command :**

The join command in UNIX is a command line utility for joining lines of two files on a common field. It can be used to join two files by selecting fields within the line and joining the files on them. The result is written to standard output.

Join syntax :

Join [option]….. file1 file2

How to join two files

To join two files using the join command files must have identical join fields. The default join field is the first field delimited by blanks. For the following example there are two files college.txt and city.txt.s

```
cat college.txt
1 pvg
2 met
3 mit

cat city.txt
1 nashik
2 nashik
3 pune
```

These files share a join field as the first field and can be joined.

```
join college city.txt
1 pvg nashik
2 met nashik
3 mit pune
```

- ***Exec()***
- The exec() system call is also used to create processes. But there is one big difference between fork() and exec() calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.
- It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist.

- There are many flavors of exec() in UNIX, one being exec1() which is shown below as an example:

```
//example2.c
#include
void main() {
  execl("/bin/ls", "ls", 0); // line A
  printf("This text won't be printed unless an error occurs in exec().");
}
```

- As shown above, the first parameter to the execl() function is the address of the program which needs to be executed, in this case, the address of the **ls** utility in UNIX. Then it is followed by the name of the program which is **ls** in this case and followed by optional arguments. Then the list should be terminated by a NULL pointer (0).

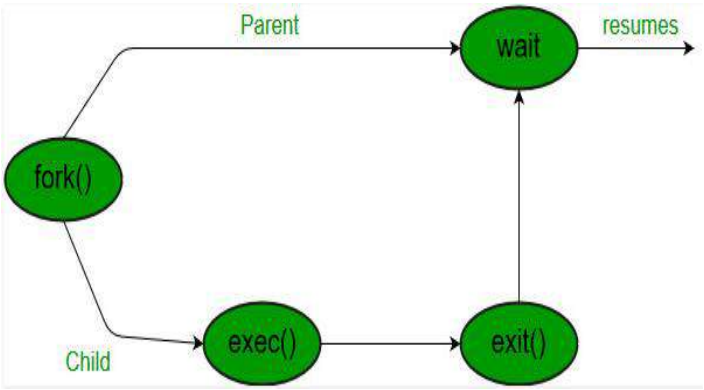
- When the above example is executed, at line A, the **ls** program is called and executed and the current process is halted. Hence the printf() function is never called since the process has already been halted. The only exception to this is that, if the **execl()** function causes an error, then the printf()function is executed.

- **Wait ()**

  A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.

  Child process may terminate due to any of these:

  - It calls exit();
  - It returns (an int) from main
  - It receives a signal (from the OS or another process) whose default action is to terminate.



``

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int **status*);

pid_t waitpid(pid_t *pid*, **int ****status*, **int** *options*);

int waitid(idtype_t *idtype*, **id_t** *id*, **siginfo_t *** infop , **int** options);

System Calls vs Library Functions

- A system call is executed in the kernel
  - p = getpid();
- A library function is executed in user space
  - n = strlen(s);
- Some library calls are implemented with system calls
  - printf() really calls the write() system call
- Programs use both system calls and library functions

8. Algorithms :

Note : you should write algorithm as per your program

9. Conclusion :

Thus , the process system call program is implemented and studied various system call.

References :

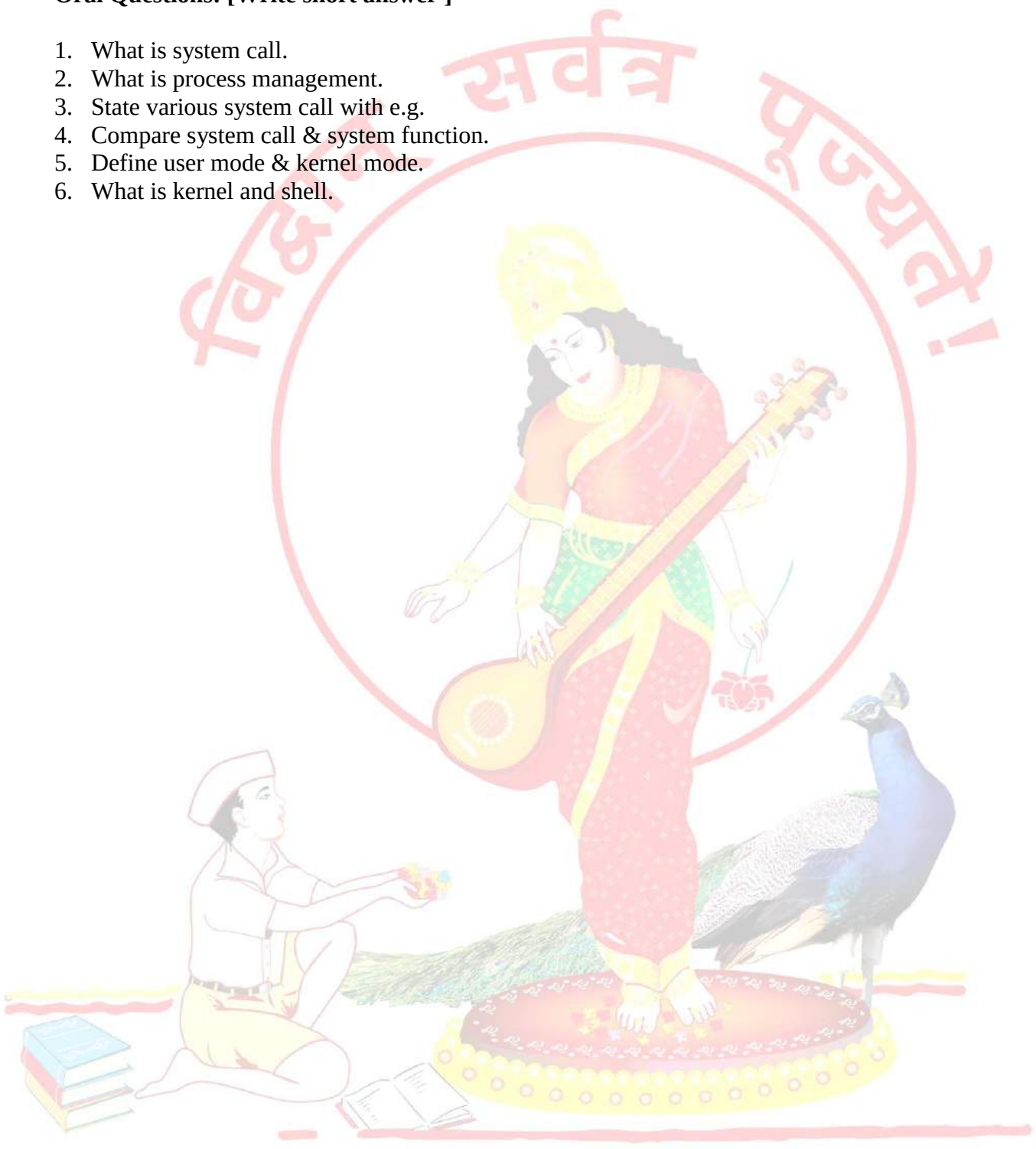
https://en.wikipedia.org/wiki/System_call
https://en.wikipedia.org/wiki/Process_management_(computing)
https://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture25.pdf
https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html
https://www.thegeekstuff.com/2012/03/c-process-control-functions/

