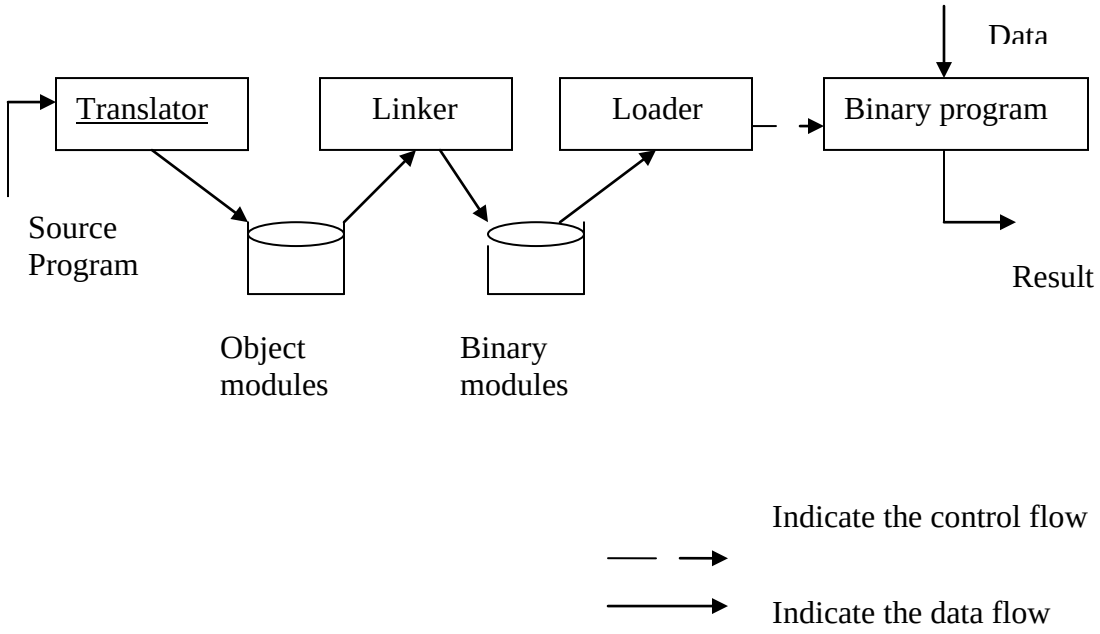
MODULE 5 – LINKERS AND LOADERS

Execution phases

The execution of a program involves 4 steps:-

1) Translation – Converting source program to object modules. The assemblers and compilers fall under the category of translators.
2) Linking – combines two or more separate object modules and supplies the information needed to allow references between them.
3) Relocation – This modifies the object program so that it can be loaded at an address different from the location originally specified.
4) Loading – which brings object program into memory for execution .It is done by a loader.



The translator converts the source program to their corresponding object modules, which are stored to files for future use. At the time of linking, the linker combines all these object modules together and convert them to their respective binary modules. These binary modules are in the ready to execute form. They are also stored to the files for future use. At the time of execution the loader, uses these binary modules and load to the correct memory location and the required binary program is obtained. The binary program in turn receives the input from the user in the form of data and the result is obtained.

The loader and linker has 4 functions to perform:

1. Allocation – space in memory for the program.
2. Linking – resolve symbolic references.
3. Relocation – adjust address dependent statements with available addresses.
4. Loading – load the code to memory.

Translated origin – While compiling a program P, a translator is given an origin specification for P. This is called the translated origin of P. The translator uses the value of the translated origin to perform the memory allocation for the symbols declared in P. This address will be specified by the user in the ORIGIN statement.

```
START 500
----------
----------
END
```

Here translated origin is 500.

Translation time (translated address) – This is the address assigned by the translator.

Execution start address: The execution start address is the address of the instruction from which its execution must begin. The start address specified by the translator is the translated start address of the program.

Linked origin – Address of the origin assumed by the linker while producing a binary program.

Linked address – This is the address assigned by the linker.

