ASSIGNMENT NUMBER: Group A - 01

Revised On: 16/12/2019

TITLE	**Pass I of a two pass assembler.**
PROBLEM STATEMENT /DEFINITION	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.
OBJECTIVE	• Understand the internals of language translators • Handle tools like LEX and YACC • Understand the operating system internals and functionalities with implementation point of view
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implementation of Pass I of assembler.

Pre requisite:
Format and type of assembly instruction. Working of an assembler.
Learning Objectives:

- Analyze of source code to solve problem.
- Identify data structures required in the design of assembler.

Learning Outcomes:

The students will be able to

- Parse and tokenize the assembly source code
- Perform the LC processing
- Generate the intermediate code file
- Design the symbol table, literal table, pooltab

Theory:

Assembler is a program which converts assembly language instructions into machine language form. A two pass assembler takes two scans of source code to produce the machine code from assembly language program.

Assembly process consists of following activities:

- Convert mnemonics to their machine language opcode equivalents
- Convert symbolic (i.e. variables, jump labels) operands to their machine addresses
- Translate data constants into internal machine representations
- Output the object program and provide other information required for linker and loader

Pass I Tasks:

- Assign addresses to all the statements in the program (address assignment)
- Save the values (addresses) assigned to all labels(including label and variable names) for use in pass II (Symbol Table creation)
- Perform processing of assembler directives(e.g. BYTE, RESW directives can affect address assignment)

Description using set theory:
Let ‘S’ be set which represents a system S={I,O,T,D,Succ,Fail}
where,

I=Input
O=Output
T=Type (Variant I or II)

D=Data Structure

I={Sf,Mf}
where,
Sf=Source Code File
Mf=Mnemonic Table

O={St,Lt,Ic}
Where,
St=Symbol
Lt=Literal
Ic=Intermediate Code File

St={N,A}
where,
N=Name Of Symbol
A=Address Of Symbol

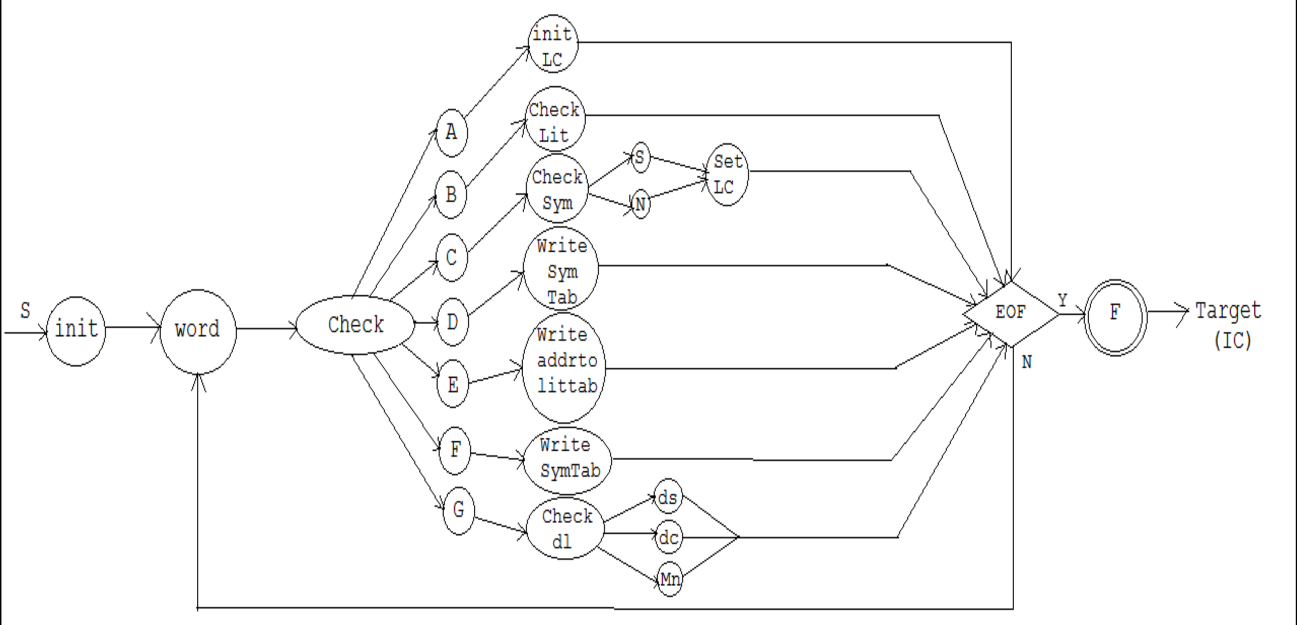
Lt={N,A}
where,
N=Name Of Literal
A=Address Of Literal

T=Variant II

D={Ar,Fl,Sr}
Where,
Ar=Array
Fl=File
Sr=Structure

Success Succ={x |x is set of all cases that are handled in program}
Succ=
{Undefined Symbol (also label),
Duplicate Symbol,
Undefined Symbol in assembler directives,
}
Failures Fail={x |x is set of all cases that are not handled in program}
Fail=
{Multiple statements in a line}

Turing machine/state diagram:



State Diagram
(Pass 1 of 2 Pass Assembler)

Steps to do /algorithm:

- Create MOT.
- Read the .asm file and tokenize it.
- Create symbol and literal tables.
- Generate intermediate code file.

Testing Method:

Use unit testing method for testing the functions. Test the functionalities using functional testing.

Sample Test cases

Test case id	Test case	Expected Output	Actual Result
1	Input all valid mnemonics	Replace the mnemonics with correct opcodes	Success
2	Input the instructions and operands in valid format	Generate valid intermediate code format	Success

FAQs:

Which variant is used in implementation? Why?
Which intermediate data structures are designed and implemented?
Which assembler is implemented?

Review Questions:

1. What is two pass assembler?
2. What is the significance of symbol table?
3. Explain the assembler directives EQU, ORIGIN.
4. How literals are handled in pass I?
5. What are the tasks done in Pass I?
6. How error handling is done in pass I?

ASSIGNMENT NUMBER: Group A - 02

Revised On: 16/12/2019

TITLE	**Pass II of a two pass assembler.**
PROBLEM STATEMENT /DEFINITION	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.
OBJECTIVE	• Understand the internals of language translators • Handle tools like LEX and YACC • Understand the operating system internals and functionalities with implementation point of view
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implementation of pass II of a two pass assembler.
Pre-requisite:
Intermediate code file format. Various data structures like symbol table.
Learning Objectives:

- Synthesis of the object code.
- Understand the use of data structures required in the design of assembler.

Learning Outcomes:

The students will be able to

- Parse and tokenize the intermediate code file
- Perform the LC processing
- Generate the target code file
- Demonstrate the use of symbol table, literal table, pooltab

Theory:

Assembler is a program which converts assembly language instructions into machine language form. A two pass assembler takes two scans of source code to produce the machine code from assembly language program.

