

UNIT-4

LOADERS AND LINKERS

Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

- **Loading** - which allocates memory location and brings the object program into memory for execution - (Loader)
- **Linking**- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)
- **Relocation** - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

4.1 Basic Loader Functions:

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 4.1. Translator may be assembler/compiler, which generates the object program and later loaded to the memory by the loader for execution. In figure 4.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure4.3 shows the role of both loader and linker.

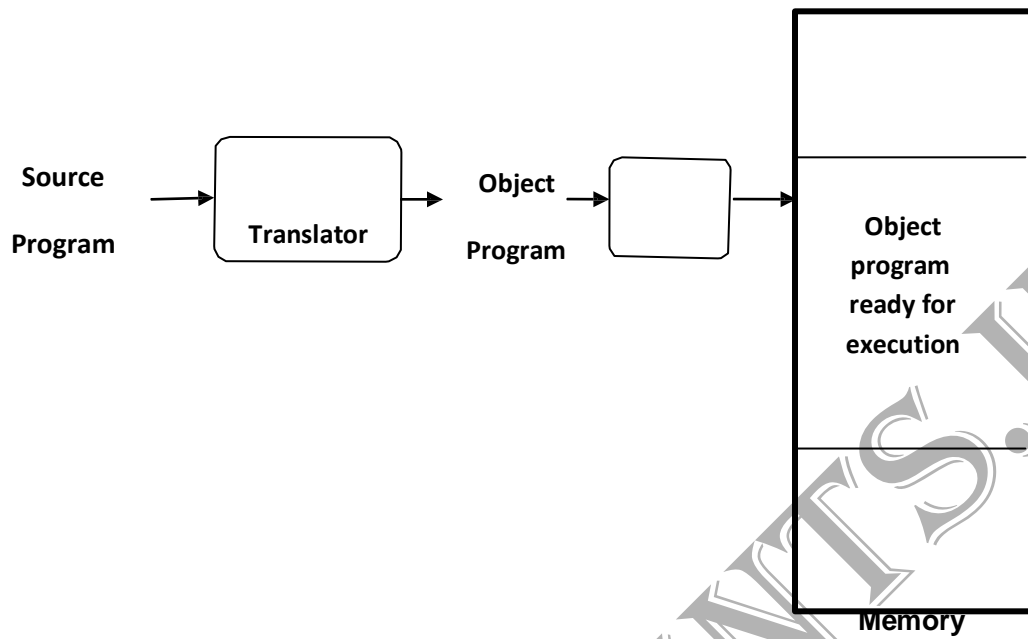


Figure 4.1 : The Role of Loader

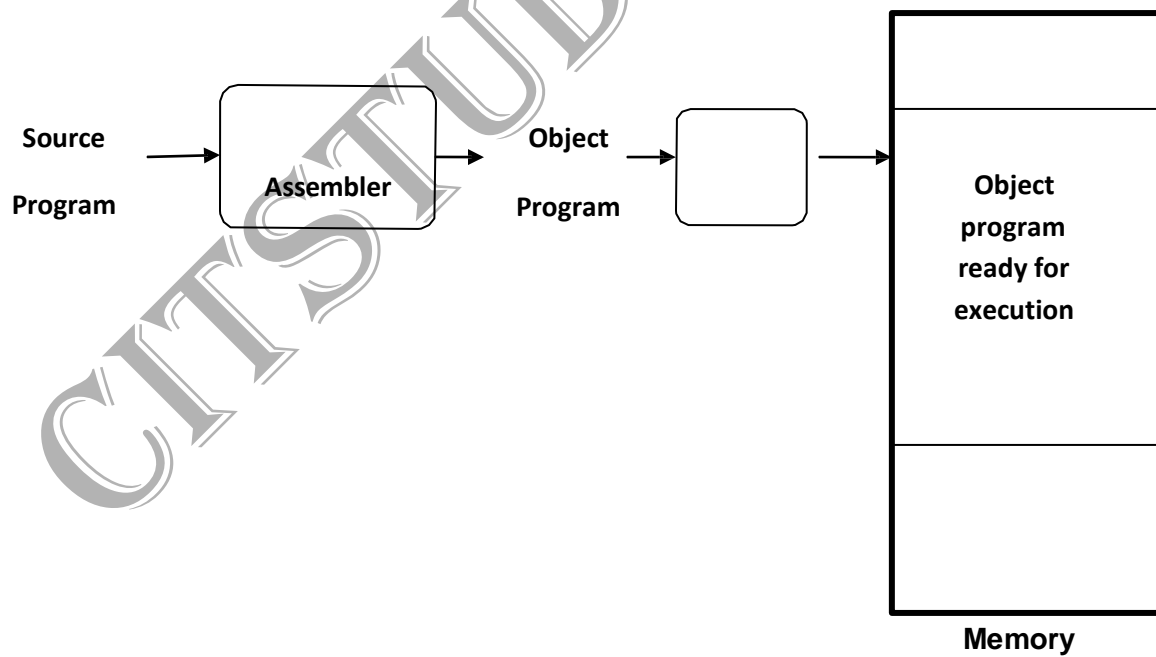


Figure 4.2: The Role of Loader with Assembler

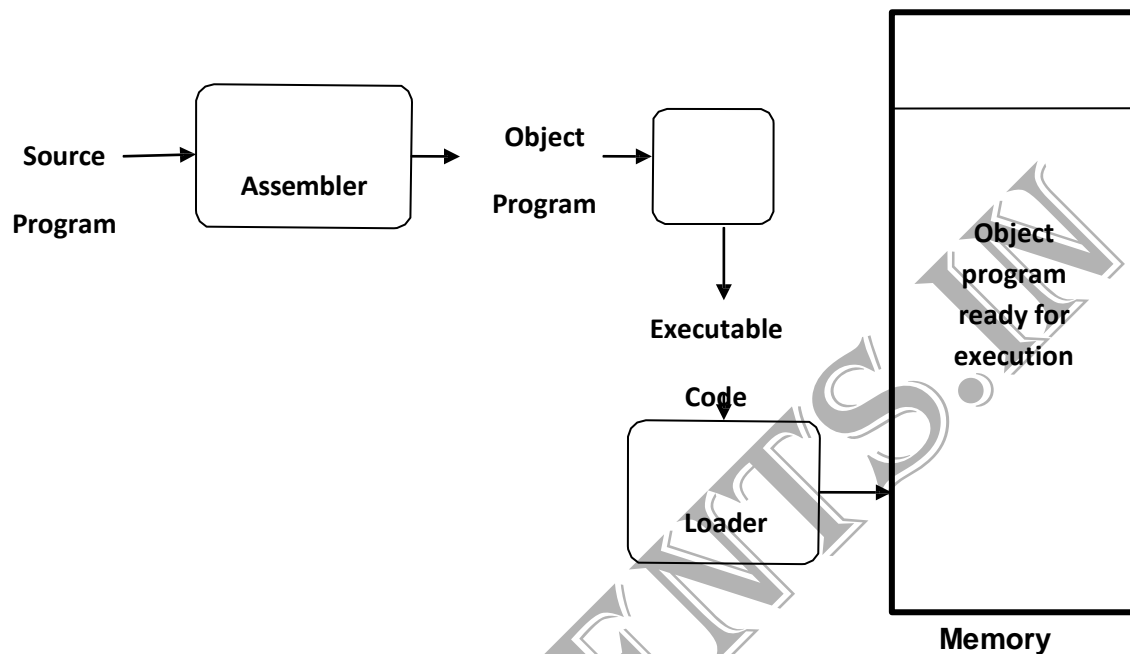


Figure 4.3: The Role of both Loader and Linker

Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

4.1.1 Design of Absolute Loader:

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure 4.4.

