Gokhale Education Society's
R. H. Sapat College of Engineering, Management Studies & Research, Nasik-422005
Department of Computer Engineering

Laboratory Manual

System Programming & Operating System Lab

For

Third Year Students
Dept: Computer Engineering

Author GESRHSCOE, Nasik

FOREWORD

LABORATORY MANUAL CONTENTS

This manual is intended for the Third year students of Computer branches in the subject of System Programming and Operating System. This manual typically contains practical/Lab Sessions related SPOS covering various aspects related the subject to enhanced understanding. Main aim of this course is to understand and solve logical & Operating System problems through Java language. Strengthen knowledge of a Assembly programming language. Further develop your skills in software development using a Object Oriented language.

This course will also prepare students with the necessary programming background for Data Structures using C/C++ and Java programming courses. We have made the efforts to cover various aspects of the subject covering these labs encompass the regular materials as well as some advanced experiments useful in real life applications. Programming aspects will be complete in itself to make it meaningful, elaborative understandable concepts and conceptual visualization.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus, as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions

Ms. R.D.Narwade/Mr. G K Bhambre
Asst. Professor, Comp Dept

Dr. D.V. Patil
HOD, Comp Dept.

General Guidelines and Rules

Guidelines

1. Students must read the lab module and make necessary preparation for each session **PRIOR TO** coming to the lab.
2. Each lab session will last for approximately TWO hours, which is divided into the following time slots:

Item	Time
Demonstration ▪ Instructor will give a brief demonstration during the allocated time and students will be given some hands-on exercises. Hence the students are expected to fully utilize this time to enquire the instructor/tutors regarding the subject matter.	40 minutes
Lab assignment ▪ Students must:: o Answer all the given questions o Report to the lab instructor and demonstrator to submit the answers before the lab ends ▪ No take-home assignment will be allowed. ▪ Solutions will ONLY be given after the module is covered by all groups.	80 minutes

DOs and Don'ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.
2. All the students should sit according to their roll numbers starting from their left to right.
3. All the students are supposed to enter the terminal number in the log book.
4. Do not change the terminal on which you are working.
5. All the students are expected to get at least the algorithm of the program/concept to be implemented.
6. Strictly observe the instructions given by the teacher/Lab Instructor.

Instruction for Laboratory Teachers::

1. Submission related to whatever lab work has been completed should be done during the next lab session. The immediate arrangements for printouts related to submission on the day of practical assignments.
2. Students should be taught for taking the printouts under the observation of lab teacher.

3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

Rules

All students are bound to adhere to the following rules. Failure of complying with any rules will be penalized accordingly.

1. Students must be in the lab before the lab activities started. No late coming is tolerated without prior consent from the respective lecturer. Failure to do so may **eliminate** your mark for that particular lab session.
2. During lab session any form of portable data storage and retrieval devices is **prohibited**. If found, then we reserve the right to confiscate the item and devoid your mark for that particular lab session. Collection of the confiscated item(s) requires approval from Deputy Dean of Academic Affairs.
3. Duplicated lab assignment: the source and duplicate will be considered **void**.
4. Submission procedure:
   a) Create a folder in the drive of your workstation. Name the folder with your **ID number and your name**. Example: 04xxxxxx Rahul Joshi
   b) Save all your answers and source codes inside the folder. Name the files according to the question, example: question1.cpp/question1.txt.

Report your completed lab assignment to the instructor/demonstrator for inspection and assessment

Special Instruction Before Each Lab Session

1. Create 2 folders in the home directory:
   a. 1 folder for **lab exercise and hand-on experience**:
      ✓ Name the folder with your **ID number and your name with (demo)**. Example: 04xxxxxx Rahul Joshi (demo)
   b. 1 folder for **submission** (refer to **Section B**)
2. This lab is aimed to apply the theories that you have learnt in Object Oriented Programming in C++, not intended to teach you everything about C++. Hence, you need to do your own homework.
3. Lab module is designed as a guideline, not a comprehensive set of notes and exercises. Read your theory notes and books pertaining to the topics to be covered (Refer to '**Objective'** section in each of the lab module).

How to get lab manual

1. You can obtain the lab manual from the selected lab representatives

2. Please get the materials ready before coming to the lab.

H/w, S/w Requirement:

Operating System recommended :- 64-bit Open source Linux or its derivative

Programming tools recommended: - Open Source C++ Programming tool like G++/GCC.

First assignment is compulsory. Set of suggested assignment list is provided in 3 groups- A, B, and C. Instructor is suggested to design assignments list by selecting/designing at least **12** suitable assignments from group A, B, and C- **compulsory assignment, 5** from group A, **4** from group B, **3** from group C.

Course Objectives: -

1. To implement basic language translator by using various needed data structures
2. To implement basic Macroprocessor.
3. To design and implement Dynamic Link Libraries
4. To implement scheduling schemes

Course Outcomes: -

1. Understand the internals of language translators
2. Handle tools like LEX & YACC
3. Understand the Operating System internals and functionalities with implementation point of view

Reference Books :-

Paul Gries Jennifer Campbll, Jason Montojo, ―Practical Programming Second Edition‖, SPD, ISBN: 978-93-5110-469-8

SUBJECT INDEX

Sr No.	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
1.	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.	Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of 'Java' program.
3.	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.
4.	Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java.
5.	Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file
	Group C
1.	Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive)
2.	Write a Java program to implement Banker's Algorithm
3.	Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming).
4.	Study assignment on process scheduling algorithms in Android and Tizen.
	Group D
1.	Write a Java Program (using OOP features) to implement paging simulation using 1. Least Recently Used (LRU) 2. Optimal algorithm

1. PASS-1 ASSEMBLER

Aim: To implement Pass-1 Assembler.

Problem Statement : 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

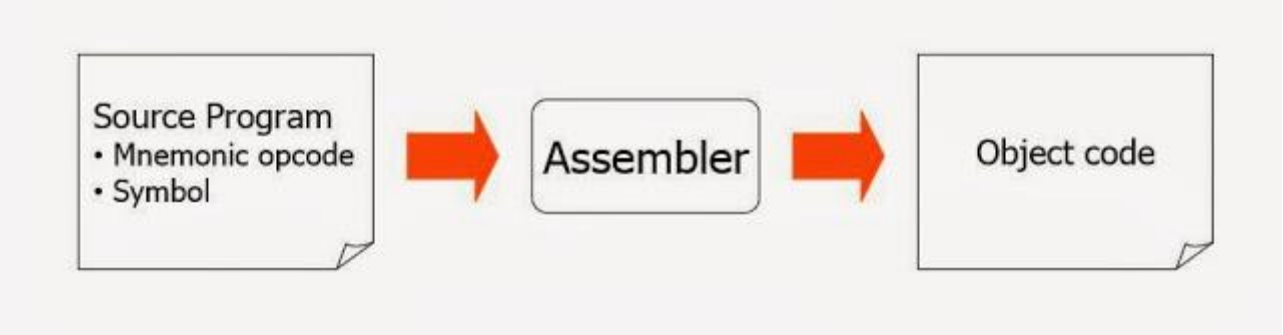
Theory:

Assembly Language

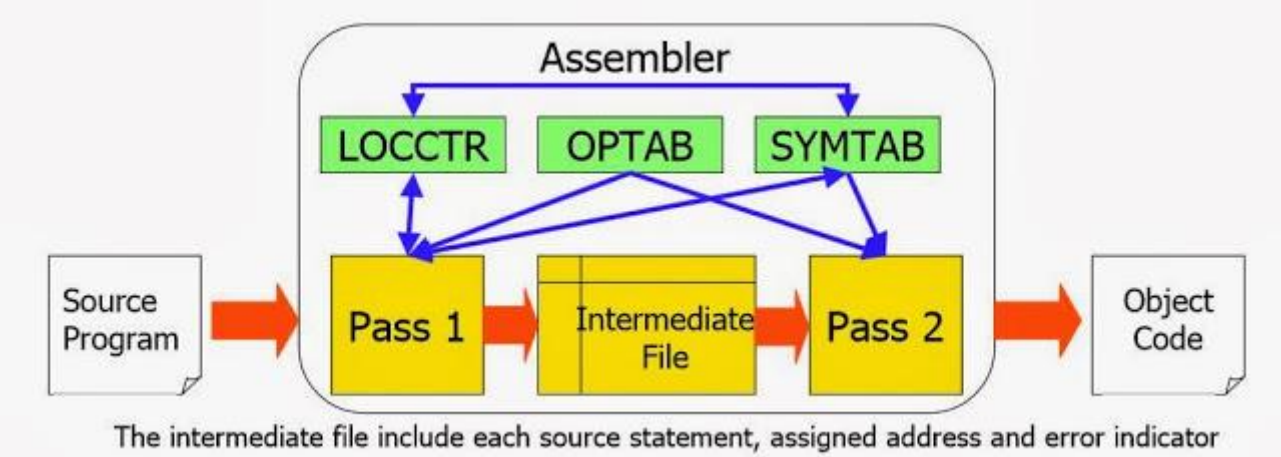
An assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Each assembly language is specific to a particular computer architecture, in contrast to most high-level programming languages, which are generally portable across multiple architectures, but require interpreting or compiling. Assembly language uses a mnemonic to represent each low-level machine operation or opcode. Some op-codes require one or more operands as part of the instruction.

Assembler

Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.



An assembler is a translator that translates an assembler program into a conventional machine language program. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created.



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

Assembler Directives

- Assembler directives are pseudo instructions
  - They will not be translated into machine instructions.
  - They only provide instruction/direction/information to the assembler.
- Basic assembler directives :
  - **START :** Specify name and starting address for the program
  - **END :** Indicate the end of the source program.
  - **EQU :** The EQU directive is used to replace a number by a symbol. For example: MAXIMUM EQU 99. After using this directive, every appearance of the label "MAXIMUM" in the program will be interpreted by the assembler as the number 99 (MAXIMUM = 99). Symbols may be defined this way only once in the program. The EQU directive is mostly used at the beginning of the program.