Assembly process consists of following activities:

- Convert mnemonics to their machine language opcode equivalents
- Convert symbolic (i.e. variables, jump labels) operands to their machine addresses
- Translate data constants into internal machine representations
- Output the object program and provide other information required for linker and loader

Pass II Tasks:

- Assemble instructios(generate opcode and look up addresses)
- Generate data values defined by BYTE, WORD
- Perform processing of assembler directives(not done in pass I)
- Write the object program and the assembly listing

Description using set theory:

Let ‘S’ be set which represents a system
S={I,O,T,D,Succ,Fail}
where,
I=Input
O=Output
T=Type
D=Data Structure

I={Ic,St,Lt}

where,

Ic=Intermediate Code File
St=Symbol table
Lt=Literal table

St={N,A}

where,

N=Name Of Symbol
A=Address Of Symbol

Lt={N,A}

where,

N=Name Of Literal
A=Address Of Literal

O={o}

Where,

o=Output File(M/C Code File)

T=Varient II

D={Ar,Fl,Sr}

Where,

Ar=Array
Fl=File
Sr=Structure

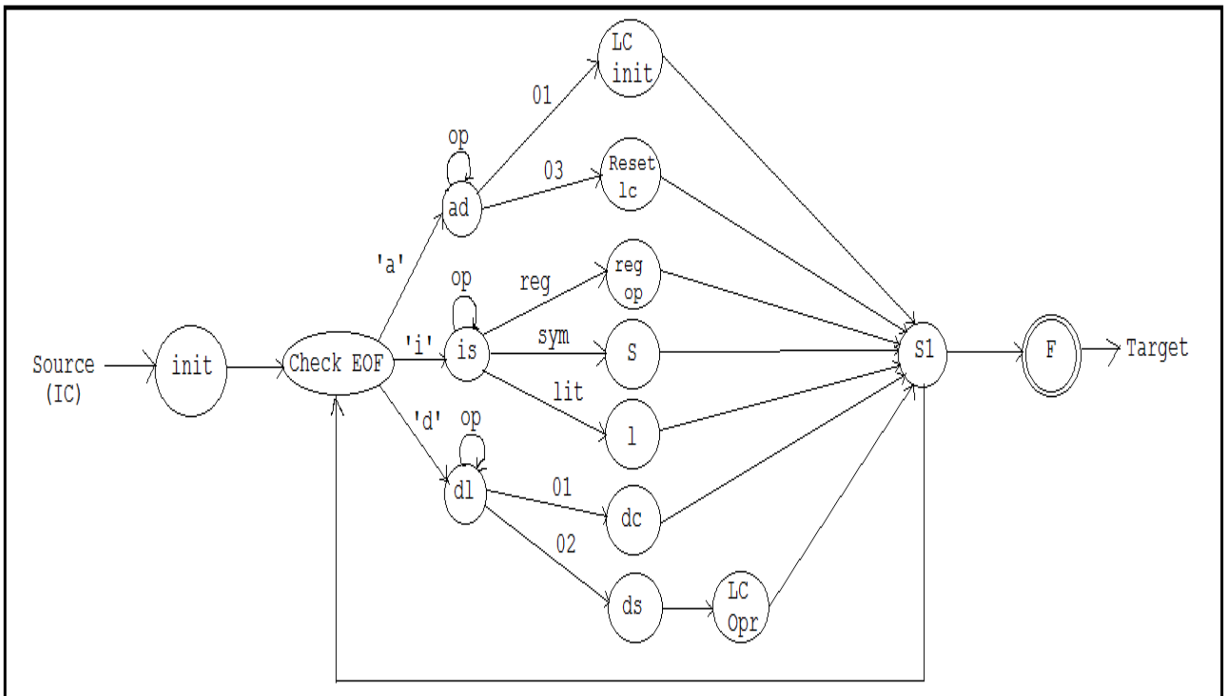
Success Succ={x |x is set of all cases that are handled in program}

Succ=
{
Undefined Symbol
Undefined   mnemonic,
}

Failures Fail={x |x is set of all cases that are not handled in program}

Fail=
{Multiple statements in a line}

Turing machine/state diagram:



State Diagram
(Pass 2 of 2 Pass Assembler)

Steps to do /algorithm:

- Read the intermediate code file generated in pass I.
- Search symbol and literal tables to use in machine code generation.
- Generate the machine code.

FAQs:

Which variant of 2 pass assembler is implemented?
What type of data structures designed, used?

Oral/Review Questions:

What is two pass assembler?
What is the significance of symbol table?
How literal table and pooltab is used in pass II?
What are the tasks done in Pass II?
How error handling is done in pass II?
How symbol and literal tables are referred in pass II?

ASSIGNMENT NUMBER: Group A -03

Revised On: 16/12/2019

TITLE	**Pass I of a two pass macro processor.**
PROBLEM STATEMENT /DEFINITION	Design suitable data structures and implement pass-I of a two-pass macro-processor using OOP features in Java
OBJECTIVE	• Understand the internals of language translators • Handle tools like LEX and YACC • Understand the operating system internals and functionalities with implementation point of view
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implementation of pass I of a two pass macro processor.

Pre requisite:

Macro formats and working of macros.

Learning Objectives:

- Identify and create the data structures required in the design of macro processor.
- Learn parameter processing in macro.

Learning Outcomes:

The students will be able to

- Identify and create the MDT, MNT
- Pass the parameters to the macro
- To separate the macro definitions from the source code

Theory:

Macro processing feature allows the programmer to write shorthand version of a program (modular programming). The macro processor replaces each macro invocation with the corresponding sequence of statements i.e. macro expansion.

Tasks done by the macro processor

- Recognize macro definitions
- Save the macro definition recognize macro calls
- Expand macro calls

Tasks in pass I of a two pass macro processor

- Recognize macro definitions
- Save the macro definition(Create MDT,MNT,ALA)Perform processing of assembler directives(e.g. BYTE, RESW directives can affect address assignment)
- Create intermediate code file.

Steps to do /algorithm:

- Read .asm file.
- Create MNT and MDT.
- Create ALA.
- Create intermediate code file.

FAQs:

Which data structures are developed?

Which form of nested macro is handled?

Oral/Review Questions:

1. What is macro and macro processor?
2. What is MDT, MNT?
3. What is nested macro?
4. What are the tasks done in pass I of macro processor?
5. How macro call definitions are handled in pass I?
6. How formal and actual parameters are linked?
7. What are the steps to implement pass I of macro processor?

ASSIGNMENT NUMBER: Group A - 04

Revised On: 16/12/2019

TITLE	**Pass II of a two pass macro processor.**
PROBLEM STATEMENT /DEFINITION	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.
OBJECTIVE	• Understand the internals of language translators • Understand the operating system internals and functionalities with implementation point of view
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implementation of pass II of a two pass macro processor.
Pre requisite:
Macro formats and working of macros.
Learning Objectives:

- Understand the macro expansion in pass II.
- Replace the formal parameters with actual parameters.
- Understand the use of data structures developed in pass I.

Learning Outcomes:
The students will be able to

- Expand the macro call statements
- Link the actual parameters with the formal parameter.
- Demonstrate the use of various data structures in Pass II which are created in Pass I.