The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

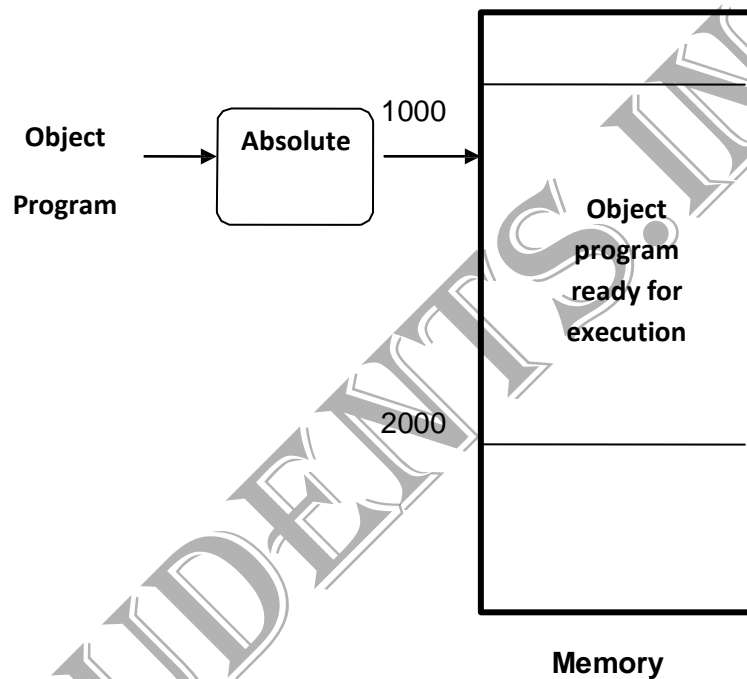


Figure 4.4: The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

Begin

read Header record

verify program name and length

read first Text record

while record type is \neq 'E' **do**

begin

{ if object code is in character form, convert into internal representation }

move object code to specified location in memory

read next object program record

end

jump to address specified in End record

end

```

HCOPY  CC10C000107A
T0010001E1410334820390010362810303010154820613C100300102A0C103900102D
T00101E150C10364820610810334C0000454F46CC0003000000
T0020391E041030001030E0205030203ED8205D2810303020575490392C205E38203F
T0020571C1010364C0000F1001000041030E02079302064509039DC20792C1036
T002073073820644C000005
E001000

```

(a) Object program

Memory address	Contents			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮

(b) Program loaded in memory

4.1.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin

X=0x80 (the address of the next memory location to be loaded)

Loop

$A \leftarrow \text{GETC}$ (and convert it from the ASCII character

code to the value of the hexadecimal digit)

save the value in the high-order 4 bits of S

$A \leftarrow \text{GETC}$

combine the value to form one byte $A \leftarrow (A+S)$

store the value (in A) to the address in register X

$X \leftarrow X+1$

End

It uses a subroutine GETC, which is

GETC $A \leftarrow \text{read one character}$

if $A=0x04$ then jump to 0x80

if $A < 48$ then GETC

$A \leftarrow A-48$ (0x30)

if $A < 10$ then return

$A \leftarrow A-7$

return

4.2. Machine-Dependent Loader Features

Absolute loader is simple and efficient, but the scheme has potential disadvantages. One of the most disadvantages is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.

This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions.

4.2.1 Relocation

The concept of program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.

Methods for specifying relocation

Use of modification record and, use of relocation bit, are the methods available for specifying relocation. In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification record

col 1: M
col 2-7: relocation address
col 8-9: length (halfbyte)
col 10: flag (+/-)
col 11-17: segment name

H_ΔCOPY Δ000000 001077

T_Δ000000 Δ1D_Δ17202D_Δ69202D_Δ48101036_Δ..._Δ4B105D_Δ3F2FEC_Δ032010

T_Δ00001D_Δ13_Δ0F2016_Δ010003_Δ0F200D_Δ4B10105D_Δ3E2003_Δ454F46

T_Δ001035 Δ1D_ΔB410_ΔB400_ΔB440_Δ75101000_Δ..._Δ332008_Δ57C003_ΔB850

T_Δ001053_Δ1D_Δ3B2FEA_Δ134000_Δ4F0000_ΔF1_Δ..._Δ53C003_ΔDF2008_ΔB850

T_Δ00070_Δ07_Δ3B2FEF_Δ4F0000_Δ05

M_Δ000007_Δ05+COPY

M_Δ000014_Δ05+COPY

M_Δ000027_Δ05+COPY

E_Δ000000

The relocation bit method is used for simple machines. Relocation bit is 0: no modification is necessary, and is 1: modification is needed. This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record:

col 1: T

col 2-7: starting address

col 8-9: length (byte)

col 10-12: relocation bits

col 13-72: object code

Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be

modified during relocation must coincide with one of these 3-byte segments. For absolute loader, there are no relocation bits column 10-69 contains object code. The object program with relocation by bit mask is as shown below. Observe FFC - means all ten words are to be modified and, E00 - means first three records are to be modified.

H_ΛCOPY _Λ000000 00107A

T_Λ000000_Λ1E_ΛFFC_Λ140033_Λ481039_Λ000036_Λ280030_Λ300015_Λ..._Λ3C0003 _Λ...

T_Λ00001E_Λ15_ΛE00_Λ0C0036_Λ481061_Λ080033_Λ4C0000_Λ..._Λ000003_Λ000000

T_Λ001039_Λ1E_ΛFFC_Λ040030_Λ000030_Λ..._Λ30103F_ΛD8105D_Λ280030_Λ...

T_Λ001057_Λ0A_Λ800_Λ100036_Λ4C0000_ΛF1_Λ001000

T_Λ001061_Λ19_ΛFE0_Λ040030_ΛE01079_Λ..._Λ508039_ΛDC1079_Λ2C0036_Λ...

