BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 5

Loaders

Table of Contents

1. INTRODUCTION

A loader is the part of an operating system responsible for loading programs. It is one of the essential stages in starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file, the file containing the program text, into memory and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

In this chapter, we'll talk about a loader's fundamental duties before moving on to its machine-dependent and machine-independent characteristics. In the last section of this unit, we are discussing the loader design options.

1.1 Learning Objectives:

After studying this unit, learneres should be able to:

- *Define a Loader.*
- *Describe the basic functions of the loader.*
- *Explain the machine dependent loader feature.*
- *Describe different loader design options.*

2. BASIC FUNCTIONS OF LOADER

The first step in developing any machine code manipulation program is understanding the source object file format – what type of information is stored, how it is stored, and how it can be accessed. Normally we are not concerned with any of these issues since the operating system (OS) automatically handles all file access operations and provides users with an interface to these functions. When a program needs to be executed, the OS first extracts all relevant file information (usually from the file header), and carries out any necessary actions before putting it into memory; in other words, the OS decodes the object file into an understandable form in memory. This behavior is often described as the loading of a program.

An operating system utility is one that copies programs from a storage device to the main memory, where they can be executed. In addition to copying a program into the main memory, the loader can replace virtual addresses with physical addresses. Most loaders are transparent; that is, we cannot directly execute them, but the operating system uses them when necessary.

A binary object file is either an executable file that runs on a particular machine or contains object code that needs to be linked. The object code (or executable code) is generated by compiler or an assembler.

Three steps are required to run an object program.. They are Relocation, Linking, Loading, and allocation. Relocation modifies the object programs so that they can be loaded at an address different from the location originally specified. Linking combines two or more separate object programs and supplies the information needed to allow references between them. Loading and allocation allocate memory location and bring the object program into the memory for execution.

Fundamental functions of a loader:

1. Bringing an object program into memory.
2. Starting its execution.

3. DESIGN OF AN ABSOLUTE LOADER

For a simple absolute loader, all functions are accomplished in a single pass as follows:

1) *The Header record* of object programs is checked to verify that the correct program has been presented for loading.
2) As each *Text record* is read, the object code it contains is moved to the indicated address in memory.
3) When the *End record* is encountered, the loader jumps to the specified address to begin the execution of the loaded program.

An example object program is shown in Figure 5.1(a) and figure 5.1(b) shows how this object program is stored in memory

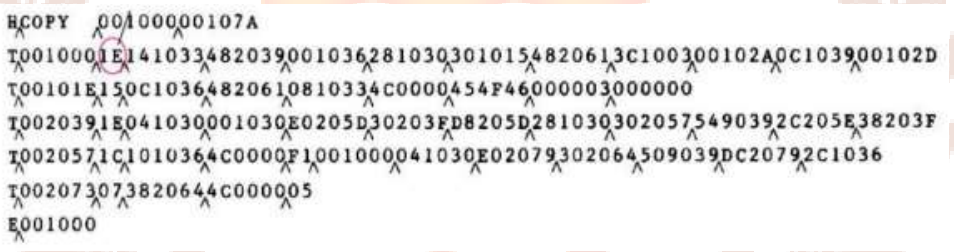
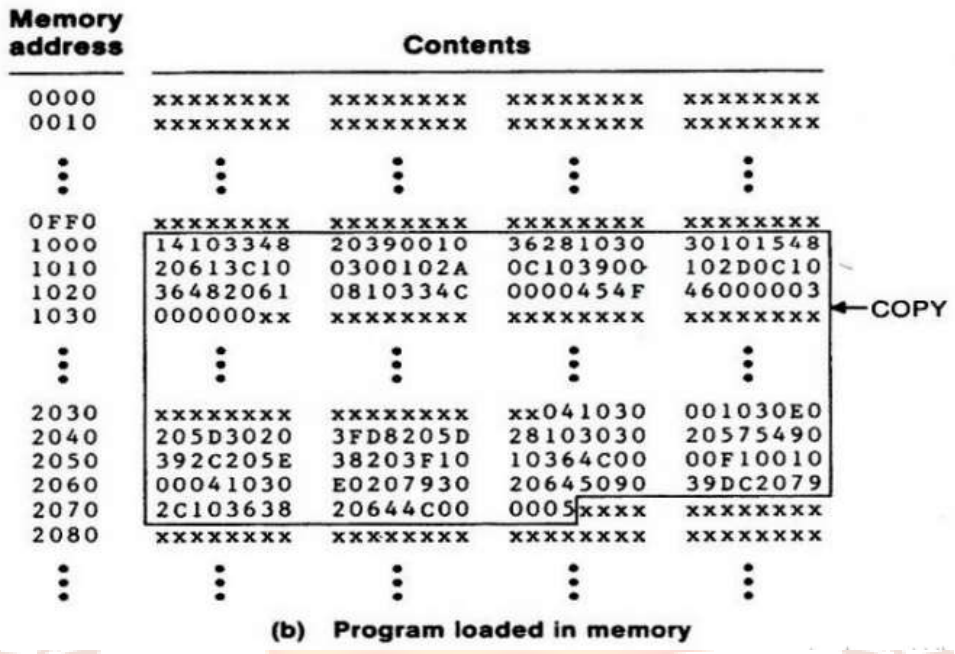


