Page No.
Date

Assignment no 1

Q1 Imagine you are designing an operating system for a time-sensitive industrial robot. How would you manage process scheduling & what objectives would you prioritize to ensure the robot operates safely & efficiently?

→

To design an operating system for time sensitive industrial robot, I would prioritize the following :-

* Process schedulling
  1) Real time schedulling : Prioritize critical tasks
  2) Priority-Based : Give higher priority to urgent task & lower to others
  3) Fixed Timers : Schedule regular tasks, like sensor checks at consistent intervals

* objectives
  1) Real time response : Ensure quick reaction to maintain performance
  2) Safety : Stop the robot or enter safe mode if task fails
  3) Efficiency
  4) Fault tolerance : Continue safely in case of hardware failures.

Q2] Write a simple bash script to access array elements ?

→

```
#! /bin/bash
# Declare an array
my_array = ("apple", "cherry", "banana")

#access Indivival elements
echo "First element : ${my_array[0]}"
echo "Second element : ${my_array[1]}"
echo "Third element : ${my_array[2]}"

# access all elements
echo "all elements : ${my_array[@]}"

# loop through the array
echo "looping through the array : "
for element in "${my_array[@]}"
    echo "$element"
done
```

Q3] Write script that acts as a basic calculator (addition, substraction, multiplication, division)

→

$ cat arithmatic_operator.sh

Script :-

```
a = 10
b = 20
```

```
val = 'expr $a + $b'
echo "a+b : $val"

val = 'expr $a + - $b'
echo " a-b : $val "

val = 'expr $a * $b'
echo " a*b : $val "

val = 'expr $b / $a'
echo " b/a : $val "
```

Output :-

```
a+b : 30
a-b : -10
a*b : 200
b/a : 2
```

Assignment 2

Q1 Consider the following set of process & Apply RR with $q = 1$, SJF & show Avg. waiting time & Turn around time

Process	Arrival Time	Processing Time
A	0	3 2 1 X
B	1	5 4 3 2 1 X
C	3	2 1 X
D	9	5 4 3 2 1 X
E	12	5 4 3 2 1 X

→ 1) Round Robin (RR) with $q = 1$

Ready queue

A	B	A	C	A	B	C	B	B	D	B	D	E	D	E	D	E	D	E	E

Running queue

A	B	A	C	A	B	C	B	B	D	B	D	E	D	E	D	E	D	E	E

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Process	Arrival Time	Processing Time	Completion Time	TAT	WT
A	0	3	5	5	2
B	1	5	11	10	5
C	3	2	7	4	2
D	9	5	18	9	4
E	12	5	20	8	3

So

Avg waiting time $= \frac{2+5+2+4+3}{5}$

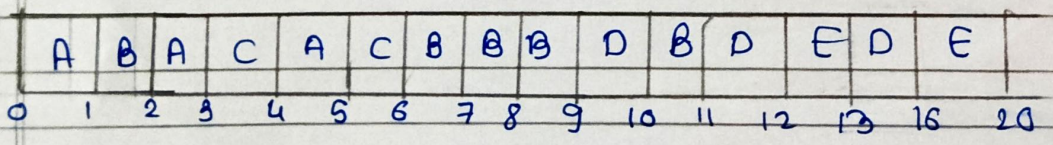
$= 3.2$

Avg Turnaround time $= \frac{5+10+4+9+8}{5}$

$= 7.2$

2] SJF

Process	Arrival Time	Processing Time	Completion Time	TAT	WT
A	0	3	5	5	2
B	1	5	11	10	5
C	3	2	6	3	1
D	9	5	16	7	2
E	12	5	20	8	3



Avg waiting time $= \frac{5+10+3+7+8}{5} = 6.6$

Page No.
Date

Avg waiting time $= \frac{2+5+1+2+3}{5}$

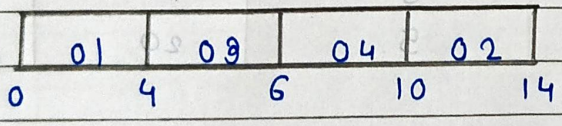
$= \frac{13}{5}$

$= 2.6$

Q2 For the table given below, draw grant chart illustrating process execution using priority non-preemptive scheduling algorithm

Process	A.T	B.T	Priority
01	0	5	4
02	2	4	2
03	2	2	6 (highest)
04	4	4	3

→



Q3 Write a problem to create thread using Pthread library

→

Problem statement :-

Write a C program that creats Thread using the pthread library. The thread should print "Hello from the thread!" to the Console. The main thread should print "Hello from the main thread!"

