5.1 Prologue :

➤➤➤ [University Exam : May 2005, Dec. 2005 III]

1. An **object program** contains **translated instructions** and **data values** from the source program and **specifies addresses in memory** where these items are to be loaded.
2. There are basically 3 processes which we will discuss in this chapter :
   (a) **Loading** : Which brings the **object program into memory** for execution.
   (b) **Relocation** : Which modifies the object program so that it can be loaded at an **address different from location originally specified**.
   (c) **Linking** : Which **combines two or more separate object programs** and supplies the information needed to allow reference between them.
3. After discussing Macro processor and assembler in the first unit, let us discuss now about loader.

   A loader is a system program that performs the loading function. In other words we can say that the loader is a program which **accepts the object program,** prepares this program for **execution** by computer and initiates the execution. See Fig. 5.1.

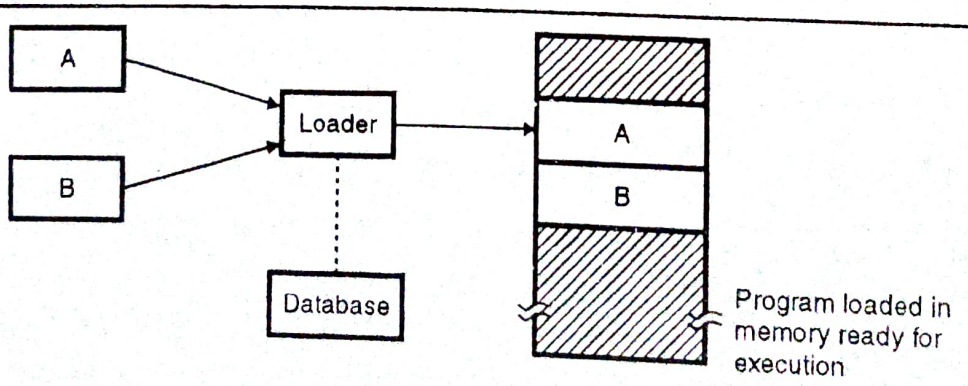


Fig. 5.1 : General loading

4. Many loaders also support relocation and linking. Some systems have a linker to perform the linking operations and a separate loader to handle relocation and loading.

5. In short a loader must perform following 4 functions :
   (a) Allocate space in memory for the programs (allocation).
   (b) Resolve symbolic references between object programs (linking).
   (c) Adjust all address dependent locations, such as address constant to correspond to the allocated space (relocation).
   (d) Physically place the machine instructions and data into memory (loading).
6. Let us elaborate some more on the concept of linking, relocation and loading. First we will discuss about linking.
7. **Linking :**
   (a) In general a program can contain :
      (i) Internal references to **externally defined symbols.**
      (ii) **Internally defined symbols** which may be used externally (i.e. public definitions).
   (b) This resolution is usually performed by a linker which comes into picture after an assembler has finished its task.
   (c) The functions of the linker are as follows :
      (i) To establish the correspondence between external symbol references and definitions and public definitions.
      (ii) To determine the relocatability of symbol attributes and expressions.
      (iii) To substitute for these relocatable addresses.

   The Fig. 5.2 shows the block diagram of a linker.

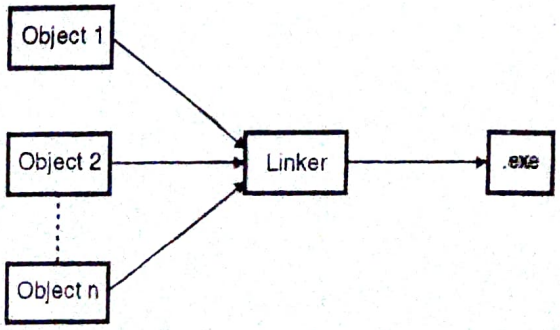


Fig. 5.2 : Block diagram of a linker

8. **Relocation :**

(a) We know that **allocation** means **fixing the memory required** by the programs. This task may be done by an assembler if the program does not contain external subroutines. It may also be done by linker if it knows the size of the total linked programs together.

(b) Task of linking is done by linker. But for execution of the program all these modules are to be brought into memory. There the question arises where ?

(c) The **linked modules** contain addresses which are still **dependent** on the **starting address** of **that module**. But the final memory location of execution is known only to the loader. So all address sensitive instructions are to be adjusted with respect to final memory locations and then only a program can execute.

(d) Generally **linker** will try to put **all modules one after other** to **form** an **executable file**. While doing this, it will try to put first module with respect to zero address, second module with respect to length of first module and so on.

(e) For this activity, linker uses the relocation information, length, machine code and data from object file.

(f) **Relocation** involves, **adjustment of addresses** of all address sensitive entities. The task of assembler is to find out which entity is address sensitive and which is not.

9. **Loading :**

(a) Final step in this process is loading. In this process **executable file** is brought into **main memory** for **execution**.

(b) There are several questions which comes to our mind before doing this activity ? How much of the main memory is required ? Where the program will be loaded into main memory ?

(c) Now, to solve these questions, answer must be there as a part of an executable file. Well, so an executable file contain the **length of the file information** which is passed to an operating system. Operating system passes this information to one of its component called as **'memory management module'**.

(d) This will go through its data structures and looking at main memory status right now (i.e. at the time of request) and depending on the policies, the information about the starting address of the required free slot in the main memory is conveyed to the loader.

(e) Now, loader has both information i.e. what to load ? i.e. an executab file residing in secondary storage device and where to load ? i.e. a starting address of the required free slot in the main memory. So, now the task is very straight forward i.e. pick from some place and dump it somewhere else, it is really a **'loading'.**

(f) Once the loading of program is done, loader loads PC with the starting address of the program and passes control back to the O.S. to execute the program.

10. Now let us study various loader schemes. We will discuss various schemes for accomplishing the four functions of a loader.

5.2 Loader Schemes :

➤➤➤ [University Exam : Dec. 2005 !!!]

5.2.1 Compile and Go Loader :

1. This is a category of loader where **everything** i.e. assembling or compiling, linking, loading etc. goes in **one shot.** So naturally, either the programmer or an assembler (which is a part of assemble and go loader) must know the starting address in the main memory where the program will be executed.

2. One method of performing the loader functions is to have the **assembler run in one part of memory and** place the assembled machine instructions and data as they are assembled, directly into their assigned memory locations as shown in Fig. 5.3.

