Assignment No. 1(a)- Group A

Problem 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.

1.1 Prerequisite:

Basic concepts of Assembler pass 1& pass2(syntax analyzer)

1.2 Learning Objectives:

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

1.3 . 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.

❖ 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.

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.

❖ 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.
II. LITERAL TABLE (LITTAB) : it contains entries such as literal and it's value.

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

1.4 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

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

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 outinput 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 andlocation counter.
VII. Generate intermediate code.
VIII. Pass this intermediate code to pass -2.

1.5 Conclusion:

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

Assignment Questions:

1. Explain the need for two pass assembler.
2. What is the job of assembler?
3. What are the various data structures used for implementing Pass-I of a two-pass assembler.
4. How are literals handled in an assembler?

PROGRAM CODE:

```
package exp1;

import java.io.;
class pass1
{
   public static void main(String args[])throws Exception
   {
   FileReader FP=new FileReader(CUsersLAB A-
   26Desktopinput.txt); BufferedReader bufferedReader = new
   BufferedReader(FP);

   String line=null;
   int line_count=0,LC=0,symTabLine=0,opTabLine=0,litTabLine=0,poolTabLine=0;

    Data Structures
    final int MAX=100;
    String SymbolTab[][]=new
    String[MAX][3]; String OpTab[][]=new
    String[MAX][3]; String LitTab[][]=new
    String[MAX][2];
    int PoolTab[]=new int[MAX];
    int litTabAddress=0;
---------------------------------------------------------------

    System.out.println(_________________________________________);
      while((line = bufferedReader.readLine()) != null)
       {
         String[] tokens = line.split(t);
        if(line_count==0)
        {
                LC=Integer.parseInt(tokens[1]); set
LC to operand of START
                for(int i=0;itokens.length;i++)              for printing the inputprogram
                        System.out.print(tokens[i]+t);
                System.out.println();
        }
        else
        {
                 for(int i=0;itokens.length;i++) for printing the input program
                        System.out.print(tokens[i]+t);
                 System.out.println();
                if(!tokens[0].equals())
                {

                        Inserting into Symbol Table
                        SymbolTab[symTabLine][0]=tokens[0];
                        SymbolTab[symTabLine][1]=Integer.toString(LC);
                        SymbolTab[symTabLine][2]=Integer.toString(1);
                        symTabLine++;
                }
                else if(tokens[1].equalsIgnoreCase(DS)tokens[1].equalsIgnoreCase(DC))
```

```
		{
			Entry into symbol table for declarative statements
			SymbolTab[symTabLine][0]=tokens[0];
			SymbolTab[symTabLine][1]=Integer.toString(LC);
			SymbolTab[symTabLine][2]=Integer.toString(1);
			symTabLine++;
		}

		if(tokens.length==3 && tokens[2].charAt(0)=='=')
		{
			Entry of literals into literal table
			LitTab[litTabLine][0]=tokens[2];
			LitTab[litTabLine][1]=Integer.toString(LC);
			litTabLine++;
		}

		else if(tokens[1]!=null)
		{
				Entry of Mnemonic in opcode table
			OpTab[opTabLine][0]=tokens[1];


if(tokens[1].equalsIgnoreCase(START)tokens[1].equalsIgnoreCase(END)tokens[1].equa
lsIgnoreCase(ORIGIN)tokens[1].equalsIgnoreCase(EQU)tokens[1].equalsIgnoreCase(LT
ORG))		if Assembler Directive
			{
				OpTab[opTabLine][1]=AD;
				OpTab[opTabLine][2]=R11;
			}
			else
if(tokens[1].equalsIgnoreCase(DS)tokens[1].equalsIgnoreCase(DC))
			{
				OpTab[opTabLine][1]=DL;
				OpTab[opTabLine][2]=R7;
			}
			else
			{
				OpTab[opTabLine][1]=IS;
				OpTab[opTabLine][2]=(04,1);
			}
		opTabLine++;
		}
	}
```