Figure 5.1 (a): An example object program

- It is very important to realize that in Fig 5.1 (a), each printed character represents one byte of the object program record.
- In Fig 5.1(b), on the other hand, each printed character represents one hexadecimal digit in memory (a half-byte).
- Therefore, to save space and execution time of loaders, most machines store object programs in a binary form, with each byte of object code stored as a single byte in the object program.



(b) Program loaded in memory

- In this type of representation, a byte may contain any binary value.

Figure 5.1(b): Program loaded in memory

Algorithm for Absolute Loader:

Figure 5.2 shows the algorithm for the absolute loader.

Algorithm for an absolute loader

```
begin
  read Header record
  verify program name and length
  read first Text record
  while record type is ≠ '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
```

Fig. 5.2: Algorithm for Absolute loader

4. A SIMPLE BOOTSTRAP LOADER

When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader,** is executed. This bootstrap loads the first program to be run by the computer – usually an operating system.

Working on a simple Bootstrap loader

- The bootstrap begins at address 0 in the memory of the machine.
- It loads the operating system at address 80.
- Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits just as it is in a SIC text record (refer to fig. 5.3) object program.
- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.
- After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the loaded program.
- Much of the work of the bootstrap loader is performed by the subroutine GETC (refer to fig. 5.3).
- GETC (refer to fig. 5.3) is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. For example, two bytes = C "D8" ‘4438’H converting to one byte ‘D8’H.
- The resulting byte is stored at the address currently in register X, using STCH (refer to fig. 5.3) instruction that refers to location 0 using indexed addressing.
- The TIXR (refer to fig.5.3) instruction is then used to add 1 to the value in X.

```
GETC    TD      INPUT   TEST INPUT DEVICE
        JEQ     GETC    LOOP UNTIL READY
        RD      INPUT   READ CHARACTER
        COMP    #4      IF CHARACTER IS HEX 04 (END OF FILE),
        JEQ     80          JUMP TO START OF PROGRAM JUST LOADED
        COMP    #48     COMPARE TO HEX 30 (CHARACTER '0')
        JLT     GETC    SKIP CHARACTERS LESS THAN '0'
        SUB     #48     SUBTRACT HEX 30 FROM ASCII CODE
        COMP    #10     IF RESULT IS LESS THAN 10, CONVERSION IS
        JLT     RETURN      COMPLETE. OTHERWISE, SUBTRACT 7 MORE
        SUB     #7          (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN  RSUB            RETURN TO CALLER
INPUT   BYTE    X'F1'   CODE FOR INPUT DEVICE
        END     LOOP
```

Source code for the bootstrap loader

Figure 5.3: Source code for Bootstrap loader

Self-Assessment Questions - 1

1. The process of bringing the object program into the memory for execution is called _____.
2. When a computer is first turned on or restarted, a special type of absolute loaderis executed, which is called _____ .
3. The first program loaded by Bootstrap to be run by _____

5. MACHINE-DEPENDENT LOADER FEATURE

The absolute loader has several potential disadvantages. One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory. On a simple computer with small memory, the actual address at which the program will be loaded can be specified easily. On a larger and more advanced machine, we often like to run several independent programs together, sharing memory between them. We have yet to determine in advance where a program will be loaded. Hence, we write relocatable programs instead of absolute ones. Writing absolute programs also makes it difficult to use subroutine libraries efficiently. This could not be done effectively if all subroutines had pre-assigned absolute addresses.

The need for *program relocation* indirectly impacts the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics. Loaders that allow for program relocation are called relocating loaders or relative loaders.

5.1 Relocation

Loaders that allow for program relocation are called *relocating loaders* or *relative loaders*. The need for program relocation results from the change to larger and more powerful computers.

Schemes for Relocation:

1) **Modification Record**

   A modification record describes each part of the object code that must be changed when the program is relocated. In the object program, there is one Modification record for each value that must be changed during relocation. Each modification record specifies the starting address and length of the field whose value is to be altered.

A Modification describes each part of the object code that must be changed when the program is relocated.

Consider the program in Figure 5.4.

