DE-1(System Programming)
Loaders and Linkers

Definition of Loader:
Loader is utility program which takes object code as input prepares it for execution and loads the executable code into the memory. Thus loader is actually responsible for initiating the execution process.

Functions of Loader:
The loader is responsible for the activities such as allocation, linking, relocation and loading
1) It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.

2) It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.

3) There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.

4) Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.

Loader Schemes:
Based on the various functionalities of loader, there are various types of loaders:

Loaders Scheme or types of Loader:

Based on the above four functions the loader is divided into different types, they are

i. Compile and go loader or Assemble and go loader
ii. General loader scheme
iii. Absolute loader
iv. Direct linking loader
v. Relocating loader
vi. Dynamic linking loader

1) **"compile and go" loader:** in this type of loader, the instruction is read

line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77, it's a FORTRAN compiler which uses such "load and go" scheme. This loading scheme is also called as "assemble and go".

Advantages:

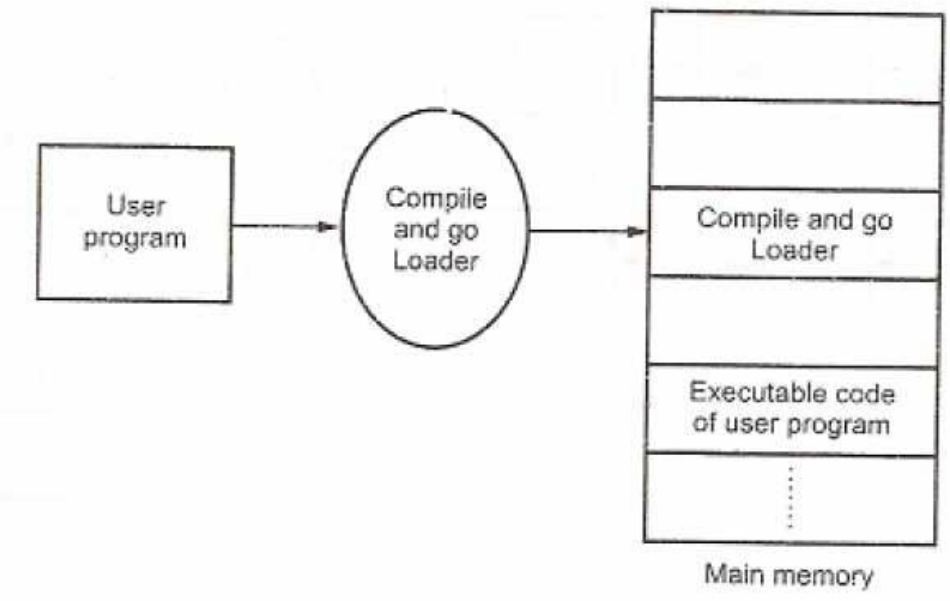
• This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.

Disadvantages:

• In this scheme some portion of memory is occupied by assembler which is simply a wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.

• 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 program it needs to be assembled and executed each time, which then becomes a time consuming activity.

• It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.



Compile and go loading scheme

2) **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader

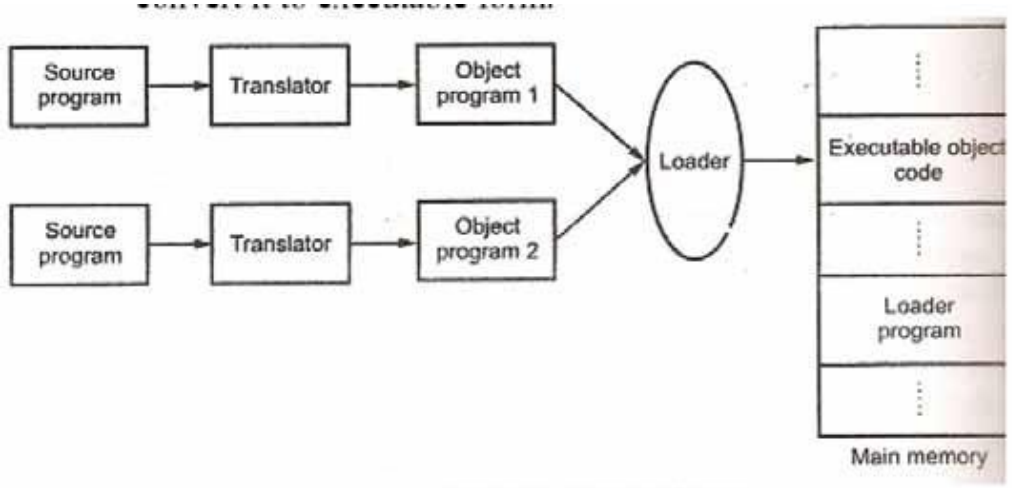
accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