Load origin – Address of the origin assigned by the loader while loading the program for execution.

Load time (or load) address – Address assigned by the loader.

The linked and the load origin of the program may change due to the one of the two reasons:-

1) The same set of translated address may have been used in different object modules constituting a program. E.g. object modules of library routines often have the same translated origins. Memory allocation to such programs would conflict unless their origins are changed.
2) An operating system may require that a program should execute from a specific area of the memory. This may require a change in its origin.

Relocation

Program relocation: It is the process of modifying the addresses used in the address sensitive instructions of a program such that the program.

If linked origin ≠ translated origin, relocation must be performed by the linker.
If load origin ≠ linked origin, relocation must be performed by the loader.

Performing relocation

Let the translated and linked origins of program P be t_origin (p) and l_origin (p), respectively. Consider a symbol *symb* in P. Let its translation time be t (symb) and link time address be l (symb). The relocation factor of P is defined as

relocation_factor = l_origin (p) – t_origin (p)…… (Eq No.1)

This value can be positive, negative or zero.

Consider a statement which uses *symb* as an operand. The translator puts the address t (symb) in the instruction generated for it.

Now,

t (symb) = t_origin (p) + d (symb) …… (Eq No.2)
where d (symb) is the offset of *symb* in P.

Hence

l (symb) = l_origin (p) + d (symb)

Using eq.no.1

l (symb) = t_origin (p) + relocation_factor (p) + d (symb)
= t_origin (p) + d (symb) + relocation_factor (p)

Substituting eq (2) in this,

l (symb) = t (symb) + relocation_factor (p)…….. (Eq No.3)

Let IRR (p) designate the set of instructions requiring relocation in program P. Following the Eq. 3, relocation of program P cam be performed by computing the relocation factor for P and adding it to the translation time address in every instruction i ε IRR (p).

Consider a sample assembly program, P and its generated code,

Statement		Address	Code
START	500		
ENTRY	TOTAL		

```
        EXTRN    MAX, ALPHA
        READ     A                  500)          + 09 0 540

LOOP    |                           501)

        |
        MOVER    AREG, ALPHA        518)          + 04 1 000
        BC       ANY, MAX           519)          + 06 6 000
        |
        |
        BC       LT, LOOP           538)          + 06 1 501
        STOP                        539)          + 00 0 000

A       DS       1                  540)          + 00 0 000
TOTAL   DS       1                  541)
        END
```

The translated origin of the program in the ex is 500.
The translation time address of symbol A is 540.

The instruction corresponding to the statement READ A (existing in the translated memory word 500) uses the address 540, hence it is an address sensitive instruction.

If the linked origin is 900, A would have the link time address 940. Hence the address in the READ instruction has to be corrected to 940.

Similarly, the instruction in translated memory word 538 contains 501, the address of LOOP. This should be corrected to 901. Same way operand address in the instructions with the addresses 518 and 519 also need to be corrected.

From the above e.g.

Relocation factor = 900 – 500 = 400

Relocation is performed as follows,

IRR (p) contains the instructions with translated addresses 500 and 538. The instruction with translated address 500 contains the address 540 in the operand field. This address is changed to (540 + 400) = 940. Similarly, 400 is added to the operand address in the instruction with the translated address 538. This achieves the relocation.

ENTRY and EXTRN statements

Consider an application program AP consisting of a set of program units SP = {P (i)}.
A program unit P (i) interacts with another program unit P (j) using addresses of P (j)'s instructions and data in its own instructions. To realize such interactions, P (j) and P (i) must contain public definitions and external references as defined in the following.