Theory:
Macro processing feature allows the programmer to write shorthand version of a program (modular programming). The macro processor replaces each macro invocation with the corresponding sequence of statements i.e. macro expansion.
Tasks done by the macro processor

- Recognize macro definitions
- Save the macro definition recognize macro calls
- Expand macro calls

Tasks in pass I of a two pass macro processor

- Recognize macro definitions
- Save the macro definition(Create MDT,MNT,ALA)Perform processing of assembler directives(e.g. BYTE, RESW directives can affect address assignment)
- Create intermediate code file.

Steps to do /algorithm:

- Read .asm file.
- Create MNT and MDT.
- Create ALA.
- Create intermediate code file.

INSTRUCTIONS FOR WRITING JOURNAL:

- Title
- Problem Definition
- Objective: Intention behind study
- Software & Hardware requirements
- Explanation of the assignment
- Algorithm or Flowchart
- Developing and testing the program
- Program listing & test results
- Conclusion

FAQs:

1. How are nested macros handled?
2. How macro call within macro definition is handled?
3. Which type of parameters handled?

Oral/Review Questions:

1. What is macro and macro processor?
2. What is macro call nested within macro definition?
3. What are the tasks done in pass II of macro processor?
4. How macro call statements are expanded in pass II?
5. How formal and actual parameters are linked?
6. What are the steps to implement pass II of macro processor?

ASSIGNMENT NUMBER: Group B - 01

Revised On: 16/12/2019

TITLE	**Dynamic Link Library**
PROBLEM STATEMENT /DEFINITION	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++).
OBJECTIVE	• To understand Dynamic Link Libraries Concepts • To implement dynamic link library concepts • To study about Visual Basic
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	1. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: To write a program to create Dynamic Link Library for Arithmetic Operation in VB.net
Learning Objectives:

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

Learning Outcomes:

The students will be able to

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

Theory:

Dynamic Link Library:

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

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

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

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

DLLs are essentially the same as EXEs, the choice of which to produce as part of the linking process is for clarity, since it is possible to export functions and data from either.
It is not possible to directly execute a DLL, since it requires an EXE for the operating system to load it through an entry point, hence the existence of utilities like RUNDLL.EXE or RUNDLL32.EXE which provide the entry point and minimal framework for DLLs that contain enough functionality to execute without much support.
DLLs provide a mechanism for shared code and data, allowing a developer of shared code/data to upgrade functionality without requiring applications to be re-linked or re-compiled. From the application development point of view Windows and OS/2 can be thought of as a collection of DLLs that are upgraded, allowing applications for one version of the OS to work in a later one, provided that the OS vendor has ensured that the interfaces and functionality are compatible.

DLLs execute in the memory space of the calling process and with the same access permissions which means there is little overhead in their use but also that there is no protection for the calling EXE if the DLL has any sort of bug.

Difference between the Application & DLL:

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

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

- Implicit linking
- Explicit linking

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

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

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

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

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

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

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

int stdcall func (int a, double b)

is decorated as:

_func@12

The C calling convention (cdecl) decorates the name as _func.

To get the decorated name, use /MAP. Use of declspec(dllexport) does the following:

- If the function is exported with the C calling convention (_cdecl), it strips the leading underscore (_) when the name is exported.

- If the function being exported does not use the C calling convention (for example, stdcall), it exports the decorated name.

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

int stdcall MyFunc (int a, double b); void stdcall InitCode (void);

In the .DEF file:
```
EXPORTS
MYFUNC=_MyFunc@12
INITCODE=_InitCode@0
```

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

DLL's Advantages:

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

Disadvantage:

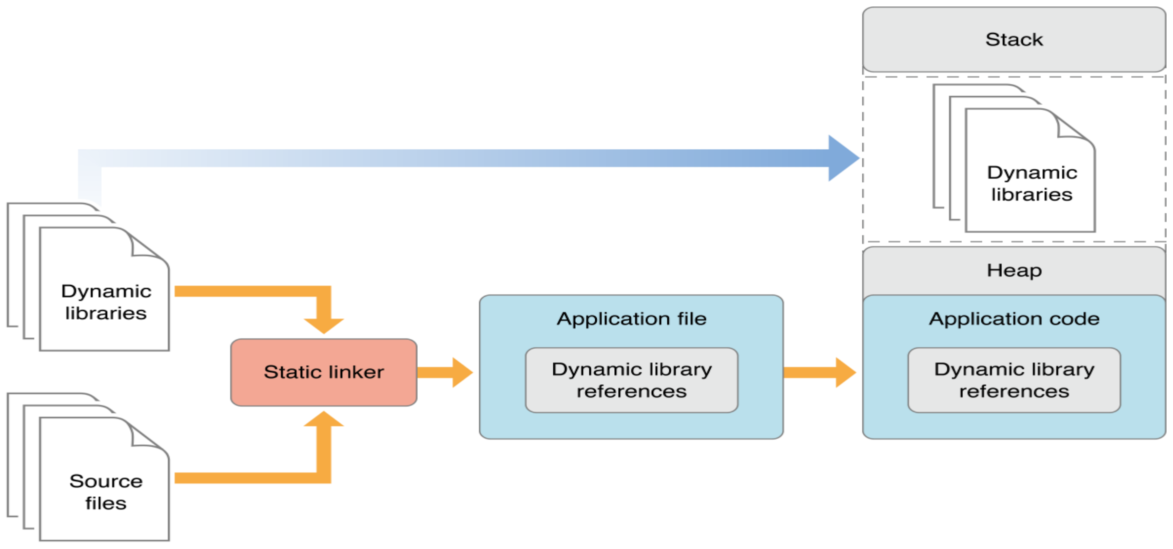
A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module.

Visual Basic:
Visual Basic is a third-generation event-driven programming language first released by Microsoft in

1991. It evolved from the earlier DOS version called BASIC. BASIC means Beginners' All-purpose Symbolic Instruction Code. Since then Microsoft has released many versions of Visual Basic, from Visual Basic 1.0 to the final version Visual Basic 6.0. Visual Basic is a user-friendly programming language designed for beginners, and it enables anyone to develop GUI window applications easily.

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

Design Architecture:



Algorithm:

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

Review Questions:

- What Is Dll And What Are Their Usages And Advantages?
- What Are The Sections In A Dll Executable/binary?
- Where Should We Store Dlls ?

4. Who Loads And Links The Dlls?

5. How Many Types Of Linking Are There?

6. What Is Implicit And Explicit Linking In Dynamic Loading?

7. How to call a DLL function from Visual Basic?

Group B-ASSIGNMENT NUMBER: 02

Revised On: 16/12/2019

TITLE	**Lexical Analysis to generate tokens**
PROBLEM STATEMENT /DEFINITION	Write a program using LEX specifications to implement lexical analysis phase of compiler to generate tokens of subset of Java program.
OBJECTIVE	• Understand the importance and usage of LEX automated tool • Appreciate the role of lexical analysis phase in compilation • Understand the theory behind design of lexical analyzers and lexical analyzer generator
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA 64-bit architecture I3 or I5 machines LEX and YACC
REFERENCES	1. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implement LEX program to generate token from a given program.