E_Λ000000

4.2.2 Program Linking

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: EXTDEF BUFFER, BUFFEND, LENGTH

EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC

EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record

The format of the Define record (D) along with examples is as shown here.

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address within this control section (hexadecimal)
Col.14-73	Repeat information in Col. 2-13 for other external symbols

Example records

D LISTA 000040 ENDA 000054

D LISTB 000060 ENDB 000070

Refer record

The format of the Refer record (R) along with examples is as shown here.

Col. 1	R
Col. 2-7	Name of external symbol referred to in this control section
Col. 8-73	Name of other external reference symbols

Example records

R LISTB ENDB LISTC ENDC

R LISTA ENDA LISTC ENDC

R LISTA ENDA LISTB ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs given here are used to illustrate linking and relocation. The following figures give the sample programs and their corresponding object programs. Observe the object programs, which contain D and R records along with other records.

```

0000  PROGA    START          0

                                EXTDEF    LISTA, ENDA

                                EXTREF     LISTB, ENDB, LISTC, ENDC

                                .....

                                .....

0020  REF1      LDA           LISTA           03201D

0023  REF2      +LDT          LISTB+4         77100004

0027  REF3      LDX           #ENDA-LISTA     050014

                                .

                                .

0040  LISTA     EQU           *

                                .

                                .

0054  ENDA      EQU           *

                                .

0054  REF4      WORD          ENDA-LISTA+LISTC 000014

0057  REF5      WORD          ENDC-LISTC-10    FFFFF6

```

005A	REF6	WORD	ENDC-LISTC+LISTA-1	00003F
005D	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000014
0060	REF8	WORD	LISTB-LISTA	FFFFC0
		END	REF1	
0000	PROGB	START	0	
		EXTDEF	LISTB, ENDB	
		EXTREF	LISTA, ENDA, LISTC, ENDC	
			
			
0036	REF1	+LDA	LISTA	03100000
003A	REF2	LDT	LISTB+4	772027
003D	REF3	+LDX	#ENDA-LISTA	05100000
0060	LISTB	EQU	*	
0070	ENDB	EQU	*	
0070	REF4	WORD	ENDA-LISTA+LISTC	000000
0073	REF5	WORD	ENDC-LISTC-10	FFFFFF6
0076	REF6	WORD	ENDC-LISTC+LISTA-1	FFFFFFF

0079	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	FFFFFF0
007C	REF8	WORD	LISTB-LISTA	000060
		END		
0000	PROGC	START	0	
		EXTDEF	LISTC, ENDC	
		EXTREF	LISTA, ENDA, LISTB, ENDB	
			
			
0018	REF1	+LDA	LISTA	03100000
001C	REF2	+LDT	LISTB+4	77100004
0020	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
0030	LISTC	EQU	*	
0042	ENDC	EQU	*	
0042	REF4	WORD	ENDA-LISTA+LISTC	000030
0045	REF5	WORD	ENDC-LISTC-10	000008
0045	REF6	WORD	ENDC-LISTC+LISTA-1	000011
004B	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000000

004E REF8 WORD LISTB-LISTA 000000
END

H PROGA 000000 000063

D LISTA 000040 ENDA 000054

R LISTB ENDB LISTC ENDC

.

.

T 000020 0A 03201D 77100004 050014

.

.

T 000054 0F 000014 FFFF6 00003F 000014 FFFFC0

M000024 05+LISTB

M000054 06+LISTC

M000057 06+ENDC

M000057 06-LISTC

M00005A06+ENDC

M00005A06-LISTC

M00005A06+PROGA

M00005D06-ENDB

M00005D06+LISTB

M00006006+LISTB

M00006006-PROGA

E000020

H **PROGB** 000000 00007F

D LISTB 000060 ENDB 000070

R LISTA ENDA LISTC ENDC

.

T 000036 0B 03100000 772027 05100000

.

T 000007 0F 000000 FFFFF6 FFFFFF FFFFF0 000060

M000037 05+LISTA

M00003E 06+ENDA

M00003E 06 -LISTA

M000070 06 +ENDA

M000070 06 -LISTA

M000070 06 +LISTC

M000073 06 +ENDC

M000073 06 -LISTC

M000073 06 +ENDC

M000076 06 -LISTC

M000076 06+LISTA

M000079 06+ENDA

M000079 06 -LISTA

M00007C 06+PROGB

M00007C 06-LISTA

E

H **PROGC** 000000 000051

D LISTC 000030 ENDC 000042

R LISTA ENDA LISTB ENDB

.

