Unit 3.

* Loaders & Linker *

Syllabus:-

Introduction, Loader schemes: compile and GO, General Loader Schemes, Absolute Loader, subroutine Linkages, Relocating Loaders, Direct linking loaders, Overlay structure, Design of Absolute Loader, Design of Direct linking loader, self relocating programs, Static and dynamic linking.

* Introduction to Loaders :-

Loader is a special kind of progeam that takes/accepts the object code produced by the assemblers, compilers and make it ready for execution and helps to execute.

In other words loader takes object code as a input and prepares it for execution.

Loader perfoems four important functions.

1. Allocation
2. Linking
3. Relocation
4. Loading.

1. Allocations-

Loader allocates memory for the progeam in main memory.

2. Linking:-

It combines two or more separate object progeams or modules & supplies necessary information

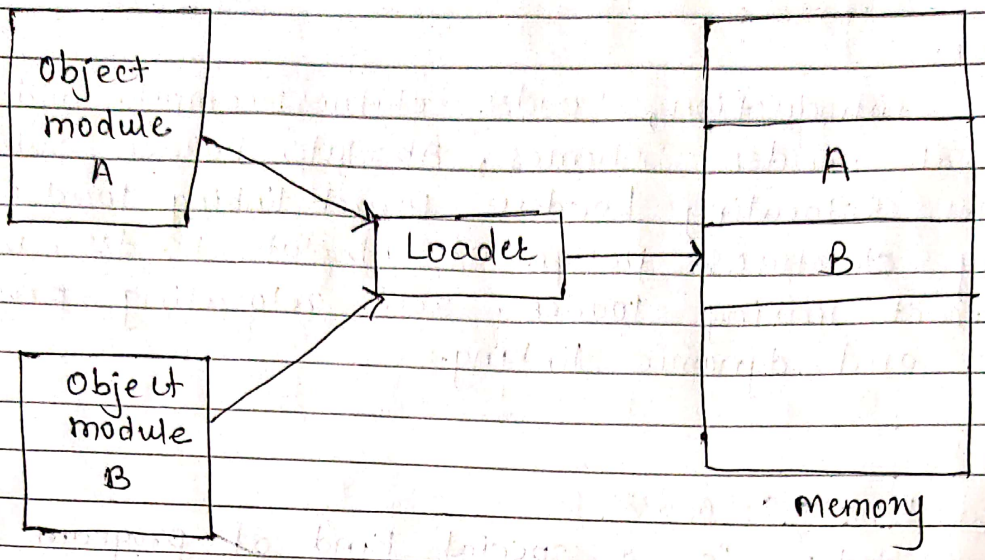
3. Relocations:-

It modifies the object progeam so that it can be loaded at an address different from the location.

4. Loadings-

It brings the object program into the main memory for execution.

* General loading scheme :-



* General Loading Scheme *

- Program modules A and B are loaded in memory after linking. It is ready for execution.

Functions of Loader :-

1. Allocation :-

In order to allocate memory to the program, the loader allocates the memory on the basis of the size of the program this is known as allocation.

The loader gives the space in memory where the object program will be loaded for execution.

2. Linking :-

The linker resolves the symbolic reference code or data between the object modules by allocating all of the user subroutine & library subroutine addresses. This process is known as linking.

For example, In C language we have a printf() function.

If we are using printf function in our program when program control encounters printf statement then program control goes to the line where the printf() is written. Here linker comes into the picture and it links that line to the module where the actual implementation of the printf() function is written.

3. Relocation:-

There are some address dependent locations in the program, and these address constants must be modified to fit the available space, this can be done by loader and this is known as relocation.

ory

4. Loading:-

The loader loads the program into the main memory for execution of that program.

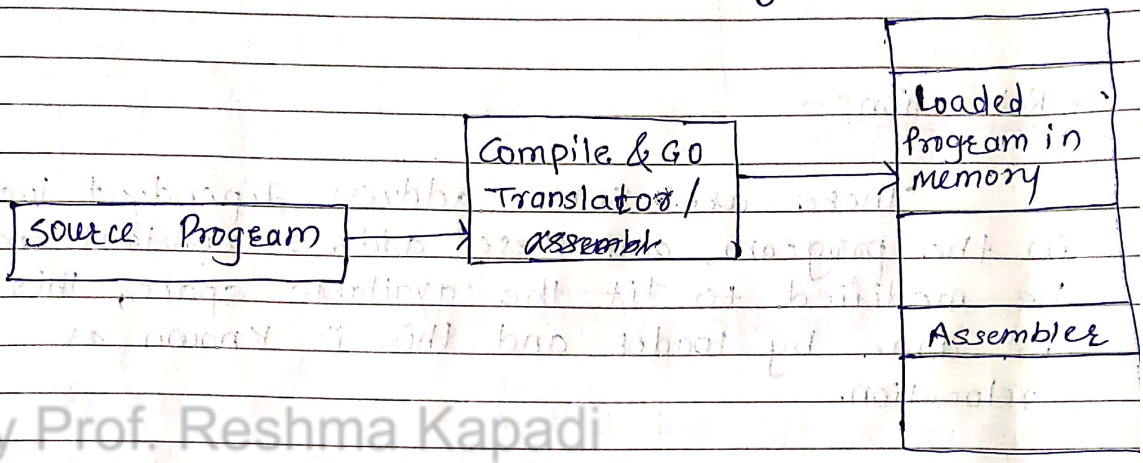
It loads the machine instruction & data of related programs & subroutines into the main memory. This process is known as loading.

