GROUP - A

EXPERIMENT NO: 01

1. Title:

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

2. Objectives:

- To understand data structures to be used in pass I of an assembler.
- To implement pass I of an assembler

3. Problem Statement:

Write a program to create pass-I Assembler

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the concept of Pass-I Assembler
- Understand the Programming language of Java

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

A language translator bridges an execution gap to machine language of computer system. An assembler is a language translator whose source language is assembly language.

An Assembler is a program that accepts as input an Assembly language program and converts it into machine language.

Language processing activity consists of two phases, Analysis phase and synthesis phase. Analysis of source program consists of three components, Lexical rules, syntax rules and semantic rules. Lexical rules govern the formation of valid statements in source language. Semantic rules associate the formation meaning with valid statements of language. Synthesis phase is concerned with construction of target language statements, which have the same meaning as source language statements. This consists of memory allocation and code generation.

TWO PASS TRANSLATION SCHEME:

In a 2-pass assembler, the first pass constructs an intermediate representation of the source program for use by the second pass. This representation consists of two main components - data structures like Symbol table, Literal table and processed form of the source program called as intermediate code(IC). This intermediate code is represented by the syntax of Variant –I.

Analysis of source program statements may not be immediately followed by synthesis of equivalent target statements. This is due to forward references issue concerning memory requirements and organization of Language Processor (LP).

Forward reference of a program entity is a reference to the entity, which precedes its definition in the program. While processing a statement containing a forward reference, language processor does not posses all relevant information concerning referenced entity. This creates difficulties in synthesizing the equivalent target statements. This problem can be solved by postponing the generation of target code until more information concerning the entity is available. This also reduces memory requirements of LP and simplifies its organization. This leads to multi-pass model of language processing.

Language Processor Pass: -

It is the processing of every statement in a source program or its equivalent representation to perform language-processing function.

Assembly Language statements: -

There are three types of statements Imperative, Declarative, Assembly directives. An imperative statement indicates an action to be performed during the execution of assembled program. Each imperative statement usually translates into one machine instruction. Declarative statement e.g. DS reserves areas of memory and associates names with them. DC constructs memory word containing constants. Assembler directives instruct the assembler to perform certain actions during assembly of a program,

e.g. START<constant> directive indicates that the first word of the target program generated by assembler should be placed at memory word with address <constant>

Function Of Analysis And Synthesis Phase:

Analysis Phase: -

Isolate the label operation code and operand fields of a statement.

Enter the symbol found in label field (if any) and address of next available machine word into symbol table.

Validate the mnemonic operation code by looking it up in the mnemonics table.Determine the machine storage requirements of the statement by considering the mnemonic operation code and operand fields of the statement.

Calculate the address of the address of the first machine word following the target code generated for this statement (Location Counter Processing)

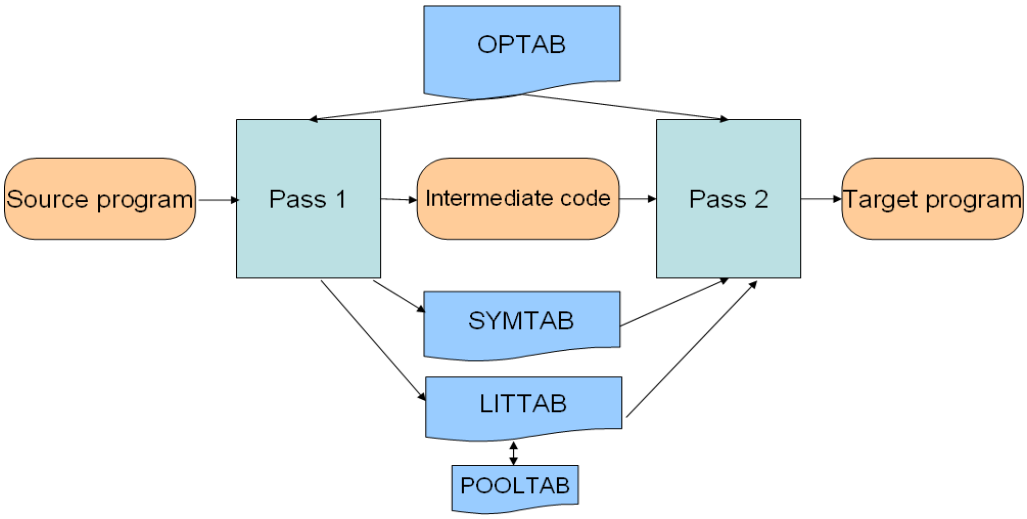
Synthesis Phase:

Obtain the machine operation code corresponding to the mnemonic operation code by searching the mnemonic table.

Obtain the address of the operand from the symbol table.

Synthesize the machine instruction or the machine form of the constant as the case may be.

DATA STRUCTURES OF A TWO PASS ASSEMBLER:



Data Structure of Assembler:

a) Operation code table (OPTAB) :This is used for storing mnemonic,

operation code and class of instruction

Structure of OPTAB is as follows

b) Data structure updated during translation: Also called as translation time data

structure. They are

I. SYMBOL TABLE (SYMTAB) : Ii contains entries such as

symbol,it's address and value.

