BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 6

Linker

Table of Contents

1. INTRODUCTION

In the previous unit, we studied Loaders and loader features. In this unit, we are going to discuss Linkers. Execution of a program written in a language, say L involves four steps. They are,

1) Translation of the program,
2) linking of the program with other programs needed for its execution,
3) Relocation of the program to execute from the specific memory area allocated to it,
4) loading of the program in memory for execution.

In these four steps, step 1 is performed by a language translator, step 2 and 3 are performed by a *linker*, and step 4 is performed by a loader.

In this unit, we are discussing Relocation and Linking concepts, the Design of a linker, and the self-relocating program linking for overlays.

1.1 Learning Objectives

After studying this unit, you should be able to:

- *Define a Linker*
- *Describe relocation and linking concepts*
- *Explain the design of a linker*
- *Describe self-relocating program*
- *Explain linking for overlays*

2. INTRODUCTION TO LINKER

A linker or link editor is a program that takes one or more objects generated by compilers and assembles them into a single executable program.

Link editors are commonly known as linkers. The compiler automatically invokes the linker as the last step in compiling a program. The linker inserts code (or maps in shared libraries) to resolve program library references and/or combines object modules into an executable image suitable for loading into memory.

Definition

Linker is a program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code. . This process involves resolving references between the modules and fixing the relocation information used by the operating system kernel when loading the file into memory to run it.

The objects are program modules containing machine code and information for the linker. This information comes mainly in the form of symbol definitions, which come in two varieties:

- Defined or exported symbols are functions or variables that are present in the module represented by the object, and which should be available for use by other modules.
- Undefined or imported symbols are functions or variables that are called or referenced by this object, but not internally defined.

In short, the linker's job is to resolve references to undefined symbols by finding out which other object defines a symbol in question and replacing placeholders with the symbol's address.

There are four different steps for the execution of a program, such as Translation, linking, relocation, and loading. Figure 6.1 shows the schematic which shows these four program execution steps.

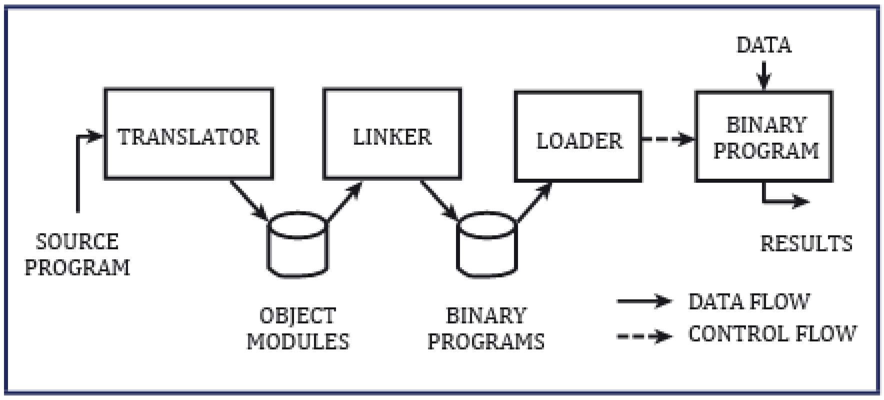

Fig 1: A schematic of program execution

The translator outputs a program form called *object module* for the program. The linker processes a set of object modules to produce a ready-to-execute program form, which we will call a *binary program*. The loader loads this program into the memory for the purpose of execution. As shown in the schematic, the object modules and ready-to-execute program forms can be stored in the form of files for repeated use.

Linkers can take objects from a collection called a library. Some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries. Libraries for diverse purposes exist, and one or more system libraries are usually linked in by default.

The linker also takes care of arranging the objects in a program's address space. This may involve relocating code that assumes a specific base address to another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads, and stores.

The executable output by the linker may need a relocation pass when it is finally loaded into memory just before execution. On hardware offering virtual memory, this is usually omitted, though – every program is put into its own address space, so there is no conflict even if all programs load at the same base address.

Self-Assessment Questions - 1

1. A program that takes one or more objects generated by compilers and assembles them into a single executable program is called __________ .
2. A Program module containing machine code and information for the linker is called ____________ .
3. The functions or variables that are present in the module represented by the object, and which should be available for use by other modules are called ________ .
4. The functions or variables that are called or referenced by this object, but not internally defined are called__________ .
5. The linker processes a set of object modules to produce a ready to execute program form called____________.

3. RELOCATION AND LINKING CONCEPTS

Different relocation and linking concepts are discussed in this section.

3.1 Program Relocation

Let AA be a set of absolute addresses-instruction or data addresses used in the instruction of a program say P. AA ≠ ∑ implies that program P assumes its instruction and data to occupy memory words with specific addresses. Such a program called an address-sensitive program contains one or more of the following:

1. An address sensitive instruction: an instruction that uses an address ai∑AA
2. An address constant: a data word that contains an address ai∑AA.