Pre-requisite:

Working of a compiler with their phases and Java programming.
Learning Objectives:

- Analyze source code to solve the problem
- Identify tokens required in the lexical analysis process.

Learning Outcomes:
The students will be able to

- Tokenize any given input source code.

Theory:

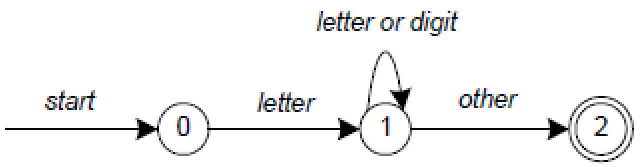
During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.
The following represents a simple pattern, composed of a regular expression that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter (letter | digit)*

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

- repetition, expressed by the “*” operator
- alternation, expressed by the “|” operator
- concatenation

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



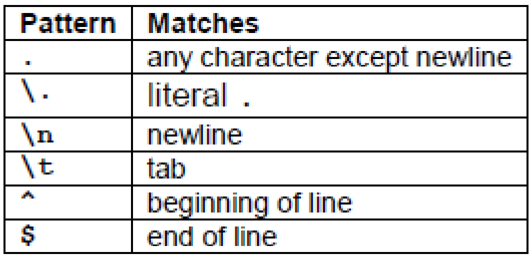
Finite State Automate

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

```
start: goto state0
state0: read c
        if c = letter goto state1
        goto state0
state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2
state2: accept string
```

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

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



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

Table 1: Special Characters

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

Table 2: Operators

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

Table 3: Character Class

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

Table 4: Lex Predefined Variables

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

... definitions ...
%%
... rules ...

```
%%
... subroutines ...
```

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

```
%%
```

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

```
%%
/* match everything except newline */
. ECHO;
/* match newline */
\n ECHO;
%%
int yywrap(void) {
return 1;
}
int main(void) {
yylex();
return 0;
}
```

FAQs:

1. What is lexical analysis?
2. What is LEX?
3. What are lexemes?
4. What is pattern recognition?
5. What is the data structure used in Lexical phase of compiler
6. What are lexical errors?

Oral/Review Questions:

1. What are lexemes?
2. What is pattern recognition?
3. What is lexical analysis?
4. What is LEX?
5. Algorithm for lexical analysis
6. Syntax for Pattern Recognition
7. Libraries for LEX in Java.
8. Difference between LEX and YACC.

Group B-ASSIGNMENT NUMBER: 03

Revised On: 16/12/2019

TITLE	**Lexical Analysis to count number of words, lines and characters.**
PROBLEM STATEMENT /DEFINITION	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.
OBJECTIVE	• Understand the importance and usage of LEX automated tool • Appreciate the role of lexical analysis phase in compilation • Understand the theory behind design of lexical analyzers and lexical analyzer generator • Count the number of words, lines and characters
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA 64-bit architecture I3 or I5 machines LEX and YACC
REFERENCES	2. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implement LEX program to count number of lines, words and characters in a given file.

Pre-requisite:

Working of a compiler with their phases and Java programming.
Learning Objectives:

- Analyze source code
- Identify tokens required in the lexical analysis process.
- Count the lines, words and characters

Learning Outcomes:
The students will be able to

- Count number of lines, words and characters

Theory:

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.
The following represents a simple pattern, composed of a regular expression that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter (letter | digit)*

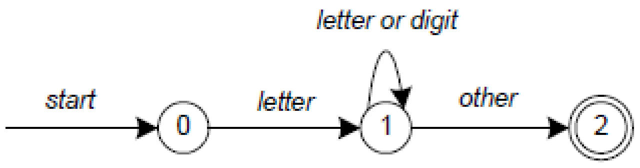
This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the “*” operator

- alternation, expressed by the “|” operator

- concatenation

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



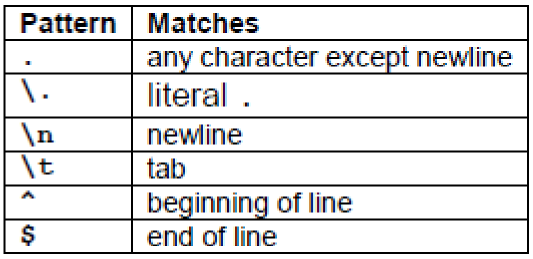
Finite State Automate

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

```
start: goto state0
state0: read c
        if c = letter goto state1
        goto state0
state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2
state2: accept string
```

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

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



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

Table 1: Special Characters

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

Table 2: Operators

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

Table 3: Character Class

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

Table 4: Lex Predefined Variables

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

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

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

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

```
%%
.
\n
```

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

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

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with “**%{**“ and “**%}**” markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit [0-9]
letter [A-Za-z]
%{
int count;
%}
%%
/* match identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
yylex();
printf("number of identifiers = %d\n", count);
return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

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

FAQs:
What is LEX?
What are lexemes?
What is pattern recognition?
What is the difference between lex and yacc?

Oral/Review Questions:
What are lexemes?
What is pattern recognition?
What is lexical analysis?
What is LEX?
Algorithm for lexical analysis
Syntax for Pattern Recognition
Libraries for LEX in Java.
Why yytext() is used?
Why yylex() is used?
Why yywrap() and yylineno() are used?

Group B-ASSIGNMENT NUMBER: 04

Revised On: 16/12/2019

TITLE	**YACC program to validate variable declarations**
PROBLEM STATEMENT /DEFINITION	Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java.
OBJECTIVE	• Be proficient on writing grammars to specify syntax • Understand the theories behind different parsing strategies-their strengths and limitations • Understand how the generation of parser can be automated • Be able to use YACC to generate parsers
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA 64-bit architecture I3 or I5 machines LEX and YACC
REFERENCES	3. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implement YACC program to validate type and syntax of variable declaration.

Pre-requisite:

Working of a compiler with their phases specifically syntactic analysis
Basics of Java programming
Basics of grammar analysis and regular expressions

Learning Objectives:

- Analyze source code for variables
- Identify data type and syntax of a variable

Learning Outcomes:
The students will be able to

- Match the variables and identify the data type.

Theory:

Lex recognizes regular expressions, whereas YACC recognizes entire grammar. Lex divides the input stream into tokens, while YACC uses these tokens and groups them together logically.

The syntax of a YACC file:

```
%{
        declaration section
%}
        rules section
%%
        user defined functions
```

Declaration section:

Here, the definition section is same as that of Lex, where we can define all tokens and include header files. The declarations section is used to define the symbols used to define the target language and their relationship with each other. In particular, much of the additional information required to resolve ambiguities in the context-free grammar for the target language is provided here.

Grammar Rules in Yacc:

The rules section defines the context-free grammar to be accepted by the function Yacc generates, and associates with those rules C-language actions and additional precedence information. The grammar is described below, and a formal definition follows.

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

```
A : BODY ;
```

The symbol A represents a non-terminal name, and BODY represents a sequence of zero or more names, literals, and semantic actions that can then be followed by optional precedence rules. Only the names and literals participate in the formation of the grammar; the semantic actions and precedence rules are used in other ways. The colon and the semicolon are Yacc punctuation.