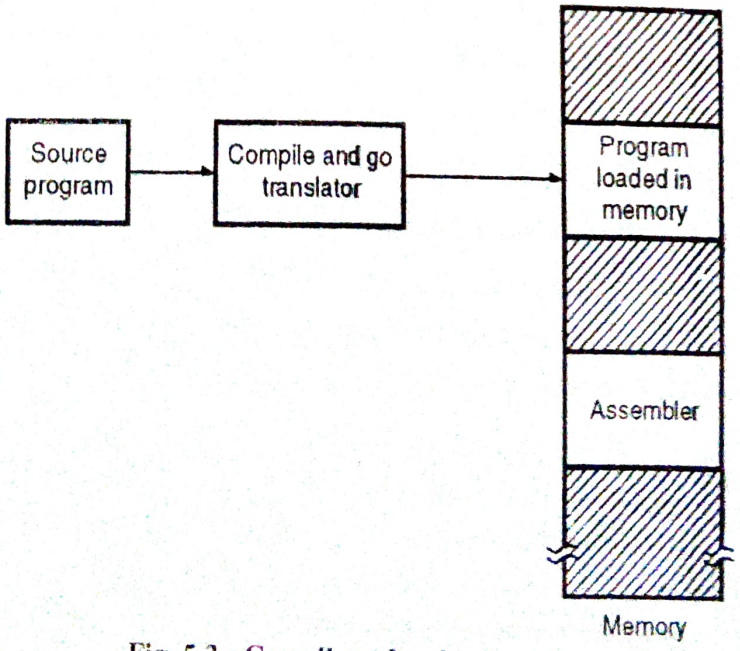


Fig. 5.3 : Compile and go loader scheme

3. When the assembly is completed, the assembler causes a transfer to the starting instruction of the program. This loading scheme is also called **assemble and go.**

4. **Advantages of this scheme :**

   Let us discuss the advantages of this scheme.

   (a) It is relatively easy to implement. The assembler simply places the code into core and the loader consists of one instruction, that transfers to the starting instruction of the newly assembled program.

   (b) It is a simple solution. It does not involve extra procedures. It is used by WATFOR FORTRAN compiler.

5. **Disadvantages of this scheme :**

   (a) A portion of memory is wasted because the core is occupied by the assembler. This portion then can not be used by object program.

   (b) It is necessary to retranslate or assemble the user's program every time it is run.

   (c) It is very difficult to handle multiple segments (a segment can be a program or data), if the source programs are in different languages i.e. one subroutine in assembly language and another in FORTRAN.

5.2.2 General Loader Scheme :

1. The problem of wasting core for the assembler can be solved by **outputting the instructions and data as they are assembled.**

   Such an output could be saved and loaded whenever the code was to be executed.

2. The **assembled programs** could be **loaded into the same area** in core that the **assembler occupied** since the translation has already been completed.

   This **output form,** containing a **coded form of the instructions** is called an **object program.**

3. Use of an object program as intermediate data requires the addition of a new program to the system called a loader as shown in Fig. 5.4.

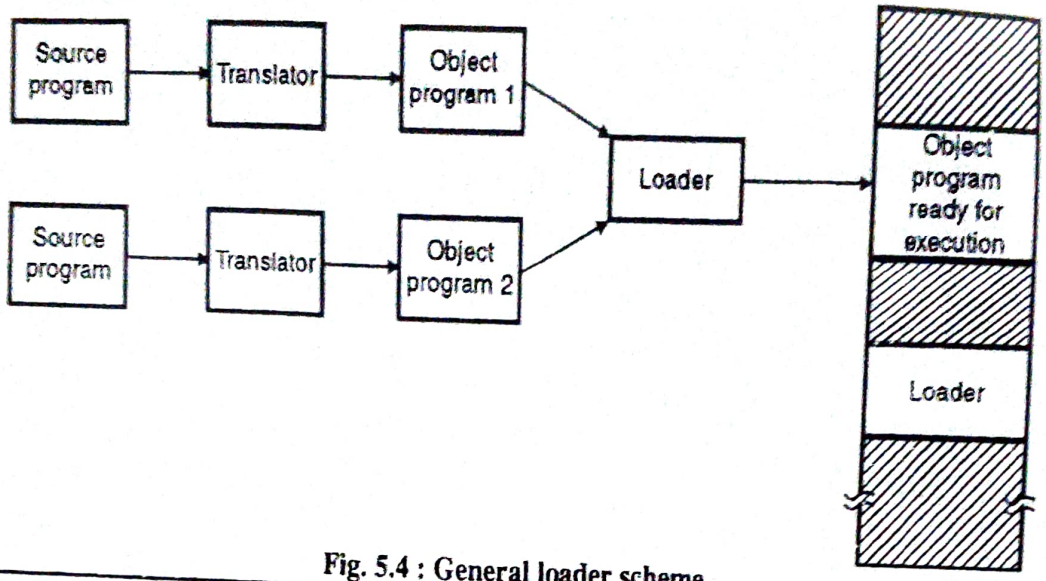


Fig. 5.4 : General loader scheme

4. The loader accepts the assembled machine instructions, data and other information present in object program and places machine instructions and data in core in an executable form.

5. **Advantages of this scheme :**

   (a) Loader is assumed to be smaller than the assembler, so more memory is available to the user.

   (b) Reassembly is no longer necessary to run the program at a later date.

   (c) If all the source program translators (assemblers and compilers) produce compatible object program and use compatible linkage conventions, it is possible to write subroutines in several different languages since the object program to be processed by the loader will all be in the machine language.

5.2.3 Absolute Loaders :

1. The simplest type of loader scheme which fits in the general loader scheme is called an absolute loader.

2. This scheme is same that of "compile and go" scheme except that the **assembler produces relocated object files** which can be stored on the secondary storage media for running it in future.

3. Absolute loader takes away few difficulties of compile and go loader by bifurcating assembling and loading functions. So **an assembler** performs its task and comes out with an **object file**. The loader accepts the assembled machine instructions, data and other information present in the

object format, and places the machine instructions and data in an executable form.

4. We note here that no relocation information is needed to be stored as a part of an object file, so the word "absolute". This comparatively saves space.

5. **Locations** being **absolute** either **programmer** or an **assembler need** to know the **memory management** issues.

   Resolution of external references and linking of different modules which are interdependent is done by the **programmer** assuming programmer knows memory management.

6. Loader in this case, is a small piece of software which will simply load each module at respective locations in main memory as prescribed by an assembler through object file.

Fig. 5.5 shows diagram of absolute loader.