SYMBOL TABLE have following fields :

Name of symbol	Symbol Address	Value

II. LITERAL TABLE (LITTAB) : it contains entries such as literal

and it's value.

Literal Table has following fields :

literal	**Address of Literal**

III . POOL TABLE (POOLTAB): Contains literal number of the

starting literal of each literal pool.

Pool TABLE (pooltab) have following fields.

LITERAL_NO

IV: Location Counter which contains address of next instruction by

calculating length of each instruction.

Design of a Two Pass Assembler: -

Tasks performed by the passes of two-pass assembler are as follows:

Pass I: -

Separate the symbol, mnemonic opcode and operand fields.

Determine the storage-required foe every assembly language statement and update the location counter.

Build the symbol table and the literal table.

Construct the intermediate code for every assembly language statement.

Pass II: -

Synthesize the target code by processing the intermediate code generated during pass1

INTERMEDIATE CODE REPRESENTATION

The intermediate code consists of a set of IC units, each IC unit consisting of the following three fields

1. Address
2. Representation of the mnemonic opcode
3. Representation of operands

Mnemonic field

The mnemonic field contains a pair of the form: *(statement class, code)*

Where *statement class* can be one of IS,DL and AD standing for imperative statement, declaration statement and assembler directive , respectively.

For imperative statement, *code* is the instruction opcode in the machine language

For declaration and assembler directives , following are the codes

Declaration Statements		Assembler directives	
DC	01	START	01
DS	02	END	02
		ORIGIN	03
		EQU	04
		LTORG	05

Mnemonic Operation Codes :

Instruction Opcode	Assembly Mnemonic	Remarks
00	**STOP**	**STOP EXECUTION**
01	**ADD**	**FIRST OPERAND IS MODIFIED CONDITION CODE IS SET**
02	**SUB**	**FIRST OPERAND IS MODIFIED CONDITION**

		CODE IS SET
03	**MULT**	**FIRST OPERAND IS MODIFIED CONDITION CODE IS SET**
04	**MOVER**	**REGISTER ⟵ MEMORY MOVE**
05	**MOVEM**	**MEMORY MOVE ⟶ REGISTER MOVE**
06	**COMP**	**SETS CONDITION CODE**
07	**BC**	**BRANCH ON CONDITION**
08	**DIV**	**ANALOGOUS TO SUB**
09	**READ**	**FIRST OPERAND IS NOT USED**
10	**PRINT**	**FIRST OPERAND IS NOT USED**

Branch On Condition :

BC <condition code > <memory address>

Transfer Control to the Memory word With Address < memory address>

Condition code	Opcode
LT	01
LE	02
EQ	03
GT	04
GE	05
ANY	06

8. Algorithms(procedure) :

PASS 1

• Initialize location counter, entries of all tables as zero.

• Read statements from input file one by one.

• While next statement is not END statement

I. Tokenize or separate out input statement as label,numonic,operand1,operand2

II. If label is present insert label into symbol table.

III. If the statement is LTORG statement processes it by making it's entry into literal table, pool table and allocate memory.

IV. If statement is START or ORIGEN Process location counter accordingly.

V. If an EQU statement, assign value to symbol by correcting entry in symbol table.

VI. For declarative statement update code, size and location counter.

VII. Generate intermediate code.

VIII. Pass this intermediate code to pass -2.

10. Conclusion:

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

References:

J. J. Donovan, "Systems Programming", McGraw Hill.[chapter 3 topic 3.0,3.1, 3.2.1 in brief ,3.2.2 figure 3.3 &3.5]

Oral Questions: [Write short answer]

1. Explain what is meant by pass of an assembler.
2. Explain the need for two pass assembler.
3. Explain terms such as Forward Reference and backward reference.
4. Explain various types of errors that are handled in passé-I.
5. Explain the need of Intermediate Code generation and the variants used.
6. State various tables used and their significance in the design of two pass Assembler.
7. What is the job of assembler?
8. What are the various data structures used for implementing Pass-I of a two-pass assembler.
9. What feature of assembly language makes it mandatory to design a two pass assembler?
10. How are literals handled in an assembler?
11. How assembler directives are handled in pass I of assembler?

GROUP - A

EXPERIMENT NO: 02

1. Title:

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

2. Objectives:

- To understand data structures to be used in pass II of an assembler.
- To implement pass I of an assembler

3. Problem Statement:

Write a program to create pass-II Assembler

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the concept of Pass-II Assembler
- Understand the Programming language of Java

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

Design of a Two Pass Assembler: -

Tasks performed by the passes of two-pass assembler are as follows:

Pass I: -

Separate the symbol, mnemonic opcode and operand fields.

Determine the storage-required foe every assembly language statement and update the location counter.

Build the symbol table and the literal table.

Construct the intermediate code for every assembly language statement.