Advantages:

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.

• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.

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



General loader scheme

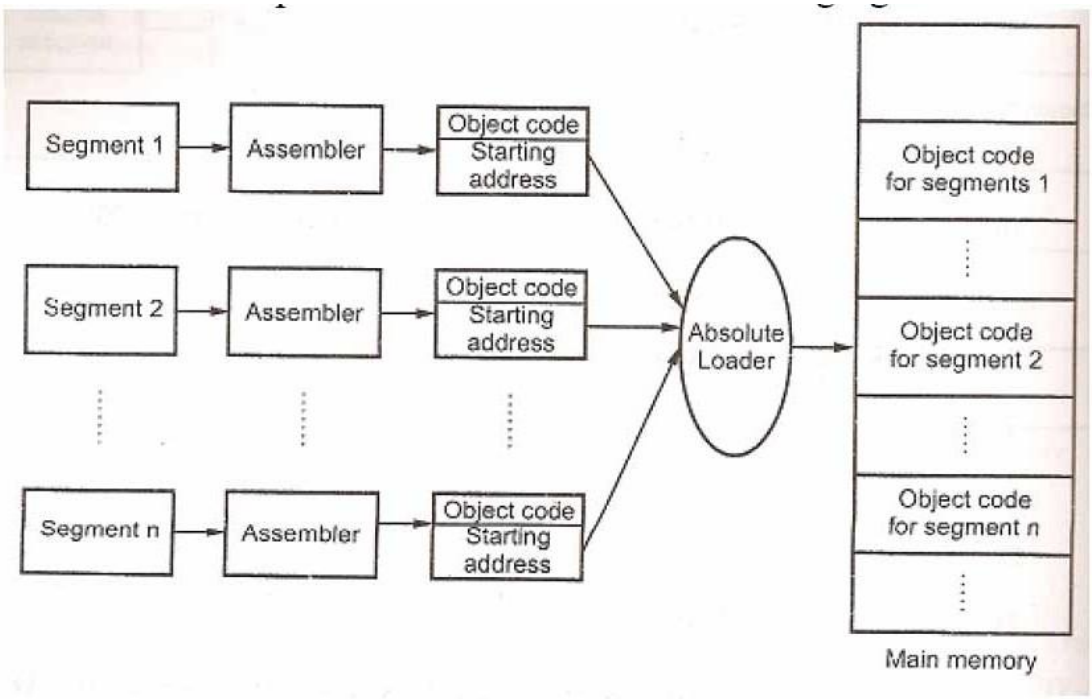
3) **Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, 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. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first

thing is : specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next . modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

```
Line number
1          MAIN          START   1000
.                          .
.                          .
.                          .
1                         JMP      5000
16                        STORE    ;instruction at location 2000
                          END
1                         SUM      START 5000
2
20                        JMP      2000
21                        END
```

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN. Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependant dresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.

The entire process is modeled in the following figure.



Process of absolute loading

Thus the absolute loader is simple to implement in this scheme-
l) Allocation is done by either programmer or assembler
2)Linking is done by the programmer or assembler
3)Resolution is done by assembler
4) Simply loading is done by the loader
As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

Advantages:

1. It is simple to implement
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient.

Disadvantages:

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.
If at all any modification is done the some segments, the starting

addresses of immediate next segments may get changed, the programmer

has to take care of this issue and he needs to update the corresponding starting addresses on any modification in the source.