* Compile and go loader :-

In this type of loader an assembler is present in memory.

In other part of memory, there is an assembled instruction which means the assembled source program.

Assembled machine instruction is placed directly into thier assigned memory location.



* Compile & GO Loader Scheme *

- In compile and go loader the assembler places the assembled instruction directly into the designated memory locations for execution.
- The instructions are read line by line, it's machine code is obtained & it is directly put in the main memory at some known address.
- After completion of assembly process, it assigns the starting address of the program to the location counter.
- As assembler is present in memory & if the user runs the same source program, every line of code will again be translated by a translator.

So here re-translation happens.

Advantages of compile & go loader:-

① They are simple and easier to implement.
② No additional routines are required to load the compiled code into the memory.

Disadvantages:-

① There is wastage of memory space due to the presence of the assembler.
② There is no production of obj file, the source code is directly converted to executable form.

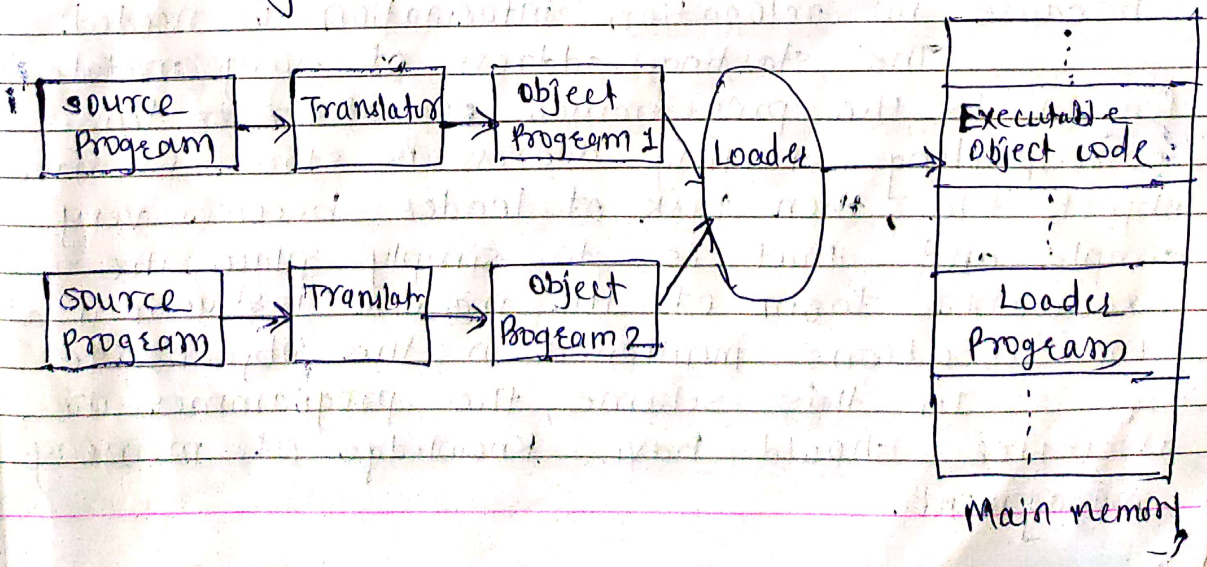
Hence even though there is no modification in the source code it needs to be assembled & executed each time.

② General Loader Scheme:-

In this loader scheme, the source program is converted to object program by some translator (assembler)

The loader accepts these object modules & puts machine instruction & data in an executable form at their assigned memory.

The loader occupies some portion of main memory.



Advantages:

① There is no wastage of memory because assembler is not placed in the memory, instead of it; loader occupies some portion of the memory.

Size of the loader is smaller than assembler so more memory is available to the user.

② It is possible to write source program with multiple languages, because source program are first converted to object programs always & loader accepts these object modules to convert it to executable form.

Disadvantage:-

1. If the program is modified it has to be retranslated.
2. Some portion of the memory is occupied by the loader.

Absolute Loader:

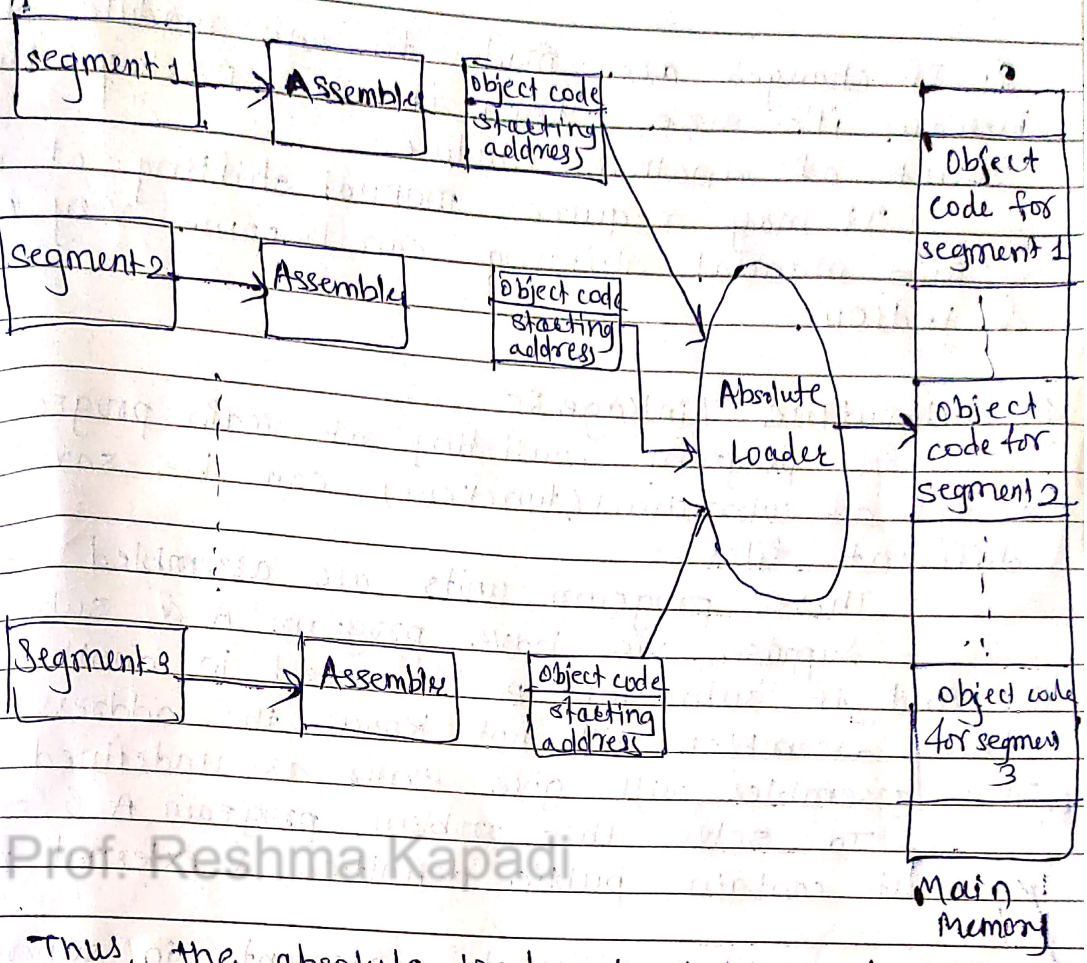
Absolute loader is a kind of loader in which relocated object files are created.

Loader accepts these files & places them at specified locations in the memory.

This type of loader is called absolute loader because, no relocation information is needed.

The starting address of every module is known to the programmer ~~of assembler~~ this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file.

In this scheme, the programmer or assembler should have knowledge of memory management.



Thus, the absolute loader is simple to implement In this scheme,

1) Allocation is done by progeammer.
2) Linking is done by progeammer.
3) Relocation is done by assembler
4) Loading is done by loader.

Advantages:-

1. No Relocation infoemation is required.
2. This scheme is very simple to implement.
3. This scheme makes more memory available for loading.
4. This scheme supports multiple object modules to reside in memory.

Disadvantages:-

1) Since the linking is handled by the progeammer, the progeammer has to remember the address of each module.

2. If changes are made to one module that increases it's size then it can overlap the start of another module.
It may require manual shifting of module. This manual shifting can become very complex & tedious.

④ Subroutine Linkage :-
- A program consisting of main program & a set of subroutines (functions) can be saved in different files.
- These program units are assembled separately.
- Suppose we have program A & subroutine B. And if subroutine B is saved in another file then assembler will not know the address of B. The assembler will give error as undefined symbol.
- To solve this problem program A & subroutine B must contain public definitions & external references.

There are two types of statements in subroutine Linkage.
1) EXTRN statement
2) ENTRY statement.

① EXTRN statement :-
The symbols which are defined in other program units are called as EXTRN symbols.
These symbols are used in current program unit but defined in other program units.

Syntax :-
EXTRN F1 or
EXTRN PG1, PG2

② ENTRY statement :-
The symbols which may defined in the current program unit and these symbols may be

referenced in other program units.

Syntax:- ENTRY PG2

Example.

```
MAIN    START
        EXTRN F1
        ____
        ____      } // some lines of code.
        ____
        CALL F1
        ____
        ____      } // some lines of code.
        ____
        END

F1      START
        ____
        ____      Subroutine F1.
        ____
        ____
        END
```


- In above Example, subroutine F1 is declared as an external symbol to the main program.

- Assembler cannot provide addresses of external symbols.
- These addresses are fixed at the time of linking.
- The external references are remain unresolved until linking is completed.