Three Main Data Structures

- Operation Code Table (OPTAB)
- Location Counter (LOCCTR)
- Symbol Table (SYMTAB)

Instruction formats

- Addressing modes · Direct addressing (address of operand is given in instruction itself) · Register addressing (one of the operand is general purpose register) · Register indirect addressing (address of operand is specified by register pair) · Immediate addressing (operand - data is specified in the instruction itself) · Implicit addressing (mostly the operation operates on the contents of accumulator)
- Program relocation · It is desirable to load and run several programs and resources at the same time · The system must be able to load programs into memory wherever there is room · The exact starting address of the program is not known until load time. · The assembler can identify (for the loader) those parts of the program that need modification. · An object program that contains this type of modification information necessary to perform modification is called a re-locatable program.

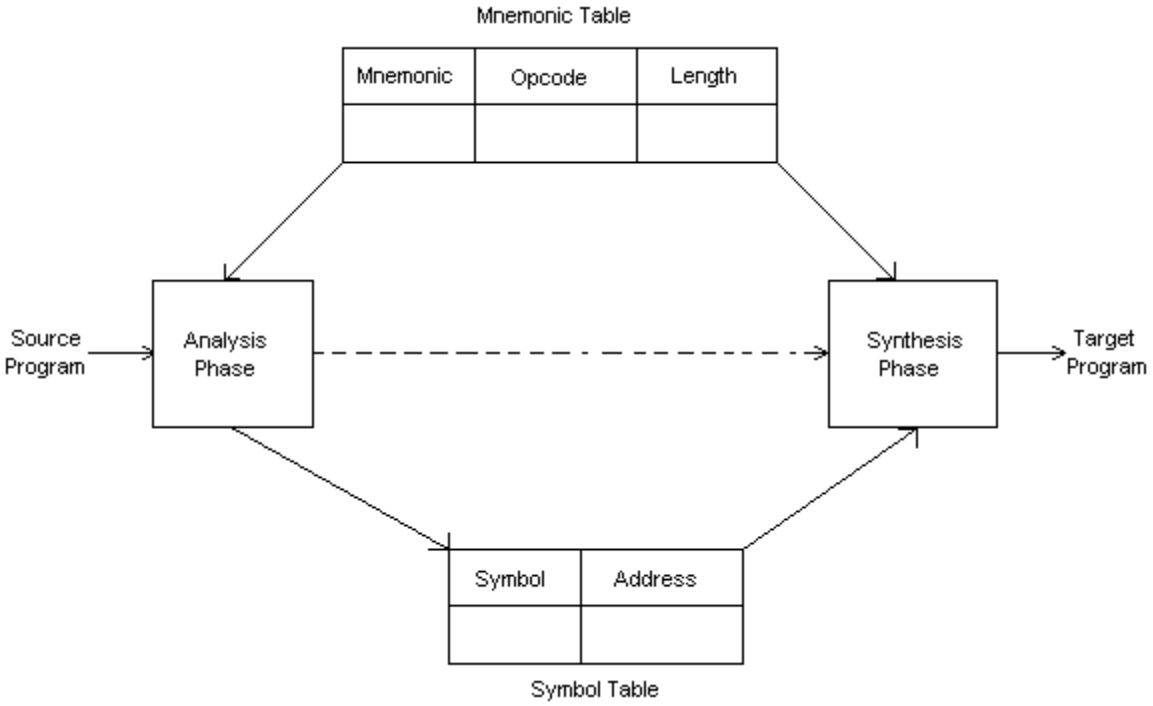
Literal

It is convenient for the programmer to be able to write the value of a constant operand as a part of the instruction that uses it. Such an operand is called a literal. In this assembler language notation, a literal is identified with the prefix '=', which is followed by a specification of the literal value.
The difference between literal operands and immediate operands

- for literal operand we use '=' as prefix, and with immediate operand we use '#' as prefix
- During immediate addressing, the operand value is assembled as part of the machine instruction, and there is no memory reference.
- With a literal, the assembler generates the specified value as a constant at some other memory location.

One-pass assemblers

A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references (the problem of forward referencing) and assemble code in one pass



Forward reference in one pass assembler

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.

Data structures for assembler:

Op code table

Looked up for the translation of mnemonic code

key: mnemonic code

Hashing is usually used once prepared, the table is not changed efficient lookup is desired
since mnemonic code is predefined, the hashing function can be tuned a priori The table may have the instruction format and length to decide where to put op code bits, operands bits, offset bits

for variable instruction size
used to calculate the address
Symbol table
Stored and looked up to assign address to labels

efficient insertion and retrieval is needed
deletion does not occur
Difficulties in hashing

non random keys
Problem

the size varies widely
pass 1: loop until the end of the program
1. Read in a line of assembly code
2. Assign an address to this line

increment N (word addressing or byte addressing)
3. Save address values assigned to labels

in symbol tables
4. Process assembler directives

constant declaration
space reservation

Algorithm for Pass 1 assembler:

```
begin
        if starting address is given
                LOCCTR = starting address;
        else
                LOCCTR = 0;
        while OPCODE != END do              ;; or EOF
                begin
                read a line from the code
                if there is a label
                        if this label is in SYMTAB, then error
                        else insert (label, LOCCTR) into SYMTAB
                search OPTAB for the op code
                if found
                        LOCCTR += N         ;; N is the length of this instruction (4 for MIPS)
                else if this is an assembly directive
                        update LOCCTR as directed
                else error
                write line to intermediate file
                end
        program size =  LOCCTR - starting address;
end
```

Algorithm 4.1 (Assembler First Pass)

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB [1] := 1;
   littab_ptr := 1;
2. While next statement is not an END statement
   (a) If label is present then
   this_label := symbol in label field;
   Enter (*this_label, loc_cntr*) in SYMTAB.
   (b) If an LTORG statement then
   (i) Process literals LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [*littab_ptr*−1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.
   (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (iii) POOLTAB [*pooltab_ptr*] := *littab_ptr*;
   (c) If a START or ORIGIN statement then
   loc_cntr := value specified in operand field;
   (d) If an EQU statement then
   (i) *this_addr* := value of <*address spec*>;
   (ii) Correct the symtab entry for *this_label* to (*this_label, this_addr*).
   (e) If a declaration statement then
   (i) *code* := code of the declaration statement;
   (ii) *size* := size of memory area required by DC/DS.
   (iii) *loc_cntr* := *loc_cntr* + *size*;
   (iv) Generate IC '(DL, *code*) ...'.
   (f) If an imperative statement then
   (i) *code* := machine opcode from OPTAB;
   (ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;
   (iii) If operand is a literal then
   this_literal := literal in operand field;
   LITTAB [*littab_ptr*] := *this_literal*;
   littab_ptr := *littab_ptr* + 1;
   else (i.e. operand is a symbol)
   this_entry := SYMTAB entry number of operand;
   Generate IC '(IS, *code*)(S, *this_entry*)';
3. (Processing of END statement)
   (a) Perform step 2(b).
   (b) Generate IC '(AD,02)'.
   (c) Go to Pass II.

Input:

```
START 200
 MOVER AREG,='4'
 MOVEM AREG,A
 MOVER BREG,='1'
LOOP MOVER CREG,B
 LTORG
 ADD CREG,='6'
 STOP
A DS 1
B DS 1
 END
```

Expected Output: Symbol Table

A	208
LOOP	203
B	209

Intermediate Code

AD	01	C	200	
IS	04	1	L	1
IS	05	1	S	1
IS	04	2	L	2
IS	04	3	S	3
AD	05			
IS	01	3	L	3
IS	00			
DL	02	C	1	
DL	02	C	1	
AD	02			

Conclusion :

Thus we have implemented PASS-1 Assembler using Object oriented features

2.	PASS-2 ASSEMBLER

Aim: To design data structure for Pass-2 Assembler

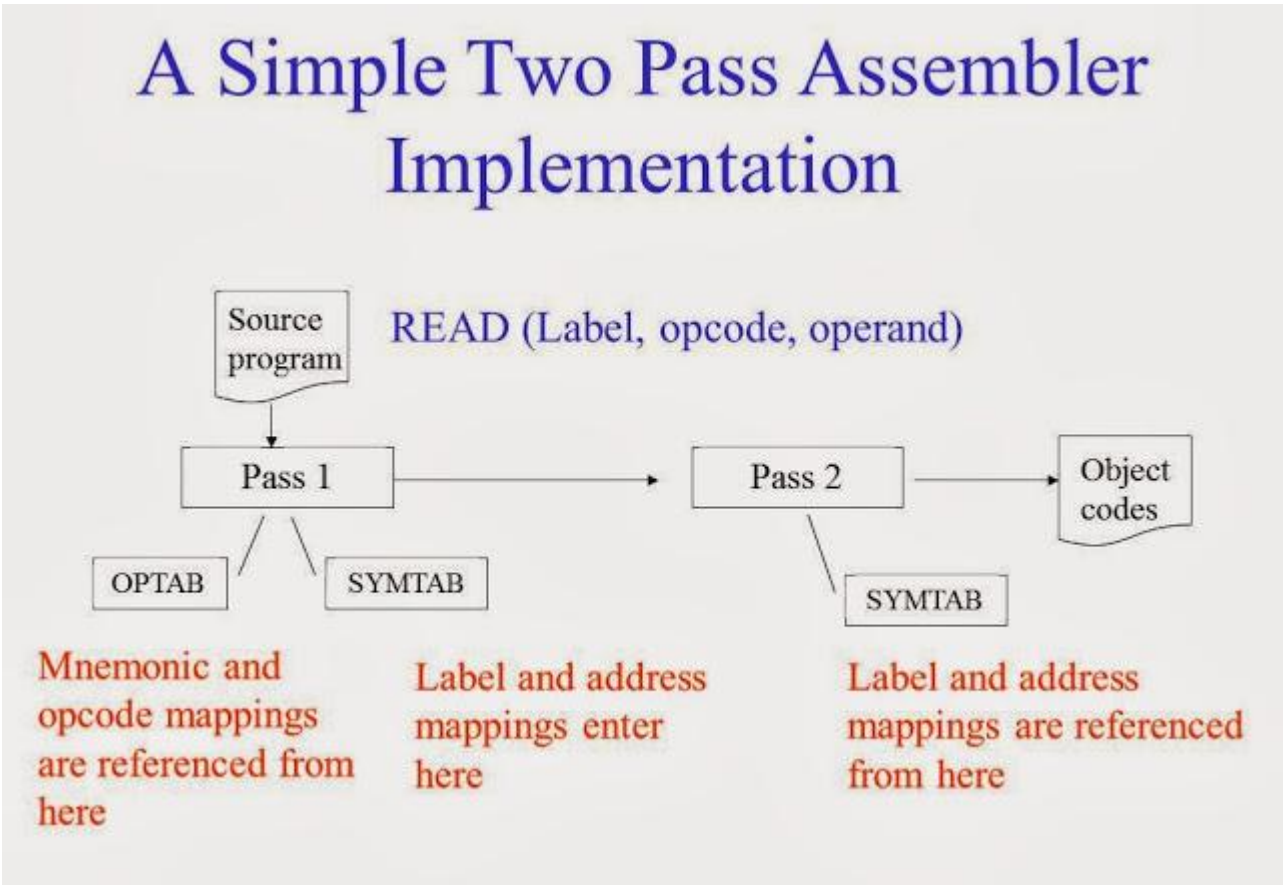
Problem Statement: Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment.