Algorithm for absolute Loader

Input: Object codes and starting address of program segments.

Output: An executable code for corresponding source program. This executable code is to be placed in the main memory

Method: Begin

```
For each program segment
 do Begin
        Read the first line from object module to
        obtain information about memory location. The
        starting address say S in corresponding object
        module is the memory location where executale
        code is to be placed.
        Hence
        Memory_location = S
        Line counter = 1; as it is first line While (!
        end of file)
        For the curent object code
do Begin
1. Read next line
2. Write line into location S
3. S = S + 1
4. Line counter Line counter + 1
```

Subroutine Linkage: To understand the concept of subroutine linkages, first consider the following scenario:

"In Program A a call to subroutine B is made. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler can not find the value of this symbolic reference and hence it will declare it as an error. To overcome problem, there should be some mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must -be declared as an external routine in A. To declare such subroutine as external, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. Similarly, if one subroutine or a variable is defined in the current segment and can be referred by other segments then those should be declared by using pseudo-ops INT. Thereby the assembler

could inform loader that these are the subroutines or variables used by other segments. This overall process of establishing the relations between the subroutines can be conceptually called a_ subroutine linkage.

For example

```
MAIN START
          EXT B
            .
            .
            .
         CALL B
           .
             .
           END
B       START
             .
             .
        RET
        END
```

At the beginning of the MAIN the subroutine B is declared as external. When a call to subroutine B is made, before making the unconditional jump, the current content of the program counter should be stored in the system stack maintained internally. Similarly while returning from the subroutine B (at RET) the pop is performed to restore the program counter of caller routine with the address of next instruction to be executed.

Concept of relocations:

Relocation is the process of updating the addresses used in the address sensitive instructions of a program. It is necessary that such a modification should help to execute the program from designated area of the memory.

The assembler generates the object code. This object code gets executed after loading at storage locations. The addresses of such object code will get specified only after the assembly process is over. Therefore, after loading,

address of object code = Mere address of object code + relocation constant.

There are two types of addresses being generated: Absolute address and, relative address. The absolute address can be directly used to map the object code in the main memory. Whereas the relative address is only after the addition of relocation constant to the object code address. This kind of adjustment needs to be done in case of relative address before actual execution of the code. The typical example of relative reference is : addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals, redefinable symbols.

Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute.
The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.
For example
LOADA(X)+5
The expression A(X) means the address of variable X. The meaning of the above instruction is that loading of the contents of memory location which is 5 more than the address of variable X. Suppose if the address of X is 50 then by above command we try to get the memory location 50+5=55. Therefore as the address of variable X is relative A(X) + 5 is also relative.