.

```
              line_count++;
              LC++;
            }

            System.out.println(_____________________________________________);

            print symbol table
            System.out.println(nn           SYMBOL TABLE              );
            System.out.println( ------------------------ );
            System.out.println(SYMBOLtADDRESStLENGTH
            ); System.out.println( --------------------- );
            for(int i=0;isymTabLine;i++)
                    System.out.println(SymbolTab[i][0]+t+SymbolTab[i][1]+t+SymbolTab[i][2]);
            System.out.println( ------------------------ );


            print opcode table
            System.out.println(nn           OPCODE TABLE              );
            System.out.println(.......................................);
            System.out.println(MNEMONICtCLASStINFO
            ); System.out.println( ------------------------ );
            for(int i=0;iopTabLine;i++)
                    System.out.println(OpTab[i][0]+tt+OpTab[i][1]+t+OpTab[i][2]);
            System.out.println(.......................................);

            print literal table
            System.out.println(nn   LITERAL TABLE           );
            System.out.println( ----------------);
            System.out.println(LITERALtADDRESS
            ); System.out.println( -------------);
            for(int i=0;ilitTabLine;i++)
                    System.out.println(LitTab[i][0]+t+LitTab[i][1]);
            System.out.println( -----------------);


            intialization of POOLTAB
            for(int i=0;ilitTabLine;i++)
            {
                    if(LitTab[i][0]!=null && LitTab[i+1][0]!=null ) if literals are present
                    {
                            if(i==0)
                            {
                                    PoolTab[poolTabLine]=i+1;
                                    poolTabLine++;
                            }
                            else
        if(Integer.parseInt(LitTab[i][1])(Integer.parseInt(LitTab[i+1][1]))-1)
                            {
                                    PoolTab[poolTabLine]=i+2;
                                    poolTabLine++;
                            }
     .              }
            }
            print pool table
            System.out.println(nn   POOL TABLE                );
            System.out.println( ----------------);
```

```
System.out.println(LITERAL
NUMBER); System.out.println( -);
for(int i=0;ipoolTabLine;i++)
        System.out.println(PoolTab[i]);
System.out.println(-----------------);


 Always close files.
bufferedReader.close();
}
}
```

.

INPUT:

```
START       100
            READ        A
LABLE       MOVER       A,B
            LTORG
                        ='5'
                        ='1'
                        ='6'
                        ='7'
            MOVEM       A,B
            LTORG
                        ='2'
LOOP        READ        B
A           DS          1
B           DC          '1'
                        ='1'
            END
```

.

Assignment No.1 (b) Group A

Problem Definition:

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

Problem Statement: - To write programs to implement pass two of a two pass assembler.

Pre-requisites:- Basic System Programming.

Software Requirements:- java, eclipse

Hardware Requirements: - No

Objectives: - 1. To design and implement Pass II of two pass assembler.
2. To implement basic language translator
3. Convert intermediate file to Target file.

Outcomes: - After completion of this program students will be able to accept input of Intermediate file & Convert to Output/Target file.

Theory:-

What is a single pass assembler?

It is an assembler that generally generates the object code directly in memory for immediate execution.

It passes through your source code only once and you are done. Now let us see how a two pass assembler works.

Simple, while on its way, if the assembler encounters an undefined label, it puts it into a symbol table along with the address where the undefined symbol's value has to be placed when the symbol is found in future.

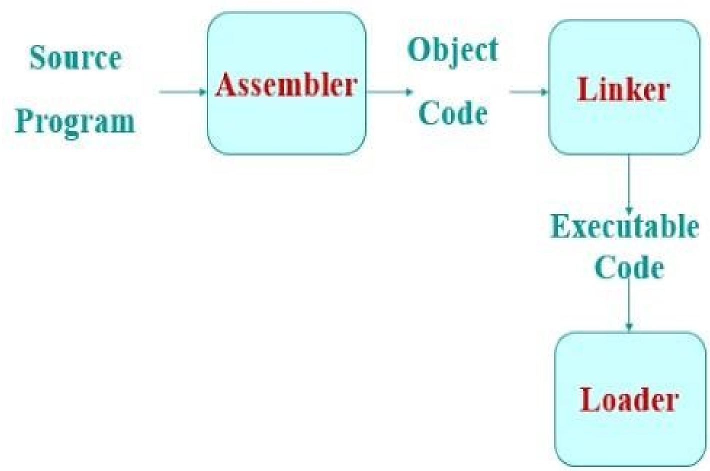
.

WHY DO WE NEED A TWO-PASS ASSEMBLER?

As explained, the one-pass assembler cannot resolve forward references of data symbols.