Theory:

Two-pass assemblers

The two pass assembler performs two passes over the source program. In the first pass, it reads the entire source program, looking only for label definitions. All the labels are collected, assigned address, and placed in the symbol table in this pass, no instructions as assembled and at the end the symbol table should contain all the labels defined in the program. To assign address to labels, the assembles maintains a Location Counter (LC).

In the second pass the instructions are again read and are assembled using the symbol table. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are**.**



Difference between One Pass and Two Pass Assemblers

The difference between one pass and two pass assemblers are:-

A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references (the problem of forward referencing) and assemble code in one pass. The one pass assembler prepares an intermediate file, which is used as input by the two pass assembler.

A two pass assembler does two passes over the source file (the second pass can be over an intermediate file generated in the first pass of the assembler). In the first pass all it does is looks for label definitions and introduces them in the symbol table (a dynamic table which includes the label name and address for each label in the source program). In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations into machine codes and so on.

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

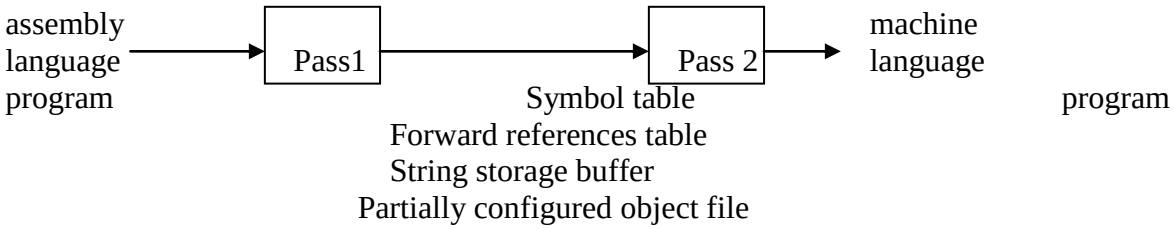


Figure 2. *A simple two pass assembler*.

Algorithm

```
begin

        if starting address is given
                LOCCTR = starting address;
        else
                LOCCTR = 0;
        while OPCODE != END do              ;; or EOF
                begin
                read a line from the code
                if there is a label
                        if this label is in SYMTAB, then error
                        else insert (label, LOCCTR) into SYMTAB
                search OPTAB for the op code
                if found
                        LOCCTR += N         ;; N is the length of this instruction (4 for MIPS)
                else if this is an assembly directive
                        update LOCCTR as directed
                else error
                write line to intermediate file
                end
        program size =  LOCCTR - starting address;
end
```

Solved Example:

```
1          START   200
2          MOVER   AREG, ='5'      200)  +04 1 211
3          MOVEM   AREG, A         201)  +05 1 217
4   LOOP   MOVER   AREG, A         202)  +04 1 217
5          MOVER   CREG, B         203)  +05 3 218
6          ADD     CREG, ='1'      204)  +01 3 212
7          ...

12         BC      ANY, NEXT       210)  +07 6 214
13         LTORG
                   ='5'            211)  +00 0 005
                   ='1'            212)  +00 0 001
14         ...
15  NEXT   SUB     AREG, ='1'      214)  +02 1 219
16         BC      LT, BACK        215)  +07 1 202
17  LAST   STOP                    216)  +00 0 000
18         ORIGIN  LOOP+2
19         MULT    CREG, B         204)  +03 3 218
20         ORIGIN  LAST+1
21  A      DS      1               217)
22  BACK   EQU     LOOP
23  B      DS      1               218)
24         END
25                 ='1'            219)  +00 0 001
```

```
20         ORIGIN  LAST+1
```

Algorithm 4.2 (Assembler Second Pass)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   loc_cntr := 0;
2. While next statement is not an END statement
   (a) Clear *machine_code_buffer*;
   (b) If an LTORG statement
      (i) Process literals in LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [POOLTAB [*pooltab_ptr*+1]]−1 similar to processing of constants in a DC statement, i.e. assemble the literals in *machine_code_buffer*.
      (ii) *size* := size of memory area required for literals;
      (iii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (c) If a START or ORIGIN statement then
      (i) *loc_cntr* := value specified in operand field;
      (ii) *size* := 0;
   (d) If a declaration statement
      (i) If a DC statement then
         Assemble the constant in *machine_code_buffer*.
      (ii) *size* := size of memory area required by DC/DS;
   (e) If an imperative statement
      (i) Get operand address from SYMTAB or LITTAB.
      (ii) Assemble instruction in *machine_code_buffer*.
      (iii) *size* := size of instruction;
   (f) If *size* ≠ 0 then
      (i) Move contents of *machine_code_buffer* to the address *code_area_address* + *loc_cntr*;
      (ii) *loc_cntr* := *loc_cntr* + *size*;
3. (Processing of END statement)
   (a) Perform steps 2(b) and 2(f).
   (b) Write *code_area* into output file.

Input :

IC.txt

AD	01	C	200	
IS	04	1	L	1
IS	05	1	S	1
IS	04	2	L	2
IS	04	3	S	3
AD	05			
IS	01	3	L	3
IS	00			
DL	02	C	1	
DL	02	C	1	
AD	02			

LITTAB.txt

='4'	204
='6'	210
='1'	205

SYMTAB.txt

A	208
LOOP	203
B	209

POOLTAB.txt

1
3

Expected Output:

200	04	1	204
201	05	1	208
202	04	2	210
203	04	3	209
204	00	0	004
205	00	0	006
206	01	3	205
207	00	0	000

208
209
210 00 0 001

Conclusion:

Thus we have generated Machine Code for the Source program .

3.	PASS-1 Macroprocessor

Aim: To design Data Structure for Macroprocessor

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

Theory:

1: Macro processor (Definition)

A macro processor is a program that reads a file (or files) and scans them for certain keywords. When a keyword is found, it is replaced by some text. The keyword/text combination is called a Macro.

2. Basic tasks performed by Macro processor:

a) Recognize macro definition

b) Save the definition

c) Recognize Call

d) Expanded calls and substitute arguments.

3. Macro definition part

It consists of

1. Macro prototype Statement – This declares the name of macro and the types of parameters.
2. Model Statement – It is a statement for which assembly language statement is generated during macro expansion.
3. Preprocessor Statement – It is used to perform auxiliary function during macro expansion.

4. Macro Call & Expansion

The operation define by macro can be used by writing a macro name in the mnemonic field And its operand field. Appearance of macro name in the mnemonic field leads to a macro call. Macro Call replaces such statements by sequence of statements comprising the macro. This is known as Macro expansion.

5. Implementation Logic

1. Definition Processing: - Scan all macro definitions and for each macro definition enter the Macro name in macro name table (MNT). Store entire macro definition in macro definition Table (MDT) and add auxiliary information in MNT such as no positional parameters (#PP) no Of key word parameters (#KP) , macro definition table position (MDTP) etc.

2. Macro Expansion: - Examine all statement in assembly source program to detect the macro Calls. For each macro call locate the macro in MNT, retrieve MDTP, establish the Correspondence between formal and actual parameters and expand the macro.

6. Data Structure required for macro definition processing

1. Macro Name Table (MNT):- Fields Name of macro, #PP(Number of positional parameters), #KP (Number of keyword parameters), MDTP (Macro Definition Table Pointer), KPDTP (Keywords Parameters Default Table Position).

2. Parameter Name Table (PNTAB):- Feilds parameter name

3. Keywords Parameters Default Table (KPDTAB):- Feilds-parameter name, default value.

4. Macro Definition Table (MDT):- Model statements are stored in intermediate code form as: Opcode and operands.

7. Algorithm/Pseudo Code

Before processing any definition initialize KPDTAB_ptr, MDT_ptr to 0 and MNT_ptr to -1. These table pointers are common to all macro definition. For each macro definition perform the Following steps.

A one pass macro processor that alternate between macro definition and macro expansion in a Recursive way is able to handle recursive macro definition. Because of one pass structure, the **Definition of macro must appear in the source program before any statements that invoke that macro.**

Algorithm:

A one-pass macro processor that alternate between macro definition and macro expansion algorithms.