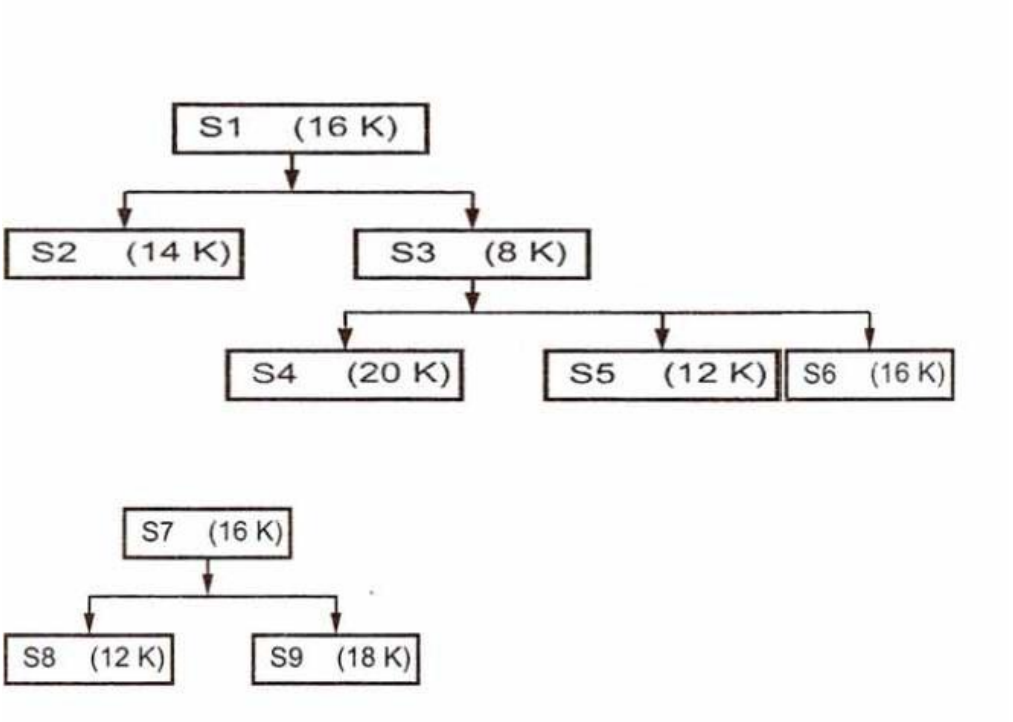
Direct Linking Loaders

The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute

which then can be directly placed at the specified location or the address can be relative. If at all the address is relative then it is the assembler who informs the loader about the relative addresses.
The assembler should give the following information to the loader
1)The length of the object code segment
2) The list of all the symbols which are not defined 111 the current segment but can be used in the current segment.
3) The list of all the symbols which are defined in the current segment but can be referred by the other segments.
The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The USE table holds the information such as name of the symbol, address, address relativity.
The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table. The definition table holds the information such as symbol, address.

Overlay Structures and Dynamic Loading:

Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such situations only those segments are resident in the memory that are actually needed at the time of execution But the question arises what will happen if the required segment is not present in the memory? Naturally the execution process will be delayed until the required segment gets loaded in the memory. The overall effect of this is efficiency of execution process gets degraded. The efficiency can then be improved by carefully selecting all the interdependent segments. Of course the assembler can not do this task. Only the user can specify such dependencies. The inter dependency of the segments can be specified by a tree like structure called static overlay structures. The overlay structure contain multiple root/nodes and edges. Each node represents the segment. The specification of required amount of memory is also essential in this structure. The two segments can lie simultaneously in the main memory if they are on the same path. Let us take an example to understand the concept. Various segments along with their memory requirements is as shown below.



Automatic Library Search:

Previously, the library routines were available in absolute code but now the library routines are provided in relocated form that ultimately reduces their size on the disk, which in turn increases the memory utilization. At execution time certain library routines may be needed. Keeping track of which library routines are required and how much storage is required by these routines, if at all is done by an assembler itself then the activity of automatic library search becomes simpler and effective. The library routines can also make an external call to other routines. The idea is to make a list of such calls made by the routines. And if such list is made available to the linker then linker can efficiently find the set of required routines and can link the references accordingly.

For an efficient search of library routines it desirable to store all the calling routines first and then the called routines. This avoids wastage of time due to winding and rewinding. For efficient automated search of library routines even the dictionary of such routines can be maintained. A table containing the names of library routines and the addresses where they are actually located in relocatable form is prepared with the help of translator and such table is submitted to the linker. Such a table is called subroutine directory. Even if these routines have made any external calls the -information about it is also given in subroutine directory. The linker searches the subroutine directory, finds the address of desired library routine (the address where the routine is stored in relocated form).Then linker prepares a

load module appending the user program and necessary library routines by doing the necessary relocation. If the library routine contains the