T 000018 0C 03100000 77100004 05100000

.

T 000042 0F 000030 000008 000011 000000 000000

M000019 05+LISTA

M00001D 06+LISTB

M000021 06+ENDA

M000021 06 -LISTA

M000042 06+ENDA

M000042 06 -LISTA

M000042 06+PROGC

M000048 06+LISTA

M00004B 06+ENDA

M00004B 006-LISTA

M00004B 06-ENDB

M00004B 06+LISTB

M00004E 06+LISTB

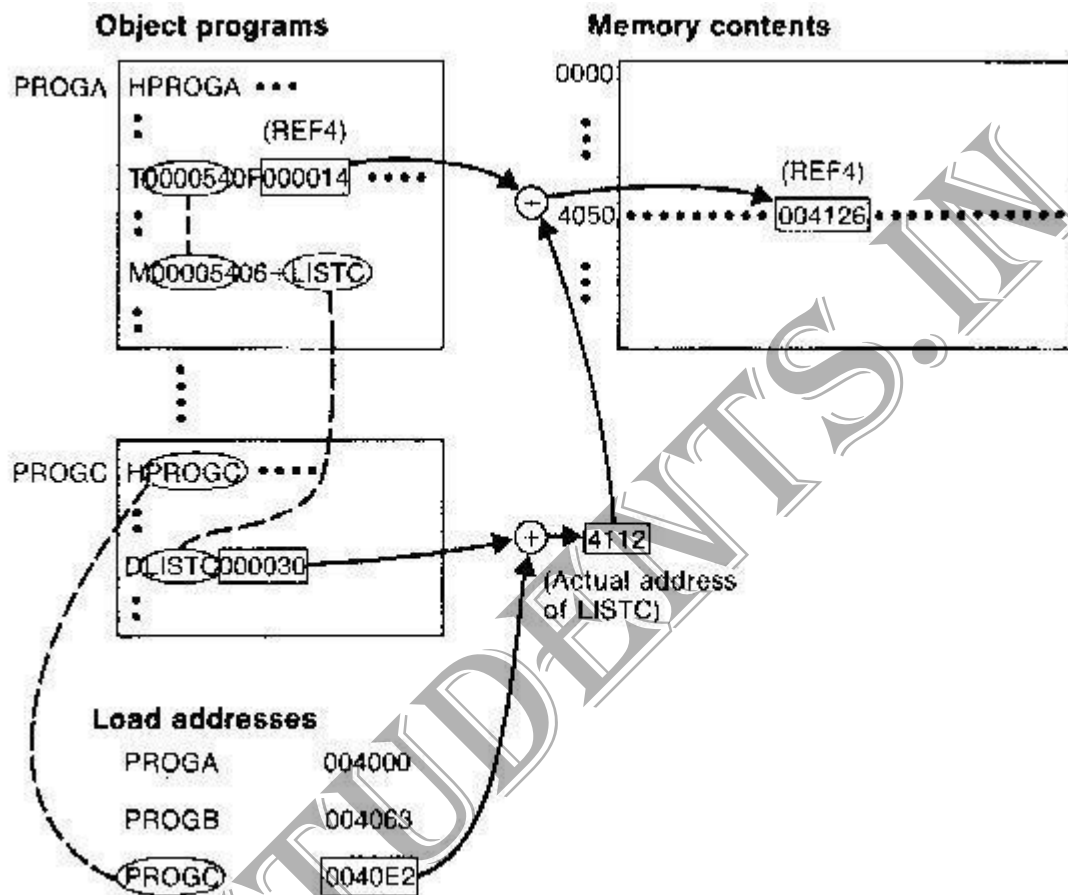
M00004E 06-LISTA

E

The following figure shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.

Memory address	Contents			
0000	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
⋮	⋮	⋮	⋮	⋮
3FF0	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
4000
4010
4020	03201077	1040C705	0014..... ← PROGA
4030
4040
4050	00412600	00080040	51000004
4060	000083.....
4070
4080
4090031040	40772027 ← PROGB
40A0	05100014
40B0
40C0
40D000	41260000	08004051	00000400
40E0	0083.....
40F00310	40407710 ← PROGC
4100	40C70510	0014.....
4110
4120	00412600	00080040	51000004
4130	000083xx	XXXXXXXX	XXXXXXXX	XXXXXXXX
4140	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
⋮	⋮	⋮	⋮	⋮

For example, the value for REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). The following figure shows the details of how this value is computed.