Algorithm:

```
begin {macro processor}
        EXPANDING : = FALSE
                while OPCODE ≠ 'END' do
                    begin
                        GETLINE
                        PROCESSLINE
                        end {while}
                end {macro processor}
        procedure PROCESSLINE
                begin
                    search NAMTAB for OPCODE
                    if found then
                        EXPAND
                    else if OPCODE = 'MACRO' then
                        DEFINE
                    else write source line to expanded file
                    end {PROCESSLINE}
```

Algorithm:

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

procedure GETLINE
    begin
        if EXPANDING then
```

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

Example

Source

```
STRG    MACRO
        STA     DATA1
        STB     DATA2
        STX     DATA3
        MEND
   .
STRG
   .
STRG
   .
   .
```

Expanded source

```
   .
   .
   .
        STA     DATA1
        STB     DATA2
        STX     DATA3
   .
        STA     DATA1
        STB     DATA2
        STX     DATA3
   .
```

Source

```
STRG    MACRO &a1, &a2, &a3
        STA     &a1
        STB     &a2
        STX     &a3
        MEND
   .
STRG    DATA1, DATA2, DATA3
   .
STRG    DATA4, DATA5, DATA6
   .
   .
```

Expanded souce

```
   .
   .
   .
        STA     DATA1
        STB     DATA2
        STX     DATA3
   .
        STA     DATA4
        STB     DATA5
        STX     DATA6
   .
```

Input

```
MACRO  INCR  &X  &Y &REG1
            ADD REG  &Y
           MOVEM &REG1 &X
MEND
           START 100
          READ  N1
         READ N2
          INCR N1 N2
         STOP
         N1 DS1
         N2 DS2
         END
```

```
C:\ABC>javac macro.java
C:\ABC>java macro
MACRO INCR    &X        &Y      &REG1
      MOVER       &REG1  &X
      ADD           &REG1  &Y
      MOVEM      &REG1  &X
 MEND
      START         100
      READ          N1
      READ          N2
      INCR          N1        N2
      STOP
      N1            DS         1
      N2            DS         2
      END
```

```
**********************************************
 MNT :

 INDEX   MACRONAME      MDT INDEX
   1            INCR                  1
**********************************************
  ALA:

INDEX   ARGUMENT
  #1       &X
  #2       &Y
  #3       &REG1
**********************************************
  MDT :

MACRO           INCR       &X      &Y      &REG1
                MOVER     #3       #1
                ADD         #3       #2
```

```
                MOVEM   #3      #1
 MEND

**********************************************
```

Conclusion:
Thus pass I of Macro processor is implemented and .MNT, MDT & ALA file is generated.

4.	PASS-2 Macroprocessor

Aim:- Design of a MACRO PASS-2

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.

Theory:-

1: Macro processor (Definition)

A macro processor is a program that reads a file (or files) and scans them for certain keywords. When a keyword is found, it is replaced by some text. The keyword/text combination is called a Macro.

2. Basic tasks performed by Macro processor:

a) Recognize macrodefinition
b) Save thedefinition
c) RecognizeCall
d) Expanded calls and substitutearguments.

In two-pass macro-preprocessor, you have two algorithms to implement, first pass and second pass. Both the algorithms examine line by line over the input data available. Two algorithms to implement two-pass macro-preprocessor are:

• Pass 1 Macro Definition
•Pass 2 Macro Calls and Expansion

Pass 1 Macro Definition

Pass 1 algorithm examines each line of the input data for macro pseudo opcode. Following are the steps that are performed during Pass 1 algorithm:
1. Initialize MDTC and MNTC with value one, so that previous value of MDTC and MNTC is Set to value one.
2.Read the first input data.
3.If this data contains MACRO pseudo opcode then
A.Read the next data input.
B.Enter the name of the macro and current value of MDTC in MNT.
C. Increase the counter value of MNT by value one.
D.Prepare that argument list array respective to the macro found.
E.Enter the macro definition into MDT. Increase the counter of MDT By value one.
F.Read next line of the input data.
G.Substitute the index notations for dummy arguments passed in Macro.
H.Increase the counter of the MDT by value one.
I.If mend pseudo opcode is encountered then next source of input data Is read.
J. Else expands data input.
4.If macro pseudo opcode is not encountered in data input then
A.A copy of input data is created.
B. If end pseudo opcode is found then go to Pass 2.
C.Otherwise read next source of input data.

Pass 2 Macro Calls and Expansion

Pass two algorithm examines the operation code of every input line to check whether it exist in MNT or not.

Following are the steps that are performed during second pass algorithm:

1. Read the input data received from Pass1.
2. Examine each operation code for finding respective entry in theMNT.
3. If name of the macro is encounteredthen
   A. A Pointer is set to the MNT entry where name of the macro isfound. This pointer is called Macro Definition Table Pointer(MDTP).
   B. Prepare argument list array containing a table of dummyarguments.
   C. Increase the value of MDTP by valueone.
   D. Read next line from MDT.
   E. Substitute the values from the arguments list of the macrofor Dummyarguments.
   F. If mend pseudo opcode is found then next source of input data is Read.
   G. Else expands datainput.
4. When macro name is not found then create expanded data file.
5. If end pseudo opcode is encountered then feed the expanded source fileto Assembler forprocessing.
6. Else read next source of datainput.

Draw flowchart w.r.t. algorithm

Input:

```
INPUT
MACRO
INCR1 &FIRST,&SECOND=DATA9
A       1,&FIRST
L       2,&SECOND
MEND MACRO
INCR2 &ARG1,&ARG2=DATA5
L       3,&ARG1
ST      4,&ARG2
MEND
PRG2  START
USING         *,BASE
INCR1 DATA1
INCR2 DATA3,DATA4
FOUR  DC      F'4'
FIVE  DC      F'5'
BASE  EQU     8
TEMP  DS      1F
DROP  8
END
.
```

Output:

```
===== PASS 1 =====
```

```
ALA:
[&FIRST, &SECOND]
[&ARG1, &ARG2]

  MNT:
  [INCR1,   0]
  [INCR2,   4]

  MDT:
  INCR      &FIRST,&SECOND=DAT
  1         A9
  A         1,#0
  L         2,#1
  MEN
  D         &ARG1,&ARG2=DATA5
  INCR
  2
  L         3,#0
  ST        4,#1
  MEN
  D
```

===== PASS 2 =====