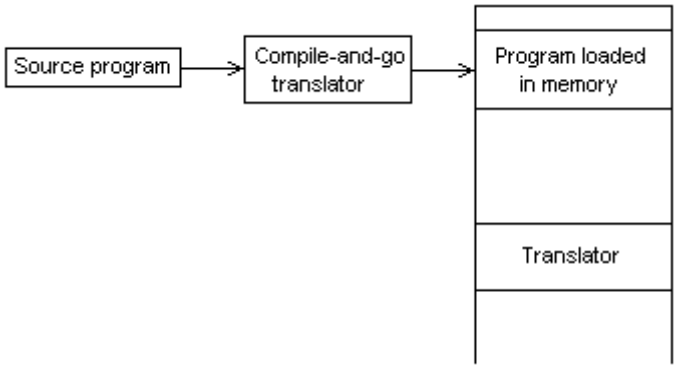
Public definition: a symbol pub_symb defined in a program unit which may be referenced in other program units. This is denoted with the keyword ENTRY and in the e.g. it is TOTAL.
External reference: a reference to a symbol ext_symb which is not defined in the program unit containing the reference. This is denoted with the keyword EXTRN and in the e.g it is MAX and ALPHA.
These are collectively called subroutine linkages. They link a subroutine which is defined in another program.
EXTRN: EXTRN followed by a set of symbols indicates that they are defined in other programs, but referred in the present program.
ENTRY: It indicates that symbols that are defined in the present program are referred in other programs.

Loader schemes

There are various loading schemes available and different loader schemes are:

1. Compile-and -go loader
2. General loading scheme
3. Absolute loader
4. Relocating loaders
5. Linking loaders

Compile-and-go loader

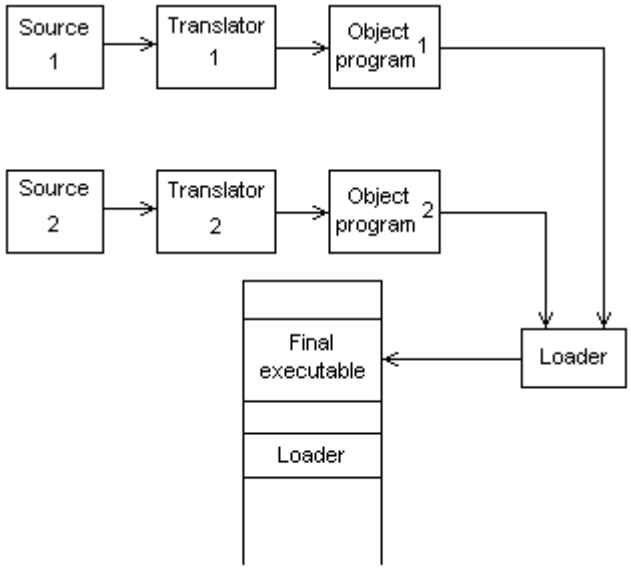


This is also called assemble-and-go loader. It is one of the easiest to implement. Here, the assembler runs in one part of the memory and places the assembled machine code, as they are assembled, directly into the assigned memory locations. So, assembler must always be present in the memory. It has 3 main disadvantages:

1. A portion of the memory is wasted.
2. Necessary to assemble user program every time it is run.
3. Very difficult to handle multiple source files in different languages.

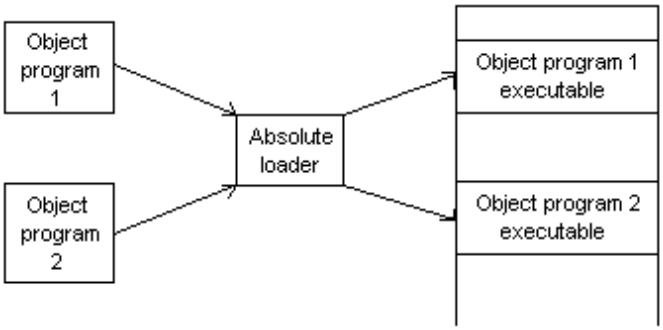
Turbo C uses this scheme.

General loader scheme



Here, different source programs are translated separately to get the respective object program. This can be the different modules of the same program also. Then, they are loaded. The loader combines the codes and executes them. Here, the object modules are saved in the secondary storage. So, the code can be loaded in the space where the assembler had been in the earlier case. But here, an extra component called loader is needed. Loader is generally smaller than the assembler. So, more space is available to the user.