The initial value from the Text record

T0000540F000014FFFFFF600003F000014FFFC0 is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). The result is 004126.

That is REF4 in PROGA is $ENDA-LISTA+LISTC=4054-4040+4112=4126$.

Similarly the load address for symbols LISTA: $PROGA+0040=4040$, LISTB: $PROGB+0060=40C3$ and LISTC: $PROGC+0030=4112$

Keeping these details work through the details of other references and values of these references are the same in each of the three programs.

4.3.3 Algorithm and Data structures for a Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program, which is already given. The concept given in the program linking section is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism.

Linking Loader uses two-passes logic. ESTAB (external symbol table) is the main data structure for a linking loader.

Pass 1: Assign addresses to all external symbols

Pass 2: Perform the actual loading, relocation, and linking

ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.

Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

Program Logic for Pass 1

Pass 1 assign addresses to all external symbols. The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR (control section address), CSLTH (control section length) and ESTAB. The pass 1 processes the Define Record. The algorithm for Pass 1 of Linking Loader is given below.

Pass 1:

```

begin
  get PROGADDR from operating system
  set CSADDR to PROGADDR {for first control section}
  while not end of input do
    begin
      read next input record {Header record for control section}
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
        set error flag {duplicate external symbol}
      else
        enter control section name into ESTAB with value CSADDR
      while record type ( ) 'E' do
        begin
          read next input record
          if record type = 'D' then
            for each symbol in the record do
              begin
                search ESTAB for symbol name
                if found then
                  set error flag {duplicate external symbol}
                else
                  enter symbol into ESTAB with value
                    (CSADDR + indicated address)
              end {for}
            end {while ( ) 'E'}
          add CSLTH to CSADDR {starting address for next control section}
        and {while not EOF}
      end {Pass 1}
    end {while not end of input}
  end

```

Program Logic for Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its address. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs. The algorithm for Pass 2 of Linking Loader is given below.

Pass 2:

```

begin
  set CSADDR to PROGADDR
  set EXECADDR to PROGADDR
  while not end of input do
    begin
      read next input record {Header record}
      set CSLTH to control section length
      while record type () 'E' do
        begin
          read next input record
          if record type = 'T' then
            begin
              {if object code is in character form, convert
               into internal representation}
              move object code from record to location
              (CSADDR + specified address)
            end {if 'T'}
          else if record type = 'M' then
            begin
              search ESTAB for modifying symbol name
              if found then
                add or subtract symbol value at location
                (CSADDR + specified address)
              else
                set error flag (undefined external symbol)
              end {if 'M'}
            end {while () 'E'}
          if an address is specified (in End record) then
            set EXECADDR to (CSADDR + specified address)
          add CSLTH to CSADDR
        end {while not EOF}
      jump to location given by EXECADDR {to start execution of loaded program}
    end {Pass 2}
  
```

How to improve Efficiency?

The question here is can we improve the efficiency of the linking loader. Also observe that, even though we have defined Refer record (R), we haven't made use of it. The efficiency can be improved by the use of local searching instead of multiple searches of ESTAB for the same symbol. For implementing this we assign a reference number to each external symbol in the Refer record. Then this reference number is used in Modification records instead of external symbols. 01 is assigned to control section name, and other numbers for external reference symbols.

The object programs for PROGA, PROGB and PROGC are shown below, with above modification to Refer record (Observe R records).

H)R0CB P.00000..0007P

1ST 0006 ENOB p00070
LISTA 21ENDA LJSC F.NDC

:

T"UOU03B"U 0310000..7 /202 "O.SI 00000

:

T0000700E000000FFFFF6FFFFF6FFFFF00000Q60

M00003705+02

M00003E05+03

M00003E05-02

M00007006+03

M00007006-02

M00007006+04

M00007306+05

M00007306-04

M00007606+05

M00007606-04

M00007606+02

M00007906+03

M00007906-02

M00007C06+01

M00007C06-02

E

HPRGA 000000000063

DLISTA 000040ENDA 000054

R02LISTA 03ENDB 04LISTC 05ENDC

:

"!JOG02 QA,0 3201D,77 100004.05011

:

"f,00GOJ HH'OvC0 14 FPF1-F60CI00::>0000 14FFFC0

Y.00GO2405+02

Y.0000.1 06+04

ll u(uji'ov+J5

ri);o liO>fo6_04


```

HPRGCG 0000000000051
DLISTC 000030ENDC 000042
EO2LISTA 03ENDA 04LISTB 05ENDB
:
:
T0000180C031000007710000405100000
:
:
T0000420F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E