```
  MDT:
  INCR      &FIRST,&SECOND=DAT
  1         A9
  A         1,#0
  L         2,#1
  MEN
  D


  PRG2      STAR
            T           *,BASE
            USIN
            G
            A           1,DATA1
            L           2,DATA9
            L           3,DATA3
            ST          4,DATA4
  FOUR      DC          F'4'
  FIVE      DC          F'5'
  BASE      EQU         8
  TEMP      DS          1F
            DRO         8
            P
            END
```

ALA:
[DATA1, DATA9]
[DATA3, DATA4]

Conclusion:

Thus pass II of Macro processor is implemented and ALA file is generated

5.	LEX Program

Aim: - Design Lex program for to generate token of given input file

Problem Statement: - Write a program using Lex specifications to implement lexical analysis Phase of compiler to generate tokens of subset of Java program.

Pre-requisites:- LEX 110, LEX 120, LEX 130, LEX 140, LEX 160, 250

Software Requirements:-

S.No.	Facilities required	Quantity
1	System	1
2	O/S	Ubuntu
3	S/W name	LEX Tool (flex)

Hardware Requirements: - No

Objectives: - 1. To understand LEX Concepts
2. To implement LEX Program
3. To study about Lex&Java
4. To know important about Lexical analyzer

Theory:-

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 users expressions and actions into the host general – purpose language; the generated program is named yylex. The yylex program will recognizeexpressions in a stream (called input in this memo) and perform the specified actions foreach expression as it isdetected.

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.

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.

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…..

%%

…...subrotines….

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.

Conclusion:-

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

6.	LEX Program

Aim: - Design Lex program to count no. of words, lines and characters of given input file.

Problem Statement: - 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.

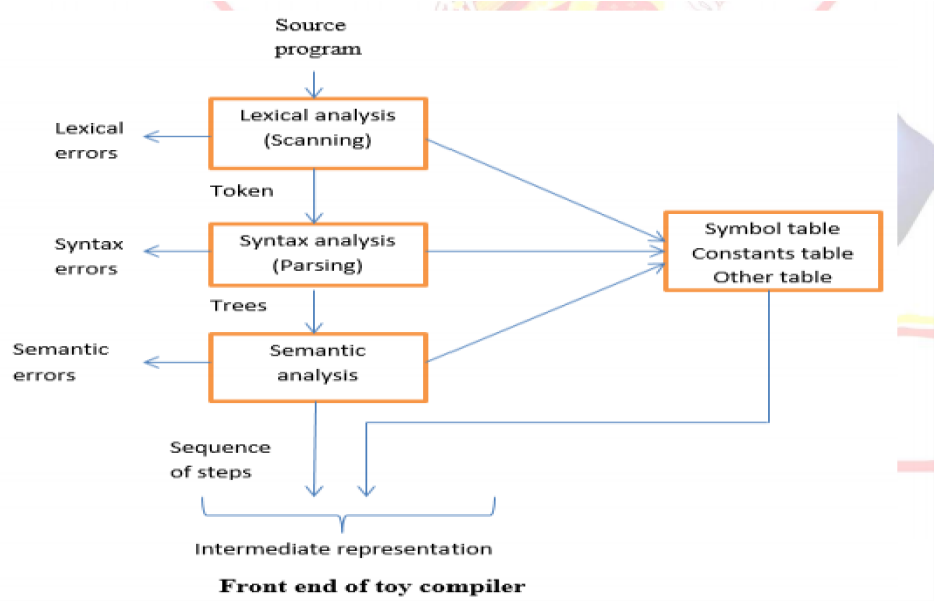
Pre-requisites:- LEX Basics

Software Requirements:-

S.No.	Facilities required	Quantity
1	System	1
2	O/S	Ubuntu Kylin
3	S/W name	LEX Tool (flex)

Objectives: - 1. To understand LEX Concepts.
2. To implement LEX Program for no's of count.
3. To study about Lex & Java.
4. To know important about Lexical analyzer.

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



Front end of toy compiler

Conclusion:-

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

7.	LEX & YACC Program

Aim: - Design Lex & Yacc program to validate type and syntax of variable declaration in Java.

Problem Statement: - Write a program using Yacc specifications to implement lexical analysis Phase of compiler to validate type and syntax of variable declaration in Java.

Pre-requisites:- LEX 110, LEX 120, LEX 130, LEX 140, LEX 160, 250

Software Requirements:-

S.No.	Facilities required	Quantity
1	System	1
2	O/S	Ubuntu Kylin
3	S/W name	FLEX, YACC (LEX & YACC)

Theory:-

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 its normal work, the parser also verifies that the input is syntactically sound.

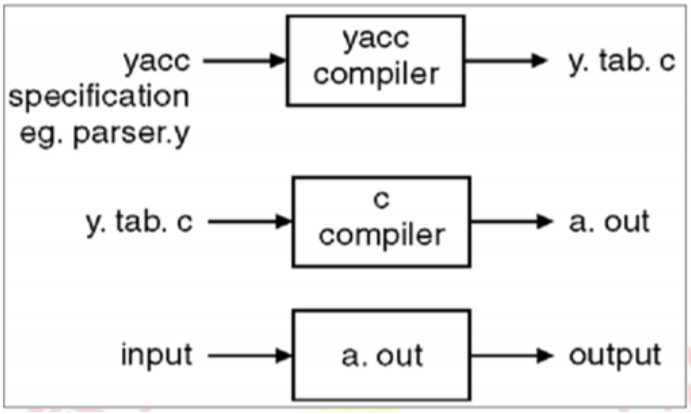


Fig :YACC Parser Generation Model

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 Backus and 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

Translating, Compiling and Executing A Yacc Program:
The Lex program file consists of Lex specification and should be named .l and the Yacc program consists of Yacc specifications and should be named .y. following command may be issued to generate the parser .

Lex <filename>.l
Yacc –d<filename> .y
cc lex.yy.c y.tab.c –ll
./a.out

Yacc reads the grammar description in .y and 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 .y and place them in file y.tab.h. Lex reads the pattern descriptions in .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 wills reads from this file or any further use.

Comparing Sentence Types:-

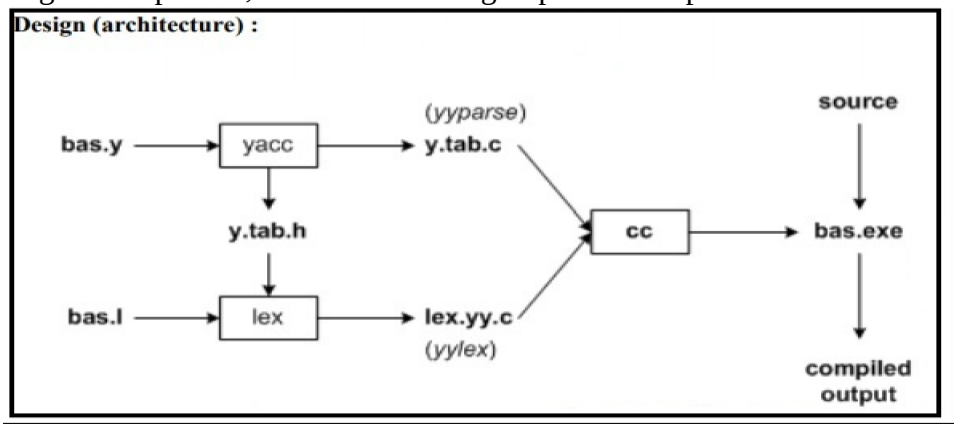
Sentences give structure to language, and in English, they come in four types: simple, compound, complex and compound-complex. When you use several types together, your writing is more interesting. Combining sentences effectively takes practice, but you'll be happy with the result.

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 & conjunctions.

Application: -

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



Conclusion:-

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

8.	Job Scheduling Algorithm

Assignment C-1

Aim: - Implement Job scheduling algorithm

1) FCFS
2) Shortest Job first
3) Priority
4) Round Robin

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

Theory:

Problem Explanation:

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allowed to utilize the CPU. The criteria for selection for an algorithm are,

1. The maximum throughput
2. Least turnaround time.
3. Minimum waiting time.
4. Maximum CPU utilization.
5. Also the variance in response time must be minimum. In Preemptive job, a currently

Executing job can be removed and a new job can take its place, however in Non-Preemptive this is not possible.

1) FIRST COME FIRST SERVE:

This is the simplest CPU scheduling algorithm. The Process that request the CPU first, is the one to which it is allocated first. The algorithm is implemented using a job queue. When a process Requests the CPU it is added at the tail of the job queue.The CPU is allocated to the process at the head of the queue. However the TAT varies, which is not favorable

Implementation:

1- Input the processes along with their burst time (bt).

2- Find waiting time (wt) for all processes.

3- As first process that comes need not to wait so

waiting time for process 1 will be 0 i.e. wt[0] = 0.

4- Find **waiting time** for all other processes i.e. for process i ->
wt[i] = bt[i-1] + wt[i-1] .

5- Find **turnaround time** = waiting_time + burst_time for all processes.

6- Find **average waiting time** =
total_waiting_time / no_of_processes.

7- Similarly, find **average turnaround time** =
total_turn_around_time / no_of_processes.

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

2) SHORTEST JOB FIRST:

This algorithm associates with it the length of the next CPU burst. When the CPU is available, it is assigned to that job with the smallest CPU burst. This algorithm provides the minimum average waiting time. The major problem with this knows the CPU burst of a job.

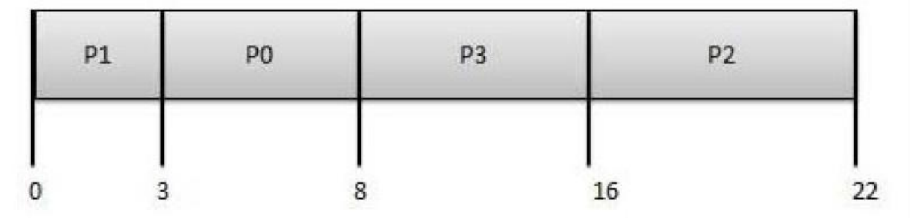
Algorithm:

1- Sort all the processes in increasing order

according to burst time.

2- Then simply, apply FCFS.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	3
P2	2	8	8
P3	3	6	16



How to compute below times in SJF using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time – Burst Time

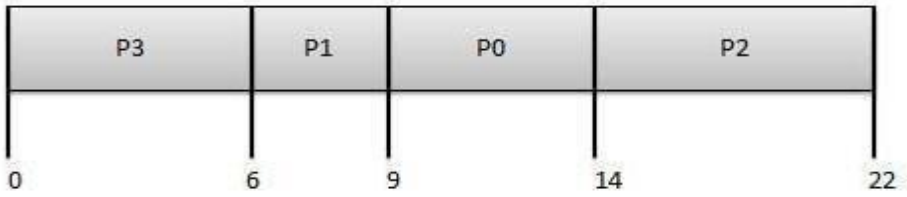
SHORTEST REMAINING TIME:

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

3) PRIORITY BASEDSCHEDULING:

- 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.

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Implementation :

1- First input the processes with their burst time and priority.

2- Sort the processes, burst time and priority according to the priority.

3- Now simply apply FCFS algorithm

Note: A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

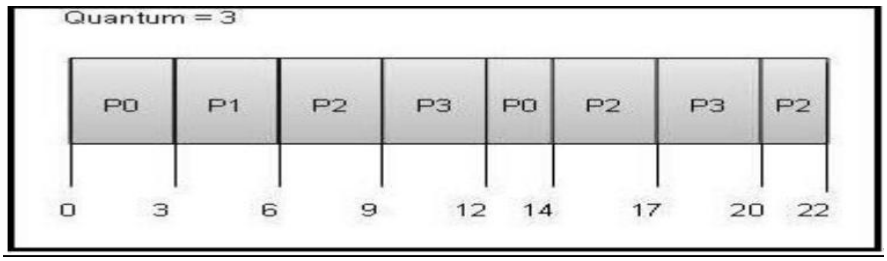
Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

4) ROUND ROBIN SCHEDULING:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.
- 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 processes.



Round Robin Example:

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit.

P1	P2	P3	P1	P2	P3	P1	P2	P3	P2

0 10

P1 waiting time : 4

P2 waiting time: 6

P3 waiting time: 6

The average waiting time(AWT) : (4+6+6)/3=5.33

```
// Java program for implementation of FCFS   scheduling
import java.text.ParseException;
class GFG {
	// Function to find the waiting time for all processes
	static void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
		// waiting time for first process is 0
		wt[0] = 0;
		// calculating waiting time
		for (int i = 1; i < n; i++) {
			wt[i] = bt[i - 1] + wt[i - 1];
		}
	}

	// Function to calculate turn around time
	static void findTurnAroundTime(int processes[], int n,
			int bt[], int wt[], int tat[]) {
		// calculating turnaround time by adding bt[i] + wt[i]
		for (int i = 0; i < n; i++) {
			tat[i] = bt[i] + wt[i];
		}
	}

	//Function to calculate average time
	static void findavgTime(int processes[], int n, int bt[]) {
		int wt[] = new int[n], tat[] = new int[n];
		int total_wt = 0, total_tat = 0;

		//Function to find waiting time of all processes
		findWaitingTime(processes, n, bt, wt);

		//Function to find turn around time for all processes
		findTurnAroundTime(processes, n, bt, wt, tat);

		//Display processes along with all details
		System.out.printf("Processes Burst time Waiting"
					+" time Turn around time\n");

		// Calculate total waiting time and total turn around time
		for (int i = 0; i < n; i++) {
			total_wt = total_wt + wt[i];
			total_tat = total_tat + tat[i];
			System.out.printf(" %d ", (i + 1));
			System.out.printf("      %d ", bt[i]);
			System.out.printf("      %d", wt[i]);
			System.out.printf("      %d\n", tat[i]);
		}
		float s = (float)total_wt /(float) n;
		int t = total_tat / n;
		System.out.printf("Average waiting time = %f", s);
```

```
            System.out.printf("\n");
            System.out.printf("Average turn around time = %d ", t);
        }
        // Driver code
        public static void main(String[] args) throws ParseException {
            //process id's
            int processes[] = {1, 2, 3};
            int n = processes.length;

            //Burst time of all processes
            int burst_time[] = {10, 5, 8};
            findavgTime(processes, n, burst_time);
        }
}
```

Output:

```
Processes  Burst time  Waiting time  Turn around time
 1       10    0        10
 2       5     10       15
 3       8     15       23
Average waiting time = 8.33333
Average turn around time = 16
```

Important Points:

1.Non-preemptive
2.Average Waiting Time is not optimal
3.Cannot utilize resources in parallel : Results in Convoy effect (Consider a situation when many IO bound processes are there and one CPU bound process. The IO bound processes have to wait for CPU bound process when CPU bound process acquires CPU. The IO bound process could have better taken CPU for some time, then used IO devices).

/// **Java program for SJF scheduling**

```
import java.util.*;
class SJF {
public static void main(String args[]) {
Scanner sc = new Scanner(System.in);
int n, BT[], WT[], TAT[];
System.out.println("Enter no of process");
n = sc.nextInt();
BT = new int[n + 1];
WT = new int[n + 1];
TAT = new int[n + 1];
float AWT = 0;
System.out.println("Enter Burst time for each process");
for (int i = 0; i < n; i++) {
System.out.println("Enter BT for process " + (i + 1));
BT[i] = sc.nextInt();
}
for (int i = 0; i < n; i++) {
WT[i] = 0;
TAT[i] = 0;
```