Absolute loader



Here, the assembler outputs the machine language translation of the source program in almost the same form as in the 1st scheme. But here, the assembler is not in memory at

loading time. So, more core is available to the user. They are actually, the combination of both previous schemes. The main problem is that the addresses have to be given by the user and they should not overlap.

Algorithm for an Absolute Loader:

Begin
Read the header record
Verify the program name and length
Read the first text record
While record type! = 'E' do
{
if object code is in character form
Convert it into internal representation
}
Move object codes to the specified location in the memory.
Read the next text record.
End while
Jump to address specified in End record
End

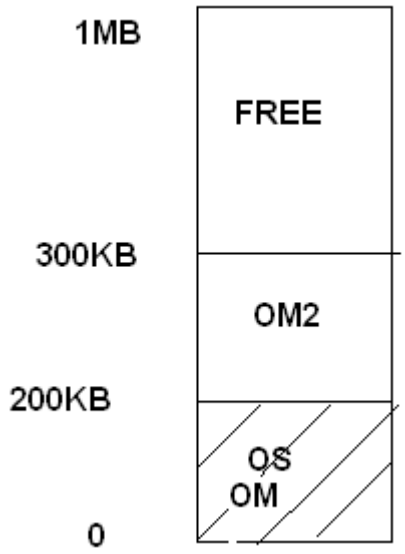
Relocating loaders:

The loaders, which allow program relocation, are called relocating loaders.

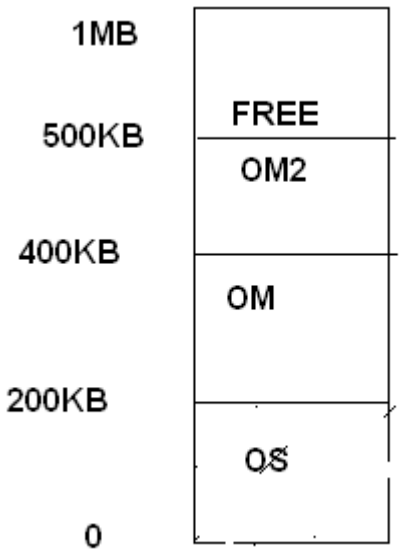
Relocation:

It is one of important concept for the design of relocating or linking loaders.

This object module occupies 200KB of memory from 0 to 200KB.This has to be loaded into the main memory.



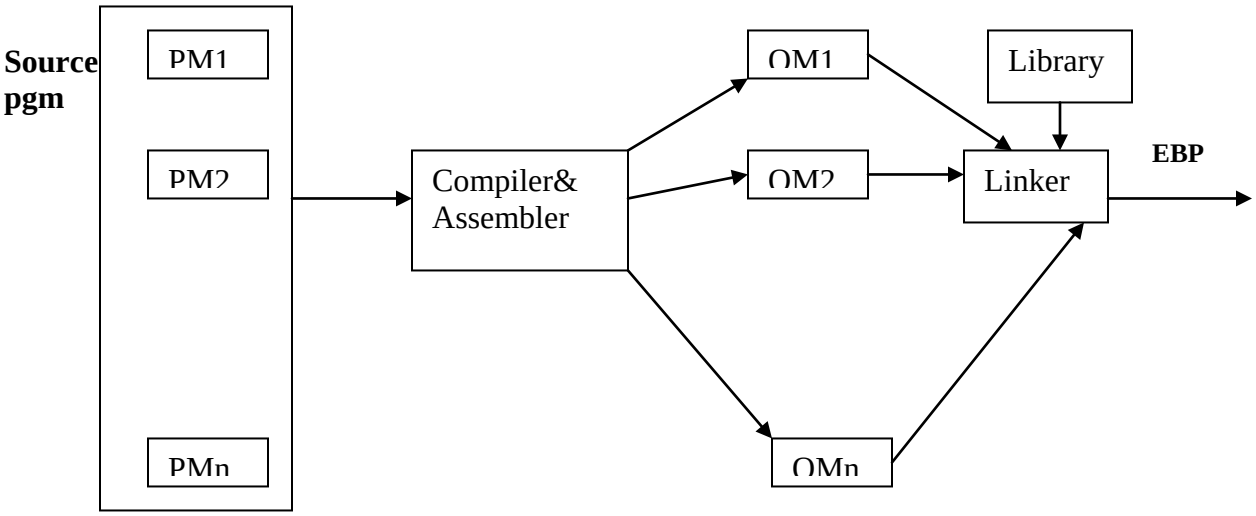
In the main memory, already OS occupies the memory from 0 to 200KB.Now the object module OM has to be placed over the OS from 200KB onwards. But OM has been placed from 0 to 200KB.Now there is an overlap, this leads to problems and confusions. The module OM has to be addressed from 200KB to 400KB.So the starting address of OM is 200.This is called relocation.