external calls then the linker searches the subroutine directory finds the address of such external calls, prepares the load module by resolving the external references. **Linkage Editor:** The execution of any program needs four basic functionalities and those are allocation, relocation, linking and loading. As we have also seen in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Moreover if the program contains many subroutines or functions and the program needs to be executed repeatedly then this activity becomes annoyingly complex .Each time for execution of a program, the allocation, relocation linking and -loading needs to be done. Now doing these activities each time increases the time and space complexity. Actually, there is no need to redo all these four activities each time. Instead, if the results of some of these activities are stored in a file then that file can be used by other activities. And performing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Thus dividing the essential four functions in groups reduces the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, creates linked executable text and stores this text in a file in some systematic manner. Such kind of module prepared by the binder execution is called load module. This load module can then be actually loaded in the main memory by the loader. This loader is also called as module loader. If the binder can produce the exact replica of executable code in the load module then the module loader simply loads this file into the main memory which ultimately reduces the overall time complexity. But in this process the binder should knew the current positions of the main memory. Even though the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the region of main memory available for loading the program is decided by the host operating system. The binder should also know which memory area is allocated to the loading program and it should modify the relocation information accordingly. The binder which performs the linking function and produces adequate information about allocation and relocation and writes this information along with the program code in the file is called linkage editor. The module loader then accepts this rile as input, reads the information stored in and based on this information about allocation and relocation it performs the task of loading in the main memory. Even though the program is repeatedly executed the linking is done only once. Moreover, the flexibility of allocation and relocation helps efficient utilization of the main memory.

Direct linking: As we have seen in overlay structure certain selective subroutines can be resident in the memory. That means it is not necessary to resident all the subroutines in the memory for all the time. Only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory. This process of postponing linking and loading of external reference until execution is called dynamic linking. For example suppose the subroutine main calls A,B,C,D then it is not desirable to load A,B,C and D along with the main in the memory. Whether A, B, C or D is called by the main or not will be known only at the time of execution. Hence keeping these routines already before is really not needed. As the subroutines get executed when the program runs. Also the linking of all the subroutines has to be performed. And the code of all the subroutines remains resident in the main memory. As a result of all this is that memory gets occupied unnecessarily. Typically 'error routines' are such routines which can be invoked rarely. Then one can postpone the loading of these routines during the execution. If linking and loading of such rarely invoked external references could be postponed until the execution time when it was found to be absolutely necessary, then it increases the efficiency of overhead of the loader. In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is done dynamically. **Advantages**

1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.
2. The system can be dynamically reconfigured.

Disadvantages The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory.

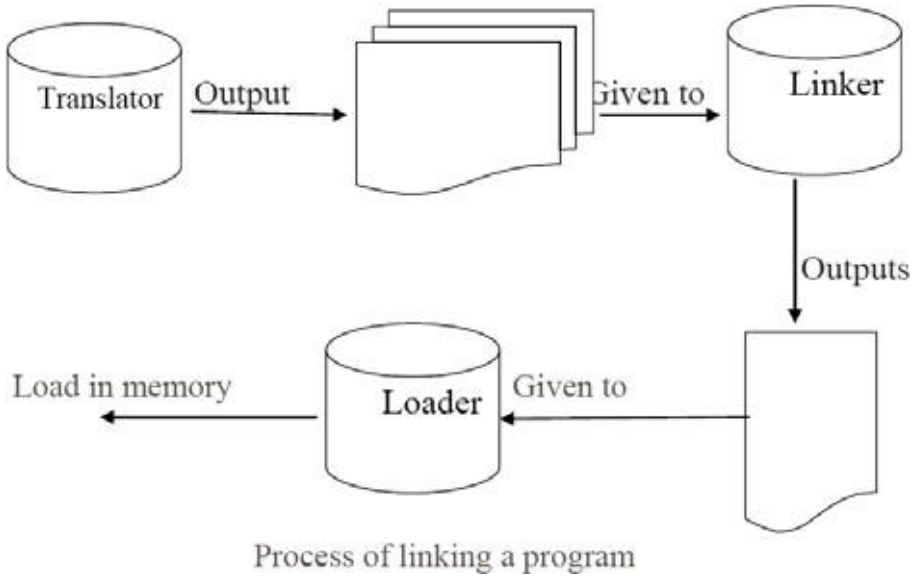
Bootstrap Loader: As we turn on the computer there is nothing meaningful in the main memory (RAM). A small program is written and stored in the ROM. This program initially loads the operating system from secondary storage to main memory. The operating system then takes the overall control. This program which is responsible for booting up the system is called bootstrap loader. This is the program which must be executed first when the system is first powered on. If the program starts from the location x then to execute this program the program counter of this machine should be loaded with the value x. Thus the task of setting