```
}
int temp;
for (int i = 0; i < n; i++) {
for (int j = 0; j < n – 1; j++) {
if (BT[j] > BT[j + 1]) {
temp = BT[j];
BT[j] = BT[j + 1];
BT[j + 1] = temp;
temp = WT[j];
WT[j] = WT[j + 1];
WT[j + 1] = temp;
}
}
}
for (int i = 0; i < n; i++) {
TAT[i] = BT[i] + WT[i];
WT[i + 1] = TAT[i];
}
TAT[n] = WT[n] + BT[n];
System.out.println(" PROCESS BT WT TAT ");
for (int i = 0; i < n; i++)
System.out.println(" " + i + " " + BT[i] + " " + WT[i] + " " + TAT[i]);
for (int j = 0; j < n; j++)
AWT += WT[j];
AWT = AWT / n;
System.out.println("**********************************************");
System.out.println("Avg waiting time=" + AWT +
"\n**********************************************");
}
}
/*

Enter no of process
4
Enter Burst time for each process
Enter BT for process 1
5
Enter BT for process 2
3
Enter BT for process 3
3
Enter BT for process 4
1
PROCESS BT WT TAT
0 1 0 1
1 3 1 4
2 3 4 7
3 5 7 12
**********************************************
Avg waiting time=3.0
```

```
**********************************************
*/
```

Java program for Priority Scheduling

```
import java.util.Scanner;

public class priority{

    public static void main(String args[]) {
          Scanner s = new Scanner(System.in);

          int x,n,p[],pp[],bt[],w[],t[],awt,atat,i;

          p = new int[10];
          pp = new int[10];
          bt = new int[10];
          w = new int[10];
          t = new int[10];

   //n is number of process
   //p is process
   //pp is process priority
   //bt is process burst time
   //w is wait time
   // t is turnaround time
   //awt is average waiting time
   //atat is average turnaround time


   System.out.print("Enter the number of process : ");
   n = s.nextInt();
    System.out.print("\n\t Enter burst time : time priorities \n");

   for(i=0;i<n;i++)
    {
       System.out.print("\nProcess["+(i+1)+"]:");
      bt[i] = s.nextInt();
      pp[i] = s.nextInt();
      p[i]=i+1;
    }

//sorting on the basis of priority
  for(i=0;i<n-1;i++)
   {
     for(int j=i+1;j<n;j++)
     {
       if(pp[i]>pp[j])
       {
     x=pp[i];
```

```
    pp[i]=pp[j];
    pp[j]=x;
    x=bt[i];
    bt[i]=bt[j];
    bt[j]=x;
    x=p[i];
    p[i]=p[j];
    p[j]=x;
     }
   }
}
w[0]=0;
awt=0;
t[0]=bt[0];
atat=t[0];
for(i=1;i<n;i++)
 {
   w[i]=t[i-1];
   awt+=w[i];
   t[i]=w[i]+bt[i];
   atat+=t[i];
 }

//Displaying the process

  System.out.print("\n\nProcess \t Burst Time \t Wait Time \t Turn Around Time   Priority \n");
for(i=0;i<n;i++)
  System.out.print("\n   "+p[i]+"\t\t   "+bt[i]+"\t\t     "+w[i]+"\t\t     "+t[i]+"\t\t     "+pp[i]+"\n");
awt/=n;
atat/=n;
  System.out.print("\n Average Wait Time : "+awt);
  System.out.print("\n Average Turn Around Time : "+atat);

       }
}


/*******OUTPUT********

Enter the number of process : 5

  Enter burst time : time priorities

Process[1]:7 2

Process[2]:6 4

Process[3]:4 1

Process[4]:5 3
```

Process[5]:1 0

Process Burst Time Wait Time Turn Around Time Priority

5 1 0 1 0

3 4 1 5 1

1 7 5 12 2

4 5 12 17 3

2 6 17 23 4

Average Wait Time : 7
Average Turn Around Time : 11

**********/

Java program for Round Robin Algorithm

Round-robin Scheduling algorithm (RR) is designed especially for a time-sharing system. It is similar to FCFS scheduling, but preempted is added to switch between processes. A small unit of time called a time quantum is defined.a time quantum is generally from 10 to 100 milliseconds.The CPU scheduler goes around the ready queue , allocating the CPU to each process for a time interval of up to 1-time quantum.

Input of Round robin Scheduling algorithm (RR) is: n bt[] q

Output of Round robin Scheduling algorithm (RR) is:

average waiting time

Average turn around time

ROUND ROBIN SCHEDULING ALGORITHM Program:

```
import java.io.*;
class round
{
        public static void main(String args[])throws IOException
        {
                DataInputStream in=new DataInputStream(System.in);
                int i,j,k,q,sum=0;
                System.out.println("Enter number of process:");
                int n=Integer.parseInt(in.readLine());
                int bt[]=new int[n];
                int wt[]=new int[n];
                int tat[]=new int[n];
                int a[]=new int[n];
                System.out.println("Enter brust Time:");
                for(i=0;i<n;i++)
                {
                        System.out.println("Enter brust Time for "+(i+1));
                        bt[i]=Integer.parseInt(in.readLine());
```

```
}
System.out.println("Enter Time quantum:");
q=Integer.parseInt(in.readLine());
for(i=0;i<n;i++)
a[i]=bt[i];
for(i=0;i<n;i++)
wt[i]=0;
do
{
        for(i=0;i<n;i++)
        {
                if(bt[i]>q)
                {
                        bt[i]-=q;
                        for(j=0;j<n;j++)
                        {
                                if((j!=i)&&(bt[j]!=0))
                                wt[j]+=q;
                        }
                }
                else
                {
                        for(j=0;j<n;j++)
                        {
                                if((j!=i)&&(bt[j]!=0))
                                wt[j]+=bt[i];
                        }
                        bt[i]=0;
                }
        }
        sum=0;
        for(k=0;k<n;k++)
        sum=sum+bt[k];
}
while(sum!=0);
for(i=0;i<n;i++)
tat[i]=wt[i]+a[i];
System.out.println("process\t\tBT\tWT\tTAT");
for(i=0;i<n;i++)
{
        System.out.println("process"+(i+1)+"\t"+a[i]+"\t"+wt[i]+"\t"+tat[i]);
}
float avg_wt=0;
float avg_tat=0;
for(j=0;j<n;j++)
{
        avg_wt+=wt[j];
}
for(j=0;j<n;j++)
{
```

```
                    avg_tat+=tat[j];
                }
                System.out.println("average waiting time "+(avg_wt/n)+"\n Average turn around time"+(avg_tat/n));
        }
}
```