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	

```
115          .       READ RECORD INTO BUFFER
120          .
125   1036   RDREC    CLEAR  X           B410
130   1038            CLEAR  A           B400
132   103A            CLEAR  S           B440
133   103C           +LDT    #4096       75101000
135   1040   RLOOP    TD     INPUT       E32019
140   1043            JEQ    RLOOP       332FFA
145   1046            RD     INPUT       DB2013
150   1049            COMPR  A,S         A004
155   104B            JEQ    EXIT        332008
160   104E            STCH   BUFFER,X    57C003
165   1051            TIXR   T           B850
170   1053            JLT    RLOOP       3B2FEA
175   1056   EXIT     STX    LENGTH      134000
180   1059            RSUB               4F0000
185   105C   INPUT    BYTE   X'F1'       F1
195          .
200          .       WRITE RECORD FROM BUFFER
205          .
210   105D   WRREC    CLEAR  X           B410
212   105F            LDT    LENGTH      774000
215   1062   WLOOP    TD     OUTPUT      E32011
220   1065            JEQ    WLOOP       332FFA
225   1068            LDCH   BUFFER,X    53C003
230   106B            WD     OUTPUT      DF2008
235   106E            TIXR   T           B850
               ...(omitted)
```

Figure 5.4: First method- Example program.

- Most of the instructions in this program use relative or immediate addressing.
- The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

Object program

Figure 5.5 shows the object program.

```
HCOPY  000000001077
T0000001D17202D69202D4B10103603202629000033200074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T00107007 3B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

Figure 5.5: Object Program

- Each Modification record specifies the starting address and length of the field whose value is to be altered.
- It then describes the modification to be performed.
- In this example, all modifications add the value of the symbol COPY, which represents the program's starting address.

Figure 5.6: Consider a Relocatable program for a Standard SIC machine.

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061
40	0012		J	CLOOP	3C0003
45	0015	ENDFIL	LDA	EOF	00002A
50	0018		STA	BUFFER	0C0039
55	001B		LDA	THREE	00002D
60	001E		STA	LENGTH	0C0036
65	0021		JSUB	WRREC	481061
70	0024		LDL	RETADR	080033
75	0027		RSUB		4C0000
80	002A	EOF	BYTE	C'EOF'	454F46
85	002D	THREE	WORD	3	000003
90	0030	ZERO	WORD	0	000000
95	0033	RETADR	RESW	1	
100	0036	LENGTH	RESW	1	
105	0039	BUFFER	RESB	4096	

Fig. 5.6: Relocatable program for a Standard SIC machine

- The Modification record is not well suited for all machine architectures. Consider, for example, the program in Fig. 5.6. This is a relocatable program written for the standard version of SIC.
- The important difference between this example and the one in Fig. 5.4 is that the standard SIC machine does not use relative addressing.
- In this program, the addresses in all the instructions except RSUB must be modified when the program is relocated. This would require 31 Modification records, which results in an object program more than twice as large as the one shown in Fig. 5.4.

2) Relocation Bit

Each word of object code has a relocation bit that determines whether or not the word should be changed when the programme is relocated. If the relocation bit corresponding to a word of object code is set to 1, the program's starting address will be added to this word when the program is relocated. The bits corresponding to unused words are set to 0, which signifies that no changes are required.

Fig. 5.7 shows an Object program with relocation by a bit mask

- There are no Modification records.
- The Text records are the same as before except for a *relocation bit* associated with each word of object code.
- Since all SIC instructions occupy one word, there is one relocation bit for each possible instruction.

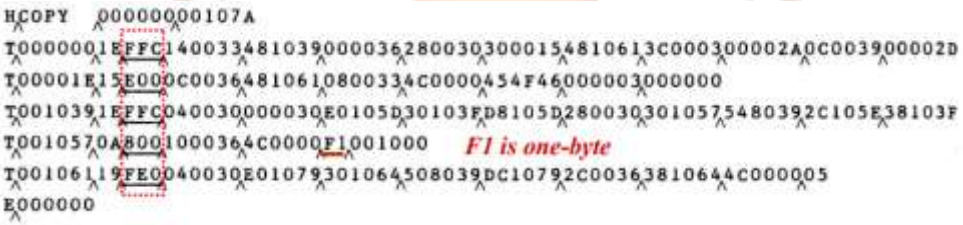


Figure 5.7: Object program with relocation by a bit mask

- The relocation bits are framed into a **bit mask by** following the length indicator in each Text record. This mask is represented(in character form) as three hexadecimal digits in Fig 5.7.
- If the relocation bit corresponding to a word of object code is set to 1, the program's starting address will be added to this word when the program is relocated. A bit value of 0 indicates that no modification is necessary.
- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
- For example, the bit masks FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.

Example: Note that the LDX instruction on line 210 (Fig 5.6) begins a new Text record. If it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the 1-byte data value generated from line 185.

3) Hardware Relocation

Some computers provide a hardware relocation capability that eliminates some of the need for the loader to perform program relocation.

5.2 Program Linking

A single control section makes up each of the three (separately constructed) programmes in figure 5.8.

Program 1 (Program A):

```
Loc       Source statement                          Object code

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
```

Program 2 (Program B):