Advantages:-

1. Subroutines can reside in several files.
2. A subroutine can be modified independently.
3. A subroutine can be written in a different languages

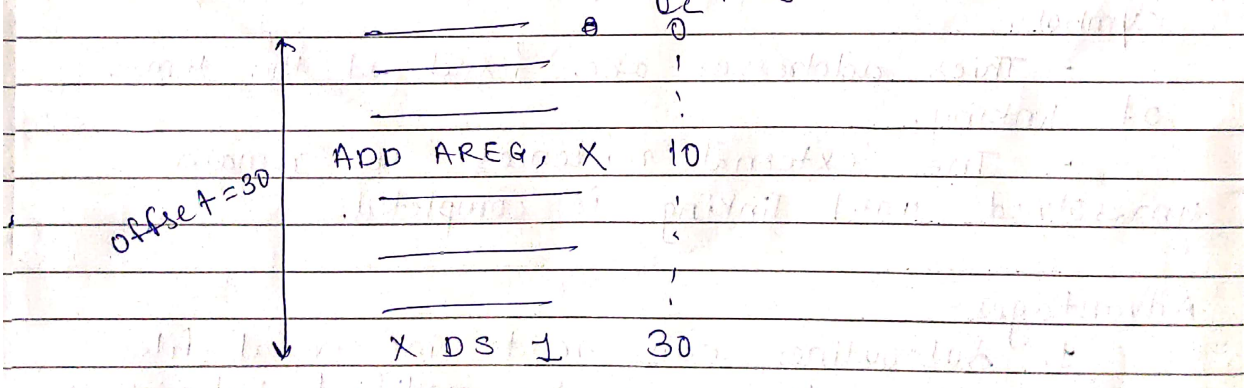
Disadvantages:-

1. The external references are unresolved until linking is completed.

⑤ Relocating Loaders:-

- When one subroutine is changed we have to assemble all subroutines.
- The task of allocation & linking for the programmer becomes very tedious so relocating loader was introduced.
- Relocating loader generates program & information about all other programs it references.
- Relocating loader also has relocation information if program is loaded in an arbitrary location in memory.
- The example of Relocating loader is BSS (Binary Symbolic loader).
- The BSS loader allows many code segment but only one data segment.
- The O/P of the assembler using a BSS loader is contains.
  1. Object program.
  2. Reference about other programs to be accessed.
  3. Information about address sensitive entities.

- We can use segment registers to reduce the relocation requirement.
- Consider the following program



- In above program the address of X is 30.
- If this program is loaded from the memory location 500 for execution then address of X becomes 530.

500 is the starting location of code segment.

- Use of segment registers make a program address insensitive.
- Here all memory addressing is performed by using displacement (offset).
- Starting memory address is stored in the segment register & the actual address is given by.

Contents of the segment register + address of the operand in the instruction.

- The address of X is

500 + 30 = 530

segment register — offset — actual address.

- The BSS loader allows multiple code segments & one common data segment.
- The assembler assembles each code segment independently & passes on to the loader the following.

1. Object program prefixed by a transfer vector.
   - Transfer vector consists of addresses of subroutines referenced by source program.

2. Relocation information:-
   locations in the program that must be changed if it is loaded in an arbitrary location in main memory.

3. Length of source program & length of transfer vector.

- The transfer vector is used to solve the problem of linking. & program length is used to solve the problem of allocation.

(6) Direct Linking Loader :-

- It is most popular loading scheme.
- It is general relocatable loader.
- It allows multiple code & multiple data segments.
- The assemblee must give the loadee the following information with each code or data segment.
  1. length of segment.
  2. Public declaration (ENTRY symbols)
  3. External declaration (EXTRN symbols)
  4. Information about address constants.
  5. The machine translation of source progeam & relative address assigned.

- The object module produced by the assembler is divided into 4 sections.
  1. External symbol directory (ESD)
  2. Actual Assembled Progeam (TXT).
  3. Relocation Directory (RLD).
  4. End of the object module (END).

1. External Symbol Directory :-

- It contains ENTRY & EXTRN symbols.
- ESD contains 3 types of symbols.
  i) SD = segment definition (symbol).
  ii) LD = ENTRY symbols.
  iii) ER = EXTRN symbols.

The format of ESD is as shown below.

Line No.	Symbol	Type	ID	Relative location	Length.

Note :- ID's are only given to SD & ER.

2. Actual Assembled progeam (TXT):-

It contains the relocatable machine language instructions & data that were produced during translation.

The format for TXT records.

Source object record No.	Relative address	contents	comments

3. Relocation Directory:-

It contains one entry for each address that must be changed when the module is loade into the main memory.

The relocation directory contains the following infoemation.

1. The address of each operand that needs to be changed due to relocation.
2. By what it has to be changed.
3. The operation to be peefoemed.

4. END:-

This indicates the End of object module.

RLD format

Source object record No.	ESD ID.	Length in bytes.	Flag +/ −	Relative address.

* Design of Direct linking loader :-

- The input to the loader is set of object programs (generated by assembler / compiler) to be linked together.

Each object module is divided into 4 sections.

1. External symbol Directory. (ESD)
2. Actual - assembled program (TXT)
3. Relocation directory.
4. End of object module.

- A direct linking loader requires two passes

i) Pass 1 :- assigns addresses to all external symbols.

ii) Pass 2 :- performs actual loading, relocation & linking.