Pass II: -

Synthesize the target code by processing the intermediate code generated during pass1

<u>Data Structure used by Pass II:</u>

1. OPTAB: A table of mnemonic opcodes and related information.

2. SYMTAB: The symbol table

3. POOL_TAB and LITTAB: A table of literals used in the program

4. Intermediate code generated by Pass I

5. Output file containing Target code / error listing.

8. Algorithms(procedure) :
Algorithm :

1. **c**ode_area_address=address of code area;

Pooltab_ptr:=1;

loc_cntr=0;

2. While next statement is not an END statement

a) clear the 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.

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 or DS:

e) if an imperative statement then

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)

9. Conclusion:

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

References:

J. J. Donovan, "Systems Programming", McGraw Hill.[chapter 3]

Oral Questions: [Write short answer]

1. Explain various types of errors that are handled in passé-II.
2. Write algorithm of passé-II.
3. Draw flowchart of passé-II.
4. State various tables used and their significance in the design of two pass Assembler.
5. How LTORG statement is handled in pass II of assembler?
6. How Declarative statement is handled in pass II of assembler?
7. What is the significance of pool table?
8. Which data structures of pass I are used in pass II of assembler?
9. Explain the handling of imperative statement.
10. What feature of assembly language makes it mandatory to design a two pass assembler?
11. How are literals handled in an assembler?
12. How assembler directives are handled in pass I of assembler?

GROUP - A

EXPERIMENT NO: 03

1. Title:

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

2. Objectives:

- To Identify and create the data structures required in the design of macro processor.
- To Learn parameter processing in macro
- To implement pass I of macroprocessor

3. Problem Statement:

Write a program to create pass-I Macro-processor

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the Programming language of Java
- Understand the concept of Pass-I Macro-processor

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

MACRO

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

A macro consist of

1. Name of the macro
2. Set of parameters
3. Body of macro

Macros are typically defined at the start of program. Macro definition consists of

1. MACRO pseudo
2. MACRO name
3. Sequence of statements
4. MEND pseudo opcode terminating

A macro is called by writing the macro name with actual parameter is an assembly program. The macro call has following syntax <macro name>.

MACRO PROCESSOR

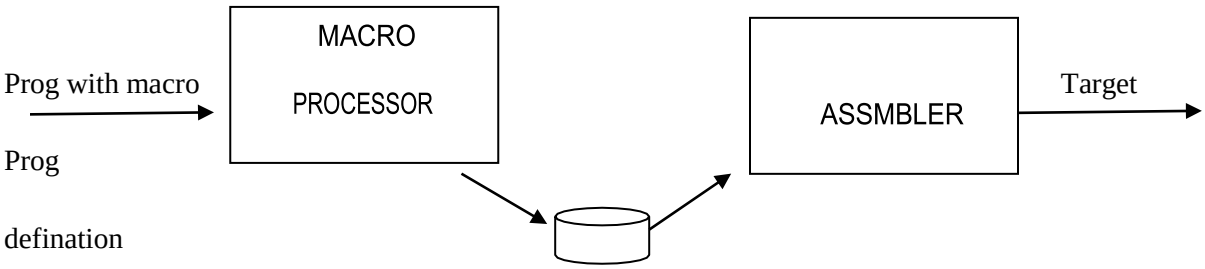


Fig 1. Assembly language program without macro

Macro processor takes a source program containing macro definition & macro calls and translates into an assembly language program without any macro definition or calls. This program can now be handled over to a conventional assembler to obtain the target language.

MACRO DEINITION

Macros are typically defined at the start of a program. A macro definition consists of

1. MACRO pseudo code
2. MACRO name
3. Sequence of statement
4. MEND pseudo opcode terminating macro definition

Structure of a macro

Example
MACRO
INCR & ARG
ADD AREG,& ARG
ADD BRA,& ARG
ADD CREG, & ARG
MEND

MACRO Expansion:

During macro expansion each statement forming the body of the macro as picked up one by one sequentially.

a. Each statement inside macro may have as it is during expansion.

b. The name of a formal parameter which is preceded by the character '&' during macro expansion an ordinary starting is retained without any modification. Formal parameters are replaced by actual parameters value.

When a call is found the call processor sets a pointer the macro definition table pointer to the corresponding macro definition started in MDT. The initial value of MDT is obtained from MDT index.

8. Design of macro processor:

- **Pass I:**
  - Generate Macro Name Table (MNT)
  - Generate Macro Definition Table (MDT)
  - Generate IC i.e. a copy of source code without macro definitions.

MNT:

Sr.No	Macro Name	MDT Index

MDT:

Sr. No	MACRO STATEMENT

ALA:

Index	Argument

Specification of Data Bases

Pass 1 data bases

1. The input macro source desk.
2. The output macro source desk copy for use by passes 2.
3. The macro definition table (MDT) used to store the names of defined macros.
4. Macro name table (MDT) used to stare the name of defined macros.
5. The Macro definition table counter used to indicate the next available entry in MNT.
6. The macro name table counter counter(MNTC) used to indicate next available entry in MNT.
7. The arguments list array (ALA) used to substitute index markers for dummy arguments before starting a macro definition.

9. Conclusion:

Thus we have successfully implemented pass-I of a two-pass Macro-processor.