```
Loc       Source statement                          Object code

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                 FFFFF6
0076  REF6    WORD    ENDC-LISTC+LISTA-1            FFFFFF
0079  REF7    WORD    ENDA-LISTA-(ENDB-LISTB)       FFFFF0
007C  REF8    WORD    LISTB-LISTA                   000060
              END
```

Program 3 (Program C):

```
Loc       Source statement                          Object code

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
0048  REF6   WORD   ENDC-LISTC+LISTA-1              000011
004B  REF7   WORD   ENDA-LISTA-(ENDB-LISTB)        000000
004E  REF8   WORD   LISTB-LISTA                    000000
             END
```

Figure 5.8: Program Linking Example

Consider first the reference marked REF1.

For the first program (Program A),

- REF1 is simply a reference to a label within the program.
- It is assembled in the usual way as a PC relative instruction.
- No modification for relocation or linking is necessary.

In Program B, the same operand refers to an external symbol.

- The assembler uses an extended-format instruction with an address field set to 00000.
- The object program for Program B contains a Modification record instructing the loader to add *the value of the symbol LISTA to this address field* when the program is linked.

For Program C, REF1 is handled in exactly the same way.

Fig 5.9 shows corresponding object programs,

Program A:

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
.
.
T0000200A03201D771000040500 14
.
.
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

Program B:

```
HPROGB 00000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
.
.
T000036 0B03100000772027051000 00
.
.
T000070 0F000000FFFFF6FFFFFFFFFFF0000060
M000037 05+LISTA
M00003E 05+ENDA
M00003E 05-LISTA
M000070 06+ENDA
M000070 06-LISTA
M000070 06+LISTC
M000073 06+ENDC
M000073 06-LISTC
M000076 06+ENDC
M000076 06-LISTC
M000076 06+LISTA
M000079 06+ENDA
M000079 06-LISTA
M00007C 06+PROGB
M00007C 06-LISTA
E
```

Program C:

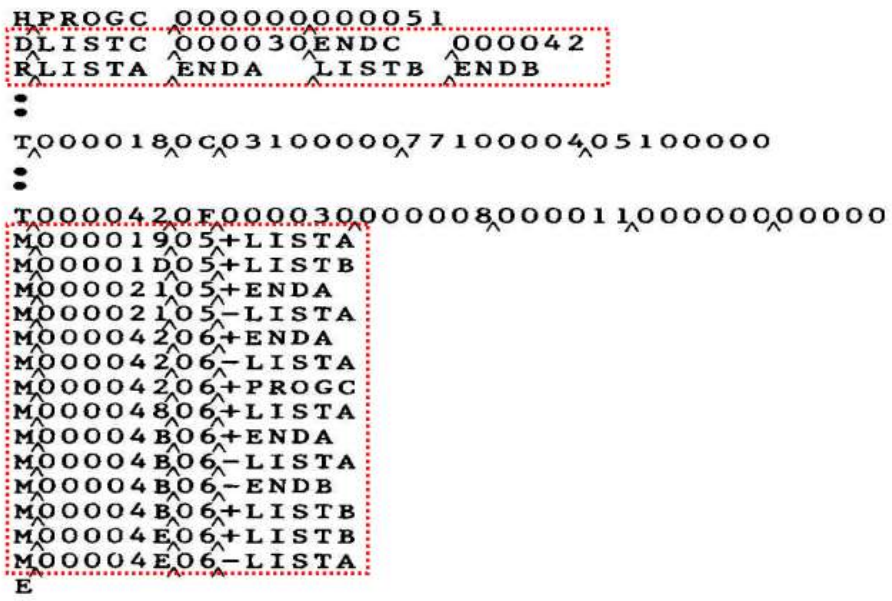


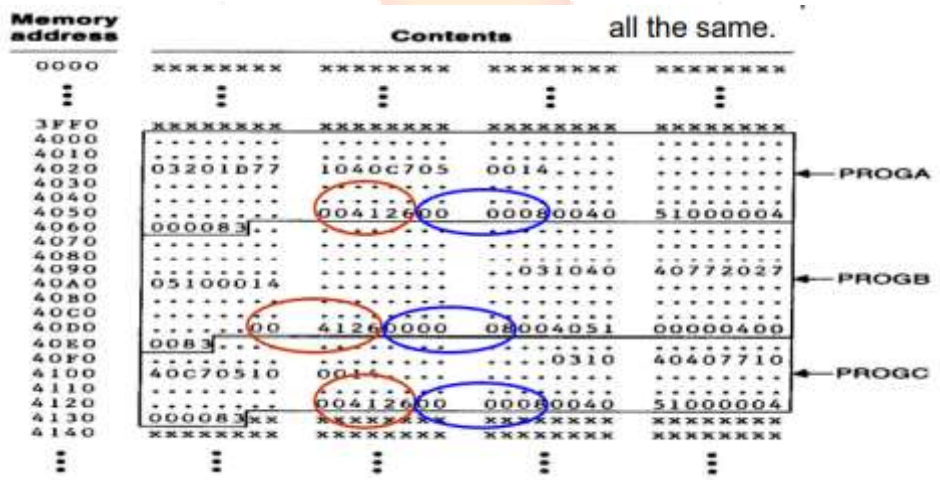
Figure 5.9: Object Programs

- The reference marked REF2 is processed in a similar manner.
- REF3 is an immediate operand whose value is to be the difference between ENDA and LISTA (that is, the length of the list in bytes).
- In Program A, the assembler has all the information necessary to compute this value. During the assembly of Program B (and Program C), the values of the labels are unknown.
- In these programs, the expression must be assembled as an external reference (*with two Modification records*) even though the result will be an absolute value independent of the locations at which the programs are loaded.

Consider REF4.

- The assembler for Program A can evaluate all of the expressions in REF4 except for the value of LISTC. This results in an initial value of '000014'H and one Modification record.

- The same expression in Program B contains no terms that can be evaluated by the assembler. The object code, therefore, contains an initial value of 000000 and *three* Modification records.
- For Program C, the assembler can supply the value of LISTC relative to the beginning of the program (but not the actual address, which is not known until the program is loaded).



- The initial value of this data word contains the relative address of LISTC ('000030'H). Modification records instruct the loader to add the beginning address of the program (i.e., the value of Program C), to add the value of ENDA, and subtract the value of LISTA.

Figure 5.10: The three programs as they might appear in memory after loading and linking.

In Figure 5.10, Program A has been loaded starting at address 4000, with Program B and Program C immediately following.

For example, the value for reference REF4 in Program A is located at address 4054 (that is, the beginning address of Program A plus 0054).

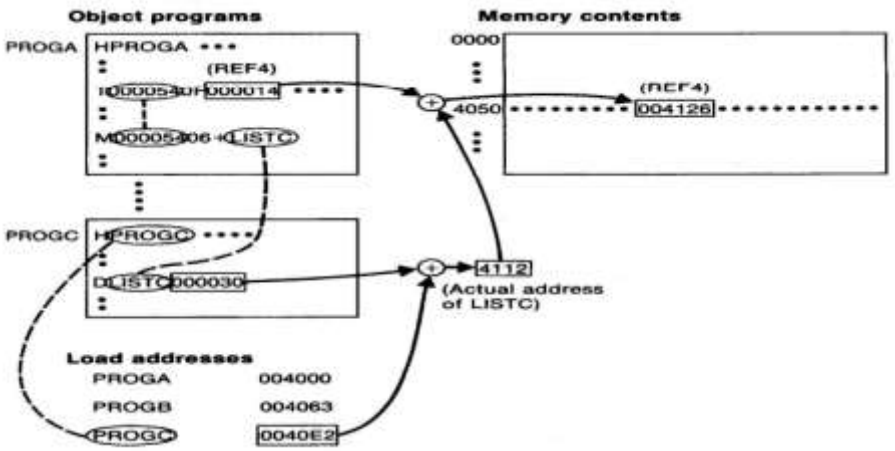


Figure 5.11: Relocation and linking operations performed on REF4 in PROGA

Consider the above figure 5.11,

The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of Program C plus 30).

5.3 Algorithm And Data Structures For A Linking Loader

The algorithm for a *linking* loader is considerably more complicated than the

absolute loader algorithm.

- A linking loader usually makes two passes over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler:
- . In Pass 1, addresses are assigned to all external symbols for further process
- Pass 2 performs the actual loading, relocation, and linking functionalities.

The main data structure needed for our linking loader is an *external symbol table (ESTAB).*

(1) This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the name and address of each external symbol in the set of control sections being loaded.

(2) A *hashed organization* is typically used for this table.

Two other important variables such as PROGADDR (program load address)

and CSADDR (control section address) are implicated in the process.

(1) PROGADDR is *the beginning address in memory* where the linked program is to be loaded. Its value is supplied to the loader by the OS.

(2) CSADDR contains *the starting* address assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses.

Pass 1

During Pass 1, the loader is concerned only with Header and Define record types in the control sections.

```
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}
   end {while not EOF}
end {Pass 1}
```

Algorithm for Pass 1 of a Linking loader:

1) The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.
2) The control section name from the Header record is entered into ESTAB, with the value given by CSADDR. All **external symbols** appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR.

3) When the End record is read, the control section length CSLTH (which was saved from the End record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.
   - At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.
   - Many loaders include as an option the ability to print a **load map** that shows these symbols and their addresses.

Pass 2

Pass 2 performs the actual *loading, relocation, and linking* of the program.

Algorithm for Pass 2 of a Linking loader

1) As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).
2) When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.
3) This value is then added to or subtracted from the indicated location in memory.
4) The last step performed by the loader is usually the transferring of control to the loaded program to begin execution.

The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution. If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered. If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point. Normally, a transfer address would be placed in the End record for the main program, but not for a subroutine.