the initial value of the program counter is to be done by machine hardware. The bootstrap loader is a very small program which is to be fitted in the ROM. The task of bootstrap loader is to load the necessary portion of the operating system in the main memory .The initial address at which the bootstrap loader is to be loaded is generally the lowest (may be at 0th location) or the highest location. .

Concept of Linking: As we have discussed earlier, the execution of program can be done with the help of following steps

1. Translation of the program(done by assembler or compiler)
2. Linking of the program with all other programs which are needed for execution. This also involves preparation of a program called load module.
3. Loading of the load module prepared by linker to some specified memory location.

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary program. The "binary program also contains some necessary information about allocation and relocation. The loader then load s this program into memory for execution purpose.



Process of linking a program

Various tasks of linker are -

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.
2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.

3. At correct locations in the load module, copy the binary machineinstructions and constant data in order to prepare ready to execute module.

The linking process is performed in two passes. Two passes are necessarybecause the linker may encounter a forward reference before knowing its address. So it is necessary to scan all the DEFINITION and USE table at least once. Linker then builds the Global symbol table with the help of USE and DEFINITION table. In Global symbol table name of each externally referenced symbol is included along with its address relative tobeginning of the load module. And during pass 2, the addresses of external references are replaced by obtaining the addresses from global symbol table.

Subroutine Linkage

To understand the concept of subroutine linkages, first consider the following scenario: "In Program A a call to subroutine B is made. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler cannot find the value of this symbolic reference andhence it will declare it as an error.

To overcome problem, there should be some mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must - be declared as an external routine in A.

To declare such subroutine as external, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. This overall process of establishing the relations between the subroutines can be conceptually called a *subroutine linkage*.

Direct Linking Loaders

The direct linking loader is the most common type of loader. The loader cannot have the direct access to the source code. The assembler should give the following information to the loader

i. The length of the object code segment
ii. The list of all the symbols which are not defined in the current segment but can be used in the current segment.
iii. The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure calledDEFINITION table.

There are 4 types of cards available in the direct linking loader. They are

i. **ESD-External symbol dictionary**
ii. ***TXT-card***
iii. **RLD-Relocation and linking dictionary**
iv. ***END-card***

i. *ESD card:* It contains information about all symbols that are defined in the program but reference somewhere, It contains:

- Reference number
- Symbol name
- Type Id
- Relative location
- Length

There are again ESD cards classified into 3 types of mnemonics. They are:

i. SD [Segment Definition]: It refers to the segment definition
ii. LD; It refers to the local definition
iii. ER: it refers to the external reference they are used in the [EXTRN] pseudo op code

***ii. TXT Card*:** It contains the actual information are text which are already translated.

iii. RLD Card: This card contains information about location in the program whose contexts depends on the address at which the program is placed.

In this we are used '+' and '–'sign, when we are using the '+' sign then no need of relocation,when we are using '-'sign relocation is necessary.

The format of RLD contains:

i. Reference number
ii. Symbol
iii. Flag
iv. Length
v. Relative location

iv. END Card: It indicates end of the object program.