If there are several successive grammar rules with the same left-hand side, the vertical bar '|' can be used to avoid rewriting the left-hand side; in this case the semicolon appears only after the last rule. The BODY part can be empty.

Programs Section

The programs section can include the definition of the lexical analyzer yylex(), and any other functions; for example, those used in the actions specified in the grammar rules. It is unspecified whether the programs section precedes or follows the semantic actions in the output file; therefore, if the application contains any macro definitions and declarations intended to apply to the code in the semantic actions, it shall place them within "%{ ... %}" in the declarations section.

Interface to the Lexical Analyzer

The yylex() function is an integer-valued function that returns a token number representing the kind of token read. If there is a value associated with the token returned by yylex() (see the discussion of tag above), it shall be assigned to the external variable yylval.

FAQs:
Which is the file generated as output as a result of YACC translation
How LEX communicates with YACC
Is there any other parser generator/ syntax analysis generator other than YACC?
Which phase is the driver of compiler?
How to debug YACC specification file?

Oral/Review Questions:
What is state machine?
Which phase is the driver of compiler?
How to debug YACC specification file?
Which is the file generated as output as a result of YACC translation
How LEX communicates with YACC
Are there any other parser generators/ syntax analysis generators other than YACC?
How YACC works?
Long form of YACC.
Why YACC is a compiler-compiler?

Group B-ASSIGNMENT NUMBER: 05

Revised On: 16/12/2019

TITLE	**YACC program to run syntactic analysis**
PROBLEM STATEMENT /DEFINITION	Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file.
OBJECTIVE	• Understand the theories behind different parsing strategies-their strengths and limitations • Understand how the generation of parser can be automated • Be able to use YACC to generate parsers • Understand how the instructions are matched and syntax is understood by machines
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA 64-bit architecture I3 or I5 machines LEX and YACC
REFERENCES	4. Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, McGraw Hill 2. Paul Gries Jennifer Campbll, Jason Montojo, “Practical Programming”, 2nd Edition, SPD
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** • Title • Objectives • Problem statement • Outcomes • Software and hardware requirements • Date of completion • Theory – Concept in brief • Algorithm • Flowchart • Design • Test cases • Conclusion/Analysis **Soft copy as follows :** Program codes with sample output of all performed assignments are to be submitted as soft copy

Aim: Implement YACC program to analyze syntax of a source code.

Pre-requisite:

Working of a compiler with their phases specifically syntactic analysis
Basics of Java programming
Basics of CFG and regular expressions

Learning Objectives:

- Analyze source code for variables
- Identify data type and syntax of a variable

Learning Outcomes:
The students will be able to

- Create a grammar and match the incoming instructions to their respective syntaxes.

Theory:

Lex recognizes regular expressions, whereas YACC recognizes entire grammar. Lex divides the input stream into tokens, while YACC uses these tokens and groups them together logically.

The syntax of a YACC file:

```
%{
        declaration section
%}
        rules section
%%
        user defined functions
```

Declaration section:

Here, the definition section is same as that of Lex, where we can define all tokens and include header files. The declarations section is used to define the symbols used to define the target language and their relationship with each other. In particular, much of the additional information required to resolve ambiguities in the context-free grammar for the target language is provided here.

Grammar Rules in Yacc:

The rules section defines the context-free grammar to be accepted by the function Yacc generates, and associates with those rules C-language actions and additional precedence information. The grammar is described below, and a formal definition follows.

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

A : BODY ;

The symbol A represents a non-terminal name, and BODY represents a sequence of zero or more names, literals, and semantic actions that can then be followed by optional precedence rules. Only the names and literals participate in the formation of the grammar; the semantic actions and precedence rules are used in other ways. The colon and the semicolon are Yacc punctuation.

If there are several successive grammar rules with the same left-hand side, the vertical bar '|' can be used to avoid rewriting the left-hand side; in this case the semicolon appears only after the last rule. The BODY part can be empty.

Programs Section

The programs section can include the definition of the lexical analyzer yylex(), and any other functions; for example, those used in the actions specified in the grammar rules. It is unspecified whether the programs section precedes or follows the semantic actions in the output file; therefore, if the application contains any macro definitions and declarations intended to apply to the code in the semantic actions, it shall place them within "%{ ... %}" in the declarations section.

Interface to the Lexical Analyzer

The yylex() function is an integer-valued function that returns a token number representing the kind of token read. If there is a value associated with the token returned by yylex() (see the discussion of tag above), it shall be assigned to the external variable yylval.

Input descriptions:

The format of the grammar rules for Yacc is:

```
name    : names and 'single character's
        | alternatives
        ;
```

Yacc definitions:

%start line	means the whole input should match line
%union	lists all possible types for values associated with parts of the grammar and gives each a field-name
%type	gives an individual type for the values associated with each part of the grammar,
	using the field-names from the %union declaration
%token	declare each grammar rule used by YACC that is recognised by LEX and give type of value

Yacc does its work using parsing and hence it is also called a parser.

PARSING:

Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not. There are two main kinds of parsers in use, named for the way they build the parse trees:
Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse

to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

FAQs:

1. Which parser is used in YACC?
2. How to debug YACC specification file?
3. What is L-R parser?
4. Types of Parsers

Oral/Review Questions:

1. Why YACC is a compiler-compiler?
2. How to debug YACC specification file?
3. What is L-R parser?
4. Types of Parsers
5. Which parser is used in YACC?
6. What is LALR parser?
7. Top-down and bottom up parser.
8. What are LL and LR Parsers?

ASSIGNMENT NUMBER: C-01

Revised On: 16/12/2019

TITLE	Scheduling algorithms.
PROBLEM STATEMENT /DEFINITION	Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF (Preemptive,Non-preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive)
OBJECTIVE	To learn and understand • Process Scheduling in Multitasking and multiusers OS. • Implementation of Scheduling Algorithms
S/W PACKAGES AND HARDWARE APPARATUS USED	C/C++ editors and compilers for Linux OS Linux OS/Fedora/Ubuntu PC with the configuration as Pentium IV 2.4 GHz. 4 GB RAM, 500 G.B HDD, 15''Color Monitor, Keyboard, Mouse
REFERENCES	• Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, ' McGraw Hill, • Stallings W., "Operating Systems", 4th Edition, Prentice Hall, 81 – 7808 – 503 – 8.
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	**Handwritten write-up as follows :** Title Objectives Problem statement Outcomes Software and hardware requirements Date of completion Theory – Concept in brief Algorithm Flowchart Design Test cases

Aim: Implementation of process scheduling algorithms

Prerequisites:

Basic Operating System Functionalities.
Concept of Multitasking and Multiuser OS .

Learning Objectives:

To learn and understand

- Process Scheduling in Multitasking and Multiuser OS
- Implementation of Scheduling Algorithms

Learning Outcomes:

The student will be able to

- Compare the scheduling algorithms
- Implement FCFS, SJF, RR Scheduling Algorithms

THEORY:

Process scheduling: It is an activity process manager that handles the task of scanning process from CPU and running of another process on basis of some strategy.Such OS allows more than one to be loaded in executable memory.