Linking loaders:

Generally a program consists of several procedures. The compiler translates all the procedures separately, independently into distinct object modules. These are the most of the time stored in the secondary memory. In order to execute the object modules, these must be linked together and loaded into main memory. Linking of various object modules are done by the linker. The linker's function is to collect the various object modules and link them together called as 'Executable binary program'.

The loader can be a linking loader if it is linking the necessary library functions and symbolic references. These receive a set of object program as input to be linked together. It eliminates the disadvantages of other loading schemes. It also takes care of relocation.



Overlays:

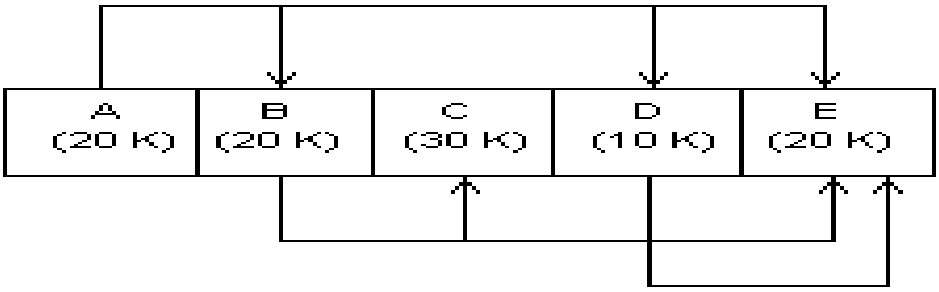
An overlay is a part of a program which has the same load origin as some other parts of the program. Overlays are used to reduce the main memory requirement of a program.

Overlay structured program:

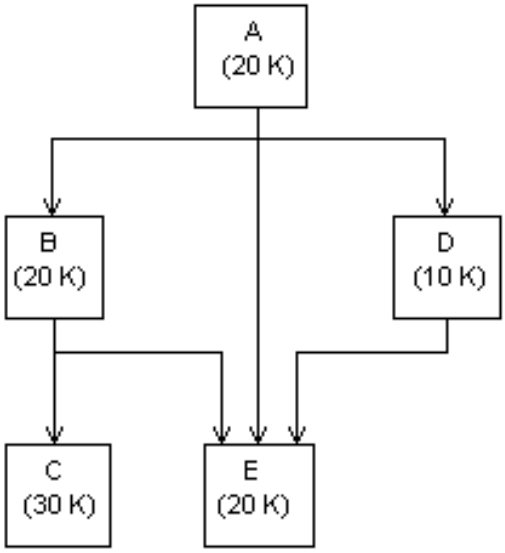
A program containing overlays as an overlay structured program, such a program consist of
1) A permanently resident position called root.
2) A set of overlays.

Execution of an overlay structured program proceeds as follows. To start with, the root is loaded into the memory and given control for the execution. Other overlays are loaded as and when needed. The loading of an overlay overwrites a previously loaded overlay with the same load origin. This reduces the memory requirement of a program.