Design of direct linking loader: Here we are taking PG1 and PG2 are two programs. The relative address and secure code of above two programs is written in the below

ESD Cards: In an ESD card table contains information necessary to build the external symbol dictionary or symbols table. In the above source code the symbols are PG1, PG1ENT2, PG2, andPG2ENT1

Source card reference	*Name*	*Type*	*Id*	*Relative address*	*length*
1	PG1	SD	01	0	60
2	PG1ENT1	LD	-	20	-
2	PG1ENT2	LD	-	30	-
3	PG2	ER	-	-	-
3	PG2ENT1	ER	-	-	-

Here, the PG1 is the segment definition it means, the header of program. PG1ENT1 and PG1ENT2 those are the local definition of program1, so that we are using the type LD. PG2 and PG2ENT1 those are using the EXTRN pseudo op code, so that we are using the type ER.

Text cards: The format of card will be

Source card reference	*Relative address*	*Content*	*Comments*
6	40-43	20	
7	44-47	45	=30+15
8	48-51	7	=30-20-3
9	52-55	0	Unknown to PG1
10	56-60	-16	-20+4

RLD Card:

Source card reference address	*ESD ID*	*Length [bytes]*	*Flag + or -*	*relative*
6	02	4	+	40
7	02	4	+	44
9	03	4	+	52
10	02	4	+	56
10	03	4	+	56
10	02	4	-	56

Specification of data structure: ***Pass1 database:***

i. Input object decks
ii. *Initial Program Load Addresses [IPLA]:* The IPLA supplied by the programmer or operating system that specifies the address to load the first segment.
iii. *Program Load Address counter [PLA]:* It is used to keep track of each segments assignedlocation

iv. *Global External Symbol Table [GEST]:* It is used to store each external symbol and its corresponding assigned core address
v. A copy of the input to be used later by pass2
vi. A printed listing that specifies each external symbol and its assigned value

Pass2 database:

i. A copy of object program is input to pass2
ii. The Initial Program Load Address [IPLA]
iii. The Program Load Address counter [PLA]
iv. A table the Global External Symbol Table [GEST]
v. The Local External Symbol Array [LESA]: which is used to establish a correspondence between the ESD ID numbers used on ESD and RLD cards and the corresponding External symbols , Absolute address value

Format of data bases:

i. *Object deck:* The object deck contains 4 types of cards
ii. ESD Card format:

Source card reference	Name	Type	ID	Relative address	length

iii. TEXT Card:

Source card reference address	Relative address	content

iv. RLD Card:

Source card references	ESD ID	Length	Flag + or -	Relative address

v. *Global External Symbol Table (GEST):* It is used to store each external symbol and its corresponding core address.

External symbol (8 bytes) character	Assigned core (4 bytes) address decimal
"PG1bbbbb"	104
"PG1ENT1b"	124

vi. *Local external symbol array (LESA):* The external symbol is used for relocation and linking purpose. This is used to identify the RLD card by means of an ID number rather than the symbols name. The ID number must match an SD or ER entry on the ESD card

Assigned core address of corresponding symbol [4 bytes]
104
124
134
….
….

This technique saves space and also increases the processing speed.

Other loading segments:

Binders:

In order to avoid the disadvantages of direct linking divides the loading process into two separateprograms:

i. A binder
ii. A module loader

Binder is a program that performs the function as direct linking loader in binding together. It outputs the text as a file or card deck, rather than placing the relocated and linked text directly into memory. The output files are in format ready to be loaded and are called a load module. The module loader loads the module into memory. The binder performs the function of the allocation,relocation and linking

The modules loader performs the function of loading. There are 2 major classes of binders:

i. *Core image builder:* A specific memory allocation of the program is performed at a time that the subroutines are bound together. It is called a core image module and the corresponding binder is called a core image builder

   Advantages: Simple to implement and Fast to execution

   Disadvantages: Difficult to allocate and load the program and Linkage editor

ii. *Linkage editors:* The linkage editor can keep track of relocation information so that the resulting load module can be further relocated and their care the module loader must performs additional allocation and relocation as well as loading but it does not worry about the problem of linking.

   Advantages: More flexible allocation and loading scheme

   Disadvantages: Implementation is so complex