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is system call.
2. What is process management.
3. State various system call with e.g.
4. Compare system call & system function.
5. Define user mode & kernel mode.
6. What is kernel and shell.



GROUP - C

EXPERIMENT NO : 13

1. Title:

Study assignment on process scheduling algorithms in Android and Tizen.

2. Objectives :

- To understand Android OS
- To understand Tizen OS
- To understand Concept of process management

3. Problem Statement :

Study assignment on process scheduling algorithms in Android and Tizen.

4. Outcomes:

After completion of this assignment students will be able to:

- Knowledge of Android and tizen OS
- Study of process management in android and tizen OS.
- Application of android and tizen os

5. Software Requirements:

Android SDK

6. Hardware Requirement:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Android OS :

- **Android** is a mobile operating system developed by Google, based on a modified version of the Linux kernel and other open source software and designed primarily for touchscreen mobile devices such as smartphones and tablets. In addition, Google has further developed Android TV for televisions, Android Auto for cars, and Android Wear for wrist watches, each with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics.
- Initially developed by Android Inc., which Google bought in 2005, Android was unveiled in 2007, with the first commercial Android device launched in September 2008. The operating

system has since gone through multiple major releases, with the current version being 8.1 "Oreo", released in December 2017.

- The android is a powerful operating system and it supports large number of applications in Smrtphones. These applications are more comfortable and advanced for the users. The hardware that supports android software is based on ARM architecture platform. The android is an open source operating system means that it's free and any one can use it. The android has got millions of apps available that can help you managing your life one or other way and it is available low cost in market at that reasons android is very popular.
- The android development supports with the full java programming language. Even other packages that are API and JSE are not supported. The first version 1.0 of android development kit (SDK) was released in 2008 and latest updated version is jelly bean.

Some android versions :

- Gingerbread (2.3)
- Honeycomb (3.0)
- Ice Cream Sandwich (4.0)
- Jelly Bean (4.3/4.2/4.1)
- KitKat (4.4)
- Lollipop (5.0)
- Marshmallow (6.0)
- Nougat (7.0)
- Oreo (8.0)

Advantages :

1. Support 2D & 3D Graphics
2. Support multiple language
3. Java support
4. Faster web browser
5. Support audio , video etc

Disadvantages :

1. Slow response
2. Heat
3. Advertisement etc

Tizen OS :

- **Tizen** is a mobile operating system developed by Samsung that runs on a wide range of Samsung devices, including smartphones; tablets; in-vehicle infotainment (IVI) devices; smart televisions; smart cameras; smartwatches; Blu-ray players; smart home appliances (refrigerators, lighting, washing machines, air conditioners, ovens/microwaves); and robotic vacuum cleaners.

- In 2010 Samsung was developing the Samsung Linux Platform (SLP) for the LiMo Foundation, whilst Intel and Nokia were leading the MeeGo project, another open source Linux mobile OS. In 2011 the MeeGo project was abandoned by its peers with Intel joining forces with Samsung to create Tizen, a new project based on code from SLP. The Linux Foundation also cancelled support of MeeGo in favor of Tizen. In 2013 Samsung merged its homegrown Bada project into Tizen.