It requires all data symbols to be defined prior to being used. A two-pass assembler solves this dilemma by devoting one pass to exclusively resolve all (data/label) forward references and then generate object code with no hassles in the next pass.
If a data symbol depends on another and another depends on yet another, the assembler resolvedthis recursively.

1. **PASS II OF THE ASSEMBLER**



.

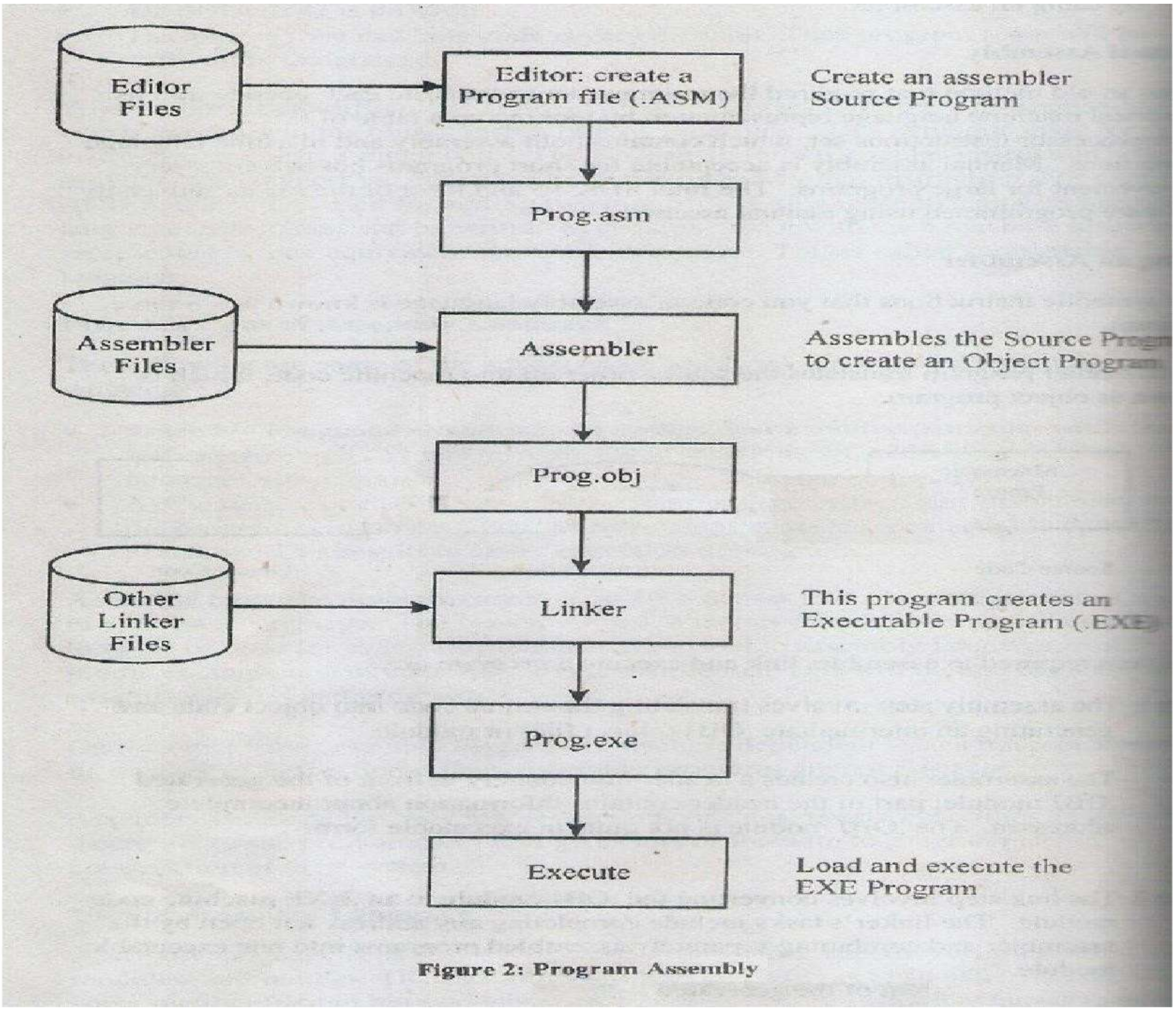


Fig. Steps for Pass II of two pass assembler

HOW IT WORKS:-

1. *READ I/P OF PASS1 AS INTERMEDIATE FILE*

2. *SYNTHESIZE THE TARGET PROGRAM*

Algorithm:-

(1) Code Area Address = address of code area (where target code is to be enabled)Pool tab ptr = '1'
Loc ctr = o (defined)

(2) While the next statement is not an END statement.