Solution :-

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void * thread_function (void * arg ) {
printf (" Hello from the thread ! " );
    return NULL ;
}

int main () {
        pthread_t thread ;

if ( pthread_create (&thread , NULL , thread_function,
NULL ))
{
    fprintf (stderr , " Error " );
```

```
        return 1;
    }

    printf ("Hello from the main thread!");

    if (pthread_join (thread, NULL))
    {
        fprintf (stderr, "Error joining thread");
        return 2;
    }
        return 0;
}
```

Output :-

```
Hello from the main thread!
Hello from the thread!
```

Assignment no : 5

1) consider the following snapshot of the system

	Allocation			Max			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

(i) What is the content of Matrix need ?

→

The need of Matrix is calculated =

Need = Max - Allocation

For P1 : Need P1 = (7-0, 5-1, 3-0) = (7, 4, 3)

For P2 : Need P2 = (3-2, 2-0, 2-0) = (1, 2, 2)

For P3 : Need P3 = (9-3, 0-0, 2-2) = (6, 0, 0)

For P4 : Need P4 = (2-2, 2-1, 2-1) = (0, 1, 1)

For P5 : Need P5 = (4-0, 3-0, 3-2) = (4, 3, 1)

Need matrix is =

7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

ii) Is the system in a safe state ?
What is safe sequence ?

→

- We will use Banker's Algorithm to find the sequence

Available Resources = (3, 3, 2)

(a) P2 :

Need : (1, 2, 2) Which is available = (3, 3, 2)

- So P2 can run

update Available = (3+2, 3+0, 2+0) = (5, 3, 2).

(b) P4 :

Need = (0, 1, 1) Which is available = (5, 3, 2)

- So P4 can run.
- update Available = (5+2, 3+1, 2+1) = (7, 4, 3)

(c) P1 :-

Need = (7, 4, 3) Which is available

so P1 can run

update Available = (7+0, 4+1, 3+0) = (7, 5, 3)

P3 :-

- Need = (6,0,0) which is available
- So P3 can run
- update available = (7+3, 5+0, 3+2) = (10, 5, 5)

P5 :-

Need = (4,3,1) which is available
- So P5 can run
- update available = (10+0, 5+0, 5+2) = (10, 5, 7)

Safe sequence :- P2, P4, P1, P3, P5

Q.2) Apply semaphore technique to solve reader Write Synchronization problem :

→

Semaphore :-

1) Mutex (Binary Semaphore) :-
Ensures mutual exclusion while Modifying shared variable like the reader count.

2) rw_mutex (Binary semaphore) -
- Ensures Mutual exclusion for Writers and prevent readers from accessing while a Writer is writing.

variables :-

read_count : keep track of the number of readers currently reading

(1) Reader process :-

```
Wait (mutex);
read_count ++;
if (read_count == 1)
    Wait (rw_mutex);
Signal (mutex);

Wait (mutex);
read_count --;
if (read_count == 0)
    Signal (rw_mutex);
Signal (mutex);
```

(b) Writer process