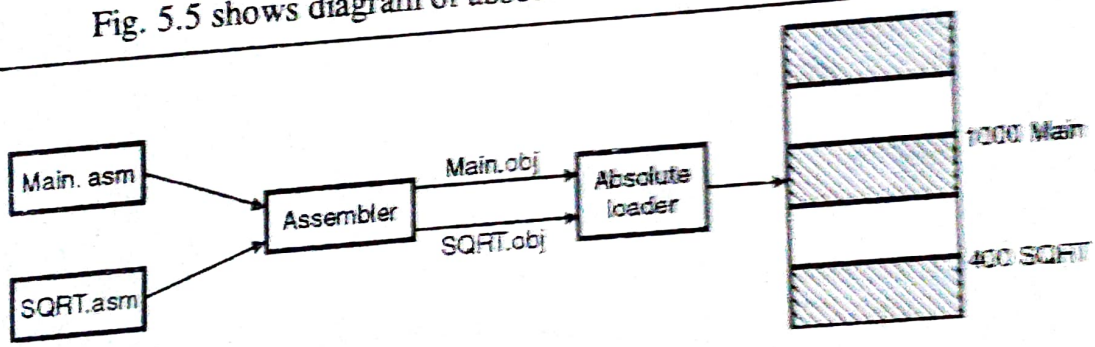


Fig. 5.5 : Absolute loader

7. In this scheme, multiple segments are allowed. Logically even source programs written in multiple languages are allowed but respective assembler has to take care of converting them into a common object format.

   For this the assembler must give following information through object file :

   (a) **Starting address and name** of each module.

   (b) **Length** of each module.

8. The general schematic for an absolute loader can be as shown in Fig. 5.6.

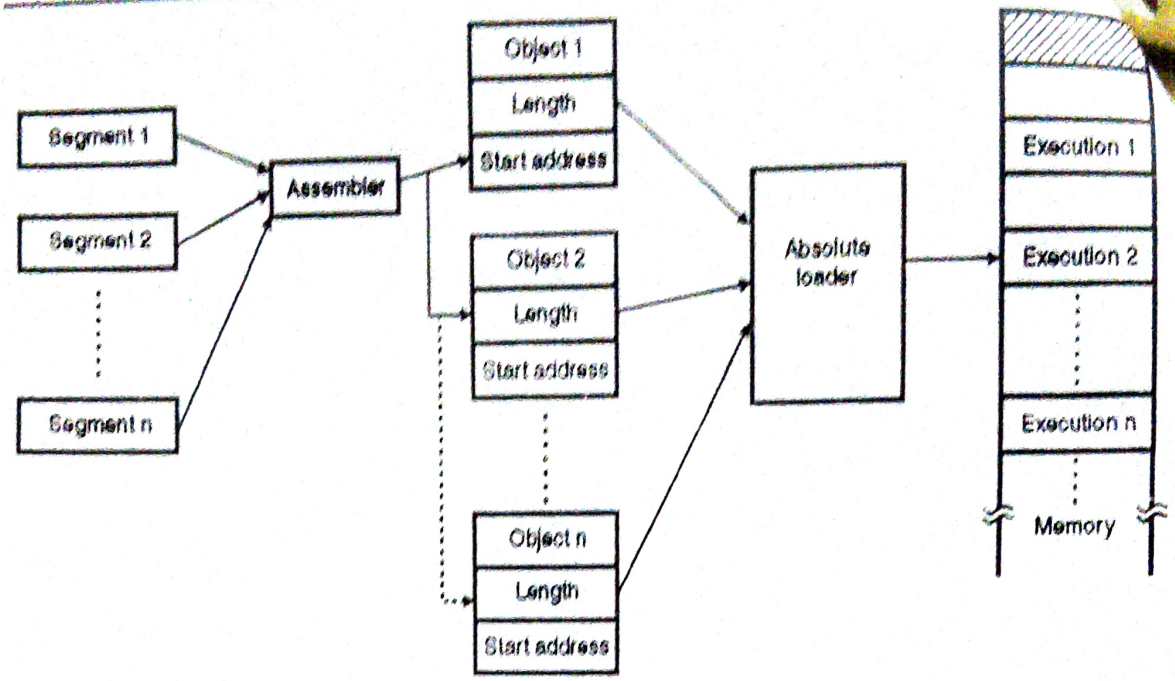


Fig. 5.6 : General schematic of absolute loader

9. **Disadvantages of absolute loader :**

   1. In this scheme programmer must specify to the assembler the address in core where the program is to be loaded.
   2. If there are multiple subroutines, the programmer must remember the address of each and use that absolute address explicitly in his other subroutines to perform subroutine linkage.
   3. The programmer has to be careful of not to assign two subroutines to the same or overlapping locations.

10. **Advantages of this scheme :**

   (a) No relocation information is required, so the size of the program is comparatively small.

   (b) This scheme makes more core available to the user since the assembler is not in memory at load time.

   (c) Since the object code is ready

   Execution time ≅ load time.

   (d) Multiple segments or multiple languages are allowed as long as multiple language assemblers converting different languages to a common object code format are available.

11. The 4 loader functions are accomplished as follows in an absolute loading scheme.:

   (i) Allocation : by programmer.

(ii) Linking : by programmer.

(iii) Relocation : by assembler.

(iv) Loader : by loader.

5.2.4 Subroutine Linkages :

1. In this section, we are going to discuss from programmer's point of view. It is about the special mechanism for calling another subroutine in an assembly language program.
2. What is the problem of subroutine linkage ?

   A main program A wishes to transfer to subprogram B. The programmer in program A could write a transfer instruction to subprogram B. But the assembler does not know the value of this symbol reference and will declare it as error saying undefined symbol, unless a special mechanism has been provided.
3. This mechanism is typically implemented with a relocating or a direct linking loader, which we will discuss in next subsequent sections.

   The assembler pseudo-opcode **EXTRN** followed by a list of symbols indicates that these symbols are defined in other programs but referenced in the present program.
4. Correspondingly, if a symbol is defined in one program and referenced in others, we insert it into a **symbol list** following the pseudo-opcode **ENTRY**.

   In turn, the assembler will inform the loader that these symbols may be referenced by other programs.
5. Let us see an example

```
MAIN    START
        EXTRN SUBROUT
        ---------
        ---------
        L       15 = A (SUBROUT)  }   Call SUBROUT
        BALR    14, 15
        :
        :
        END
```

In this example the variable SUBROUT is declared as an external variable that is a variable referenced but not defined in this program.

The **load instruction** loads the address of that variable into register 15. The **BALR** instruction **branches** to the contexts of register 15, which is the address of SUBROUT and leaves the value of the next instruction in register 14.

6. Assembler symbols are either external or internal. External means that their value is not known to the assembler but will be provided by the loader.

Before discussing about the relocating loader, let us first understand the concept of relocation in some more detail.

5.2.5 Relocation :

➤➤➤ **[University Exam : May 2004 !!!]**

1. Let AA be the set of absolute addresses-instruction or data addresses used in the instruction of a program P. AA $\neq \phi$ implies that program P assumes its instructions and data to occupy memory words with specific addresses. Such a program is called an **address sensitive** program.
2. Relocation of programs is always considered if they have address sensitive data or address sensitive information.