```
/*Round robin Scheduling algorithm output:
Enter number of process:  4
Enter brust Time:
Enter brust Time for 1 □ 4
Enter brust Time for 2  □ 5
Enter brust Time for 3□ 6
Enter brust Time for 4 □ 7
Enter Time quantum: □ 4
process         BT    WT      TAT
process1        4      0        4
process2       5      12       17
process3       6      13       19
process4       7      15       22
average waiting time 10.0
Average turn around time15.5
*/
```

9.	Bankers Algorithm

Assignment No: - C2

Aim: - Bankers algorithm for deadlock detection and avoidance.

Problem Statement: Write a Java program to implement Banker's Algorithm

Operating System | Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$

for its execution.

- Need [i, j] = Max [i, j] – Allocation [i, j]
  - $Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.
  - Banker's algorithm consist of Safety algorithm and Resource request algorithm

- **Safety Algorithm**
- The algorithm for finding out whether or not a system is in a safe state can be described as follows:
- *1) Let Work and Finish be vectors of length 'm' and 'n' respectively.*
  Initialize: Work = Available
  Finish[i] = false; for i=1, 2, 3, 4….n
- *2) Find an i such that both*
  a) Finish[i] = false

b) $Need_i$ <= Work
if no such i exists goto step (4)

- *3) Work = Work + Allocation[i]*
  Finish[i] = true
  goto step (2)
- *4) if Finish [i] = true for all i*
  then the system is in a safe state

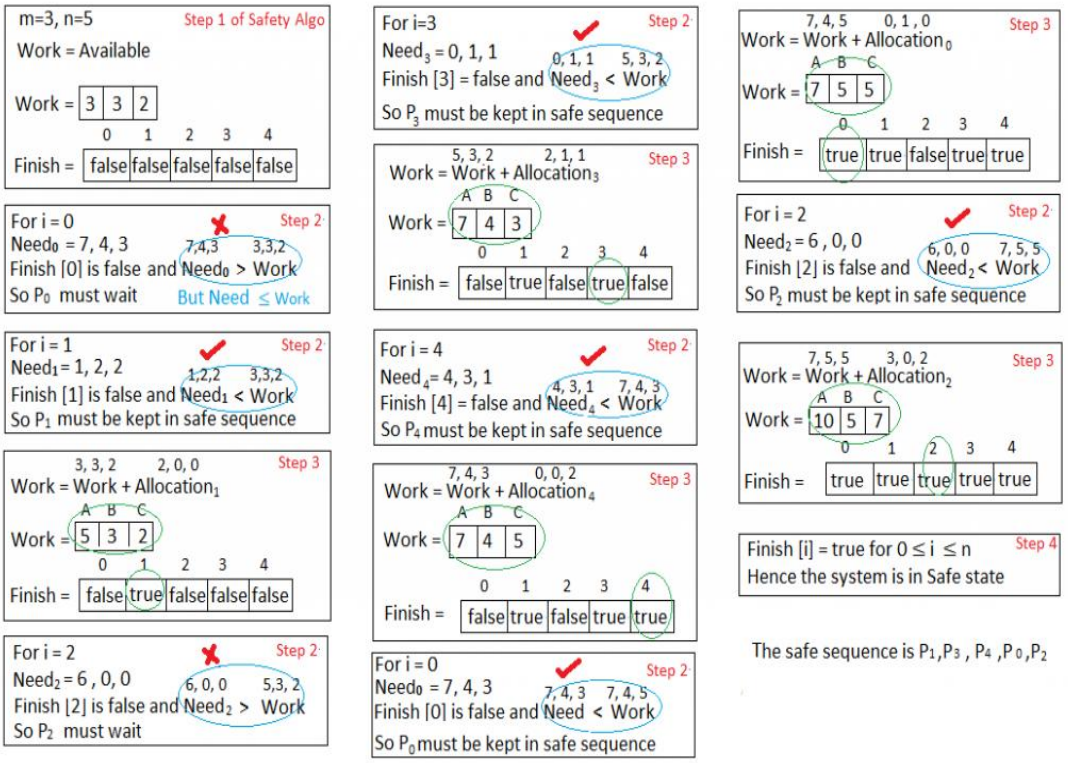
- **Example:**
- **Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:**

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

- **Question1. What will be the content of the Need matrix?**
- Need [i, j] = Max [i, j] – Allocation [i, j]
- So, the content of Need Matrix is:

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- **Question2. Is the system in safe state? If Yes, then what is the safe sequence?**
- Applying the Safety algorithm on the given system,

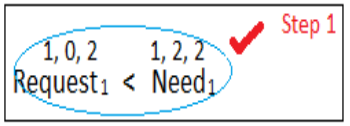


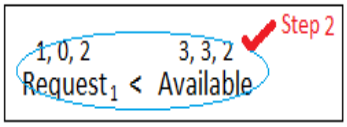
- **Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?**

A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm





Step 3

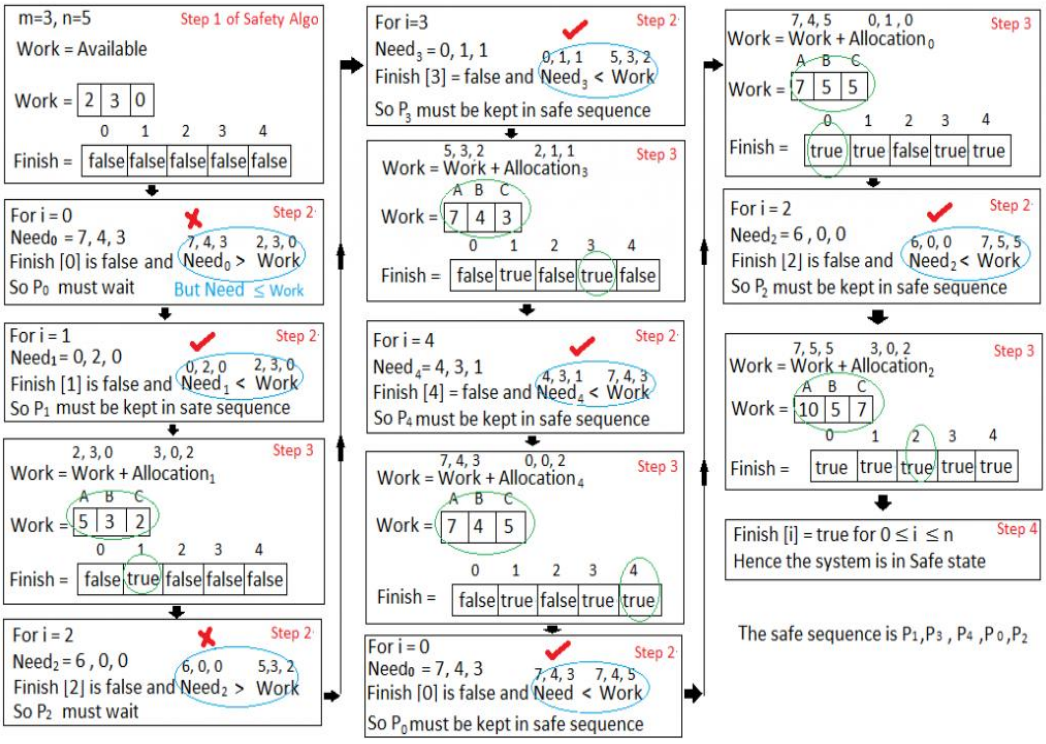
$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

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

- We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



- Hence the new system state is safe, so we can immediately grant the request for process **P_1**

// Java Code for Banker's Algorithm

```
import java.util.Scanner;
        public class Bankers{
           private int need[][],allocate[][],max[][],avail[][],np,nr;

           private void input(){
            Scanner sc=new Scanner(System.in);
            System.out.pr4int("Enter no. of processes and resources : ");
            np=sc.nextInt();  //no. of process
            nr=sc.nextInt();  //no. of resources
            need=new int[np][nr];  //initializing arrays
            max=new int[np][nr];
            allocate=new int[np][nr];
            avail=new int[1][nr];

            System.out.println("Enter allocation matrix -->");
            for(int i=0;i<np;i++)
                for(int j=0;j<nr;j++)
               allocate[i][j]=sc.nextInt();  //allocation matrix

            System.out.println("Enter max matrix -->");
            for(int i=0;i<np;i++)
                for(int j=0;j<nr;j++)
               max[i][j]=sc.nextInt();  //max matrix

              System.out.println("Enter available matrix -->");
```

```
    for(int j=0;j<nr;j++)
     avail[0][j]=sc.nextInt();  //available matrix

    sc.close();
}

private int[][] calc_need(){
  for(int i=0;i<np;i++)
    for(int j=0;j<nr;j++)  //calculating need matrix
     need[i][j]=max[i][j]-allocate[i][j];

  return need;
}

private boolean check(int i){
  //checking if all resources for ith process can be allocated
  for(int j=0;j<nr;j++)
  if(avail[0][j]<need[i][j])
     return false;

return true;
}

public void isSafe(){
  input();
  calc_need();
  boolean done[]=new boolean[np];
  int j=0;

  while(j<np){  //until all process allocated
  boolean allocated=false;
  for(int i=0;i<np;i++)
   if(!done[i] && check(i)){  //trying to allocate
       for(int k=0;k<nr;k++)
       avail[0][k]=avail[0][k]-need[i][k]+max[i][k];
    System.out.println("Allocated process : "+i);
    allocated=done[i]=true;
          j++;
        }
     if(!allocated) break;  //if no allocation
  }
  if(j==np)  //if all processes are allocated
   System.out.println("\nSafely allocated");
  else
   System.out.println("All proceess cant be allocated safely");
}

public static void main(String[] args) {
  new Bankers().isSafe();
}
```

```
    }
```

Output

```
Enter no. of processes and resources: 4
3
Enter allocation matrix -->
0 1 0
2 0 0
3 0 2
2 1 1
Enter max matrix -->
7 5 3
3 2 2
9 0 2
2 2 2
Enter available matrix -->
3 3 2
Allocated process: 1
Allocated process: 3
Allocated process: 0
Allocated process: 2