Scheduler:

They are special system software that handle process scheduling in various ways:
Its main task is to select jobs to be submitted into system and to be divided and decide which process to run.

Types of Scheduler:
1) Long-term scheduler
2)Short-term scheduler.
3)Medium term scheduler.

Arrival: The request arrives in the system hen user submits it to OS. When request arrives, the time is called arrival time.
Scheduling: The pending request is scheduled for service when scheduler selects it for servicing.
Preemption: The process is 39re-empted when CPU switches to another process before completing it and this process is added to pending request.
Non 39re-empted39: The scheduled process is always completed before next scheduling of the process.
Completion: The process is completed and next process is selected for processing by CPU.
The CPU scheduling takes the information about arrival time, size of request in CPU seconds, CPU time already consumed by the request, deadline of the process for scheduling policy.
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 of an algorithm are

- The maximum throughput
- Least turnaround time.

- Minimum waiting time.
- Maximum CPU utilization.

Some definitions in scheduling are:

Arrival time is when the process arrives in the system.(Ai)
Process time is the execution time required for the process(Xi)
Completion time is time at which the process is completed.(Ci)
Deadline is the time by which the process output is required(Di)
Turnaround time is the time to complete the process after arrival.(Ci-Ai)
Average or Mean turnaround time is the average of turnaround time of all processes.(1/n $\sum \quad (Ci - Ai)$
Weighted time around time is the turnaround time of a process to its execution time. (Ci-Ai)/Xi
Throughput is the measure of performance and no of processes completed per unit time.(n/(max(Ci)-min(Ai))

Scheduling Policies:

FCFS Scheduling:
The process requests are schedules in the order of their arrival time. The pending requests are in a queue. The first request in the queue is scheduled first. The request that comes is added to the end of the queue.

Performance of FCFS scheduling: (Time in sec)

Process No	Arrival time	Burst Time	Turnaround time	Waiting time
1	0	5	0	0
2	1	3	5	4
3	2	8	8	6
4	3	6	16	13

Average waiting time=(0+4+6+13)/4=5.75

```
P0          P1          P2          P3
0          5           8           16        22
```

<u>Algorithm:</u>
1) Input the processes along with burst times.
2)Input arrival time for all processes
3)Sort according to their arrival time along with indices.
4)Perform processes in sorted order
5)Stop.

Shortest job first (SJF) scheduling:

est approach to minimize waiting time. It is easy to implement in batch systems where required CPU time is known in advance.

a)**<u>Non-preemptive</u>**

Performance of SJF scheduling: (Time in sec)

Process No	Arrival time Ai	Burst time	Waiting time
1	0	2	1
2	0	3	3
3	1	4	5
4	0	1	0

Average waiting time=2.25

```
P3          P0          P1          P2
0         1          3           6          10
```

b) **<u>Preemptive:</u>**

Process	Arrival time	Burst time	Waiting time
P1	0	300	425
P2	0	125	150
P3	0	400	725
P4	0	150	275
P5	0	100	0
P6	150	50	0

```
   P5    P2    P6    P2    P4    P1    P3
0    100   150   200   275   425   725   1125
```

Average waiting time=262.5

Algorithm

1)Calculate burst time.
2)Sort all processes in increasing order of burst time.
3)Apply FCFS to sorted list.
4)Perform all processes
5)Stop

Round Robin scheduling: Schedules using time slicing. The amount of CPU time a process may use when allocated is limited. The process is 41re-empted if the process requires more time or if process requires I/O operation before the time slice. It makes weighted turnaround time approximately equal all time but throughput may not be well as all processes are treated equally.

Performance of Round Robin scheduling:

Proces	Arrival	Burst time	Waiting time

s No	time Ai		
1	1	150	250
2	2	100	200
3	3	200	300
4	4	50	150

Time quantum=50
Average waiting time=225

P1	P2	P3	P4	P1	P2	P3	P1	P3	
0	50	100	150	200	250	300	350	400	550

Algorithm:

1)Get input for processes with arrival time and burst time. Take quantum.
2)Sort processes according to arrival time.
3)Process till all processes are done:
4)End

Priority based Scheduling:

It is non-preemptive algorithm and one of the common scheduling algorithm in batch system. Each process is assigned a priority and process with highest priority is executed first and so on. Processes with same priority are executed on FCFS basis.

Process	Arrival time	Burst time	Priority	Waiting time
P0	0	5	1	9
P1	1	3	2	5
P2	2	8	1	12
P3	3	6	3	0

P3	P1	P0	P2	
0	**6**	**9**	**14**	**22**

Algorithm:

1)Get input for process including arrival time,burst time and priority.
2)Sort process according to arrival time.
3)If process have same arrival time,sort them by priority.

4)Print process according to index.
5)End

Steps to do /algorithm:

1. Create s menu to select various scheduling algorithms
2. Take number of tasks and CPU time as input.
3. Calculate average waiting time and turnaround time for each scheduling strategy.
4. Perform a comparative assessment of best policy for given set of processes.

FAQs

1. What are the inputs to be taken?
   Ans: The inputs for each process with process no, Arrival time, Execution time to be taken.
2. What all parameters to be calculated?
   Ans: The turn around time, Waiting time,Average Turnaround Time,Average Waiting Time.
3. How the output to be shown?
   Ans: The output for each algorithm for the same set of inputs to be shown as table and Gant chart to be shown in write-up.

Oral/Review Questions:

1. Why job scheduling is required in OS?
2. What are basic job scheduling policies used in OS?
3. What is preemptive scheduling?
4. What is the computational complexity of RR, SJF, FCFS?
5. What is Gant chart?

Group C (Assignment No – 2)

Reevised on:16/12/2019

TITLE	**Implementation of Banker's algorithm.**
PROBLEM STATEMENT /DEFINITION	Write a Java program to implement Banker's Algorithm
OBJECTIVE	• To study the algorithm for finding out whether a system is in a safe state. • To study the resource-request algorithm for deadlock avoidance. • To study and implement the Banker's algorithm to avoid deadlock.
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20) Eclipse IDE, JAVA I3 and I5 machines
REFERENCES	• Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, ' McGraw Hill, • Stallings W., "Operating Systems", 4th Edition, Prentice Hall, 81 - 7808 - 503 - 8.
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Aim: Write a Java program to implement Banker's Algorithm

Pre-requisite:

Basic conditions for deadlock and deadlock handling approaches.

Learning Objectives:

To study the algorithm for finding out whether a system is in a safe state.
To study the resource-request algorithm for deadlock avoidance.
To study and implement the Banker's algorithm to avoid deadlock.