```
Wait (rw_mutex);
Signal (rw_mutex);
```

3) How semaphore is applied to dining philosopher problem

→ Semaphore

1) Mutex :- Ensures mutual exclusion when philosophers pick up or put down forks

2) semaphore fork[N]:- an array of semaphores, where each fork[i] represents a fork between philosphe. i and i+1

Solution :-

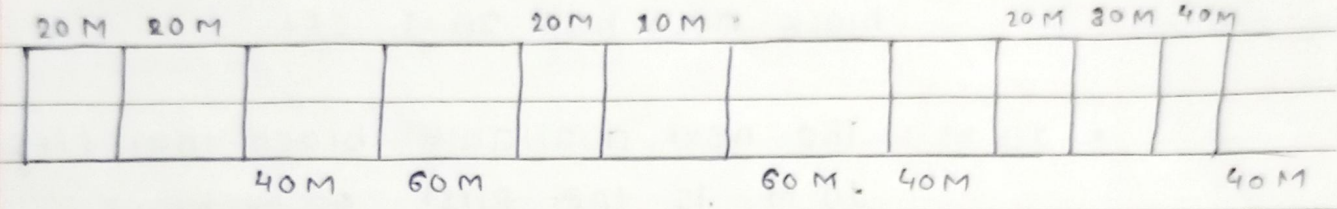
- mutex = 1 : controls critical section
- fork[i] = 1 : for each fork :

Philosopher problem :-

```
do {
    wait(mutex);
    wait (fork[i]);
    wait (fork[(i+1) % N]);
    signal(mutex);

    signal (fork[i]);
    signal (fork[(i+1) % N]);