- The Tizen Association was formed to guide the industry role of Tizen, including requirements gathering, identifying and facilitating service models, and overall industry marketing and education.[6] Members of the Tizen Association represent major sectors of the mobility industry. Current members include: Fujitsu, Huawei, Intel, KT, NEC Casio, NTT DoCoMo, Orange, Panasonic, Samsung, SK Telecom, Sprint and Vodafone

- Samsung announced in November 2016 that they would be collaborating with Microsoft to bring .Net support to Tizen.

- Samsung is currently the only Tizen member developing and using the operating system.

- As of 2017 Tizen is second largest smartwatch platform, behind watchOS and ahead of Android Wear

- On January 1, 2012, the LiMo Foundation was renamed Tizen Association. The Tizen Association works closely with the Linux Foundation, which supports the Tizen open source project.

- April 30, 2012: Tizen 1.0 released.
- February 18, 2013: Tizen 2.0 released.
- May 20, 2017: Tizen 3.0 released

- The first Tizen tablet was shipped by Systena in October 2013. Part of a development kit exclusive to Japan, it was a 10-inch quad-core ARM with 1920×1200 resolution

- On February 21, 2016, Samsung announced the Samsung Connect Auto, a connected car solution offering diagnostic, Wi-Fi, and other car-connected services. The device plugs directly into the OBD-II port underneath the steering wheel

Android vs Tizen Operating system :

Note : No need to draw above diagram. It is just to get difference between them.

Android vs Tizen Operating system :

- **Easy and Convenient Navigation:** Scrolling and navigation becomes smooth with Tizen
- **Fast and Lightweight:** Tizen Operating System is easy to operate and fast as compared to Google's Android Wear
- **Visual Effects:** Tizen extends 3D visual effects of various gaming apps installed on the device
- **UI:** TouchWiz UI
- **Resizable boxes:** One of the amazing features of Tizen is its ability to dynamically resize the icons on screen to display more information or less
- **Enhanced Processors:** Tizen 3.0 will bring 64 bit processors with it, compatible with x86 processors and 64 bit RAM, which Google is also anticipating with its update.
- **Tizen vs. Android Gaming Platform:** Tizen 3.0 will be able to make use of Vulkan API's and will prove to be a good gaming platform unlike Android.
- **Supporting Devices:** Tizen is being used in smart TV's, refrigerators, smart watches, smart phones, washing machines, light bulbs, vacuum cleaners while Android is visible only in smart phones, computers or smart watches.

- **IoT Devices:** Tizen 3.0 is compatible with Artik cloud which will extend cloud services for IoT devices.
- **Battery Consumption:** Samsung's devices with Tizen OS consume less power than Android devices according to mobile experts.
- **Pricing:** Devices with Tizen support will be made available at various price points but focus will be on lower end markets. Unlike Android, that has its presence in both upper as well as lower end markets.

Advantages of using Tizen OS

- It is an open source Operating System
- The OS is Compatible with various mobile platform. Application built on Tizen can be launched on iOS and Android too with few changes.
- The Tizen OS is so Flexible to offer many applications and adapt too, with little changes
- Immense personalization capability supported by ARM x86 processor

➢ PROCESS SCHEDULING ALGORITHMS IN ANDROID AND TIZEN OS :

- Normal scheduling

Android is based on Linux and uses the Linux kernel's scheduling mechanisms for determining scheduling policies. This is also true for Java code and threads.
The Linux's time sliced scheduling policy combines static and dynamic priorities. Processes can be given an initial priority from 19 to -20 (very low to very high priority). This priority will assure that higher priority processes will get more CPU time when when needed. These level are however dynamic, low level priority tasks that do not consume their CPU time will fine their dynamic priority increased. This dynamic behaviour results is an overall better responsiveness.

In terms of dynamic priorities it is ensured that lower priority processes will always have a lower dynamic priority than processes with real-time priorities.

Android uses two different mechanisms when scheduling the Linux kernel to perform process level scheduling

- Real-time scheduling

The standard Linux kernel provides two real-time scheduling policies, SCHED_FIFO and SCHED_RR. The main real-time policy is SCHED_FIFO. It implements a first-in, first-out scheduling algorithm. When a SCHED_FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. It has no timeslices. All other tasks of lower priority will not be scheduled until it relinquishes the CPU. Two equal-priority SCHED_FIFO tasks do not preempt each other. SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted timeslices based on their priority and run until they exhaust their timeslice. Non-real-time tasks use the SCHED_NORMAL scheduling policy (older kernels had a policy named SCHED_OTHER).

- Thread Scheduling

A thread scheduler decides which threads in the Android system should run, when, and for how long

Android's thread scheduler uses two main factors to determine the scheduling:

- Niceness Values
- Control Groups (Cgroups)

Niceness Values

- a thread with a higher niceness value will run less often than those with a lower niceness value (this sounds paradoxical)
- niceness value has the range of -20 (most prioritized) to 19 (least prioritized); default value is 0
- a new Thread inherits its priority from the thread where it is started
- it is possible to change the priority via:
  - *thread.setPriority(int priority)* - values: 0 (least prioritized) to 10 (most prioritized)
  - *process.setThreadPriority(int priority)* - values: -20 (most prioritized) to 19 (least prioritized)

Control Groups (Cgroups)

- Android has multiple control groups. The most important are:
  - the Foreground Group
  - the Background Group
- every thread belongs to a thread control group (e.g. Foreground Group)
- threads in the different control groups are allocated different amounts of CPU execution time
- threads in the Foreground Group receive a lot more execution time than threads in the Background Group