References:

J. J. Donovan, "Systems Programming", McGraw Hill[chapter 4 topic 4.0, 4.1, 4.3, 4.3.1 figure 4.1]

Oral Questions: [Write short answer]

1. What are macros? Why do we need macros?
2. Explain data structures that are used for implementing Pass I of a macro processor.
3. Explain the macro assembler facilities such as Nested Macro, Labels within Macro, Macro Parameters.
4. What are the contents of MDT and MNT?
5. Explain the algorithm of pass I of macro processor

GROUP - B

EXPERIMENT NO: 04

1. Title:

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

2. Objectives:

- To Identify and create the data structures required in the design of macro processor.
- To Learn parameter processing in macro
- To implement pass II of macroprocessor

3. Problem Statement:

Write a program to create pass-II Macro-processor

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the Programming language of Java
- Understand the concept of Pass-II Macro-processor

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

Pass II:

Replace every occurrence of macro call with macro definition. (Expanded Code)

There are four basic tasks that any macro instruction process must perform:

1. **Recognize macro definition:**

   A macro instruction processor must recognize macro definition identified by the MACRO and MEND pseudo-ops. This tasks can be complicated when macro definition appears within macros. When MACROs and MENDs are nested, as the macro processor must recognize the nesting and correctly match the last or or outer MEND with first MACRO. All intervening text, including nested MACROs and MENDs defines a single macro instruction.

2. Save the definition:

The processor must store the macro instruction definition, which it will need for expanding macro calls.

3. Recognize calls:

The processor must recognize the macro calls that appear as operation mnemonics. This suggests that macro names be handled as a type of op-code.

4. Expand calls and substitute arguments:

The processor must substitute for dummy or macro definition arguments the corresponding arguments from a macro call; the resulting symbolic text is then substitute for macro call. This text may contain additional macro definition or call.

Implementation of a 2 pass algorithm

1. We assume that our macro processor is functionally independent of the assembler and that the output text from the macro processor will be fed into the assembler.
2. The macro processor will make two independent scans or passes over the input text , searching first for macro definitions and then for macro calls
3. The macro processor cannot expand a macro call before having found and saved the corresponding macro definitions.
4. Thus we need two passes over the input text , one to handle macro definitions and other to handle macro calls.
5. The first pass examines every operation code, will save all macro definitions in a macro Definition Table and save a copy of the input text, minus macro definitions on the secondary storage.
6. The first pass also prepares a Macro Name Table along with Macro Definition Table as seen in the previous assignment that successfully implemented pass – I of macro pre-processor.

The second pass will now examine every operation mnemonic and replace each macro name with the appropriate text from the macro definitions.

8. SPECIFICATION OF DATABASE

Pass 2 database:

1. The copy of the input source deck obtained from Pass- I
2. The output expanded source deck to be used as input to the assembler
3. The Macro Definition Table (MDT), created by pass 1
4. The Macro Name Table (MNT), created by pass 1
5. The Macro Definition Table Counter (MNTC), used to indicate the next line of text to be used during macro expansion
6. The Argument List Array (ALA), used to substitute macro call arguments for the index markers in stored macro definition

9. Conclusion:

Thus we have successfully implemented pass-II of a two-pass macro-processor

References:

J. J. Donovan, "Systems Programming", McGraw Hill[chapter 4]

Oral Questions: [Write short answer]

1. Write an algorithm for PASS-II of a two pass macro processor.
2. Explain macro expansion.
3. Draw flowchart of PASS-II of a two pass macro processor.
4. Explain nested Macro calls
5. Explain data structures that are used for implementing Pass II of a macro processor.
6. What is the input and output of pass II of macro processor?
7. What functions are involved in pass II of macro processor?
8. What is Conditional Macro Expansion?
9. Explain the contents of all the tables of pass II of macro processor.

1.

GROUP - B

EXPERIMENT NO: 05

1. **Title:**

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

2. **Objectives:**
   - To understand Dynamic Link Libraries Concepts
   - To implement dynamic link library concepts
   - To study about Visual Basic

3. **Problem Statement:**

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

4. **Outcomes:**

After completion of this assignment students will be able to:

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

5. **Software Requirements:**
   - Visual Studio 2010

6. **Hardware Requirement:**

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

7. **Theory Concepts:**

Dynamic Link Library:

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

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

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

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

Feature of DLL:

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

Difference between the Application & DLL:

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

Executable file links to DLL:

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

- Implicit linking
- Explicit linking

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

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

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

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

Calling DLL function from Visual Basic Application:

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

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

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

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

is decorated as:

```
_func@12
```

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

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

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

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

In the .DEF file:

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

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

DLL's Advantages:

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

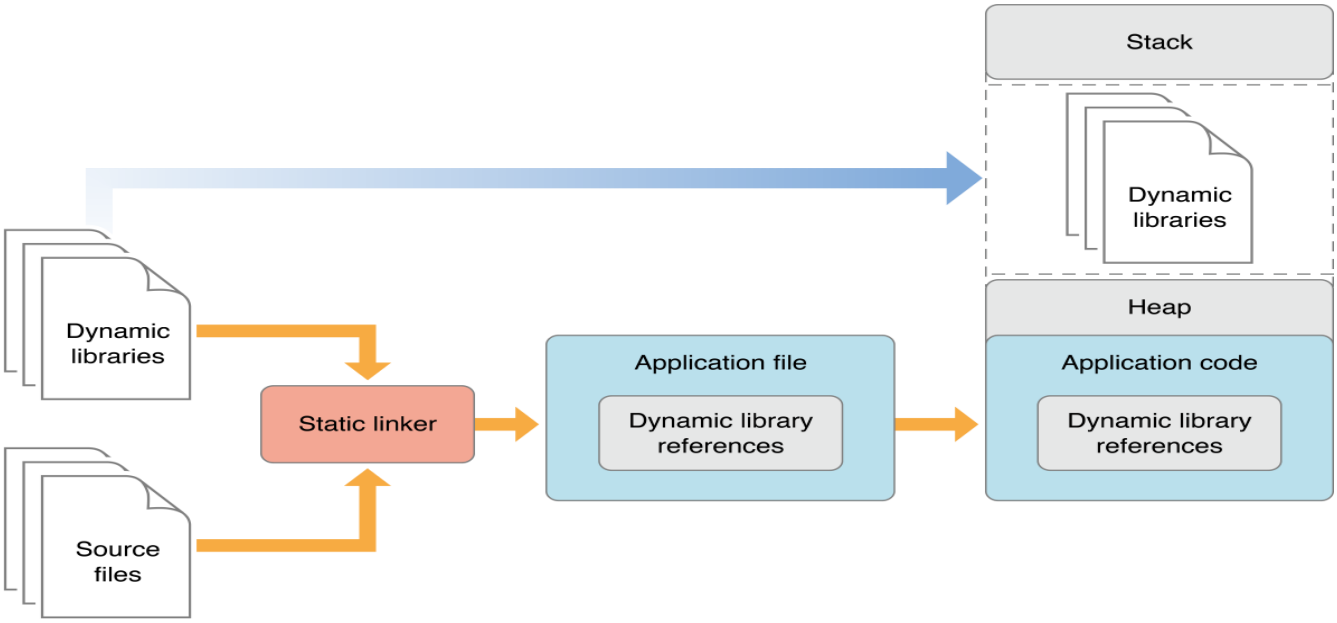
Disadvantage:

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

Visual Basic:

Visual Basic is a third-generation event-driven programming language first released by Microsoft in 1991. It evolved from the earlier DOS version called BASIC. **BASIC** means **B**eginners' **A**ll- purpose **S**ymbolic **I**nstruction **C**ode. Since then Microsoft has released many versions of Visual Basic, from Visual Basic 1.0 to the final version Visual Basic 6.0. Visual Basic is a user-friendly programming language designed for beginners, and it enables anyone to develop GUI window applications easily. In 2002, Microsoft released Visual Basic.NET (VB.NET) to replace Visual Basic 6. Thereafter, Microsoft declared VB6 a legacy programming language in 2008. Fortunately, Microsoft still provides some form of support for VB6. VB.NET is a fully object-oriented programming language implemented in the .NET Framework. It was created to cater for the development of the web as well as mobile applications. However, many developers still favor Visual Basic 6.0 over its successor Visual Basic.NET.

8. Design (architecture):



9. Algorithms(procedure) :

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

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

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

References:

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

Oral Questions: [Write short answer]

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

GROUP - B

EXPERIMENT NO: 6

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

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

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

4. **Outcomes:**

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

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

6. **Hardware Requirement:**

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

7. **Theory Concepts:**

CPU Scheduling:

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

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

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

What is scheduling?

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

What is scheduler?

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

Necessity of scheduling

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

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

• Primary objective of scheduling is to optimize system performance.