} while (true);
```

Assignment : 4

1. A dynamic partitioning scheme is being used, and the following is the Memory configuration at a given point of time



The shaded area are allocated blocks ; the white areas are free blocks.

The next three memory requests are 40M, 20M and 10M

Indicate the starting address for each of three blocks using following placement algorithm

(i) first-fit

(ii) Best-fit

(iii) Worst-fit.

Ans: Memory configuration :

free blocks : 60M, 40M, 60M, 40M

1) First - Fit

- 40 M : First free block that fit 40 M is the first 60 M block
- 20 M : The remaining part of same 60 M block now has 20 M left.
- 10 M : The next available block that fits 10 M is the first 40 M block.

∴ 40 M → Start at 40 M
∴ 20 M → Starts at 80 M
∴ 10 M → Starts at 160 M

2) Best - Fit

- 40 M :- (starting address = 160 M)
- 20 M = The smallest free block that fits 20 M is Remaining 20 M of first 60 M block (starting address = 80 M)
- 10 M = (starting address = 120 M)

40 M → Starts at 160 M
20 M → Starts at 80 M
10 M → Starts at 120 M

(3) Worst-fit :-

40 M : Worst-fit is the largest block.
20 M : The remaining part of same 60 M block
10 M : The largest remaining block is the second 60 M block.

∴ 40 M → starts at 40 M
20 M → starts at 80 M
10 M → starts at 120 M.

Q.2) Assume the disk with 200 tracks and disk request queue has random requests in it as follows: 55, 58, 39, 18, 90, 160, 150, 38, 184

Find the number of tracks traversed and average seek length is

(1) SSTF :-
- Shortest Seek Time First.

Initial head = 100

Track Traversal in order :
100 → 90 → 55 → 58 → 39 → 38 → 18 → 150 → 160 → 184.

Total tracks traversed :-

$= |100-90| + |90-55| + |55-38| + |58-39| + |39-38| + |38-18| + |18-150| + |150-160| + |160-184| + 35 + 3 + 19 + 1 + 20 + 132 + 10 + 24$

$= 254$ tracks.

Average seek length =

Total tracks traversed / Number of requests

$\therefore 254/9 = 28.22$ tracks

(2) SCAN (Elevator Algorithm)

In SCAN, the head moves in one direction servicing all requests in its path until it reaches at end, the reverses direction.

Initial head : 100

Move towards 0,

servicing : 90 → 58 → 55 → 39 → 38 → 18.

Reaching 0, reverse direction. servicing :

150 → 160 → 184.

Average seek length : 284/9 = 31.56 tracks.

3) C-SCAN (Circular SCAN)

- Initial head : 100
- Move towards 0, servicing 90 → 58 → 55 → 39 → 38 → 18