(a) Clear memory buffer area.

(b) If an LTORG statement.
(i) Process the literals and assemble the literals in memory buffer.
(ii) Size = size of memory area reqd. for literals.
(iii) Pooltab ptr = Pool tab ptr + 1

(c) If a START or ORIGIN statement then
Loc ctr = value specified in operand field.
size = 0

(d) If a declaration statement
(i) If a DC statement
Assemble the constant in memory buffer
(ii)If DS statement
Generate machine code
Size = size of memory req. by DC/DS

(e) If an imperative statement
(i) Assemble instruction in memory buffer.
(ii) Size = size reqd. to store instruction

(f) If size != 0
Store the memory buffer code in code area address.
Loc ctr = loc ctr + size

(3) END statement
(a) Perform steps 2(a) and 2(f)
(b) Write code area into o/p file.

Conclusion:-

Thus pass II of two passes assembler is implemented and .txt pass2 (.exe target) file is generated.

.

Assignment no 5 – Group B

Problem Definition:

Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive).

1.6 Prerequisite:

Basic concepts of CPU Scheduling algorithm

1.7 Learning Objectives:

- To understand CPU Scheduling Algorithms: FCFS, SJF (Preemptive).
- To implement Priority (Non-Preemptive) and Round Robin (Preemptive).

1.8 . Theory Concepts:

We are given with the n number of processes i.e. P1, P2, P3,.......,Pn and their corresponding burst times. The task is to find the average waiting time and average turnaround time using FCFS CPU Scheduling algorithm.

What is Waiting Time and Turnaround Time?

- Turnaround Time is the time interval between the submission of a process and its completion.
  Turnaround Time = completion of a process – submission of a process
- Waiting Time is the difference between turnaround time and burst time
  Waiting Time = turnaround time – burst time

What is FCFS Scheduling?

First Come, First Served (FCFS) also known as First In, First Out(FIFO) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue.

FCFS follows non-preemptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not get terminated or may get halted due to some I/O interrupt.

Example

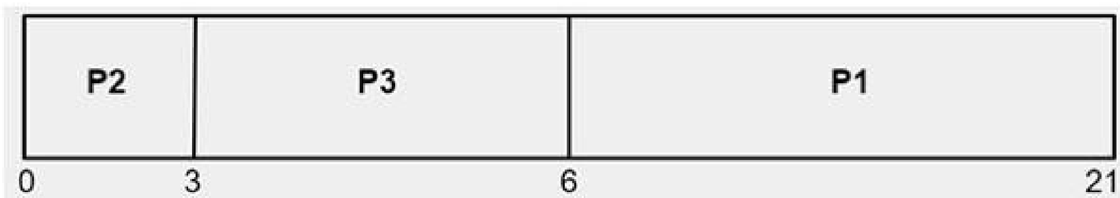
Let's say, there are four processes arriving in the sequence as P2, P3, P1 with their corresponding execution time as shown in the table below. Also, taking their arrival time to be 0.

.

Process	Order of arrival	Execution time in msec

Process	Order of arrival	Execution time in msec
P1	3	15
P2	1	3
P3	2	3

Gantt chart showing the waiting time of processes P1, P2 and P3 in the system



As shown above,

The waiting time of process P2 is 0

The waiting time of process P3 is 3

The waiting time of process P1 is 6

Average time = (0 + 3 + 6) / 3 = 3 msec.

As we have taken arrival time to be 0 therefore turn around time and completion time will be same

1.1 Conclusion:

Thus, I have studied CPU Scheduling Algorithms : FCFS, SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive).