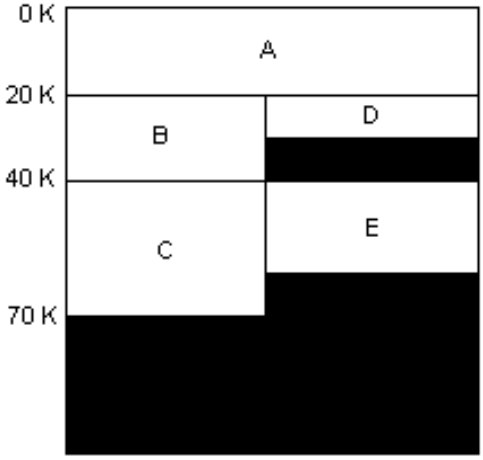
Eg:



The overlay structure will be:



So, probable allocation is like:



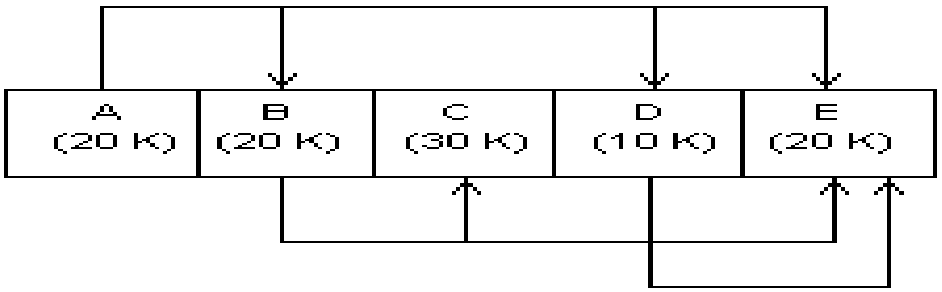
Dynamic loading

In each of the loader schemes we have assumed that all of the subroutines needed are loaded into the memory at the same time. If the total amount of memory required by all

these subroutines exceeds the amount available, then there is trouble. Then we use dynamic loading schemes to solve this problem.

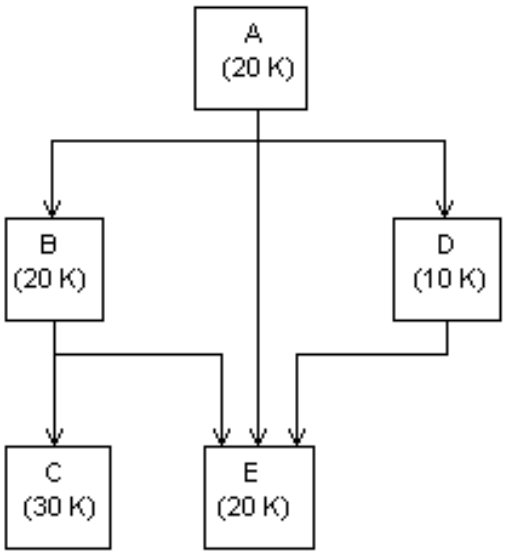
Dynamic loading is also called load-on-call. This is done to save memory. If all the subroutines are loaded simultaneously, a lot of space is taken up. But only one is used at a time. So, here, only the required subroutines are loaded. To identify the call sequence, we use a data structure called OVERLAY STRUCTURE. It defines mutually exclusive subroutines. So, only the ones needed are loaded and a lot of memory is saved. In order for the overlay to work, it is necessary for the module loader to load the various subroutines as they are needed.

Eg:



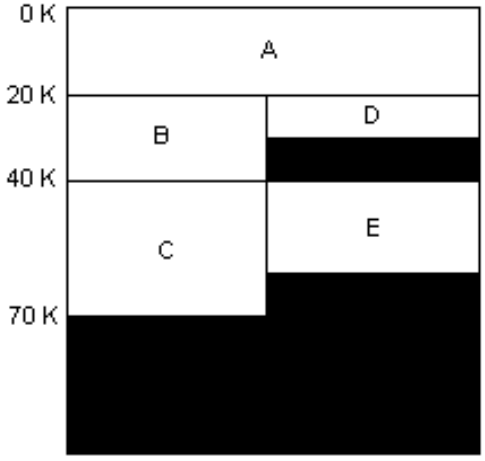
The figure illustrate a program consisting of five subprograms A,B,C,D,E that require 100 bytes of core. The arrow indicates that the subprogram A only calls B, D and E does not call any other subroutines.