An address-sensitive program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin. To execute correctly from any other memory area, the address used in each address-sensitive instruction of P must be 'corrected'.

Definition: Program relocation is the process of modifying the addresses used in the address-sensitive instructions of a program such that the program can execute correctly from the des¬ignated area of memory.

If linked origin ≠ translated origin, relocation must be performed by the linker. If load ori-gin ≠ linked origin, relocation must be performed by the loader. In general, a linker always performs relocation, whereas some loaders do not.

For simplicity, it has been assumed that loaders do not perform relocation-that is, load origin = linked origin. Such loaders are called absolute loaders. Hence the terms 'load origin' and 'linked origin' are used interchangeably. However, it would be more precise to use the term 'linked origin'.

Performing relocation

Let the translated and linked origins of a program P be t_originp, and *l*_origin$_p$ respectively. Consider a symbol *symb* in P. Let its translation time address be t$_{symb}$ and link time address is l$_{symb}$. The relocation factor of P is defined as

$$\text{relocation\_factor}_p = l\_\text{origin}_p - t\_\text{origin}_p \quad (6.1)$$

Note that relocation_factor$_p$ can be positive, negative or zero.

Consider a statement that uses symb as an operand. The translator puts the address t$_{symb}$ in the instruction generated for it. Now,

$$t_{symb} = t\_\text{origin}_p + d_{symb}$$

Where d$_{symb}$ is the offset of *symbin* P. Hence

$l_{symb} = l\_origin_p + d_{symb}$

using (6.1),

$l_{symb} = t\_\text{origin}_p + \text{relocation\_factor}_p + d_{symb}$

$= t\_\text{origin}_p + d_{symb} + \text{relocation\_factor}_p$

$= t_{symb} + \text{relocation\_factor}_p \quad (6.2)$

Let IRR$_p$ designate the set of instructions requiring relocation in program P (IRR$_p$). From (6.2), relocation of program P can be performed by computing the relocation factor for P and add¬ing it to the translation time address in every instruction I ∑ IRR$_p$.

3.2 Linking

Definition

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

Consider an application program AP consisting of a set of program units SP = {P_i}. A program unit Pi interacts with another program unit Pj by using addresses of Pj's instructions and data in its own instructions. To realize such interactions, Pj and Pi 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
- *External reference:* A reference to a symbol *ext_symb* that is not defined in the program unit containing the reference.

The handling of public definitions and external references is described in the following.

EXTRN and ENTRY statements

The ENTRY statement lists the public definitions of a program unit, i.e. it lists those symbols defined in the program unit which may be referenced in other program units. The EXTRN statement lists the symbols to which external references are made in the program unit.

Resolving external references

Before the application program AP can be executed, it is necessary that for each P_i in SP, every external reference in P_i should be bound to the correct link time address.

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

Binary Programs

A binary program is a machine language program comprising a set of program units SP (Source Program) such that all Pi belongs to SP.

1. P_i has been relocated to the memory area starting at its link origin, and
2. Linking has been performed for each external reference in P_i.

To form a binary program from a set of object modules, the programmer invokes the linker using the command

Linker <link origin>, <object module names>

[, <execution start address>]

Where <link origin> specifies the memory address to be given to the first word of the binary program.<execution start address> is usually a pair (program unit name, offset in program unit). The linker converts this into the linked start address. This is stored along with the binary program for use when the program is to be executed. If the specification of <execution start address> is omitted the execution start address is assumed to be the same as the linked origin.

Note that a linker converts the object modules in the set of program units SP into a binary program. Since we have assumed link address = load address, the loader simply loads the binary program into the appropriate area of memory for the purpose of execution.

3.3 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: Symbolic 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 mem¬ory word which is required to contain the address of the symbol.

Self-Assessment Questions - 2

6. The process of modifying the addresses used in the address-sensitive instructions of a program such that the program can execute correctly from the designated area of memory is called ________ .
7. A symbol pub_symb defined in a program unit which may be referenced in other program units is called _______ .
8. A reference to a symbol ext_symb which is not defined in the program unit containing the reference is called _______ .
9. The process of binding an external reference to the correct link time address is called _______ .

4. DESIGN OF A LINKER

In this section, we are discussing the design of a linker.

4.1 Relocation And Linking Requirements In Segmented Addressing

The relocation requirements of a program are influenced by the addressing structure of the computer system on which it is to execute. The use of the segmented addressing structure reduces the relocation requirements of the program.