```
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  // the next control section
    end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

This algorithm can be made more efficient. Assign a reference number, which is used (instead of the symbol name) in Modification records, to each external symbol referred to in a control section. Suppose we always assign the reference number 01 to the control section name.

6. MACHINE INDEPENDENT LOADER FEATURES

Machine independent loader features are:

- Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler.
- They include the use of an automatic library search process for handling external references and some common options that can be selected at the time of loading and linking.

6.1 Automatic Library Search

Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. Linking loaders that support *automatic library* search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references. The loader searches the library or libraries specified for routines that contain the definitions of these symbols and processes the subroutines found by this search exactly as if they had been part of the primary input stream. The subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved. If unresolved external references remain after the library search is completed, these must be treated as errors.

6.2 Loader Options

Many loaders allow the user to specify options that modify the standard processing

- **Typical loader option 1**: Allows the selection of alternative sources of input.

Example: INCLUDE program-name (library-name) might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.

- **Loader option 2:** Allows the user to delete external symbols or entire control sections.

Example: DELETE csect-name might instruct the loader to delete the named control section(s) from the set of programs being loaded.

CHANGE name1, name2 might cause the external symbol name1 to be changed to name2 wherever it appears in the object programs.

- **Loader option 3:** Involves the automatic inclusion of library routines to satisfy external references.

Example: LIBRARY MYLIB

Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

NOCALL STDDEV, PLOT, CORREL

To instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

Self-Assessment Questions - 2

4. Loaders that allow for program relocation are called _____ .
5. Each part of the object code that must be changed when the program is relocated is described in _____ .
6. In the case of Relocation bits, the bits corresponding to unused words are set to _____ .
7. Capability that eliminates some of the need for the loader to perform program relocation is called _____ .

7. 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 the linking function is performed at execution time

Linking Loaders

The below figure 5.12 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.

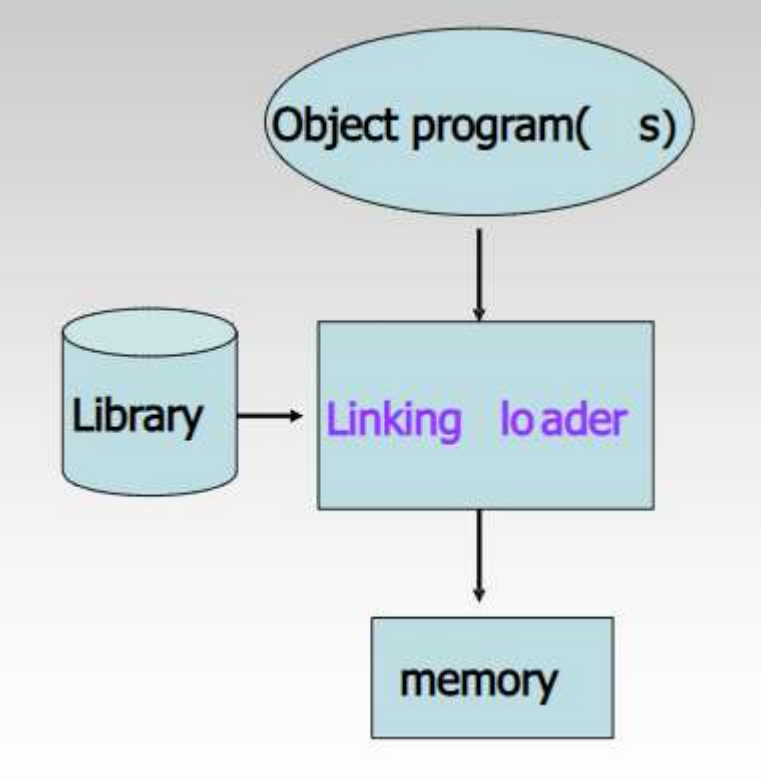


Figure 5.12: Processing of an object program using Linking Loader

Linking loaders perform all linking and relocation at load time. There are two alternatives:

1. **Linkage editors**, which perform linking prior to load time.
2. **Dynamic linking**, in which the linking function is performed at execution time.

Precondition: 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, and loads the linked program directly into memory for execution.
- A **linkage editor** produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

7.1 Linkage Editors

The figure 5.13 shows the processing of an object program using the Linkage editor.