   An address sensitive program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin.
3. Until now, whatever loaders we have seen they do not require relocation information. Now, here onwards most of the loaders will require that information. So let us have a little bit discussion on program relocation. I have already told what is program relocation ? Why it is required ? and so on.
4. **Program relocation** is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.
5. It is an **assembler who decides whether** an entity is absolute or relocatable. **Absolute entities** are machine code or register etc. Even though they are put at any location in memory, their value does not change, so these entities are address insensitive.
6. Mostly labels, symbols, literal addresses redefinables are relocatable. A relocating program can be processed to relocate it to a desired area of memory. **Object modules** are example of relocatable program.

7. A **self relocating** program is a program which can perform the relocation of its own address sensitive instructions. There are two provisions for this purpose :
   (a) A table of information concerning the address sensitive instructions.
   (b) Code to perform the relocation of address sensitive instructions. This is called as **relocating logic**.
8. A self relocating program can execute in any area of memory. This is very important in timesharing O.S. where the load address of a program is likely to be different for different executions.
9. To determine whether the result of address expression is absolute or relative, replace each term in address expression either by 0, if the term is for an absolute address or by 1, if the term is for a relative address. The + or – signs in the expression are not changed. Thus the address expression is transformed to an algebraic sum 0's and 1's. If the sum is 0 then expression is absolute otherwise it is relative.

Address expression	Relocation attribute
X – Y	absolute
X + Y – Z	relative
X + 7 + V – W	relative

Where, X, Y, Z, V, W may be either symbol or label.

After having enough discussion about the concept of relocation let us move towards relocating loaders.

5.2.6 Relocating Loader :

1. To avoid possible reassembling of all subroutines when a single subroutine is changed and to perform tasks of allocation and linking for the programmer, relocating loader was introduced.

   An example of relocating loader can be considered as Binary Symbolic Subroutines (BSS) loader used in IBM 7094.
2. BSS loader allows **many procedure segments** and **one common data segment**. The assembler assembles each procedure segment independently and passes on to the loader the text and information as to relocation and intersegment references.
3. The output of a relocating assembler is the object program and information about all other programs it references. In addition, there is **relocation**

information as to locations in this program that need to be changed if it is to be loaded in an arbitrary place in core.

4. For each source program, the assembler outputs a text which is prefixed by a **transfer vector.**

   Transfer vector consists of addresses containing **names of the subroutines** referenced by the source program.

5. Assembler also provide loader with additional information as length of entire program and length of transfer vector.

   If we consider a program, in which SQRT subroutine is called. Then transfer vector would contain symbolic name to SQRT. Whenever SQRT would be called, then that calling would be branched to the location of the transfer vector associated with SQRT.

6. So in short **assembler** provides with following information
   (a) Machine translation of a program.
   (b) Information about all other programs it references.
   (c) Relocation information.
   (d) Machine translation of a program prefixed by a transfer vector.
   (e) Length of entire program and length of transfer vector portion.

7. Now its loader's turn. A loader loads transfer vector and text into core, then the loader would **load each subroutine identified** in transfer vector.

8. Then the loader would place a **transfer instruction** to the corresponding subroutine in each entry in the transfer vector.

   The transfer vector is used to solve the problem of linking and the program length information is used to solve the problem of allocation.

9. The four functions of loader i.e. allocation, linking, relocation and loading, all are performed by BSS loader.

   The problem of relocation can be solved by **relocation bits**. Let me elaborate on relocation bits.

10. The assembler associates a bit with each instruction or address field. If this bit is one then the corresponding address field must be relocated, otherwise the field is not relocated.

11. **Advantages of relocating loader :**
    (a) It avoids reassembling of all subroutines, when a single subroutine changes.
    (b) All four function i.e. allocation, linking, relocation and loading are performed by loader.
    (c) It also allows independent translation of each module.
12. **Disadvantages of relocating loader :**
    (a) The transfer vector linkage is only useful for transfers and is not well suited for loading or storing external data.
    (b) Transfer vector increases the size of the object program in memory.
    (c) The BSS loader processes procedure segment but does not facilitate access to data segments that can be shared.

The last disadvantage can be over come by allowing one common data segment, which can be implemented by extending the relocation bits to be used as two bits per half word address field.

If bits are 01, then the half word is relocated relative to the procedure segment, if bits are 10, then it is relocated related to the address of the single common data segment.

If the bits are 00 or 11, the half word is not relocated.

5.2.7 Linking :

➤➤➤ **[University Exam : May 2004 !!!]**

We had seen the concept of relocation in section 5.2.5. Now let us go through the concept of linking.

1. A program consists of set of program units i.e. they may be called as subroutines.

   Let us assume that a program contains a set of program units

   SP as Pi, Pj ..... and so on.

   If Pi and Pj need to interact with each other then they must contain **public definitions** and **external references**.

2. **Public definitions :** Any symbol or variable defined as public can be referenced in other program units.

   External reference : Any symbol which is not defined in the program containing the reference.

3. The ENTRY statement lists the public definitions of a program unit i.e. symbols defined in the program unit which may be referenced in other program units.

   The EXTRN statement lists the symbol to which external references are made in program unit.

4. Before any program can be executed, it is necessary that every external reference in Pi should be bound to the correct link time address.

   "Linking is the process of binding an external reference to the correct link time address."

5. An external reference is said to be **unresolved** until linking is performed for it. It is said to be **resolved** when linking is completed.

After going through the basic concepts of linking, let us move on to direct linking loader.

5.2.8 Direct Linking Loader :

1. A direct linking loader is a general relocatable loader. It is most widely used scheme presently.
2. It allows multiple procedure segments and multiple data segments. Any segment can reference data or instructions contained in other segments.
3. The assembler must give the loader the following information with each procedure or data segment :
   (a) The length of segment.
   (b) A list of all symbols in segment that may be referenced by other segments i.e. public definition and their relative location within the segment.
   (c) A list of all symbols not defined in the segment but referenced in the segment i.e. external reference.
   (d) The machine code translation of source program and the relative address assigned.
4. Object module is produced by an assembler. Object modules correspond to the binary symbolic form of a program. Each module is divided into 4 sections as shown in Fig. 5.7.

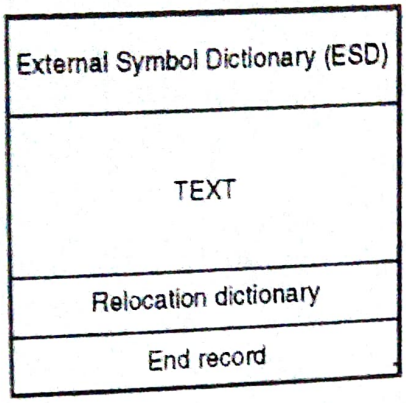


Fig. 5.7 : Object module format

5. **External Symbol Dictionary (ESD) :** It is a table that contains an entry for each external symbol defined within the module.