- In pass I. GEST (global external symbol table is prepared.)

- It contains every external symbol & the corresponding absolute address value.

Q. consider following program & show the entries of ESD, TXT & RLD, GEST for PG1 & PG2.

Object record NO.	Relative address	Source Program.	
1	0	PG1	START
2			ENTRY PG1ENT1, PG1ENT2
3			EXTRN PG2ENT1, PG2
4	20	PG1ENT1	=
5	30	PG1ENT2	=
6	40		DCA (PG1ENT1)
7	44		DCA (PG1ENT2 + 15)
8	48		DCA (PG1ENT2 - PG1ENT1 - 3)
9	52		DCA (PG2)
10	56		DCA (PG2ENT1 + PG2 - PG1ENT1 + 4)
11	60		END.

12	0	PG2	START
13			ENTRY PG2ENT1
14			EXTRN PG1ENT1, PG1ENT2
15	16	PG2ENT1	
16	24		DC A(PG1ENT1)
17	28		DC A(PG1ENT2+15)
18	32		DC A(PG1ENT2-PG1ENT1-3)
19	36		END.

1. ESD.

source object record No.	Name	Type	ID	Relative address	Length
1	PG1	SD	01	0	60
2	PG1ENT1	LD	—	20	—
2	PG1ENT2	LD	—	30	—
3	PG2ENT1	ER	02	—	—
3	PG2	ER	03	—	—

2. TXT Record.

source object record No.	Relative address	contents	comments
6	40-43	20	Address of PG1ENT1 is 2
7	44-47	45	30+15=45
8	48-51	7	PG1ENT2 - PG1E 1-3 = 30-20-3 = 7
9	52-55	0	Addr of PG2 is not known
10	56-59	-16	PG2ENT1 + PG2 - PG1ENT1 +4 = 0+0-20+4 = -16 Address of PG2ENT1 & PG2 is not known

Difference of two relative address is constant there line no:8 will not come in RLD. (Difference bet^n two Entry symbols)

3. RLD records.

source obje	ESD ID	Length in bytes	Flag + or −	Relative address
6	-	4	+	40.
7	-	4	+	44
9	03	4	+	52
10. (PG2ENT2)	02	4	+	54
10 (PG2)	03	4	+	56
10 (PG1ENT1)	-	4	-	56.

PG2

258
+16
274

4. ESD records.

Line NO.	Name	Type	ID	Relative address	Length.
12	PG2	SD	01	0	36
13	PG2 ENT1	LD	-	16	-
14	PG1 ENT1	ER	02	-	-
14	PG1 ENT2	ER	03	-	-

2. TXT Records.

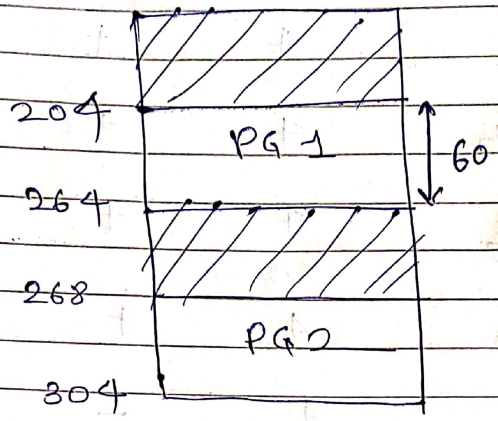
Line NO.	Relative address	contents	comments.
16	24-27	0	address of PG1ENT1 is not known so, 0.
17	28-31	0+15=15	PG1ENT1+15 0+15=15
18	32-35	-3.	PG1ENT2-PG1ENT1 -3. 0-0-3 =-3

3. RLD.

Line NO.	ESD ID	Length in Bytes	Flag +/-	Relative add^s.
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	-	32

Global External Symbol Table.

If we assume that PG1 is loaded at location 204 & PG2 is loaded at location 268.



External Symbol	Assigned memory location
PG1	204
PG1ENT1	224
PG1ENT2	234
PG2	268
PG2ENT1	284

- The GEST contains every external symbol & it's corresponding assigned memory location. i.e. local to current program segment but used in other program segment.

Q. Consider following program show ESD, TXT & RLD cards for the source code given below. Also show the contents of GEST table & LESA table.

source Program.

source code record NO.	Relative address		
1	0	MAIN	START
2			ENTRY M1, M2
3			EXTRN PROG, P1, P2
4	30	M1	≡
5	40	M2	≡
6	50		DC A(M2)
7	54		DC A(M1+5)
8	58		DC A(M2-M1-4)
9	62		DC A(PROG)
10	66		DC A(P1+P2-PROG)
11			END.
12	0	PROG	START
13			ENTRY P1, P2
14			EXTRN M1, M2
15	16	P1	≡
16	26	P2	≡
17	36		DC A(M1)
18	40		DC A(M2+10)
19	44		DC A(P2-P1-3)
20			END.

- Assume initial program load address as 400.

⇒ Ans:-

(1) object Records for Module MAIN.