The overlay structure will be:



The figure high lights the inter dependencies between procedures. The procedure B and D are never in use at the same time and procedure C and E also. If we load only those procedures that are actually be used at any particular time. The amount of memory needed is equal to the longest path of the overlay structure.

So, probable allocation is like:



The figure illustrates a storage assignment for each procedure consistent with the overlay structure. This over all scheme is called dynamic binding or load-on-call.

Linking

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

Object Module

The object module of a program contains all information necessary to relocate and link the program with other programs. The object module of a program P consists of 4 components:

1. Header: The header contains translated origin, size and execution start address of P.
2. Program: This component contains the machine language program corresponding to P.
3. Relocation table (RELOCTAB) This table describes IRRp. Each RELOCTAB entry contains a single field.
   Translated address: Translated address of an address sensitive instruction.
4. Linking table (LINKTAB) This table contains information concerning the public definitions and external references in P.
   Each LINKTAB entry contains three fields:

Symbol: Symbol name
Type: PD/EXT indicating whether public definition or external reference.
Translated address: For a public definition, this is the address of the first memory word allocated to the symbol. For an external reference, it is the address of the memory word which is required to contain the address of the symbol.

```
Statement                Address          Code
        START   500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A                500)     + 09 0 540

LOOP    |                        501)
        |
        MOVER   AREG, ALPHA      518)     + 04 1 000
        BC      ANY, MAX         519)     + 06 6 000
        |
        |
        BC      LT, LOOP         538)     + 06 1 501
        STOP                     539)     + 00 0 000

A       DS      1                540)     + 00 0 000
TOTAL   DS      1                541)
        END
```

Header

Translated origin = 500, size =42, execution start address = 500.

Relocation table:

500
538

Linking table:

ALPHA	EXT	518
MAX	EXT	519
TOTAL	PD	541

Relocation Algorithm

1) Program_linked_origin:= <link origin> from linker command
2) For each object module
   a) t_origin = translated origin of object module
      OM_Size = size of the object module
   b) Relocation factor = program_link_origin – t_origin
   c) Read the machine language code of the program in work area.
   d) Read the RELOCTAB of the object module.
   e) For each entry in RELOCTAB

i) translated address = address by translator
ii) address_in_work_area = address_of_work_area+ translated address – t_origin
iii) Add relocation factor with the address of the operand in the memory word containing the instruction with the address, address_in _work_area.

   f) Program_linked_origin = program_linked_origin + OM_Size.

Algorithm for Program Linking

1) Program_linked_origin:= <link origin> from linker command
2) For each object module
   a) t_origin = translated origin of object module
      OM_Size = size of the object module
   b) Relocation factor = program_link_origin – t_origin
   c) Read the machine language code of the program in work area.
   d) Read the LINKTAB of the object module
   e) For each LINKTAB entry with type=PD
      name: =symbol;
      linked_ address: =translated _address+relocation_factor;
      Enter (name, linked_address) in NTAB
   f) program_linked_origin = program_linked_origin + OM_Size.

3) For each object module
   a)t_origin = translated origin of object module;
      Program_linked_origin:=linked_address from NTAB;
   b) For each LINKTAB entry with type= EXT

i) address_in_work_area = address_of_work_area + program_linked_origin-<link origin> + translated address – t_origin;

ii) Search symbol in NTAB and copy its linked address. Add linked address to the operand address in the word with the address, address_in _work_area.

<u>Linking Requirements</u>

References to built in functions require linking. A name table (NAMTAB) is defined for use in program linking. Each entry of the table contains the following fields.