- Reaching 0, jump to 199, then service
  ∴ 150 → 160 → 184.

Average seek length : 183/9
= 20.33 tracks.

3) In a Buddy System of Memory Management successive requests of 50K, 25K and 35K are satisfied with 256K of available memory, how many blocks (with size) are left.

a. Two Blocks (32K, 64K)
b. One block (94K)
c. One block (145K)
d. Many blocks with varying sizes.

Initial Memory :-

Available Memory : 256 K

(i) Request for 50 K :-

- Closest power of 2 that can fit 50 K is 64 K.
- Memory split from 256 K → 128 K → 64 K
- allocate 64 K block to satisfy 50 K request.
- Remaining : 128 K + 64 K (unused)

(ii) Request for 25 K :-

- Closest power of 2 that can fit 25 K is 32 K.
- Memory split remaining 64 block → 32 K.
- Remaining memory after allocation : ~~64 K~~

128 K + 32 K (unused)

(iii) Request for 35 K :-

- Closest power OF 2 that can fit 35 K is 64 K.
- allocate 64 K block from the 128 K chunk.
- Remaining memory : 64 K (unused).

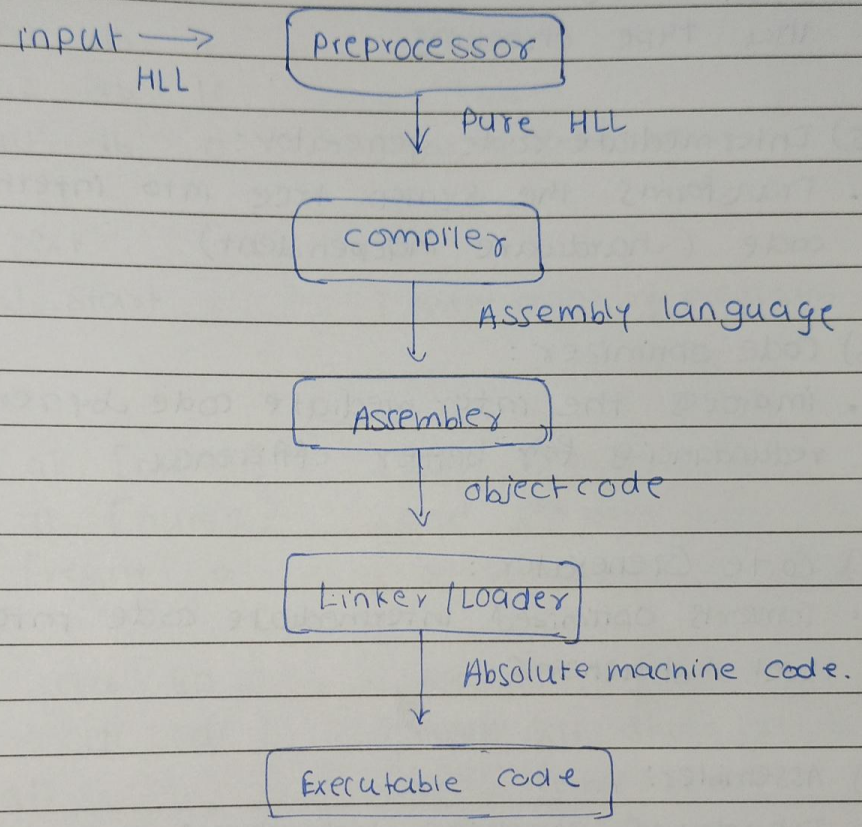
So, after three allocations only a 64 K block remains from the last split.

- one block (145 K)

Unit 5

1) Draw and explain practical arrangement of language processing system.

=



1) Source Program:
- Input is high-level source code written by the programmer (Eg:, C. Java)

2) lexical Analyzer:
- Converts the source code into tokens (Keywords, variables, operators)

3) Syntax Analyzer (parser) - ~~check the token~~
- checks the token sequence against the language's

grammar rules and creates a syntax tree.

4) Semantic Analyzer :
- Ensures logical correctness, performing tasks like type checking.

5) Intermediate code generator :
- Transforms the syntax tree into intermediate code (hardware independent)

6) Code optimizer :
- improves the intermediate code by removing redundancies for better efficiency.

7) Code Generator :
- Converts optimized intermediate code into machine level instructions

8) Assembler :
- Translates machine instructions into binary format. (object code)

9) Linker :
- Combines multiple object files & libraries into a single executable file

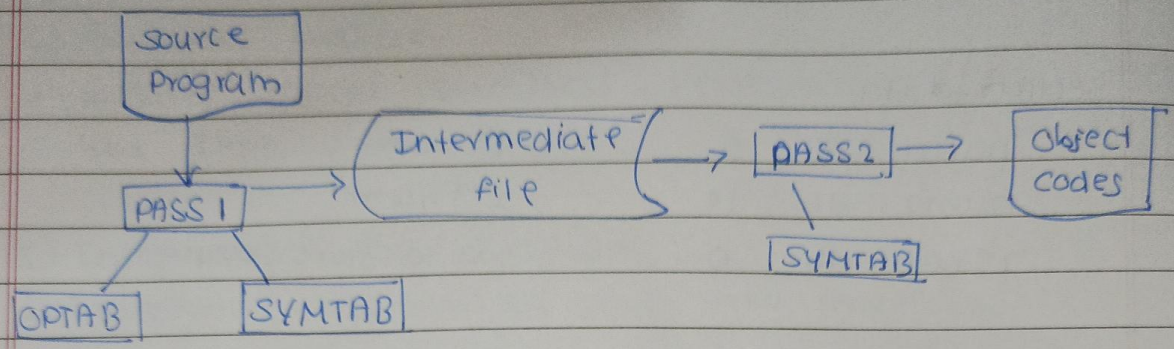
10) loader :
- Loads the executable into memory for execution by the CPU.

2) Write a program to add two numbers using assembly language scheme.

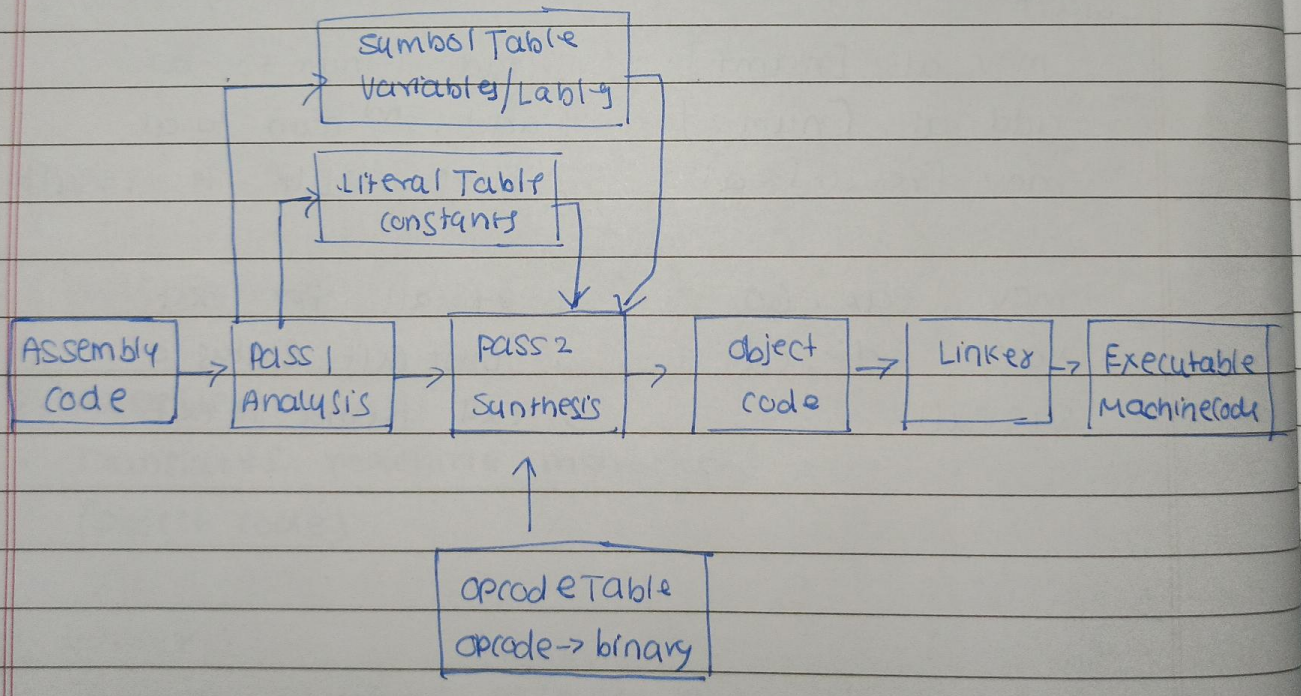
= section.data