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

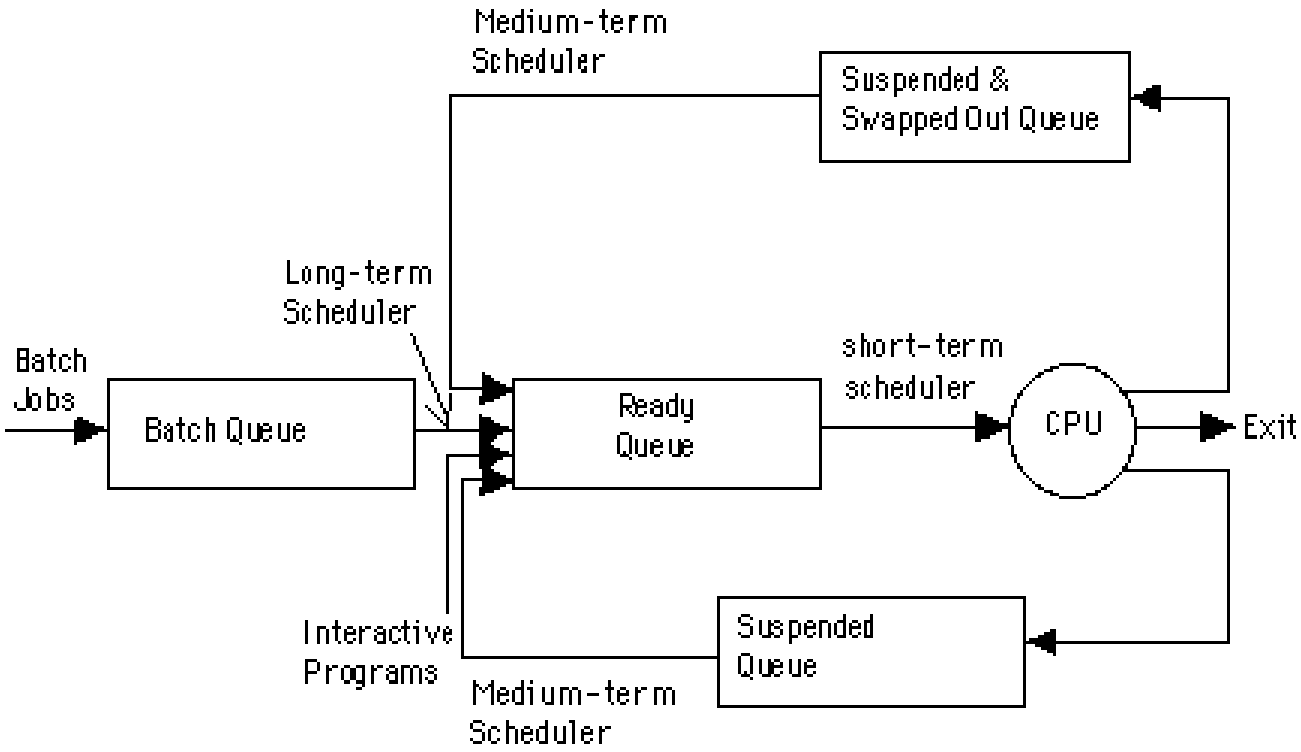
Types of schedulers

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

• Long term scheduler

• Medium term scheduler

• Short term scheduler.



Long Term Scheduler

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

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

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

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

Medium Term Scheduler

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

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

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

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

Short Term Scheduler

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

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

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

• Clock ticks

• Interrupt and I/O completions

• Most operational OS calls

• Sending and receiving of signals

• Activation of interactive programs.

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

Scheduling Criteria :

- **CPU Utilization:**

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

- **Throughput:**

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

- **Turnaround time:**

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

- **Waiting time:**

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

- **Response time:**

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

Non-preemptive Scheduling :

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

Preemptive Scheduling :

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

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

Types of scheduling Algorithms

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

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

There are different types of scheduling algorithms such as:

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

First Come First Serve Algorithm

• FCFS is working on the simplest scheduling discipline.

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

• FCFS scheduling may result into poor performance.

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

Advantages

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

Disadvantages

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

First Come, First Served

Process	Burst Time
P1	24
P2	3
P3	3

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

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

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

Shortest Job First Algorithm :

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

Advantages

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

Disadvantages

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

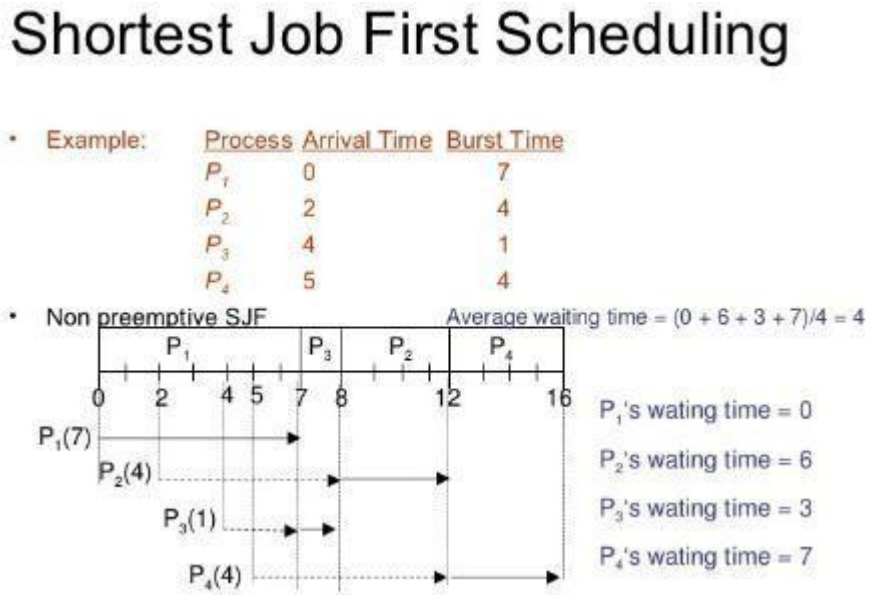
This algorithm is divided into two types:

• Pre-emptive SJF

• Non-pre-emptive SJF

• **Pre-emptive SJF Algorithm:**

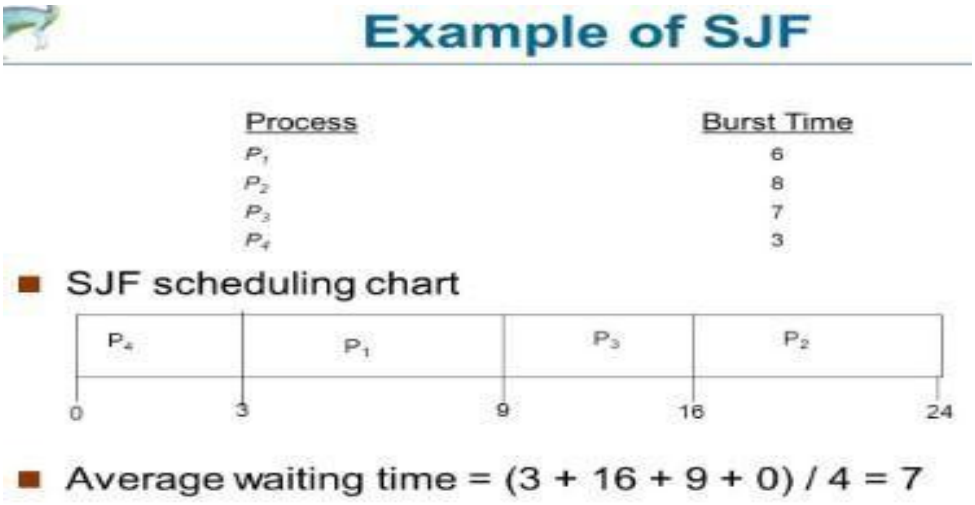
In this type of SJF, the shortest job is executed 1st. the job having least arrival time is taken first for execution. It is executed till the next job arrival is reached.



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

Non-pre-emptive SJF Algorithm:

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



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

Round Robin Scheduling :

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

Advantages

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

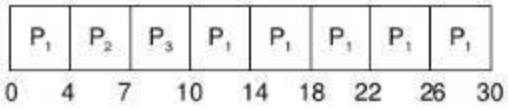
Disadvantages

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

Round Robin

Process	Burst Time
P1	24
P2	3
P3	3

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



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

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

Priority Scheduling :

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

Advantage

- Good response for the highest priority processes.

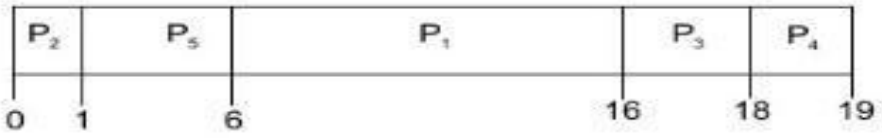
Disadvantage

- Starvation may be possible for the lowest priority processes.

Priority

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

- **Gantt Chart**



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

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

8. Algorithms(procedure) :

FCFS :

Step 1: Start the process

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

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

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

Step 5: for each process in the Ready Q calculate

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

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

Step 6: Calculate

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

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

Step 7: Stop the process

SJF :

Step 1: Start the process

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

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

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

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

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

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

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

Step 6: Calculate

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

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

Step 7: Stop the process

RR :

Step 1: Start the process

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

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

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

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

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

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

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

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

Step 7: Calculate

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

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

Step 8: Stop the process.

Priority Scheduling :

Algorithms :

Step 1: Start the process

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

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

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

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

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

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

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

Step 6: Calculate

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

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

Step 7: Stop the process

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

9. Flowchart :

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

10. Conclusion:

Hence we have studied that-

- CPU scheduling concepts like context switching, types of schedulers, different timing parameter like waiting time, turnaround time, burst time, etc.
- Different CPU scheduling algorithms like FIFO, SJF,Etc.
- FIFO is the simplest for implementation but produces large waiting times and reduces system performance.
- SJF allows the process having shortest burst time to execute first.

References :

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

Oral Questions: [Write short answer]

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

GROUP - B

EXPERIMENT NO: 7

1. Title:

Write a Java Program (using OOP features) to implement paging simulation using
1. Least Recently Used (LRU)
2. Optimal algorithm

2. Objectives:
- To understand concept of paging.
- To Learn different page replacement algorithms.

3. Problem Statement:
Write a program to implement paging simulation

4. Outcomes:
After completion of this assignment students will be able to:
- Understand the Programming language of Java
- Understand the concept of Paging

5. Software Requirements:
- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

Paging

Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous.

In paging, the physical memory is divided into fixed-sized blocks called **page frames** and logical memory is also divided into fixed-size blocks called **pages** which are of same size as that of page frames.

When a process is to be executed, its pages can be loaded into any unallocated frames (not necessarily contiguous) from the disk.

Consider the size of logical address space is 2^m. Now, if we choose a page size of 2^n., then n bits will specify the page offset and m-n bits will specify the page number.

Consider a system that generates logical address of 16 bits and page size is 4 KB. How many bits would specify the page number and page offset?

How a logical address is translated into a physical address:

In paging, address translation is performed using a mapping table, called **Page Table**. The operating system maintains a page table for each process to keep track of which page frame is allocated to which page. It stores the frame number allocated to each page and the page number is used as index to the page table.