   Each entry in the ESD has a type assigned to it that indicates its function.

6. **Text :** Text portion of object module contains the relocatable machine language instructions and data that were produced during translation.

7. **Relocation Dictionary :** It contains one entry for each address that must be relocated when the module is loaded into main memory. For this the assembler must specify information enabling the loader to correct their contents.

8. **End record :** Indicates the end of object module. In linking together a set of modules, this loader is responsible for

   (i) assigning addresses

   (ii) relocating address constants

   (iii) creating an output module called load module.

5.3 Machine Independent Features of Loader :

As seen in earlier sections relocation of few of the entities of the object code as well as external references makes it machine dependent. There are certain aspects of loading that are machine neutral.

They are :

(a) Automatic library search

(b) Loader options

Let us discuss each of them a little more in detail.

Automatic library search :

1. It is basically used for handling external references. This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.
2. Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. In most cases there is a standard system library that is used in this way.
3. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The programmer only has to mention the subroutine names as external references in the source program. On some systems this feature is referred to as **automatic library call** while on others **automatic library search**.
4. If after linking together a set of modules the linkage editor or linker detects any unresolved external references, it automatically searches a specified library the call library in an attempt to resolve these external references.
5. The subroutines fetched from a library may themselves contain external references. It is therefore necessary to repeat the library search process until all references are solved. If unresolved external references remain after the library search is completed, these must be treated as errors.
6. The **libraries** to be searched contains a special **file structure**. This structure contains a directory that gives the name of each routine and a pointer to its address within the file. If a subroutine is to be called by more than one name, both names are entered into the directory.
7. Both directory entry points to the same copy of routine. Thus the **library search** itself involves a **search of the directory**, followed by reading the object programs indicated by this search.

Loader options :

1. Many loaders allow the user to specify options that modify the standard processing. Many loaders have a special command language that is used to specify options.
2. There are many ways of doing it :
   (a) Separate input file.
   (b) Statements embedded in the primary input stream.
   (c) Loader control statements in the source program.

3. Sometimes there is a separate input file to the loader that contains such control statements. Sometimes these same statements can also be embedded in the primary input stream between object programs.

On a few systems the programmer can even include loader control statements in the source program and the assembler or compiler retains these commands as a part of the object program.

5.4 Dynamic Linking :

➤➤➤ [University Exam : May 2005 !!!]

1. Linking could be carried out at seven different times :
   (a) At source program coding time.
   (b) After coding but before translation time.
   (c) At translation time.
   (d) After translation but before loading time.
   (e) At loading time.
   (f) After loading but before execution time.
   (g) At execution time.
2. In this section we will discuss a scheme that **postpones** the linking function **until execution time**. A subroutine is loaded and linked to the rest of the program when it is first called. This type of function is called as dynamic linking.
3. Dynamic linking is used to allow several executing programs to share one copy of a subroutine or library.

   In an object oriented system, dynamic linking is often used for reference to software objects. This allows the implementation of the object and its methods to be determined at the time the program is run.
4. The implementation can be changed at any time, without affecting the program that makes use of the object. Dynamic linking also makes its possible for one object to be shared by several programs.
5. Dynamic linking provides the ability to **load** the routines only **when they are needed**. If the subroutines involved are large or have many external references, this can result in substantial savings of time and memory space.
6. Suppose that in any one execution a program uses only a few out of a large number of possible subroutines, but the exact routines needed can not be predicted until the program examines its input. Dynamic linking avoids the necessity of loading the entire library for each execution.

7. Dynamic linking may make it unnecessary for the program even to kno the possible set of subroutines that might be used. The subroutines name might simply be treated as another input then.

8. Dynamic linking generally requires some help form the O.S. If the processes in the memory are protected from one another, then the O.S. is the only entity that can check to see whether the needed routine is the another processes memory space and can allow multiple processes to access the same memory addresses.

   This concept is expanded when used in conjunction with paging.

9. Dynamic linking offers advantages over other types of linking. Suppose that a program contains subroutines that correct or clearly diagnose errors in the input data during execution. If such errors are rare, the correction and diagnostic routines may not be used at all during most execution of the program.

10. However if the program were completely linked before execution these subroutines would need to be loaded and linked every time the program is run.

11. Dynamic linking provides the ability to load the routines only when they are needed. If the subroutines involved are large or have many external references. This can result in substantial savings of time and memory space.

12. In this, the assembler produces text, binding and relocation information from a source program. The loader loads only the main program. If the main program should execute a transfer instruction to an external address or should reference an external variable the loader is called.

    Only then is the segment containing the external reference loaded.

13. Advantage here is that no overhead is incurred unless the procedure to be called or referenced is actually used. A further advantage is that the system can be dynamically reconfigured.

14. The major drawback to using this type of loading scheme is the considerable overhead and complexity incurred, due to the fact that we have postponed most of the binding process until execution time.

Dynamic Loading :

➤➤➤ [University Exam : May 2005 !!!]

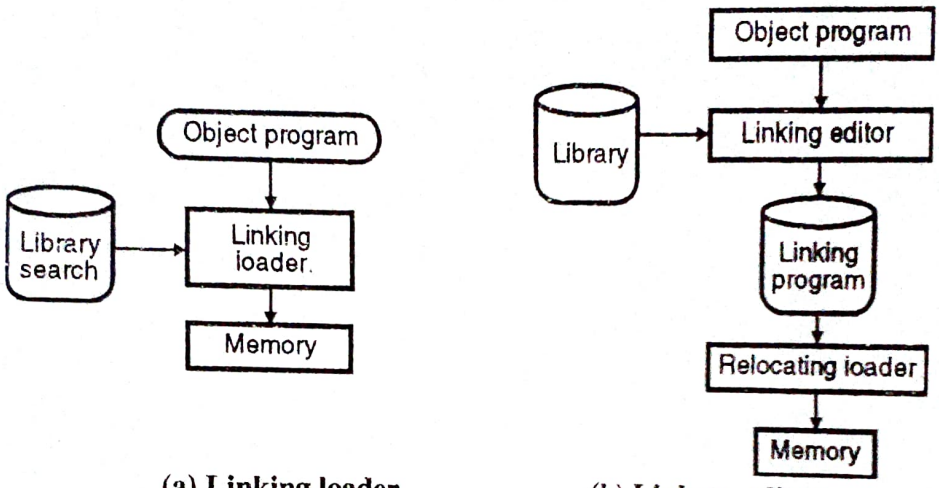
1. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disc in a relocated load format.
2. The main program is loaded into memory and executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
3. If it has not been, the relocated linking loader is called to load the desired routine into memory and to update the program's address table to reflect this change. Then, control is passed to the newly loaded routine.
4. It can so happen that a programmer in his/her program has *defined* so many subroutines (functions) but called only one of them. The natural question is, if the subroutines are never called in a program (neither they are public), then there is no need to load them into main memory and waste main memory space.
5. So, the advantage of dynamic loading is that an unused routine is never loaded. This scheme is particularly useful when large amount of code are needed to handle infrequently occurring cases, such as error routine.
6. In this case, although the total program size may be large, the portion that is actually used (and hence actually loaded) may be much smaller. But the drawback of this scheme is that, the activity of loading is to be done at run time (i.e. on the fly), so definitely there will be a slight bit reduction in run time efficiency, as lot of earlier said activities are to be performed at run time.
7. Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a scheme. Operating system may help the programmers, however, by providing library routines to implement dynamic loading.