Learning Outcomes:

The students will be able to

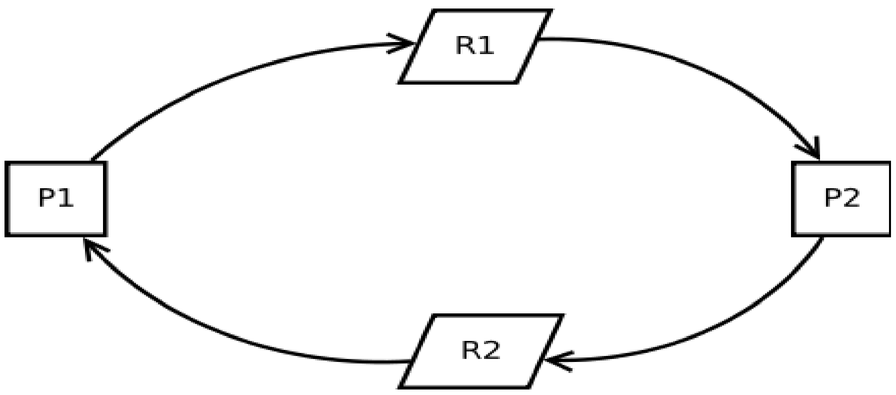
- Implement deadlock avoidance algorithm
- Compute resource allocation sequences which lead to safe state
- Demonstrate the limitations of deadlock avoidance algorithms

Theory:

Deadlock: A set of processes is in a deadlock state when every process in the set is waiting for an event that can only cause by another process in the set. Examples of such processes are resources acquisition and release.

As shown in the diagram process P1 is holding the resource R2 and requesting resource R1.

Process P2 is holding the resource R1 and requesting resource R2. So no process can proceed further, indicating the deadlock.



Four basic conditions for deadlock to happen:

1. **mutual exclusion**: at least one resource must be held in a non-sharable mode.
2. **hold and wait:** there must be a process holding one resource and waiting for another.
3. **no preemption**: resources cannot be preempted.
4. **circular wait**: there must exist a set of processes [p1, p2, …, pn] such that p1 is waiting for p2, p2 for p3, and so on and pn waits for p1….

The approaches used to handle the deadlock are :

Deadlock Avoidance
Deadlock Prevention
Deadlock Detection and Recovery

Banker’s algorithm is a deadlock avoidance algorithm. The name was chosen since this algorithm can be used in a banking system to ensure that the bank never allocates it’s available cash in such a way that it can no longer satisfy further requests for cash.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. The resources are allocated; otherwise, the process must wait until some other process releases enough resources.

The Basic algorithm:

1. If request[i] > need[i] then error (asked for too much)
2. If request[i] > available[i] then wait (can’t supply it now)
3. Resources are available to satisfy the request:

Let’s *assume* that we satisfy the request. Then we would have:
available = available - request[i]
allocation[i] = allocation [i] + request[i]
need[i] = need [i] - request [i]
Now, check if this would leave us in a safe state; if yes, grant the request, if no, then leave the state as is and cause process to wait.

Steps To Do/algorithm:

- Input need the Claim matrix (C) and Allocation matrix (A) and Resource Vector (R)
- Calculate (C-A) and Available Vector V.
- Test for safety condition of the system.
- Decide on whether the resources have to be allocated or not.

Let Request[i] be the request vector for process P[i]. If Request[i,j] = k, then process P[i] wants k instances of resource type R[j]. When a request for resources is made by process P[i], the following action are taken:
If Request[i] <= Need[i], go to step 2. Otherwise, raise an error condition, since the process has exceeded it’s maximum claim.
If Request[i] <= Available, go to step 3. Otherwise, P[i] must wait, since the resources are not available.
Have the system pretend to have allocated the required resources to process P[i] by modifying the state as follows:

```
Available: = Available - Request[i];
Allocation[i]:= Allocation[i] + Request[i];
Need[i]:= Need[i]- Request[i];
```

If the resulting resource-allocation state is safe, the transaction is completed and process P[i] is allocated it’s resources. However, if the new state is unsafe, then P[i] must wait for Request[i] and the old resource-allocation state is restored.

Oral/Review Questions:

What is deadlock? When does it occur?

What is deadlock prevention?
What is deadlock avoidance?
Which algorithm is used for deadlock avoidance?
What is deadlock detection?

ASSIGNMENT NUMBER: C3

Revised On: 16/12/2019

TITLE	**Study of UNIX system calls for process management.**
PROBLEM STATEMENT /DEFINITION	Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming).
OBJECTIVE	• To get familiar with Linux programming • To study basic Linux commands and utilites • Learn process and thread management calls in Linux.
S/W PACKAGES AND HARDWARE APPARATUS USED	C/C++ editors and compilers for Linux OS Linux OS/Fedora/Ubuntu PC with the configuration as Pentium IV 2.4 GHz. 1 GB RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse
REFERENCES	1. Adam Hoover, "System Programming with C and UNIX", Pearson Education 2. Stallings W., "Operating Systems", 5th Edition, Prentice Hall, 81 - 7808 - 503 - 8. Web references: Linux.org
STEPS	Refer to details

Aim: Study of Linux system calls for process management.
Pre-requisite:
Basics of process management and Linux environment.
Learning Objectives:

- To get familiar with Linux programming
- To study basic Linux commands and utilites
- Learn process and thread management calls in Linux.

Learning Outcomes:

The students will be able to

- Execute basic Linux commands.
- Make use of Linux system calls related to process management.
- Implement and execute programs in Linux environment.

Theory:

- **fork** - create a child process

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group or session.

* The child's parent process ID is the same as the parent's process ID.

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

- **An exec** call will load a *new* program into the process and replace the current running program with the one specified. For example, consider this program, which will execute the ls -l command in the current directory:

There are three main versions of exec which we will focus on:

- execv(char * path, char * argv[]) : given the path to the program and an argument array, load and execute the program

- execvp(char * file, char * argv[]) : given a file(name) of the program and an argument array, find the file in the environment PATHand execute the program
- execvpe(char * file, char * argv[], char * envp[]) given a file(name), an argument array, and the enviroment settings, within the enviroment, search the PATH for the program named file and execute with the arguments.
- **Waiting on a child with wait()**

The wait() system call is used by a parent process to *wait* for the status of the child to change. A status change can occur for a number of reasons, the program stopped or continued, but we'll only concern ourselves with the most common status change: the program terminated or exited. (We will discuss stopped and continued in later lessons.)

System calls provide the interface between a process and the operating system. *These system calls are the routine services of the operating system*.
Linux system call fork () creates a process Exec() ,join() etc.
Steps To Do/algorithm:

1. Study the various Linux process handling system calls.
2. Execute basic Linux commands.
3. Print the information about a process its task structure ids etc.

FAQs:

1. What is use of exe and wait command?
2. Explain multi-threading and threads related system calls in Linux.
3. Which command is used to get process ids?
4. How to kill some running process from command prompt?
5. Which system call is used to create new processes?