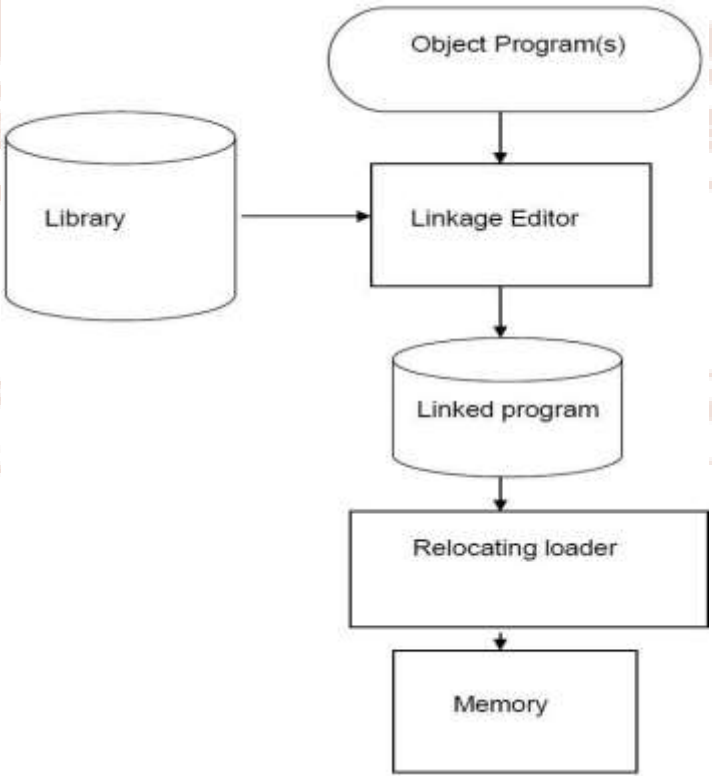


Figure 5.13: Processing of an object program using Linkage editor

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.

The Linkage editor has some helpful features, such as the ability to generate an absolute object programme if the 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.

The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program.

This means that the loading can be accomplished in one pass with no external symbol table required. If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.

Linkage editors are capable of much more than just getting an object programme ready for execution.. 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 languages.

Linkage editors often include a variety of other options and commands. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control.

7.2 Dynamic Linking

Dynamic linking is a method that delays the linking operations until execution. That is 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, that it allows 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 they are needed. The actual loading and linking can be accomplished using an operating system service request. Figure 5.14 (a, b, c, d, e) shows the operations of Dynamic Linking.

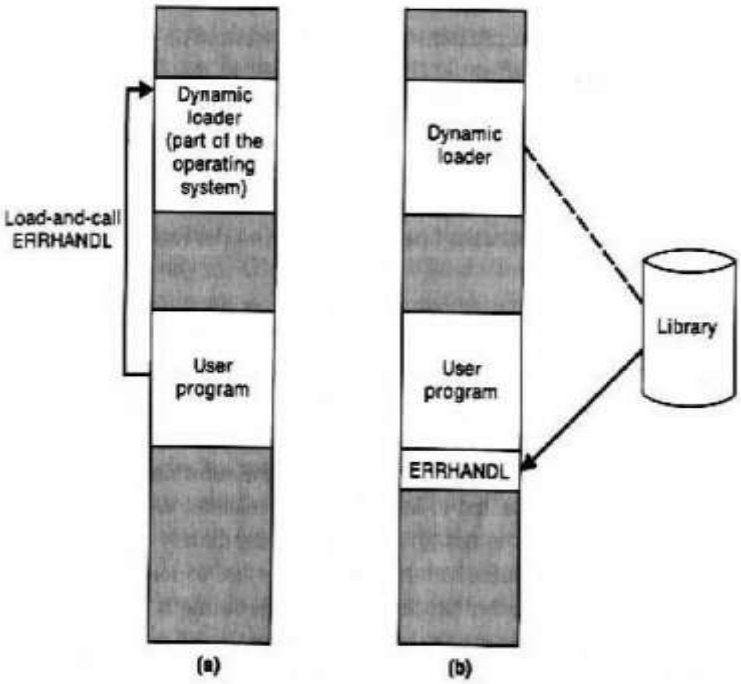


Figure 5.14 (a and b): Dynamic Linking.

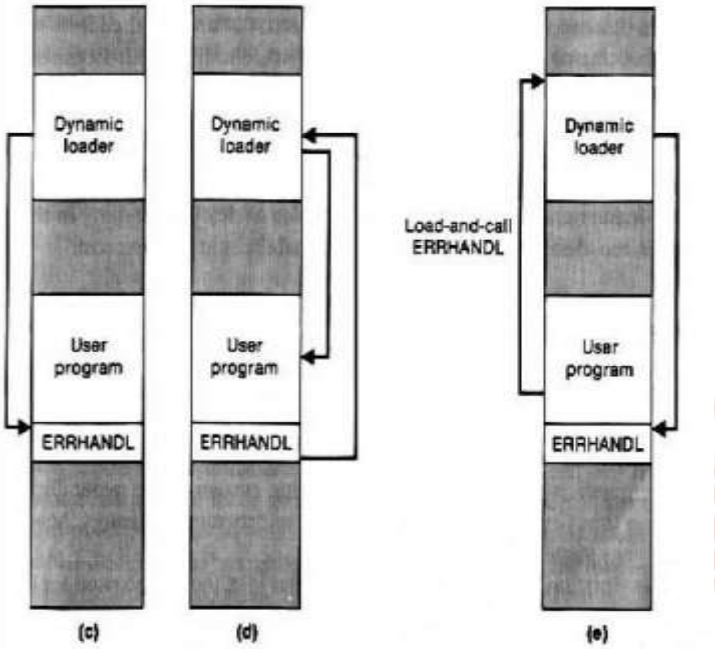


Figure 5.14 (c, d, e): Dynamic Linking.