(i) ESD card:-

source card reference	Name	Type	ID	Relative address	Length.
1	MAIN	SD	01	0	70
2	M1	LD	-	30	-
3	M2	LD	-	40	-
4	PROG	ER	02	-	-
3	P1	ER	03	-	-
2	P2	ER	04	-	-

(i) Object Records for module MAIN.

(1) ESD :-

Source card Reference	Name	Type	ID	Relative address	Length
1	MAIN	SD	01	0	70
2	M1	LD	-	30	-
2	M2	LD	-	40	-
3	PROG	ER	02	-	-
3	P1	ER	03	-	-
3	P2	ER	04	-	-

(2) TXT :-

source card Reference	Relative address	contents	comments
6.	50-53	40	Address of M2 is 40.
7.	54-57	35	A(M1+5) 30+5=35
8.	58-61	6	A(M2-M1-4) 40-30-4 = 6
9.	62-65	0	External
10.	66-69	0	P1+P2-PROG 0+0-0=0

(3) RLD :-

Source card Reference	ESDID	Length	Flag +/-	Relative address
6	-	4	+	50
7	-	4	+	54
9	02	4	+	62
10	03	4	+	66
10	04	4	+	66
10	02	4	-	66

Note:- Difference between two relative address is constant (or difference between two ENTRY symbols) Hence, line no. 8 will not [illegible] in RLD.

(ii) object Records for module PROG.

(1) ESD:-

source card reference	Name	Type	ID	Relative address	Length
12	PROG	SD	01	0	48.
13	P1	LR	-	16	-
13	P2	LR	-	26	-
14	M1	ER	02	-	-
14	M2	ER	03	-	-

(2) TXT :-

source card reference	Relative address	contents	comment
17	36-39	0	M1 is External
18	40-43	10	M2+10 = 0+10=10
19	44-47	7	P2-P1-3
			26-16-3
			= 10-3 = 7

(3) RLD.

source card reference	ESD ID	Length in bytes	Flag +/-	Relative address.
17	02	4	+	40
18	03	4	+	44

Line NO- 19 will not come in RLD as difference bet? P2 & P1 is constant.

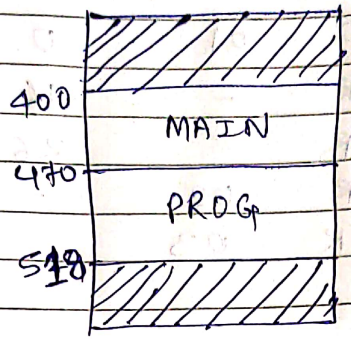
(iii) Global External Symbol Table:-

Since the initial load address is 400; MAIN module will be loaded from the initial address.

The segment/module PROG will be loaded from

Initial load address + size of the segment

PROG = 400 + 70
= 470.



External Symbol	Assigned memory address
MAIN	400
M1	430
M2	440
PROG	470
P1	486
P2	496

(iv) LESA :- Local External Symbol Array.
- LESA is created for each segment.
- It contains SD and LD type entries of each segment.

i) LESA of MAIN.

1.	MAIN	400
2.	M1	430
3.	M2	440

ii) LESA of PROG.

1.	PROG	470
2.	P1	486
3.	P2	496

* Design of Absolute Loader :-

In absolute loading scheme the progeammer & the assemblee peefoem following tasks:

1. Allocation.
2. ~~Relocation.~~
3. Linking.

- The absolute loadee reads the object program line by line. & moves the text of the progeam into the memory at the location specified by the assemblee.

- The object progeam generated by the assemblee must communicate the following infoematin to the loadee.

1. It must convey the machine instructions that the assemblee has created along with the memory address.

2. It must convey the staeting execution point. Progeam execution will start at this point afte the progeam is loaded.

3. The object progeam is a sequence of object records.

Each object record specifies some specific aspect of the program in the object module.

- There are two types of records.

1. Text Record.
2. Transfee record.

1. Text record - contains binary image of the assembly program.

2. Transfer Record -
containing the starting (entry point) of the execution.

The formats of text and transfer records are given in

Record Type	Number of bytes of information	Memory address	Binary image of data or instruction

a) Text Record.

Record Type	Number of bytes of information = 0	Address of the entry point

Record type = 0 for Text Record.
Record type = 1 for Transfer Record.

Algorithm :-

- The algorithm for absolute loader is very simple.
- The loader reads the object file record by record & moves the binary image at the locations specified in the record.

The last record is transfer record On reaching the transfer record the control is transferred to the entry point of execution.

Self Relocating Program:-

A self-relocating program alters itself to execute from a different location of the memory. Self-relocation eliminates the need to have several copies of a program on a disk, with each copy having its own different load origin. However, self-relocating programs are not necessary when the computer uses virtual memory.

During the relocation process, the program relocates its sensitive address-sensitive instructions, which allows it to execute from any part of the memory. The requirements for the relocation process are as follows:

- A table of the address-sensitive instructions. The program must know the translated origin and the execution start address as well as the addresses of the address-sensitive instructions.
- A relocating logic, which is the code that performs the relocation process.

The two functions are inbuilt into the program; the start address of the relocating code is usually specified as the execution start address of the program. Once the program is loaded into the memory for execution, the relocating logic takes control and performs the relocation using the load address and the information regarding the address-sensitive instructions.