```

Symbol and Addresses in PROGA, PROGB and PROGC are as shown below. These are the entries of ESTAB. The main advantage of reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section

Ref No.	Symbol	Address
1	PROGA	4000
2	LISTB	40C3
3	ENDB	40D3
4	LISTC	4112
5	ENDC	4124

Ref No.	Symbol	Address
1	PROGB	4063
2	LISTA	4040
3	ENDA	4054
4	LISTC	4112
5	ENDC	4124

Ref No.	Symbol	Address
1	PROGC	4063
2	LISTA	4040
3	ENDA	4054
4	LISTB	40C3
5	ENDB	40D3

4.3. Machine-independent Loader Features

Here we discuss some loader features that are not directly related to machine architecture and design. Automatic Library Search and Loader Options are such Machine-independent Loader Features.

4.3.1 Automatic Library Search

This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking. This allows programmer to use subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program.

4.3.2 Loader Options

Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways. They are, specified using a command language, specified as a part of job control language that is processed by the operating system, and can be specified using loader control statements in the source program.

Here are the some examples of how option can be specified.

- `INCLUDE program-name (library-name)` - read the designated object program from a library

- DELETE csect-name – delete the named control section from the set of programs being loaded
- CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs

LIBRARY MYLIB – search MYLIB library before standard libraries

NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines

Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

LIBRARY UTLIB

INCLUDE READ (UTLIB)

INCLUDE WRITE (UTLIB)

DELETE RDREC, WRREC

CHANGE RDREC, READ

CHANGE WRREC, WRITE

NOCALL SQRT, PLOT

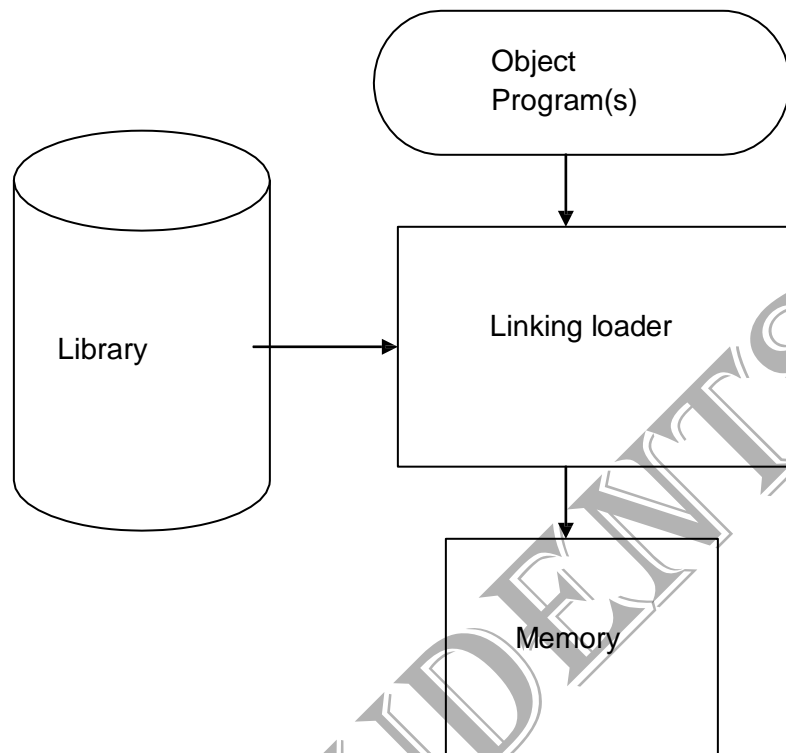
The commands are, use UTLIB (say utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol RDREC to be changed to the symbol READ, similarly references to WRREC is changed to WRITE, finally, no call to the functions SQRT, PLOT, if they are used in the program.

4.4 Loader Design Options

There are some common alternatives for organizing the loading functions, including relocation and linking. Linking Loaders – Perform all linking and relocation at load time. The

Other Alternatives are Linkage editors, which perform linking prior to load time and, dynamic linking, in which linking function is performed at execution time