Fig (a): Instead of executing a JSUB instruction referring to an external symbol, the program makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called.

Fig (b): OS checks its internal tables to see if the procedure has previously been loaded. If necessary, the routine is loaded from the specified user or system libraries.

Fig (c): Control is then passed from OS to the routine being called.

Fig (d): When the called subroutine completes its processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.

Fig (e): If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine.

Linkage editors perform linking operations before the program is loaded for execution. Linking loaders perform these same operations at load time.

Dynamic linking, dynamic loading, or load on call postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.

Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library. With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.

Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines

7.3 Bootstrap Loaders

If the question is how the loader is itself loaded into the memory? then the answer is when the 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. All object programmes that are to be loaded into a system that is empty and idle have one of these loaders added to the beginning of them. on some computers, an absolute loader is permanently resident in ROM. When the power is on, the machine begins to execute this ROM program. But it is inconvenient to change a ROM program if modifications in the absolute loader are required. An intermediate solution is to have a built-in hardware function that reads a fixed length record from some device into memory at a fixed location and then jump to execute it. This record contains machine instructions that load the absolute program that follows.

If the loading process requires more instructions, the first record causes the reading of the others, and these in turn can cause the reading of still more records. Hence the term is called Bootstrap. The first record (or records) is generally referred to as a bootstrap loader.

Self-Assessment Questions - 3

8. Some common alternatives for organizing the oading functions are_____.
9. Loaders which perform all linking and relocation at load time are called_____.
10. _____perform linking prior to load time.
11. Linking in which the linking function is performed at execution time is called _____ .
12. A loader which loads OS and prepares it for execution is called_____.

8. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- A loader is the part of an operating system that is responsible for loading programs. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution.
- Loading a program involves reading the contents of an executable file, and then carrying out other required preparatory tasks to prepare the executable for running.
- Loaders that allow for program relocation are called *relocating loaders* or *relative loaders.*
- A modification record is used to describe each part of the object code that must be changed when the program is relocated.
- A relocation bit associated with each word of object code is used to indicate whether or not this word should be changed when the program is relocated.
- There are some common alternatives for organizing the loading functions, including relocation and linking.
- Linking Loaders perform all linking and relocation at load time.
- Linkage editors perform linking prior to load time and, dynamic linking, in which the linking function is performed at execution time.

9. GLOSSARY

Bootstrap: A short computer program that is permanently resident or easily loaded into a computer and whose execution brings a larger program, such as an operating system or its loader, into memory.

Loader: A program that copies other [object] programs from auxiliary [external] memory to main [internal] memory prior to its execution.

Modification Record: This is to describe which part of the object code that be changed when the program is relocated.

Operating System: The program that runs on the hardware creating information objects such as files, processes, accounts, and access control of data.

Relative loaders: Loaders that allow for program relocation are called relocating loaders

10. TERMINAL QUESTIONS

Short Answer Questions

1. Compare the working of an absolute loader and a simple Bootstrap loader.
2. Explain in detail about Machine dependent loader features.
3. List and Describe different schemes of Relocation.
4. Describe briefly machine independent loader features.
5. Explain different loader design options.

11. ANSWERS

A. Self-Assessment Questions

1. Loading
2. Bootstrap Loader
3. An Operating System
4. Relocating Loader
5. Modification record
6. Zero
7. Hardware relocation
8. Relocation and linking
9. Linking loaders
10. Linkage editor
11. Dynamic linking
12. Bootstrap loader.

Short Answer Questions

1. For a simple absolute loader, all functions are accomplished in a single pass; Bootstrap loader loads the first program to be run by the computer – usually an operating system. Refer section 3 and 4 for more detail.
2. Main Machine dependent features are Relocation and Program linking. Refer section 5 for detail.
3. Different schemes of relocation are Modification record, relocation bit and hardware relocation. Refer section 5.1 for detail
4. Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler. Refer section 6 for detail
5. Different loader design options are linkage editor, dynamic linking and linking loader. Refer section 7 for detail

12. SUGGESTED BOOKS

- Dhamdhere (2002). Systems programming and operating systems Tata McGraw-Hill.
- M.Joseph (2007). System software, Firewall Media.