5.6 Linkage Editors :

➤➤➤ [University Exam : Dec. 2004, May 2005 !!!]

1. The source program is first assembled or compiled, producing an **object program** (which may contain several control sections).

2. A **linking loader** performs all linking and relocation operations, including automatic library search if specified and loads the linked program directly into memory for execution.
3. A **linker editor** on the other hand, produces a linked version of the program, which is often called as **load module** or an **executable image.** This is written to a file or library for later execution.
4. Figs. 5.8(a) and (b) shows the difference between linking loader and linkage editor.



(a) Linking loader **(b) Linkage editor**

Fig. 5.8

5. In linkage editor, when the user is ready to run the linked program, a simple **relocating loader** can be used to load the program.
6. The only **object code** modification necessary is the addition of **actual load address** relative values within the program.

   The linkage editor performs **relocation of all control sections** relative to start of the linked program.
7. Therefore, all items that need to be modified at load time have values that are relative to the start of the linked program. In this case **loading** can be accomplished in **one pass** with no external **symbol table** required.
8. If a program is to be executed many times without being reassembled, the use of a linkage editor substantially **reduces the overhead** required.
9. Resolution of external references and library searching are only performed once. If the actual address at which the program will be loaded is known in advance, the linkage editor can perform all of the needed relocation.
10. The result is a linked program that is an exact image of the way the program will appear in memory during execution.

11. **Linkage editors** can also be used to build **packages of subroutines** or **other control sections** that are generally used together. This can be useful when dealing with subroutine libraries that support high level programming language.
12. For e.g. in FORTAIN implementation there are large number of **subroutines** that are used to handle **formatted input and output.** These include routines to read and write data blocks, to encode and decode data item according to format specifications.
13. There are a large number of **cross references** between these subprograms because of their closely related functions. However it is desirable that they remain as **separate control sections** for reasons of **program modularity** and maintainability.
14. Sometimes, however a program is reassembled for nearly every execution. This situation might occur in program development and testing environment. It also occurs when a program is used so **infrequently** that is not worthwhile to store the **assembled version** in a **library.**

    In such cases it is more efficient to use a **linking loader,** which avoids the steps of writing and reading the linked program.
15. Another issue is that the **linked program** produced by the linkage editor is generally in a form that is **suitable** for **processing** by a **relocating loader.**

    All external references are resolved and relocation is indicated by some mechanism such as **modification records.**
16. Even though all linking has been performed, information concerning external references is often retained in the linked program. This allows subsequent relinking of the program to replace control sections, modify external references etc.

    If this information is not retained, the linked program can not be reprocessed by the linkage editor, it can only be loaded and executed.
17. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control with a corresponding increase in complexity and overhead.

5.7 Bootstrap Loader :

1. One important question is left behind is our discussion of loaders is that how is the loader itself loaded into memory and then we would say that the **operating system loads the loader.**

2. But then we are left with the same question with respect to operating system. Given an idle computer with no program in memory how do we get things started ?
3. In this situation, with the machine empty and idle there is no need for program relocation. We can simply specify the absolute address for whatever program is first loaded.

   This means that we need some means of accomplishing the functions of an absolute loader.
4. We can have a built in hardware function (or short ROM program) that reads a **fixed length** record from some device into **memory** at fixed location.

   After the read operation is complete control is automatically transferred to the address in memory where the record was stored.
5. This record contains machine instructions that load the absolute program that follows. If the loading process requires more instructions that can be read in a **single record.** This record causes **reading of other** and these in turn can cause the reading of still more records hence the **term bootstrap.**

   The **first record** is generaliy referred to as **bootstrap loader.**
6. When a computer is first turned on or restarted, a **special type of absolute loader** called Bootstrap loader is executed.
7. This bootstrap loads the **first program** to be run by the computer usually an **operating system.**
8. Such a loader is added to the **beginning** of **all object programs** that are to be loaded into an empty and idle system.
9. The bootstrap itself begins at address 0 in memory of the machine. It loads the operating system or some other program starting at address 80.
10. Because this loader is used in a unique situation i.e. the initial program load for the system, the program to be loaded is represented on **device F1** as two hexadecimal digits.
11. However there is no header record, end record or control information. The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
12. After all of the **object code** from **device F1** has been loaded, the **bootstrap** jumps to **address 80,** which begins the execution of the program that was loaded.

5.8 Overlay Structure :

1. An overlay is a part of a program which has the same load origin as some other part of the program. Overlays are used to reduce the main memory requirement of a program.

   Overlay allows a programmer the option of specifying certain dynamic memory management.

2. We refer to a program containing overlays as an **overlay structured program**. Such a program consists of
   i) A permanently resident portion called the root.
   ii) A set of overlays.
3. Execution of an overlay structured program proceeds as follows :
   i) To start with, the root is loaded in memory and given control for the purpose of execution.
   ii) Other overlays are loaded as and when needed.
   iii) Loading of an overlay over write a previously loaded overlay with the same load origin.
4. This reduces the memory requirement of a program. It also makes it possible to execute programs whose size exceeds the amount of memory which can be allocated to them.
5. A program in overlay form consists of a set of segments, each of which is composed of one or more control sections. The overlay structure of a program can be represented by a tree as shown in Fig. 5.9.