ASSIGNMENT NO:Gr D-1:

Revised on:16/12/2019

TITLE	**Implementation of Page replacement algorithms.**
PROBLEM STATEMENT /DEFINITION	Write a Java Program (Using OOP features) to implement paging simulation using 1. FIFO 2. Least Recently Used (LRU) 3. Optimal Algorithms
OBJECTIVE	• Understand virtual memory management Analyze the need of page replacement algorithms • Compare various page replacement algorithms
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit open source Linux (Fedora 20), Eclipse IDE, JAVA, I3 and I5 machines
REFERENCES	• Dhamdhere D., "Systems Programming and Operating Systems", 2nd Edition, ' McGraw Hill, • Stallings W., "Operating Systems", 4th Edition, Prentice Hall, 81 - 7808 - 503 - 8.
STEPS	Refer to details
INSTRUCTIONS FOR WRITING JOURNAL	• Title • Problem Definition • Objectives • Theory • Class Diagram/UML diagram • Test cases • Program Listing • Output • Conclusion

Aim: Implement paging simulation using

1. FIFO
2. Least Recently Used (LRU)
3. Optimal Algorithms

Pre-requisite:

Basic functionalities of OS.
Memory management in OS.

Learning Objectives:

Understand virtual memory management

Analyze the need of page replacement algorithms

Compare various page replacement algorithms

Learning Outcomes:

The students will be able to

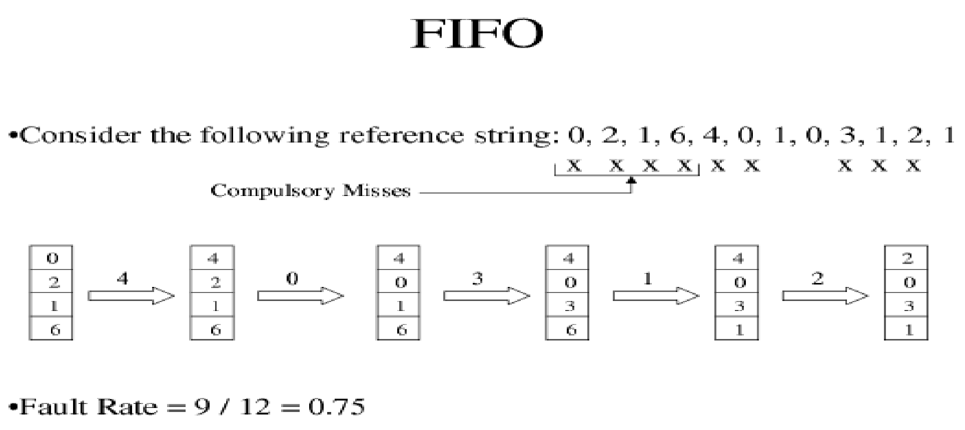
- Implement various page replacement algorithms like FIFO, LRU and Optimal
- Compare the page replacement algorithms based on hit ratio

Theory:

Whenever there is a page reference for which the page needed in not present 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.

First in First out (FIFO)

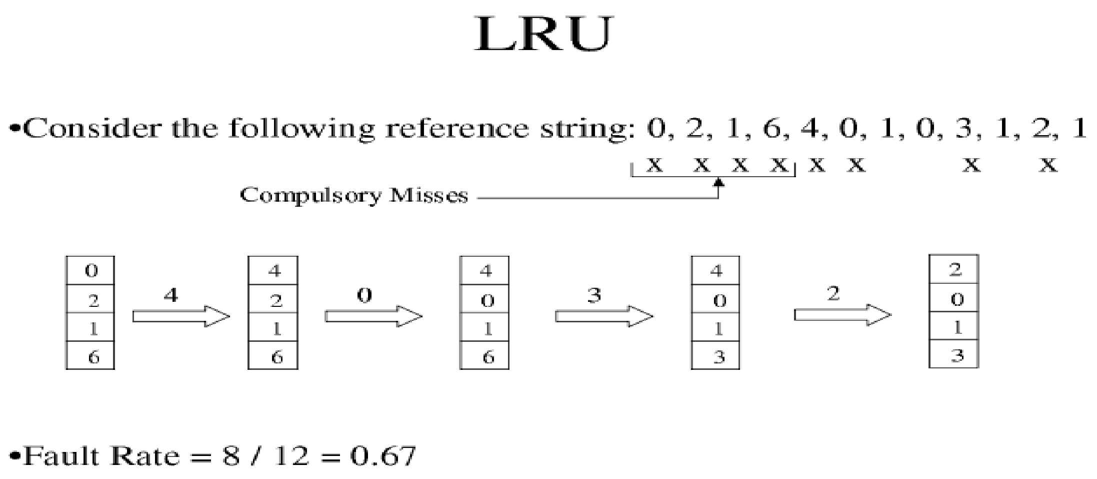
The oldest page in the physical memory is the one selected for replacement. Keep a list On a page fault, the page at the head is removed and the new page added to the tail of the list
Example -


FIFO

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

•Fault Rate = 9 / 12 = 0.75

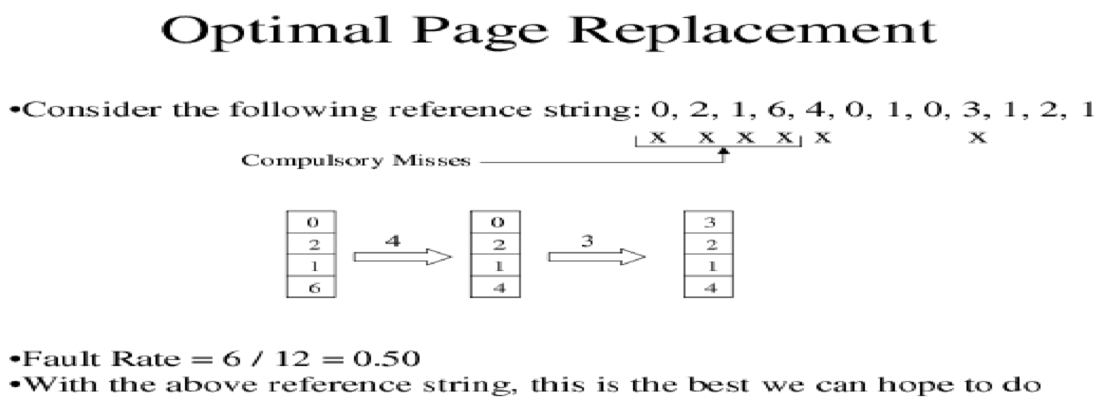
LRU(Least Recently Used): In this algorithm, the page that has not been used for longest period of time is selected for replacement. Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).


LRU

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

•Fault Rate = 8 / 12 = 0.67

The Optimal Page Replacement Algorithm:
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.



Optimal Page Replacement

- Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

- Fault Rate = 6 / 12 = 0.50
- With the above reference string, this is the best we can hope to do

Steps to do /algorithm:

4. Create s menu to select various page replacement algorithms
5. Take no of page frames and pages along with reference strings.
6. Calculate the number of page faults.
7. Perform a comparative assessment of best policy for given reference string.

FAQs:

1. Which page replacement algorithm gives minimum hit ratio?
2. Which data structures are used to implement FIFO policy?
3. Which data structures are used to implement optimal page replacement policy?

Oral/Review Questions:

1. What is page replacement? When does it needed?
2. Name the page replacement algorithms?
3. What is optimal page replacement policy?
4. Which policy assumes locality of references?
5. Which policy needs future knowledge about usage pattern?