- if an application runs at the Foreground or Visible process level (see above), the threads created by that application will belong to the Foreground Group
- all threads belonging to applications which are not currently running in the foreground are implicitly moved to the Background Group

- Priority Based Pre-Emptive Task Scheduling for Android Operating System

The key concept present in any operating system which allows the system to support multitasking, multiprocessing, etc. is Task Scheduling . Task Scheduling is the core which refers to the way the different processes are allowed to share the common CPU. Scheduler and dispatcher are the softwares which help to carry out this assignment . Android operating system uses O (1) scheduling algorithm as it is based on Linux Kernel 2.6. Therefore the scheduler is names as Completely Fair Scheduler as the processes can schedule within a constant amount of time, regardless of how many processes are running on the operating system . Pre-emptive task scheduling involves interrupting the low priority tasks when high priority tasks are present in the queue. This scheduling is particularly used for mobile operating system as the CPU utilization is medium, turnaround time and response time is high. Mobile phones are required to meet specific time deadlines for the tasks to occur.

- Fixed-priority pre-emptive scheduling

Fixed-priority preemptive scheduling is a scheduling system commonly used in real-time systems. With fixed priority preemptive scheduling, the scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute.

The preemptive scheduler has a clock interrupt task that can provide the scheduler with options to switch after the task has had a given period to execute—the time slice. This scheduling system has the advantage of making sure no task hogs the processor for any time longer than the time slice. However, this scheduling scheme is vulnerable to process or thread lockout: since priority is given to higher-priority tasks, the lower-priority tasks could wait an indefinite amount of time. One common method of arbitrating this situation is aging, which gradually increments the priority of waiting processes and threads, ensuring that they will all eventually execute. Most Real-time operating systems (RTOSs) have preemptive schedulers. Also turning off time slicing effectively gives you the non-preemptive RTOS.

Preemptive scheduling is often differentiated with cooperative scheduling, in which a task can run continuously from start to end without being preempted by other tasks. To have a task switch, the task must explicitly call the scheduler. Cooperative scheduling is used in a few RTOS such as Salvo or TinyOS.

- **Dynamic priority pre-emptive scheduling**

earliest-deadline first scheduling: a job's priority is inversely proportional to its absolute deadline. The difference between deadline monotonic scheduling and earliest-deadline first scheduling is that DM is a static priority algorithm, EDF is a dynamic priority algorithm.[3]EDF can guarantee that all deadlines are met provided that the total CPU utilization is less than 1. .

8. Conclusion :

Thus , I have studied concept of process scheduling of Android and Tizen Operating System.

References :

https://en.wikipedia.org/wiki/Android_(operating_system)
https://en.wikipedia.org/wiki/Tizen
https://github.com/dweinstein/android_notes/wiki/AndroidScheduling
https://arxiv.org/ftp/arxiv/papers/1304/1304.7889.pdf

NOTE : don't write references.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is android os.
2. What is tizen os.
3. Compare Android vs Tizen.
4. What is process management.
5. State scheduling in android.
6. Application of Android and Tizen OS.

GROUP - D

GROUP - D

GROUP - D

EXPERIMENT NO : 14

1. **Title:**

   Write a java program to implement Page Replacement Policies LRU & OPT.

2. **Objectives :**

   - To understand Page replacement policies
   - To understand paging concept
   - To understand Concept of page fault, page hit, miss, hit ratio etc

3. **Problem Statement :**

   Write a java program to implement Page Replacement Policies LRU & OPT..

4. **Outcomes:**

After completion of this assignment students will be able to:

   - Knowledge of Page Replacement Policies in OS
   - Implemented LRU & OPT Page replacement Policies
   - Understood concept of paging.