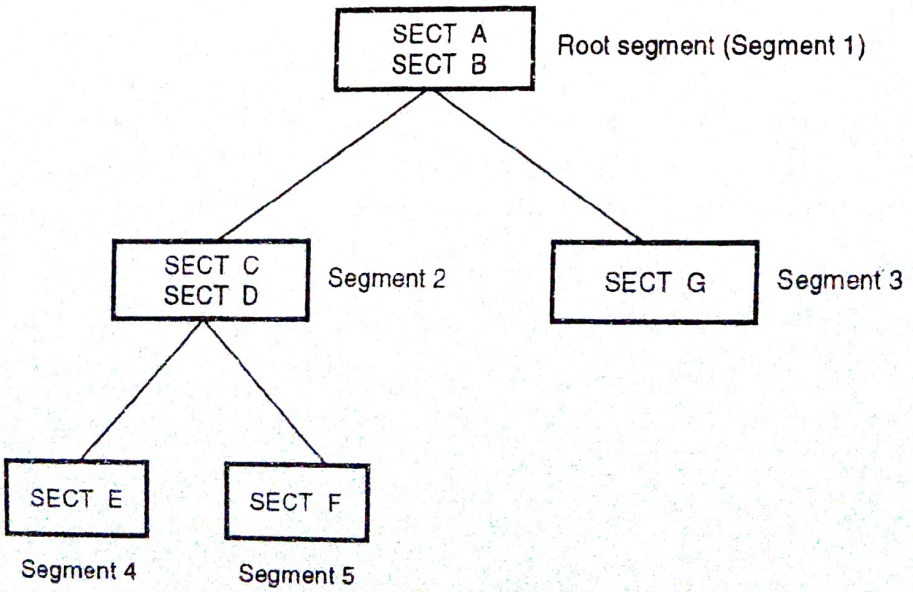


Fig. 5.9 : Example of an overlay structure in tree form

6. The **Root segment** contains all **control sections** that must remain in main memory throughout execution of the program. Segments that lie in a **path are logically related**, when control is passed to a segment, all segments in the path between the root and the segment in question are loaded into main memory if not already there.

7. For example, when control is passed to segment **4**, the overlay supervisor must insure that both segment 4 and segment 2 are in main memory. Segments that lie on the **same level** are not **logically related** and thus can overlay each other. For e.g. segment 2 and segment 3 in Fig. 5.9.

8. Once the programmer has designed the overlay structure of his program, he must indicate that structure to linker, this is accomplished with the overlay Linker Control Statement.

   Each overlay statement specifies :

   a) A set of control sections that are to be grouped into a segment.

   b) The relationship of that segment to other segments.

9. The implementation of overlay is simple. Once the linker determines the layouts of the segments relocates the code in each segment appropriately based on memory location of the segment.

10. For linking and execution of an overlay structured program in MS-DOS the linker produces a single executable file at the output, which contain two provision to support overlays. First an overlay manager module is included in the executable file. This module is responsible for loading the overlays when needed.

11. Second, all calls that cross overlay boundaries are replaced by an interrupt producing instruction. To start with, the overlay manager receives control and loads the root. A procedure call, which crosses overlay boundaries leads to an interrupt.

12. This interrupt is processed by the overlay manager and the appropriate overlay is loaded into memory.

13. As in dynamic loading, overlays do not require any special support from O.S. They can be implemented completely by the user with simple file structures, reading from the files into memory and then jumping to the memory and executing the newly read instructions.

After discussing the core concept of linking, loading and their types now its for us to go into some implementation details.

5.9 Design of an Absolute Loader :

➤➤➤ [University Exam : May 2004, May 2005 !!!]

1. In absolute loader the programmer and the assembler perform the tasks of allocation, relocation and linking. Therefore it is necessary for the loader to read the object program and move the text of the program into the absolute locations specified by the assembler.
2. There are **two types of information** that the object module must communicate from the assembler to the loader.
   i) It must convey the machine instructions that the assembler has created along with the assigned core locations.
   ii) It must convey the entry point of the program, which is where the loader has to transfer control when all instructions are loaded.
3. Fig. 5.10 shows the record format for an absolute loader.

Text Record (for instructions and data) :

Record No.	Contents
1	Record type = 0 (for text card identifier)
2	Count number of bytes of information in a record
3-5	Address at which data on record is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Record sequence number

Transfer Records (to hold entry point to program) :

Record No.	Contents
1	Card type = 1 (transfer record identifier)
2	Count = 0
3-5	Address of entry point
6-72	Empty
73-80	Record sequence number

Fig. 5.10 : Record format for absolute loader

4. The algorithm for an absolute loader is simple : The object program for this loader consists of a series of text records terminated by a transfer record. Therefore the loader should read one record at a time moving the text to the location specified on record, until transfer record is reached.
5. At this point the assembled instructions are in core and it is only necessary to transfer to the entry point specified on transfer record.

Let us make flowchart of this process.

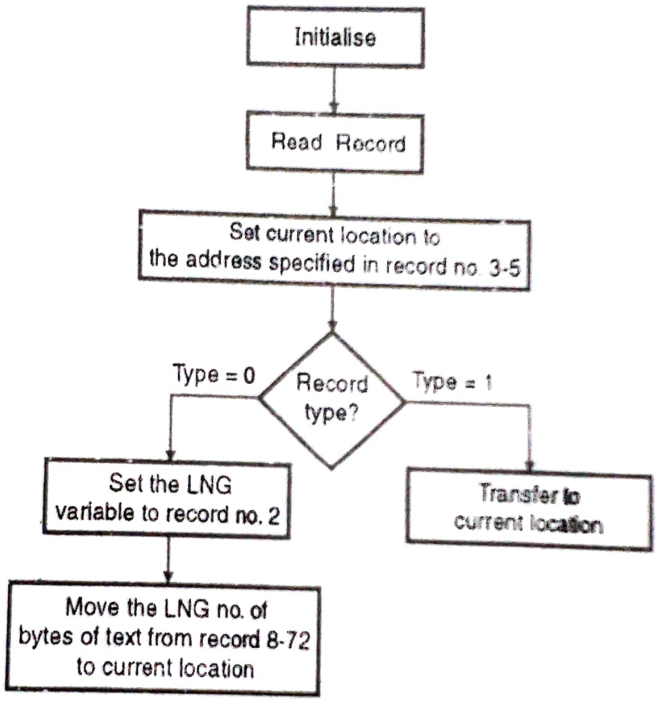


Fig. 5.11 : Flowchart for absolute loader

5.10 Design of Direct Linking Loader :

➤➤➤ **[University Exam : Dec. 2004 !!!]**

1. The design of direct linking loader is more complicated than that of the absolute loader. The input to this loader consists of a set of object programs that are to be linked together.
2. It is possible for a object program to make an external reference to a symbol whose definitions does not appear until later in this input program.
3. In such a case the required linking operation can not be performed until an address is assigned to the external symbol involved. Therefore a direct linking loader usually makes two passes over its input. **Pass1** assigns address to all external symbols and **Pass2** performs actual loading, relocation and linking.