```
num1    db    5          ; 1st num
num2    db    10         ; 2nd num
result  db    0          ; var to store result

section.text
    global _start        ; entry point of program

_start
    mov al, [num1]       ; load 1st num to al
    add al, [num2]       ; add 2nd num to al
    mov [result], al     ; store result to result var

    mov   eax, 60        ; syscall for exit.
    xor   edi, edi       ; exist exit status 0
    syscall.             ; call kernel to exit.
```

3) Draw and explain the structure of 2 pass assembler.



2 pass assembler process.



A 2 pass assembler processeses the assembly code in two distinct phases, ensuring proper handling of forward references and generating machine code from symbolic instructions.

1) Pass 1 : Analysis phase.

- objective: The 1st pass scans the source code & performs symbol definition and address assignment.
- steps :
  - scans each instruction in the source program line by line
  - builds the symbol table, which stores lables and their corresponding memory address.
  - Assigns memory adaress to instructions & data.
  - Detects forward references, where symbols (lables) are used before defined, but dosen't fully resolve them yet.
  - Produces intermediate code than contain symbolic instructions and addresses.

2) Intermediate code.

- This is a temporary representation of the program generated during pass1
- It includes symbolic names and unresolved addresses, which will be handled in pass 2.

3) pass 2: Synthesis phase.

- objective: Converts the intermediate code into actual machine code, resoluing all addresses.
- steps :
  - Re-Scans the intermediate code produced in Pass 1.

- Resolves addresses using the symbol table.
- Translates symbolic instructions into actual machine instruction.
- Generates the final machine code, which is ready to be loaded and executed by the computer.

Advantages of 2-pass assembler -

- Handles forward references easily, where labels or symbols are used before they are defined.
- Ensures all addresses and symbolic references are resolved correctly, which is critical for code correctness.

Unit 6

1) Write a sample program using macro, consisting Macro definition, calling Macro, & Macro expansion.

```
section. data
    msg db " The sum is: ", 0 .

sector. bss
    result resb 4

section. text
    global _start

%macro    ADD_NUMBERS 2
    mov   eax , %1
    add   eax , %2
    mov   [result], eax
%endmacro

_start
    ADD_NUMBERS  5,10

    mov   eax, 60
    xor   edi , edi
    syscall
```

Date
Page

Explanation -

1) • The macro ADD_NUMBERS is defined using %macro & %endmacro.
• It takes 2 parameters
• Macro performs the following operations.
- Moves the 1st parameter to EAX register
- Adds the 2nd parameter to EAX
- Stores the result in a preserved space (result)

2) Calling the Macro.
• The macro is called with the values 5 & 10 as arguments : ADD_NUMBERS 5,10
• This invocation will expand to the sequence of instruction defined in the macro.

3) Macro expansion.
• When the assembler encounters the macro call, it expands it to the following instructions.

```
mov   eax , 5
add   eax , 10
mov   [result], eax
```

2) Write a sample program using Nested Macro, consisting of macro definition, calling Macro & Macro expansion.