5. **Software Requirements:**

   Latest jdk., Eclipse

6. **Hardware Requirement:**

   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

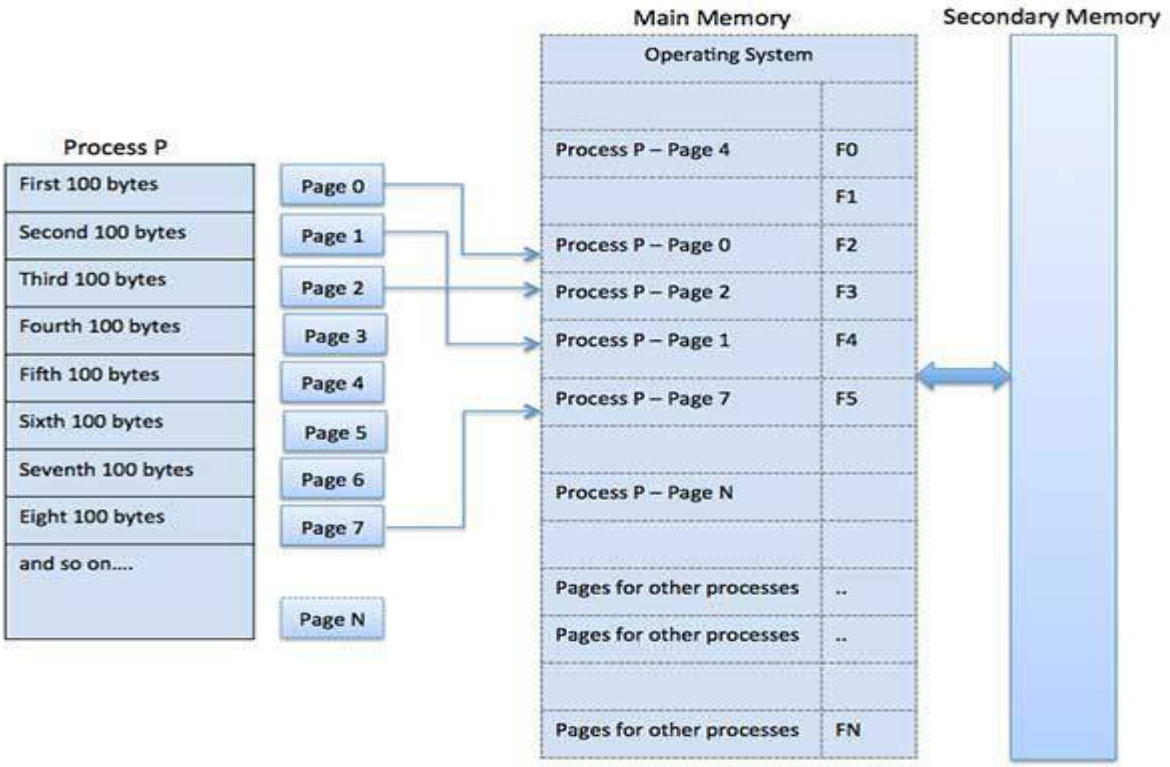
7. **Theory Concepts:**

   Paging :

   A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

   Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

   Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

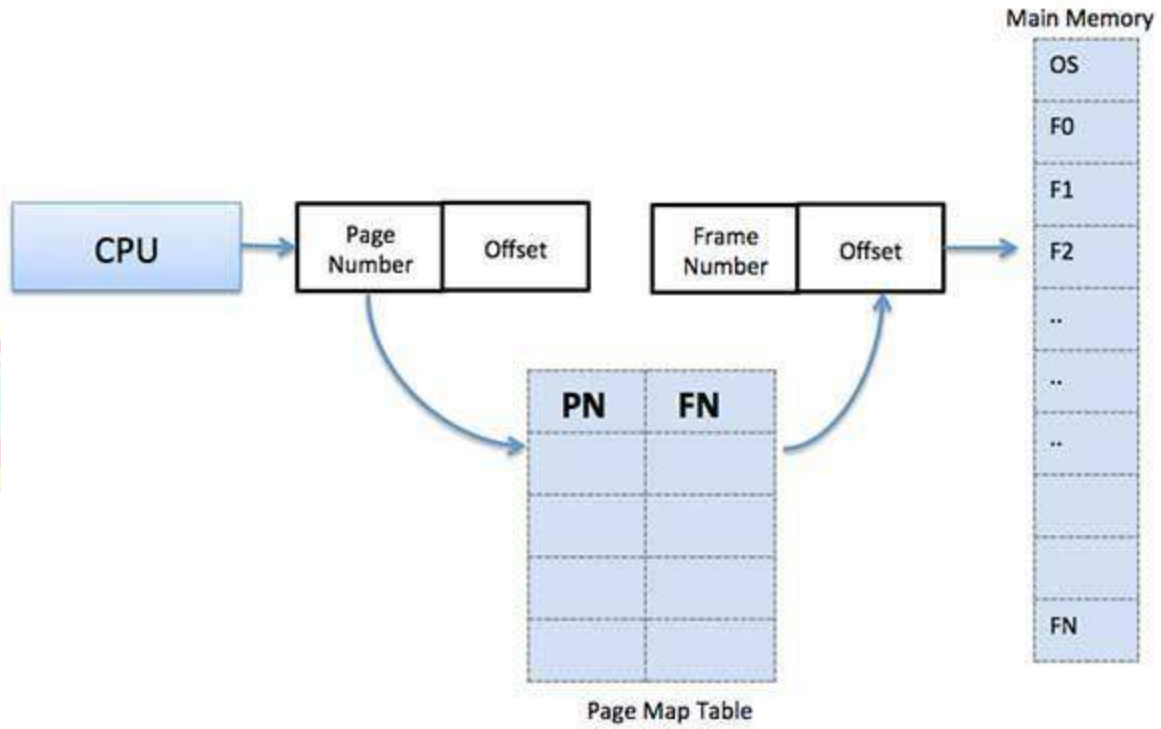
Page address is called **logical address** and represented by **page number**and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