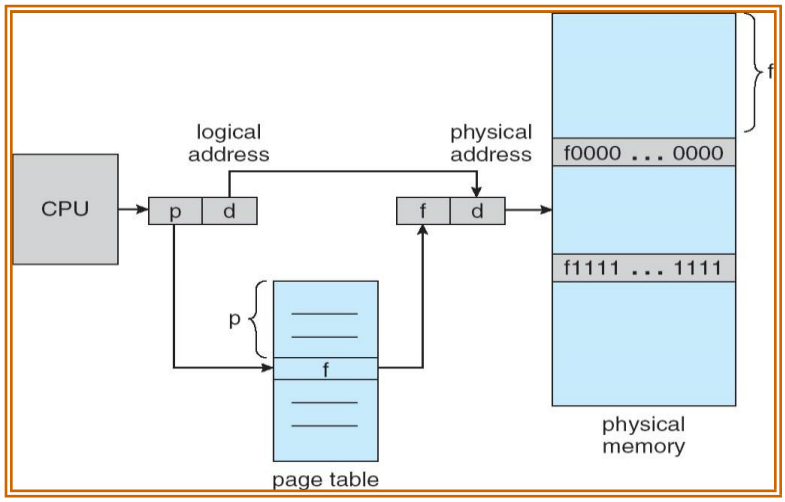
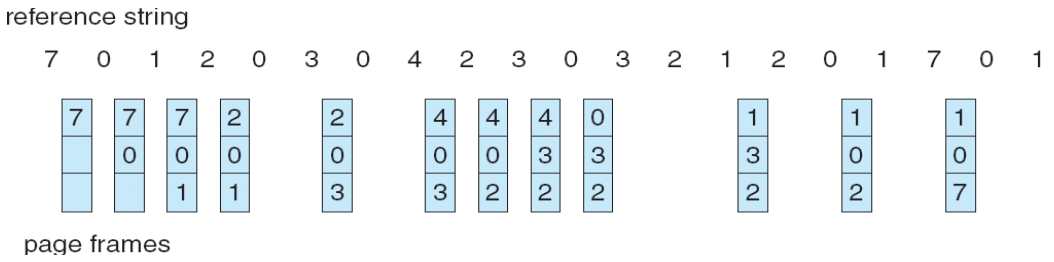


Fig1 paging

LRU

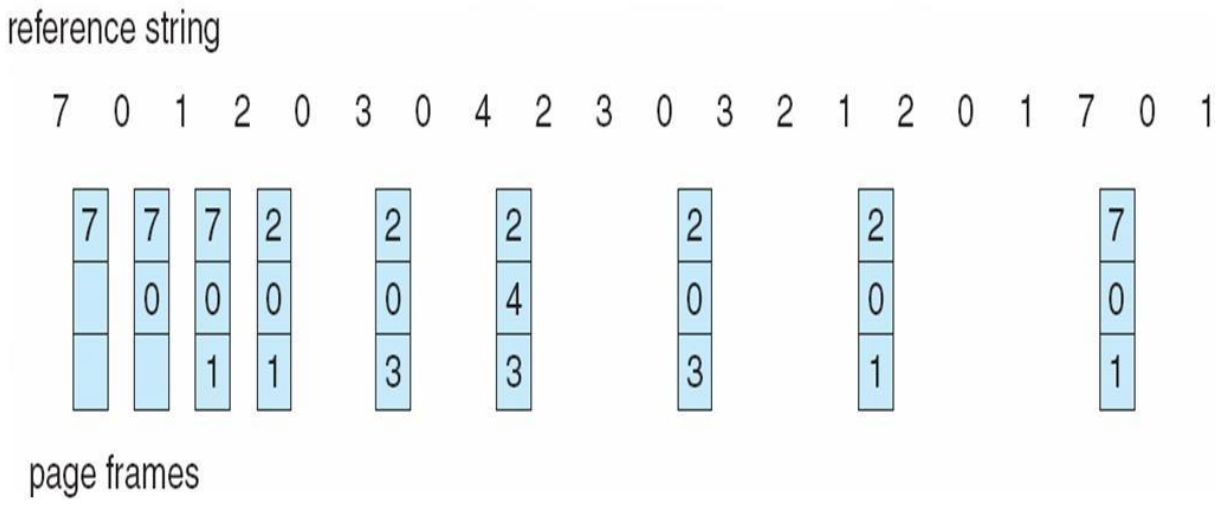
- Replaces the page that has not been referenced for the longest time:
  - By the principle of locality, this should be the page least likely to be referenced in the near future.
  - Performs nearly as well as the optimal policy.



Example of LRU

Optimal

Optimal Page Replacement refers to the removal of the page that will not be used in the future, for the longest period of time.



9. Conclusion:

The various memory management page replacement algorithms were studied and successfully implemented.

References:

Andrew S. Tanenbaum, "Modern Operating Systems", Second Edition, PHI.

[Chapter 3 topic 4.3.1, 4.4, 4.4.3, 4.4.6]

Oral Questions: [Write short answer]

1. What is page fault?

2. What is the difference between physical memory and logical memory?
3. Explain virtual memory.
4. What is the difference between paging and segmentation?
5. What is Belady's Anomaly?
6. Define the concept of thrashing? What is the scenario that leads to the situation of thrashing?