Safely allocated
```

10.	UNIX System Calls

Assignment No: - C3

Aim: - Implement UNIX system calls like for process management.

Problem Statement: - To write a program to implement UNIX system calls like for process Management.

Pre-requisites:- 1.Explain concept of system call.

2. Explain state diagram working of new process

Software Requirements:-

S.No.	Facilities required	Quantity
1	System	1
2	O/S	Ubuntu Kylin
3	S/W name	C Turbo or GCC

Hardware Requirements: - No

Objectives: - 1) To understand UNIX system call.
2) To understand Concept of process management.
3) Implementation of some system call of OS.

Theory:-

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.

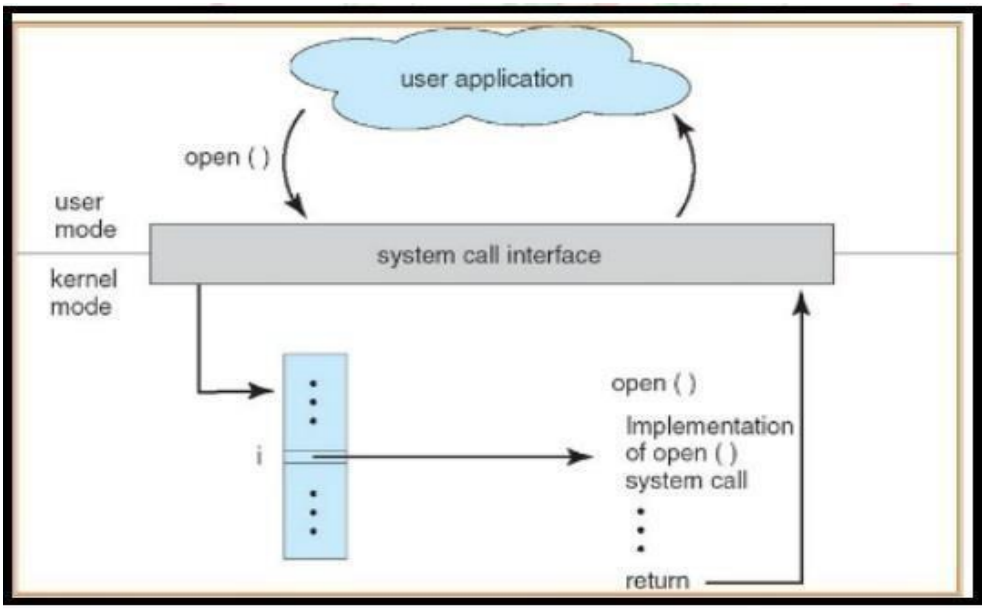
Generally, system calls are made by the user level programs in the following situations:

- 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.
- 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.
-



SYSTEM CALLS 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;

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 process.
  - ✓ 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]

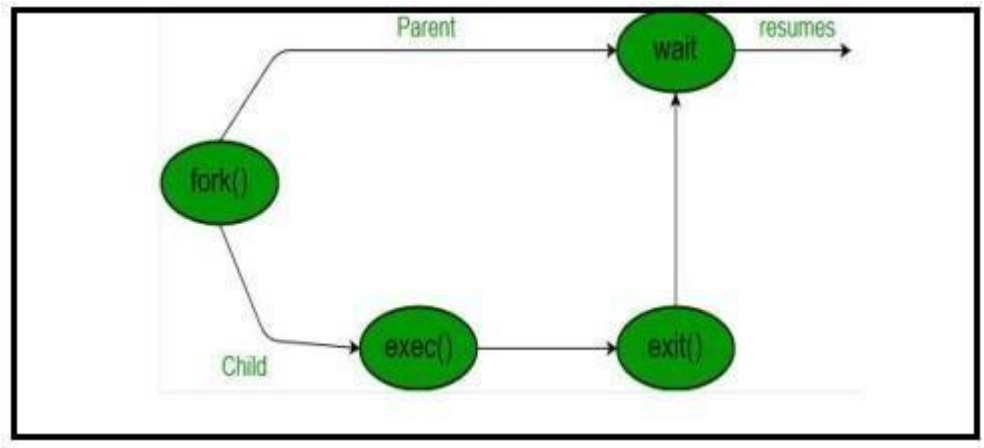
- **fork command :**
  The fork () system call is used to create processes. When a process (a program in execution) makes a fork () call, an exact copy of the process is created. Now there are two processes, one being the parent process and the other being the child process.

- **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

- **Exec() command :**
  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.

- **Wait() command :**
  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.



Conclusion:

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

11.	Study Assignment

Title:

Study assignment on process scheduling algorithms in Android and Tizen

Objectives :

1. To understand Android OS - To understand Tizen OS - To understand Concept of process management
2. Problem Statement : Study assignment on process scheduling algorithms in Android and Tizen
3. 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
4. Software Requirements: Android SDK
5. Hardware Requirement: - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD
6. **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 :

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)

- 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.

- 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.

Conclusion :

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

12.	Page Replacement Algorithm

Assignment D-1

Aim: - Implementing page replacement algorithm.

1) LRU
2) Optimal

Problem Statement: - To write a java program (using OOP feature)to implement LRU & Optimal algorithm for Page Replacement.

Pre-requisites:-

1. Explain the concept of virtual memory.
2. Define page replacement algorithm: LRU & Optimal.
3. Explain address translation in paging system.
4. Explain Belady's Anomaly

Theory:-

Whenever there is a page reference for which the page needed in memory, that event is called page fault or page fetch or page failure situation. In such case we have to make space in memory for this new page by replacing any existing page. But we cannot replace any page. We have to replace a page which is not used currently. There are some algorithms based on them. We can select appropriate page replacement policy. Designing appropriate algorithms to solve this problem is an important task because disk I/O is expensive.

There are several algorithms to achieve.

1) Last recently used (LRU)
2) Optimal

1) LRU page replacement:

The main difference between FIFO and optimal page replacement is that the FIFO algorithms Uses the time when the page was brought in to memory and the. Optimal algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the future then we will replace the page that has not been used for the longest period of time. This approach is called as least recently used (LRU) algorithm.

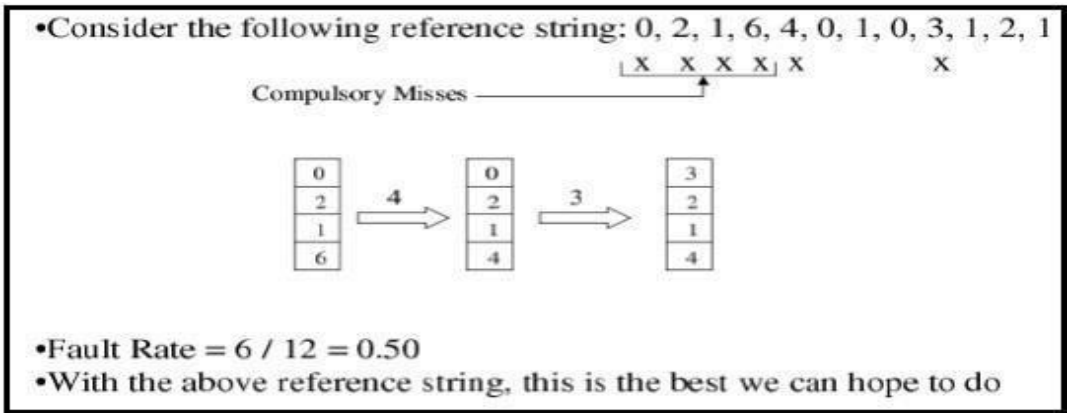
LRU replacement associates with each page must be replaced. LRU chooses that page that has not been used for the longest period of time. Now, consider reference string 7,0,1,2,0,3,4,2,3,0,3 with three memory frames or blocks The first three reference cases page fault that fill the empty frames.

1	2	3	4	5	6	7	8	9	10	11	12
	7	0	1	2	0	3	0	4	2	3	0
	7	7	7	2	2	2	2	4	4	4	0
		0	0	0	0	0	0	0	0	3	3
		1	1	1	3	a.	3	2	2	2	2
	+	+	+	+		+		+			+

2) Optimal page replacement:

The algorithm has lowest page fault rate of all algorithm. This algorithm state that: Replace the page which will not be used for longest period of time i.e future knowledge of reference string is required.

- ✓ Often called Balady's Min Basic idea: Replace the page that will not be referenced for the Longest time.
- ✓ Impossible to implement



ALGORITHM FOR LRU:

Let **capacity** be the number of pages that memory can hold. Let **set** be the current set of pages in memory.

1- Start traversing the pages.

i) **If set holds less pages than capacity.**

a) Insert page into the set one by one until the size of **set** reaches **capacity** or all page requests are processed.

b) Simultaneously maintain the recent occurred index of each page in a map called **indexes**.

c) Increment page fault

ii) **Else**

If current page is present in **set**, do nothing.
Else
a) Find the page in the set that was least recently used. We find it using index array.We basically need to replace the page with minimum index.
b) Replace the found page with current page.
c) Increment page faults.
d) Update index of current page.

2. Return page faults.

ALGORITHM FOR OPTIMAL

1. Start the process

2. Declare the size

3. Get the number of pages to be inserted

4. Get the value
5. Compare counter label and stack

6. Select the Optimal page by counter value

7. Stack them according the selection.

8. Print pages with fault pages.

9. Stop the process.

//Java program for LRU

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

class Test
{
   // Method to find page faults using indexes
   static int pageFaults(int pages[], int n, int capacity)
   {
      // To represent set of current pages. We use
      // an unordered_set so that we quickly check
      // if a page is present in set or not
      HashSet<Integer> s = new HashSet<>(capacity);

      // To store least recently used indexes
      // of pages.
      HashMap<Integer, Integer> indexes = new HashMap<>();

      // Start from initial page
      int page_faults = 0;
```

```
for (int i=0; i<n; i++)
{
    // Check if the set can hold more pages
    if (s.size() < capacity)
    {
        // Insert it into set if not present
        // already which represents page fault
        if (!s.contains(pages[i]))
        {
            s.add(pages[i]);

            // increment page fault
            page_faults++;
        }

        // Store the recently used index of
        // each page
        indexes.put(pages[i], i);
    }

    // If the set is full then need to perform lru
    // i.e. remove the least recently used page
    // and insert the current page
    else
    {
        // Check if current page is not already
        // present in the set
        if (!s.contains(pages[i]))
        {
            // Find the least recently used pages
            // that is present in the set
            int lru = Integer.MAX_VALUE, val=Integer.MIN_VALUE;

            Iterator<Integer> itr = s.iterator();

            while (itr.hasNext()) {
                int temp = itr.next();
                if (indexes.get(temp) < lru)
                {
                    lru = indexes.get(temp);
                    val = temp;
                }
            }

            // Remove the indexes page
            s.remove(val);

            // insert the current page
            s.add(pages[i]);

            // Increment page faults
```

```
                page_faults++;
            }

            // Update the current page index
            indexes.put(pages[i], i);
          }
        }

        return page_faults;
    }

    // Driver method
    public static void main(String args[])
    {
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

        int capacity = 4;

        System.out.println(pageFaults(pages, pages.length, capacity));
    }
}
```

Output:
6

```
// Java program for optimal page replacement algorithm
import java.io.BufferedReader;
                import java.io.IOException;
                import java.io.InputStreamReader;

public class optimal {



                public static void main(String[] args) throws IOException
                {
                    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                    int frames, pointer = 0, hit = 0, fault = 0,ref_len;
                    boolean isFull = false;
                    int buffer[];
                    int reference[];
                    int mem_layout[][];

                    System.out.println("Please enter the number of Frames: ");
                    frames = Integer.parseInt(br.readLine());

                    System.out.println("Please enter the length of the Reference string: ");
                    ref_len = Integer.parseInt(br.readLine());
```

```
reference = new int[ref_len];
mem_layout = new int[ref_len][frames];
buffer = new int[frames];
for(int j = 0; j < frames; j++)
        buffer[j] = -1;

System.out.println("Please enter the reference string: ");
for(int i = 0; i < ref_len; i++)
{
    reference[i] = Integer.parseInt(br.readLine());
}
System.out.println();
for(int i = 0; i < ref_len; i++)
{
 int search = -1;
 for(int j = 0; j < frames; j++)
 {
  if(buffer[j] == reference[i])
  {
   search = j;
   hit++;
   break;
  }
 }
 if(search == -1)
 {
  if(isFull)
  {
   int index[] = new int[frames];
   boolean index_flag[] = new boolean[frames];
   for(int j = i + 1; j < ref_len; j++)
   {
    for(int k = 0; k < frames; k++)
    {
     if((reference[j] == buffer[k]) && (index_flag[k] == false))
     {
      index[k] = j;
      index_flag[k] = true;
      break;
     }
    }
   }
   int max = index[0];
   pointer = 0;
   if(max == 0)
    max = 200;
   for(int j = 0; j < frames; j++)
   {
    if(index[j] == 0)
     index[j] = 200;
```

```
                    if(index[j] > max)
                    {
                     max = index[j];
                     pointer = j;
                    }
                   }
                  }
                  buffer[pointer] = reference[i];
                  fault++;
                  if(!isFull)
                  {
                   pointer++;
                     if(pointer == frames)
                     {
                      pointer = 0;
                      isFull = true;
                     }
                  }
                 }
                    for(int j = 0; j < frames; j++)
                        mem_layout[i][j] = buffer[j];
                }

                for(int i = 0; i < frames; i++)
                {
                    for(int j = 0; j < ref_len; j++)
                        System.out.printf("%3d ",mem_layout[j][i]);
                    System.out.println();
                }

                System.out.println("The number of Hits: " + hit);
                System.out.println("Hit Ratio: " + (float)((float)hit/ref_len));
                System.out.println("The number of Faults: " + fault);
            }


        }
```

```
Please enter the number of Frames:
3
Please enter the length of the Reference string:
20
Please enter the reference string:
7
0
1
2
0
3
0
4
```

2
3
0
3
2
1
2
0
1
7
0
1

The number of Hits: 11
Hit Ratio: 0.55
The number of Faults: 9