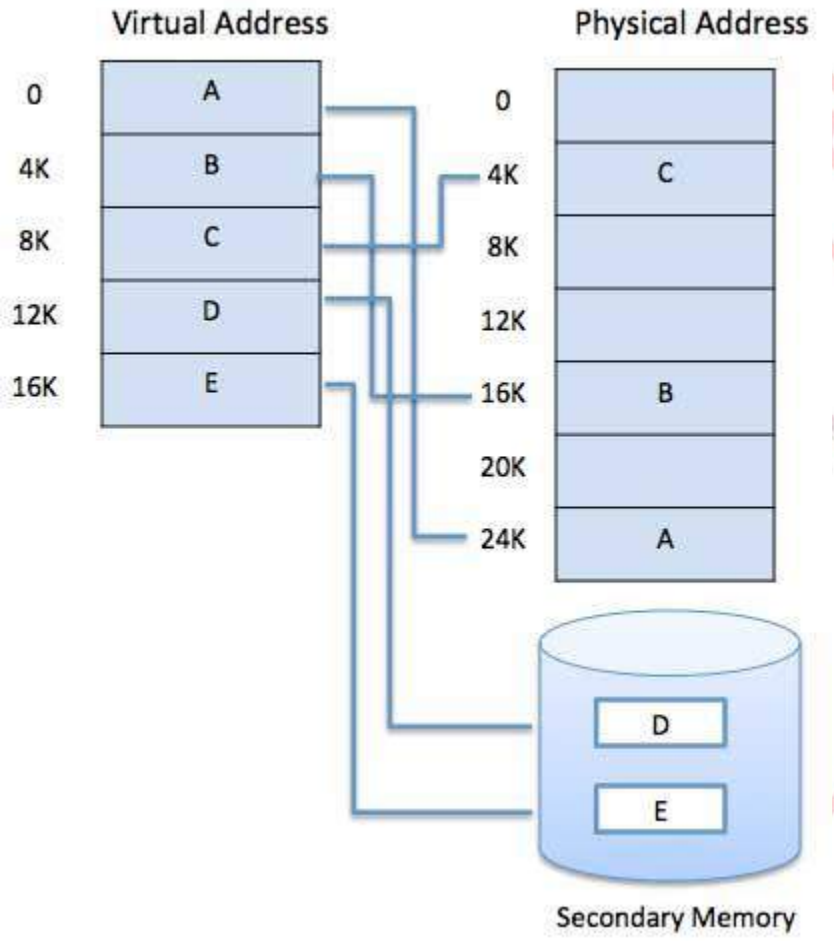
The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.

- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

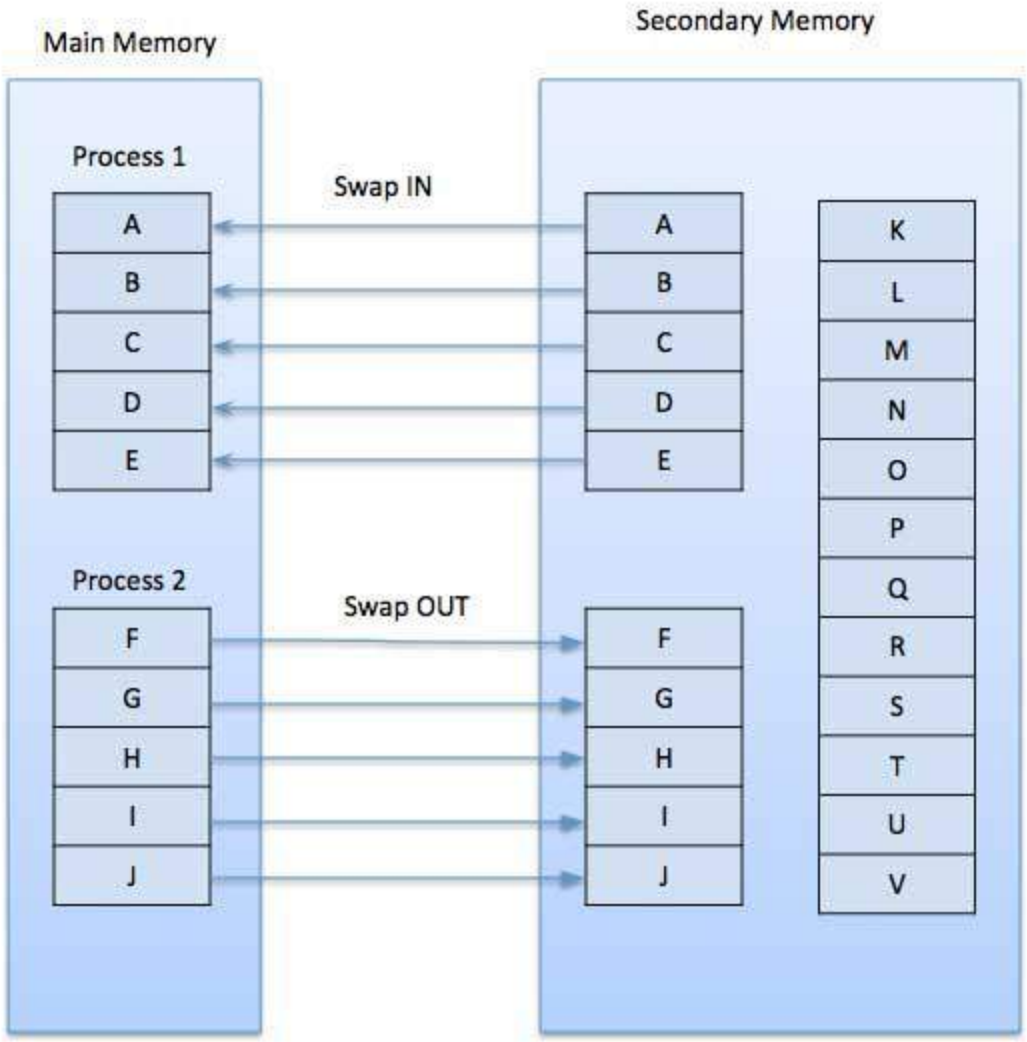
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −



Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging −

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

``

Page Replacement Algorithm :

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Page fault :

A **page fault** (sometimes called #PF, PF or hard **fault**) is a type of exception raised by computer hardware when a running program accesses a memory **page**that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.