```
section .data
    msg db "The sum of ", 0
    result_msg db " is: ", 0
    num1 db 5
    num2 db 10

section .bss
    result resb 4

section .text
    global _start

%macro PRINT_SUM 0
    %macro CALCULATE_SUM 2
        mov al, %1
        add al, %2
        mov [result], al
    %endmacro
    CALCULATE_SUM num1, num2
%endmacro

_start
    PRINT_SUM

    mov eax, 60
    xor edi, edi
    syscall
```

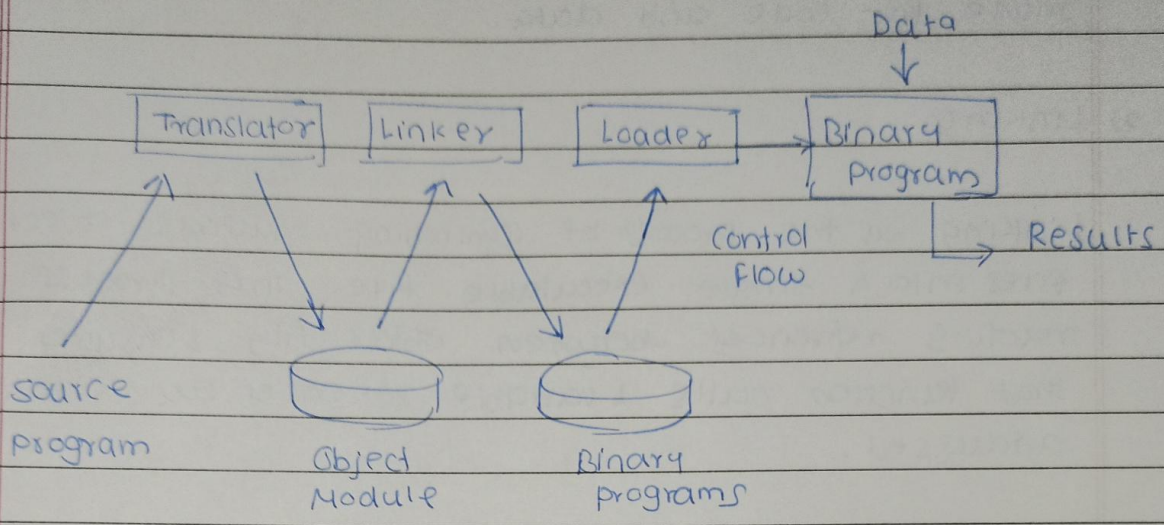
Explanation –

1) Macro definitions
   - The outer macro PRINT_SUM is defined using %macro & %endmacro
   - Inside the macro we defined a nested macro CALCULATE_SUM, which takes 2 parameters. It does –
     - moves the 1st parameter to al
     - adds the 2nd parameter to al
     - stores the result in the result variable.

2) Calling the nested macro.
   - Inside PRINT_SUM, we called the nested macro CALCULATE_SUM with parameters num1 & num2
   - This invocation expands to the sequence of instructions defined in the nested macro.

3) Explain the relocation & Linking concept with schematic diagram

=



1) Relocation.

Relocation refers to the process of adjusting the address of code and data within a program so that they can run concurrently in memory. This is especially important when a program is ~~located~~ loaded at different memory addresses using different execution.

Key Points:

- Static vs dynamic Relocation:
  - Static = Adjustments are made at compile time.
  - Dynamic - Adjustments are made at load time or runtime.
- Addrss Resolution : when a program is compiled, it may contain addresses that are relative (not absolute) or labels that refer to memory locations. The relocation process resolves these addresses into actual memory addresses.

- Relocation table - A relocation table is created to keep track of where adjustments need to be made for code and data.

2) Linking

Linking is the process of combining multiple object files into a single executable file. This process resolves refrences between object files & ensures that function calls & variable refrences are correctly addressed.

Key points:

- Static linking - All required libraries are included in the final executable at compile time, This result in a larger executable but faster execution since all necessary code is available.

- dynamic linking - libraries not included in executables; instead, they are linked at runtime, This reduces executable size & allows for shared libraries.

- linker - The linker is a tool that performs the linking process, resolving external refrences and combining object files into an executable.