The self-relocation process may be either static or dynamic. Static relocation is performed before the program is executed, whereas dynamic relocation is performed during program execution. Dynamic relocation may first suspend the program execution and perform the relocation, or use a relocation register.

Self-relocating programs are less efficient compared to reloadable programs.

Apart from genuine self-relocating programs, malware programs use the same self-relocation method to propagate through systems and networks to spread their malicious code.

Overlay Structure:-

In memory management, overlays refer to a technique used to manage memory efficiently by overlaying a portion of memory with another program or data.

The idea behind overlays is to only load the necessary parts of a program into memory at a given time, freeing up memory for other tasks. The unused portions of the program are kept on disk or other storage, and are loaded into memory as needed. This allows programs to be larger than the available memory, but still run smoothly.

The main problem in Fixed partitioning is the size of a process has to be limited by the maximum size of the partition, which means a process can never be span over another.In order to solve this problem, earlier people have used some solution which is called as Overlays.

The concept of **overlays** is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part you required, you load it and once the part is done, then you just unload it, means just pull it back and get the new part you required and run it.

Formally,
"The process of **transferring a block** of program code or other data into internal memory, replacing what is already stored".
Sometimes it happens that compare to the size of the biggest partition, the size of the program will be even more, then, in that case, you should go with overlays.

So overlay is a technique to run a program that is bigger than the size of the physical memory by keeping only those instructions and data that are needed at any given time.Divide the program into modules in such a way that not all modules need to be in the memory at the same time.

Advantages of using overlays include:

- Increased memory utilization: Overlays allow multiple programs to share the same physical memory space, increasing memory utilization and reducing the need for additional memory.
- Reduced load time: Only the necessary parts of a program are loaded into memory, reducing load time and increasing performance.
- Improved reliability: Overlays reduce the risk of memory overflow, which can cause crashes or data loss.
- Reduce memory requirement
- Reduce time requirement

Disadvantages of using overlays include:

- Complexity: Overlays can be complex to implement and manage, especially for large programs.
- Performance overhead: The process of loading and unloading overlays can result in increased CPU and disk usage, which can slow down performance.
- Compatibility issues: Overlays may not work on all hardware and software configurations, making it difficult to ensure compatibility across different systems.
- Overlap map must be specified by programmer
- Programmer must know memory requirement
- Overlapped module must be completely disjoint
- Programming design of overlays structure is complex and not possible in all cases

Example –
The best example of overlays is assembler. Consider the assembler has 2 passes, 2 pass means at any time it will be doing only one thing, either the 1st pass or the 2nd pass. This means it will finish 1st pass first and then 2nd pass.Let assume that available main memory size is 150KB and total code size is 200KB

```
Pass 1.....................70KB
Pass 2.....................80KB
Symbol table...............30KB
Common routine.............20KB
```

As the total code size is 200KB and main memory size is 150KB, it is not possible to use 2 passes together. So, in this case, we should go with the overlays technique.

According to the overlays concept at any time only one pass will be used and both the passes always need symbol table and common routine.

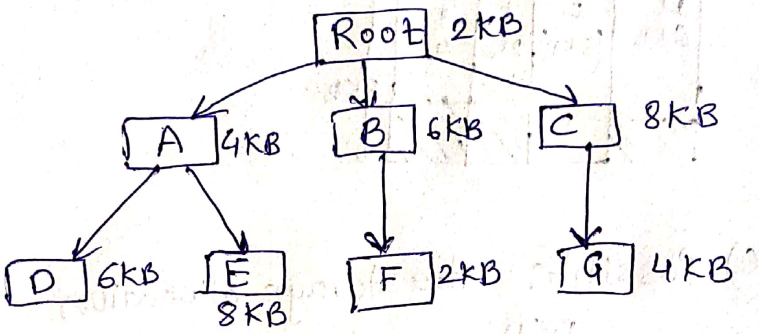
Now the question is if overlays-driver* is 10KB, then what is the minimum partition size required?

For pass 1 total memory needed is = (70KB + 30KB + 20KB + 10KB) = 130KB and

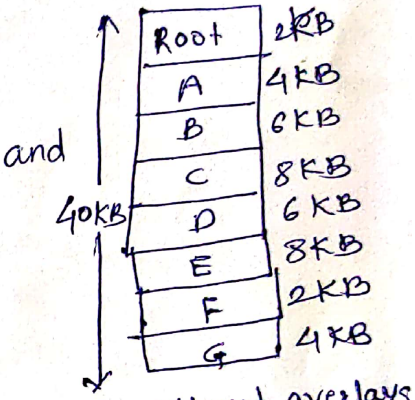
For pass 2 total memory needed is = (80KB + 30KB + 20KB + 10KB) = 140KB.

So if we have minimum 140KB size partition then we can run this code very easily.

Example:- Consider the following diagram.