Page hit :

A **hit** is a request to a web server for a file, like a web **page**, image, JavaScript, or Cascading Style Sheet. When a web **page** is downloaded from a server the number of "**hits**" or "**page hits**" is equal to the number of files requested.

Page frame :

The **page frame** is the storage unit (typically 4KB in size) whereas the **page** is the contents that you would store in the storage unit ie the **page frame**. For eg) the RAM is divided into fixed size blocks called **page frames** which is typically 4KB in size, and each **page frame** can store 4KB of data ie the **page**.

Page table :

A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.

Reference String :

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size, we need to consider only the page number, not the entire address.

- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

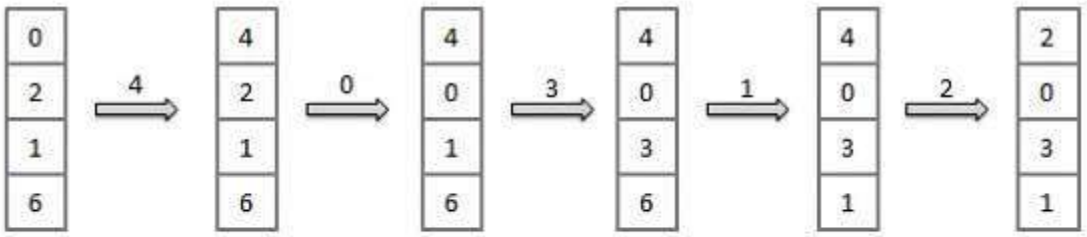
- For example, consider the following sequence of addresses − 123,215,600,1234,76,96
- If page size is 100, then the reference string is 1,2,6,12,0,0

First In First Out (FIFO) algorithm :

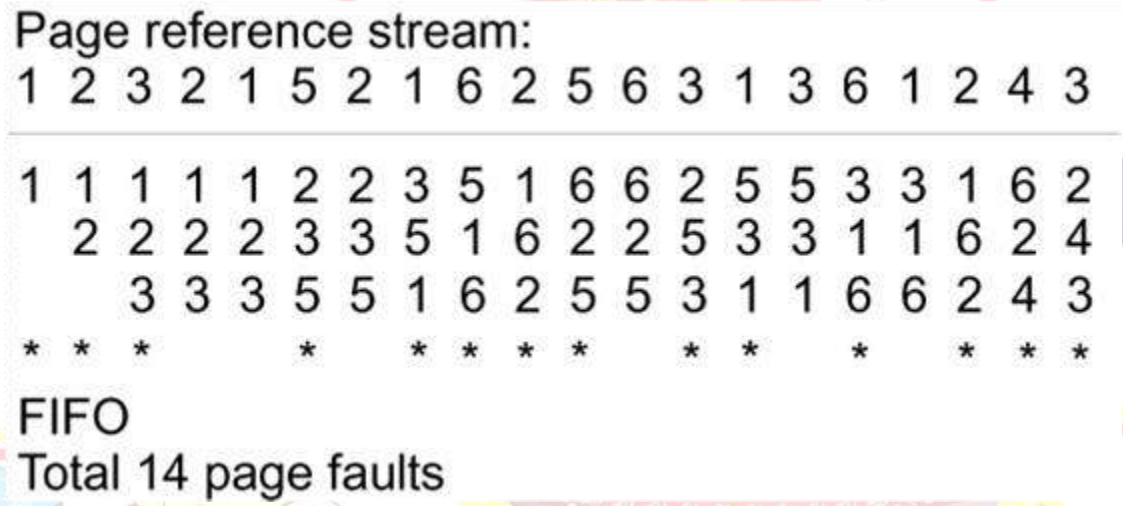
- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x



Fault Rate = 9 / 12 = 0.75



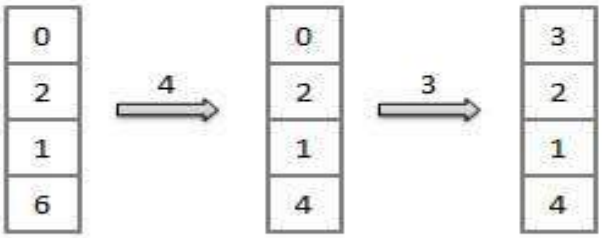
Note : you can take other example also. This just for reference. (you must calculate page fault, page hit and hit ratio)

Optimal Page algorithm :

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



Fault Rate = 6 / 12 = 0.50

Page reference stream:

1	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6	2	2	2
	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	4	4
		3	3	3	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3
*	*	*			*			*				*	*				*	*	

Optimal
Total 9 page faults

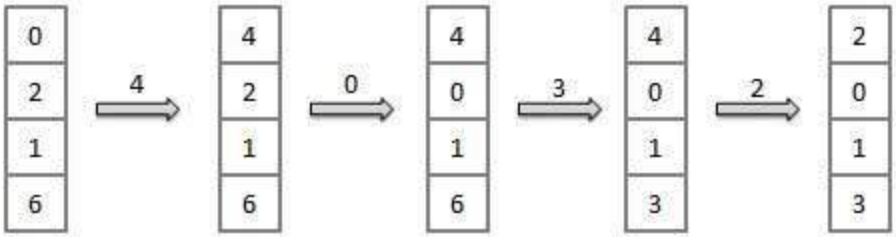
Note : you can take other example also. This just for reference. (you must calculate page fault, page hit and hit ratio)

Least Recently Used (LRU) algorithm :

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

Page reference stream:

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

1	1	1	1	3	2	1	5	2	1	6	2	5	6	6	1	3	6	1	2
	2	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4
		3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
*	*	*			*			*		*		*	*				*	*	*

LRU

Total 11 page faults

Note : you can take other example also. This just for reference. (you must calculate page fault, page hit and hit ratio)

Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used(LFU) algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

``

Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

8. Conclusion :

Thus , I have implemented page replacement policies `LRU and OPT.