Program Code :

```
import java.util.*;

public class FCFS {
	public static void main(String args[])
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("enter no of process: ");
		int n = sc.nextInt();
		int pid[] = new int[n];  // process ids
		int ar[] = new int[n];     // arrival times
		int bt[] = new int[n];     // burst or execution times
		int ct[] = new int[n];     // completion times
		int ta[] = new int[n];     // turn around times
		int wt[] = new int[n];     // waiting times
		int temp;
		float avgwt=0,avgta=0;

		for(int  i = 0 ; i< n;  i++)
```

```
{
    System.out.println("enter process " + (i+1) + " arrival time: ");
    ar[i] = sc.nextInt();
    System.out.println("enter process " + (i+1) + " brust time: ");
    bt[i] = sc.nextInt();
    pid[i] = i+1;
}

//sorting according to arrival times
for(int i = 0 ; i <n; i++)
{
    for(int  j=0;  j < n-(i+1) ; j++)
    {
        if( ar[j] > ar[j+1] )
        {
            temp = ar[j];
            ar[j] = ar[j+1];
            ar[j+1] = temp;
            temp = bt[j];
            bt[j] = bt[j+1];
            bt[j+1] = temp;
            temp = pid[j];
            pid[j] = pid[j+1];
            pid[j+1] = temp;
        }
    }
}

// finding completion times
for(int  i = 0 ; i < n; i++)
{
    if( i == 0)
    {
        ct[i] = ar[i] + bt[i];
    }
    else
    {
        if( ar[i] > ct[i-1])
        {
            ct[i] = ar[i] + bt[i];
        }
        else
            ct[i] = ct[i-1] + bt[i];
    }
    ta[i] = ct[i] - ar[i] ;          // turnaround time= completion time- arrival
time
    wt[i] = ta[i] - bt[i] ;          // waiting time= turnaround time- burst time
    avgwt += wt[i] ;                 // total waiting time
    avgta += ta[i] ;                 // total turnaround time
}

System.out.println("\npid  arrival  brust  complete turn waiting");
for(int  i = 0 ; i< n;  i++)
```

```
        {
            System.out.println(pid[i] + " \t " + ar[i] + "\t" + bt[i] + "\t" + ct[i] + "\t"
+ ta[i] + "\t"  + wt[i] ) ;
        }
        sc.close();
        System.out.println("\naverage waiting time: "+ (avgwt/n));     // printing
average waiting time.
        System.out.println("average turnaround time:"+(avgta/n));    // printing average
turnaround time.
    }
}
```

.

Assignment No. 6 – Group B

Problem Definition:

Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit.

1.1 Prerequisite:

Basic concepts of Memory placement strategies

1.2 Learning Objectives:

2 To understand different memory placement strategies
3 - To implement different memory placement strategies
4 - To study different memory placement strategies

4.1 . Theory Concepts:

A.First Fit Memory Allocation

This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to it's size. The operating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

Example :

Input :

block Size[]= {100, 500, 200, 300, 600};process Size[] = {212, 417, 112, 426};