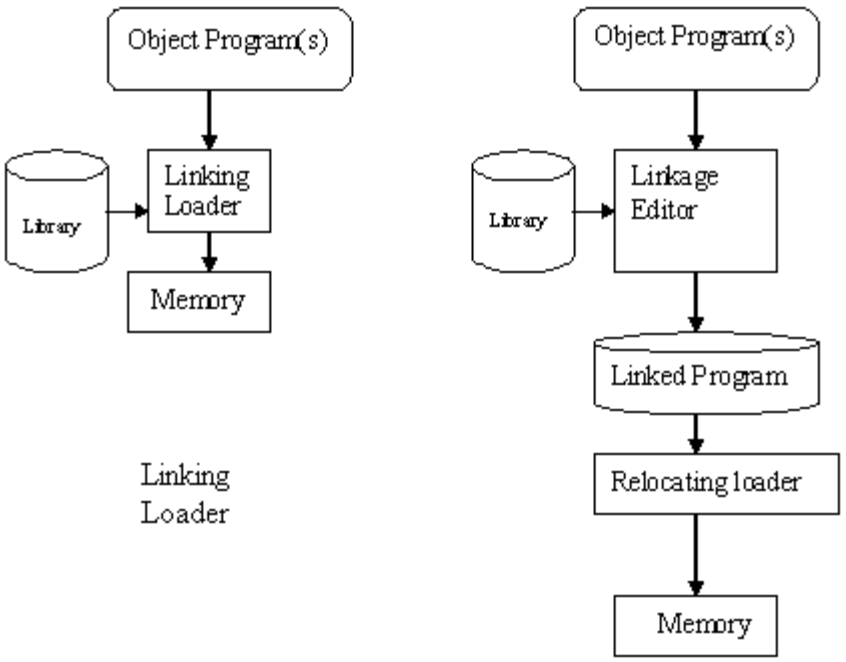
Symbol: symbolic name of an external reference or an object module.
Linked address: For a public definition this field contains linked address of the symbol. For an object module, it contains the linked origin of the object module.
Public definition: A symbol P defined in a program unit which may be referenced in other program units.
External reference: A reference to a symbol E which is not defined in the program unit containing the reference.

Linkage Editors

Linkage editors perform linking operations before the program is loaded for execution. There is a separate editor, which performs all linking operations. Linking loaders perform these same operations at load time.

It is an editor, which is used to do the linking operation before the load time. This is a separate editor it does all the operation of a linker before the load time, not at the runtime. The linkage editor receives the various object modules and some library function as input. It does the linking of the object modules and libraries as a linker. This is not directly loaded into the main memory. Instead this linked program is stored in a file or library.

The essential difference b/w a linkage editor and linkage loader is illustrated in this figure



The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and relocation operations, including automatic library search if specified loads the linked program directly into memory for execution. A linkage editor, on the other hand, produces a linked version of the program which is written to a file or library for later execution.

When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory.

Comparison between linker and linkage editor

Linker	Linkage Editor
1. Linking of object modules and necessary libraries are done and immediately loaded into main memory.	1. Linking of object modules and necessary libraries are done and stored in a file or library. The output is called linked program.
2. Only once it can be used.	2. Many number of times it can be used.
3. Library search and resolution of external reference must be done each time once.	3. Library search and resolution of external reference must be done only once
4. This type of linking is not suitable for program, which is executed repeatedly	4. This is very much suitable for program, which is executed repeatedly.

Dynamic Linking

A major disadvantage of all the loading schemes is that if a subroutine is referenced but never executed, the loader would still incur the overhead of linking the subroutine.

Dynamic linking is a mechanism, by which loading and linking of external references are pospond until execution time. If the program to be executed is larger in size than the available memory in the system, then all of its modules can't be together loaded into the main memory. So the program is divided into segments or pages. Only needed segments or pages will alone be linked into main memory. After the execution of these segments or pages, another set of segments or pages will be linked and loaded into the main memory. This is called dynamic linking and loading.