References :

https://en.wikipedia.org/wiki/Paging
https://en.wikipedia.org/wiki/Page_replacement_algorithm
https://www.geeksforgeeks.org/operating-system-paging/
https://www.tutorialspoint.com/operating_system/os_virtual_memory.htm

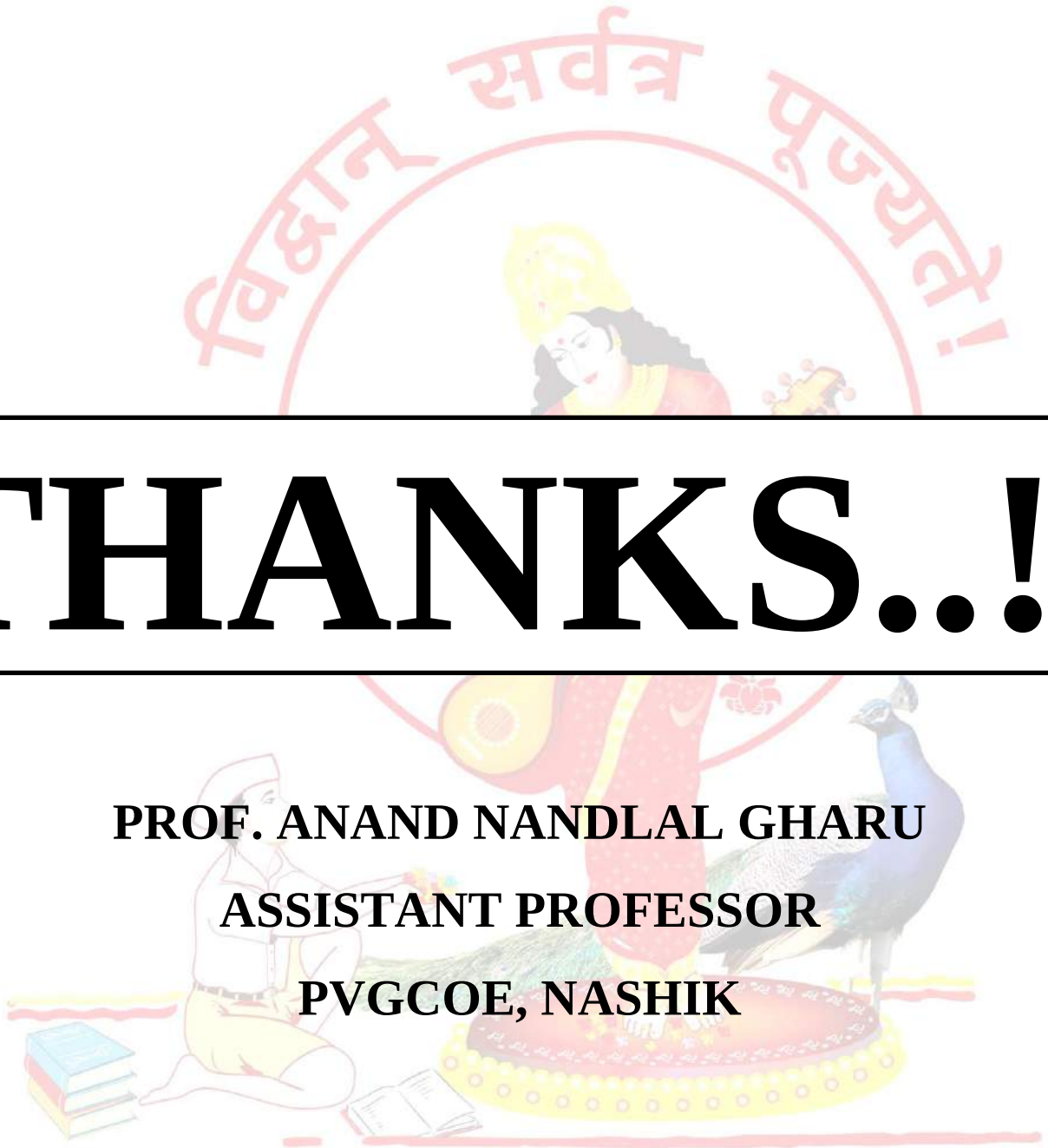
NOTE : don't write references.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Oral Questions: [Write short answer]

1. What is paging.
2. What is page replacement policies.
3. Define page table, page hit, page fault, page reference.
4. What is FIFO page replacement.
5. What is LRU and OPT page replacement.
6. State virtual memory.
7. Define demand paging.



THANKS..!

PROF. ANAND NANDLAL GHARU

ASSISTANT PROFESSOR

PVGCOE, NASHIK

Blog : anandgharu.wordpress.com