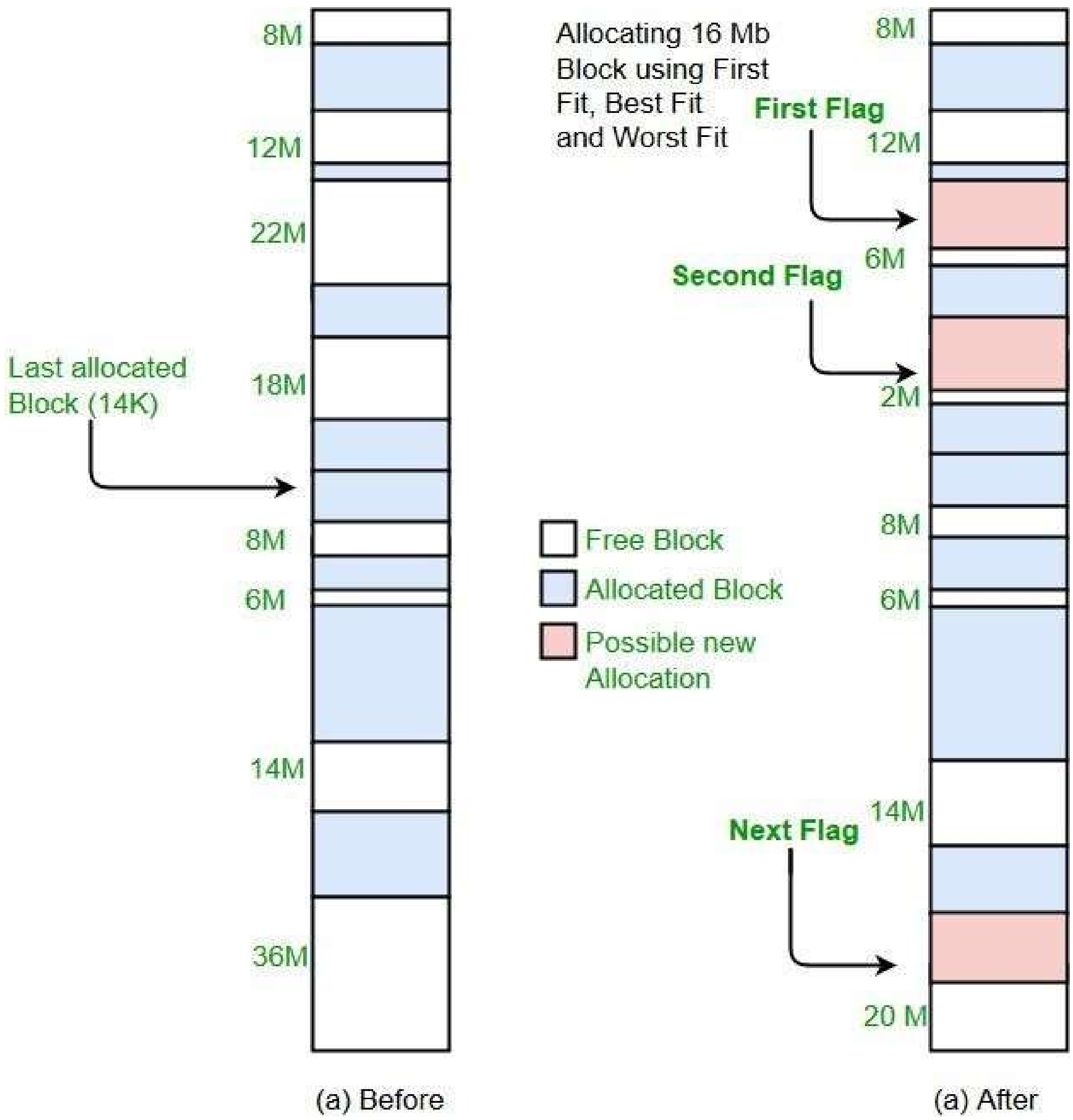
Output:

Process No.	Process Size	Block no.
1	212	2
2	417	5
3	112	2
4	426	Not Allocated

Implementation:

```
1- Input memory blocks with size and processes with size.
2- Initialize all memory blocks as free.
3- Start by picking each process and check if it can
   be assigned to current block.
4- If size-of-process <= size-of-block if yes then
   assign and check for next process.
5- If not then keep checking the further blocks.
```

.



Program Code :

```
// Java implementation of First - Fit algorithm

// Java implementation of First - Fit algorithm
class GFG
{
    // Method to allocate memory to
    // blocks as per First fit algorithm
    static void firstFit(int blockSize[], int m,
                         int processSize[], int n)
    {
        // Stores block id of the
        // block allocated to a process
        int allocation[] = new int[n];

        // Initially no block is assigned to any process
        for (int i = 0; i < allocation.length; i++)
```

```
            allocation[i] = -1;

        // pick each process and find suitable blocks
        // according to its size ad assign to it
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                if (blockSize[j] >= processSize[i])
                {
                    // allocate block j to p[i] process
                    allocation[i] = j;

                    // Reduce available memory in this block.
                    blockSize[j] -= processSize[i];

                    break;
                }
            }
        }

        System.out.println("\nProcess No.\tProcess Size\tBlock
no.");
        for (int i = 0; i < n; i++)
        {
            System.out.print(" " + (i+1) + "\t\t" +
                             processSize[i] + "\t\t");
            if (allocation[i] != -1)
                System.out.print(allocation[i] + 1);
            else
                System.out.print("Not Allocated");
            System.out.println();
        }
    }

    // Driver Code
    public static void main(String[] args)
    {
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = blockSize.length;
        int n = processSize.length;

        firstFit(blockSize, m, processSize, n);
    }
}
```

.

Output :

Process No.	Process Size	Block no.
1	212	2
2	417	5
3	112	2
4	426	Not Allocated

.