Linking Loaders

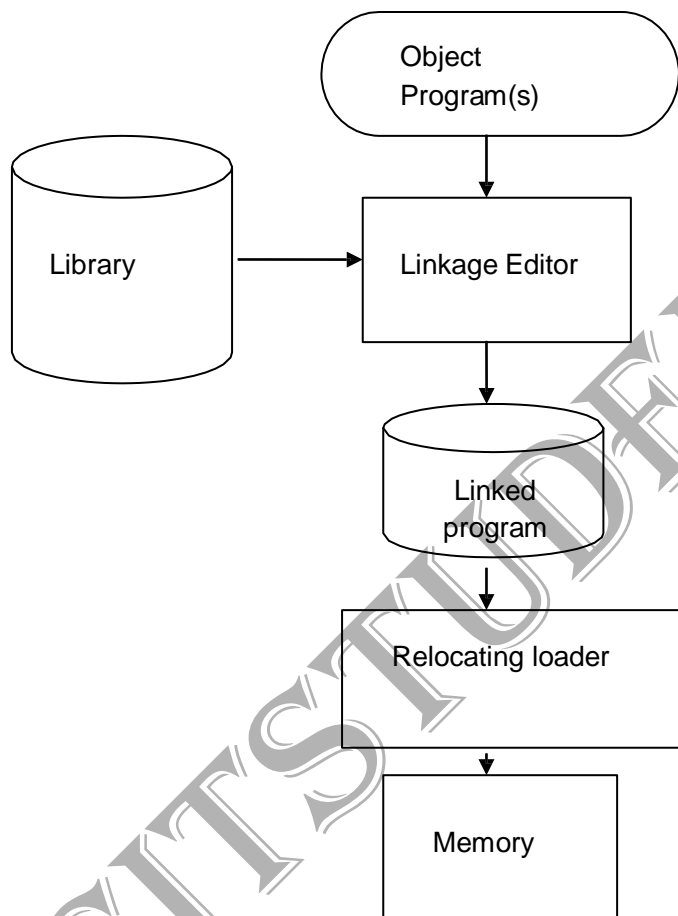


The above diagram shows the processing of an object program using Linking Loader. The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and loading operations, and loads the program into memory for execution.

4.4.1 Linkage Editors

The figure below shows the processing of an object program using Linkage editor. A linkage editor produces a linked version of the program – often called a load module or an executable image – which is written to a file or library for later execution. The linked program produced is generally in a form that is suitable for processing by a relocating loader.

Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known. New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space



4.4.2 Dynamic Linking

The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system,

dynamic linking makes it possible for one object to be shared by several programs. Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request.

4.4.3 Bootstrap Loaders

If the question, how is the loader itself loaded into the memory? is asked, then the answer is, when computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or A Bootstrap loader, which in turn loads OS and prepares it for execution. The first record (or records) is generally referred to as a bootstrap loader – makes the OS to be loaded. Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.

4.5 Implementation Examples

This section contains brief description of loaders and linkers for actual computers. They are, MS-DOS Linker - Pentium architecture, SunOS Linkers - SPARC architecture, and, Cray MPP Linkers – T3E architecture.

4.5.1 MS-DOS Linker

This explains some of the features of Microsoft MS-DOS linker, which is a linker for Pentium and other x86 systems. Most MS-DOS compilers and assemblers (MASM) produce object modules, and they are stored in .OBJ files. MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program - .EXE file; this file is later executed for results.

The following table illustrates the typical MS-DOS object module

Record Types	Description
THEADR	Translator Header
TYPDEF,PUBDEF, EXTDEF	External symbols and references
LNAMES, SEGDEF, GRPDEF	Segment definition and grouping

LEDATA, LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information
MODEND	End of object module

THEADR specifies the name of the object module. MODEND specifies the end of the module. PUBDEF contains list of the external symbols (called public names). EXTDEF contains list of external symbols referred in this module, but defined elsewhere. TYPDEF the data types are defined here. SEGDEF describes segments in the object module (includes name, length, and alignment). GRPDEF includes how segments are combined into groups. LNAMEs contains all segment and class names. LEDATA contains translated instructions and data. LIDATA has above in repeating pattern. Finally, FIXUPP is used to resolve external references.

RECOMMENDED QUESTIONS:

- 1) Write an algorithm for an absolute loader. (7)
- 2) Explain bootstrap loaders. (6)
- 3) Write an algorithm for Bootstrap loader. (7)
- 4) Explain relocation w.r.t. loader. (8)
- 5) Explain bitmask with an example.(5)
- 6) Explain program linking with an example. (7)
- 7) Write the algorithm for pass 1 of an linking loader. (8)
- 8) Write the algorithm for pass 2 of an linking loader. (8)
- 9) Explain CSADDR, PROGADDR, ESTAB.(6)
- 10) Explain linkage editors. (8)
- 11) Explain dynamic linking. (8)
- 12) Write shortnotes on (10)
 - a. MS-DOS Linker
 - b. Sun OS linker