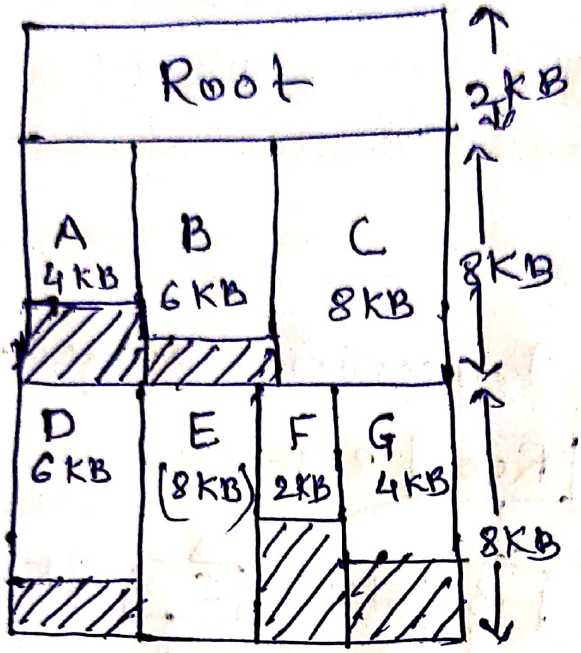


b) Dynamic loading



a) Without overlays

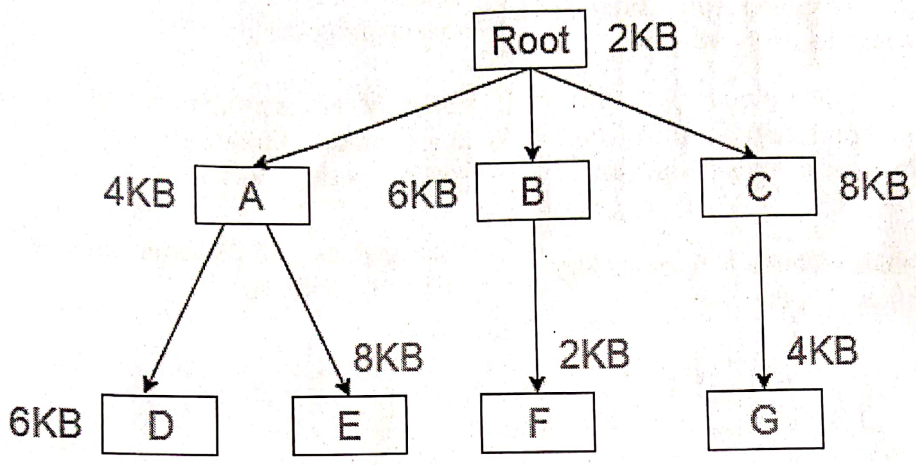
- The program consisting of 8 subprograms including Root, and requires 40KB memory
- In fig b arrow indicates that Root program calls subprogram A, B and C.
- The subprogram A only calls to D & E.
- The subprogram B only calls to F
- The subprogram C only calls to G
- The subprograms A, B and C are never used at same time.
- Similarly, subprograms D, E, F & G are never used at same time.
- So we can create overlay structure by considering above situation.
- It will require 18KB of memory whereas without overlays it may require 40KB of memory.



c) Possible ~~stro~~ storage allocation with overlays.

*Overlays driver:-It is the user responsibility to take care of overlaying, the operating system will not provide anything.Which means the user should write even what part is required in the 1st pass and once the 1st pass is over, the user should write the code to pull out the pass 1 and load the pass 2.That is what is the responsibility of the user, that is known as the Overlays driver.Overlays driver will just help us to move out and move in the various part of the code.

Question –
The overlay tree for a program is as shown below:



What will be the size of the partition (in physical memory) required to load (and run) this program?
(a) 12 KB (b) 14 KB (c) 10 KB (d) 8 KB

Explanation –
Using the overlay concept we need not actually have the entire program inside the main memory.Only we need to have the part which are required at that instance of time, either we need Root-A-D or Root-A-E or Root-B-F or Root-C-G part.

```
Root+A+D = 2KB + 4KB + 6KB = 12KB
Root+A+E = 2KB + 4KB + 8KB = 14KB
Root+B+F = 2KB + 6KB + 2KB = 10KB
Root+C+G = 2KB + 8KB + 4KB = 14KB
```

So if we have 14KB size of partition then we can run any of them.
Answer -(b) 14KB

Difference between Static Linking and Dynamic Linking:

	Static Linking	Dynamic Linking
Definition	The process of combining all necessary library routines and external references into a single executable file at compile-time.	The process of linking external libraries and references at runtime, when the program is loaded or executed.

	Static Linking	Dynamic Linking
Linking Time	Occurs at compile-time.	Occurs at runtime.
File Size	Generally larger file size, as all required libraries are included in the executable.	Smaller file size, as libraries are linked dynamically at runtime.
Flexibility	Less flexible, as any updates or changes to the libraries require recompilation and relinking of the entire program.	More flexible, as libraries can be updated or replaced without recompiling the program.
Performance	Faster program startup and direct execution, as all libraries are already linked.	Slightly slower program startup due to the additional linking process, but overall performance impact is minimal.
Examples	Executables with file extensions like .exe, .elf, .a, .lib, etc.	Executables with file extensions like .dll, .so, .dylib, etc.