4. There are four sections to the object programs :
   a) External Symbol Dictionary record (ESD)
   b) Instructions and data records called "text" of program (TXT).
   c) Relocation and Linkage Directory records (RLD).
   d) End card (END).
5. For example consider a program B has a table called NAMES it can be accessed by program A as follows.

```
A           START
            EXTRN   NAMES
              :
              :
            L       1, ADDRNAME get address of NAME table
              :
              :
ADDRNAME    DC      A(NAMES)
            END
-------------------------------
B           START
            ENTRY   NAMES
              :
              :
NAME        DC      _____
            END
```

There are three types of external symbols, as illustrated in above example :

i) **Segment Definition (SD) :** name on START or CSECT card.

ii) **Local Definition (LD) :** Specify on ENTRY record. There must be a label in same program with same name.

iii) **External References (ER) :** Specified on EXTRN record. There must be corresponding ENTRY, START or CSECT card in another program with same name.

6. Each SD and ER symbol is assigned a unique number by the assembler. This number is called the symbol's identifier and used in conjunction with the RLD cards.

The TXT cards contains blocks of data and the relative address at which the data is to be placed.

7. Once the loader has decided where to load the program, it merely adds t Program Load Address (PLA) to the relative address and moves the data into the resulting location.

Data Structures Required :

Pass1 databases :

1. Input Object Program.
2. Initial Program Load Address (IPLA) supplied by the programmer or O.S., that specifies the address to load the first segment.
3. A program Load Address (PLA) counter, used to keep track of each segment's assigned location.
4. A table, the Global External Symbol Table (GEST) that is used to store each external symbol and its corresponding assigned core address.
5. A copy of input to be used by later pass2.

Pass2 databases :

1. Copy of object program inputted to pass1.
2. The IPLA.
3. The Program Load Address (PLA).
4. The Global External Symbol Table (GEST), prepared by pass1, containing each external symbol and its corresponding absolute address value.
5. An array, 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 symbol's absolute address value.

The Fig. 5.12 shows the two pass direct linking loader.

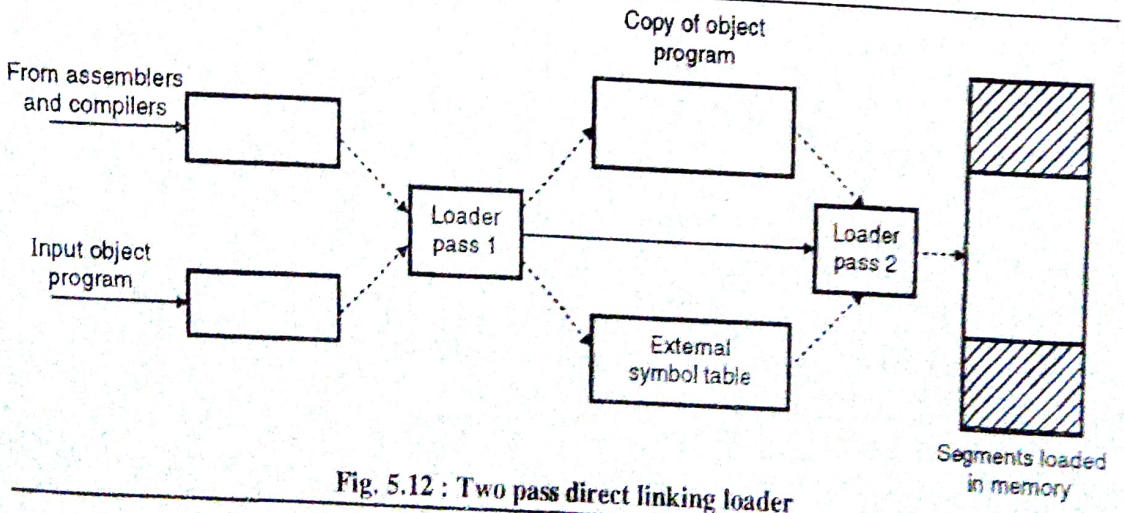


Fig. 5.12 : Two pass direct linking loader

Algorithm for Pass1 :

1. Purpose of first pass is to assign a location to each segment, and thus to define the values of all external symbols.
2. We wish to minimize the amount of core storage required for the total program, we will assign each segment the next available location after the preceding segment.
3. The Initial Program Load Address (IPLA) is normally determined by the O.S. Initially the Program Load Address (PLA) is set to the IPLA. An object program is then read and a copy written for use by pass2.
4. For TXT or RLD record, there is no processing required during pass1 so next record is read. An ESD card is processed in different ways depending upon the type of external symbol SD, LD or ER.
5. If a segment definition ESD record is read, the length field LENGTH, from the record is temporarily saved in the variable, SLENGTH. The value, VALUE to be assigned to this symbol is set to the current value of PLA.
6. The symbol and its assigned value are then stored in the GEST. A similar process is used for **LD symbols**, the value to be assigned is set to the current PLA plus the relative address ADDR indicated on the ESD record.
7. ER symbols do not require any processing during pass1. When END record is encountered, the program load address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment.
8. When the LDT or EOF record is finally read, pass1 is completed and control transfer to pass2.

Algorithm for pass2 :

1. In pass2 the loader loads the text and adjust address constants. At the end of pass2 the loader will transfer control to the loaded program. Following rule is used to determine where to commence execution :
   a) if an address is specified on the END record, that address is used as the execution start address.
   b) Otherwise, execution will commence at the beginning of first segment.
2. At the beginning of pass2 the program load address is initialized as in pass1 and the execution start address (EXADDR) is set to IPLA. The records are read one by one from the object program file left by pass1.

3. Each of the ESD card types is processed differently.

   a) **SD type ESD :** The LENGTH of the segment is temporarily saved in the variable SLENGTH. The appropriate entry in the local external symbol array, LESA (ID) is set to the current value of the Program Load Address.

   b) **LD type ESD :** It does not require any processing during pass2.

   c) **ER type ESD :** GEST is searched for a match with the ER symbol. If it is not found, this is an error. If the symbol is found in the GEST, its value is extracted and the corresponding Local External Symbol Array entry LESA (ID) is set equal to it.

   d) **TXD Record :** When a TXT record is read, the text is copied from the record to the appropriate relocated core location (PLA + ADDR).

   e) **RLD Record :** The value to be used for relocation and linking is extracted from the local external symbol array as specified by the ID field.

      i.e. LESA (ID).

   f) **END Record :** If an execution start address is specified on the END record, it is saved in the variable EXADDR after being relocated by the PLA. The PLA is incremented by the length of the segment and saved in SLENGTH, becoming PLA for next segment.

   g) **LDT/EOF Record :** The loader transfer control to the loaded program at the address specified by current contents of the execution address variable (EXADDR).

5.11 Case Study of MS-DOS Linker :

Uptil now we have seen the various features of linkers and loaders, various loading scheme. But we have not gone through a full description of linkers and loaders used as examples.

Now let us go through the example of MS-DOS linker. In this section we will go through some of the features of MS-DOS linker for Pentium and other X86 systems.