Example

Consider the program written in the assembly language of intel 8088 shown in table 6.1. The ASSUME statement declares the segment registers CS and DS to the available for memory addressing. Hence all memory addressing is performed by using suitable displacements from their contents. Translation time address of A is 0196. In statement 16, a reference to A is assembled as a displacement of 196 from the contents of the CS register. This avoids the use of an absolute address; hence the instruction is not address sensitive. Now no relocation is needed if segment SAMPLE is to be loaded with address 2000 by a calling program (or by the OS). The effective operand address would be calculated as <CS>+0196, which is the correct address 2196. A similar situation exists with the reference to B in statement 17. The reference to B is assembled as a displacement of 0002 from the contents of the DS register. Since the DS register would be loaded with the execution time address of DATA_HERE, the reference to B would be automatically relocated to the correct address.

Table 1: An 8088 assembly program for linking

Sr. No		statement			offset
0001	DATA_HERE	SEGMENT			
0002	ABC	DW	25		0000
0003	B	DW	?		0002
.		.			
.		.			
0012	SAMPLE	SEGMENT			
0013		ASSUME		CS: SAMPLE, DS: DATA_HERE	
0014		MOV		AX, DATA_HERE	0000
0015		MOV		DS, AX	0003
0016		JMP		A	0005
0017		MOV	AL,	B	0008
.					
.					
0027	A	MOV	AX,	BX	0196
.		.			
.		.			
0043	SAMPLE	ENDS			
0044		END			

Though the use of a segment register reduces the relocation requirements, it does not completely eliminate the need for relocation. Consider statement 14 in table 6.1.

MOV AX, DATA_HERE

This loads the segment base of DATA_HERE into the AX register preparatory to its transfer into the DS register. Since the assembler knows DATA_HERE to be a segment, it makes provision to load the higher order 16 bits of the address of DATA_HERE into the AX register. However it does not know the link time address of DATA_HERE, hence it assembles the MOV instruction in the immediate operand format and puts zeroes in the operand field. It also makes an entry for this instruction in RELOCTAB so that the linker would put the appropriate address in the operand field. Inter-segment calls and jumps are handled in a similar way.

Relocation is somewhat more involved in the case of intra-segment jumps assembled in the FAR format. For example, consider the following program:

```
FAR_LAB  EQU THIS FAR; FAR_LAB is a FAR label

JMP      FAR_LAB; A FAR jump
```

Here the displacement and the segment base of FAR_LAB are to be put in the JMP instruction itself. The assembler puts the displacement of FAR_LAB in the first two operand bytes of the instruction and makes a RELOCTAB entry for the third and fourth operand bytes which are to hold the segment base address. A segment like

```
ADDR_A   DW      OFFSET A
```

(Which is an 'address constant') does not need any relocation since the assembler can itself put the required offset in the bytes. In summary, the only RELOCATAB entries that must exist for a program using segmented memory addressing are the bytes that contain a segment base address.

For linking, however, both segment base address and offset of the external symbol must be computed by the linker. Hence there is no reduction in the linking requirements.

4.2 Relocation Algorithm

Let us go through the algorithm of program relocation.

Algorithm 6.1 (Program relocation)

```
1. Program_linked_origin :=<link origin> from linker command;
2. For each module
   a. t_origin :=translated origin of the object module; OM_size := size of the object module;
   b. relocation_factor := program_linked_origin – t_origin;
   c. Read the machine language program in work_area.
```

d. Read RELOCTAB of the object module.

e. For each entry in RELOCTAB

 i. Translated_addr:=address in the RELOCTAB entry.

 ii. Address_in_work_area := address of work_area +translated_address – t_origin;

 iii. Add relocation_factor to the operand address in the word with the address address_in_work_area.

f. Program_linked_origin :=program_linked_origin + OM_size;

The computation of the work area address of the word requiring relocation (step 2(e) (ii)). Step 2(f) increments program_linked_origin so that the next object module would be granted the next available load address.

4.3 Linking Requirements

Features of a programming language influence the linking requirement of programs. Procedure references do not require linking, they can be handled through relocation. References to built-in functions, however, require linking.

A reference to an external symbol says alpha can be resolved only if alpha is declared as a public definition in some object module. This observation forms the basis of program linking. The linker processes all object modules being linked and builds a table of all public definitions and their load time addresses. Linking for alpha is simply a matter of searching for alpha in this table and copying its linked address into the word containing the external reference.

A name table (NTAB) is defined for use in program linking. Each entry of the table contains the following fields:

Symbol: Symbolic names of an external reference or an object module.

Linked_address: For a public definition, this field contains the linked address of the symbol. An object module contains the linked origin of the object module.

Most information in NTAB is derived from LINKTAB (Linking Table) entries with type = PD.

Algorithm 6.2 (Program Linking)

```
1. Program_linked_origin :=<link origin> from linker command.
2. For each object module
   a. t_origin := translated origin of the object module; OM_size:=size of the object
      module;
   b. relocation_factor :=program_linked_origin – t_origin;
   c. Read the machine language program in work_area.
   d. Read LINKTAB of the object module.
   e. For each LINKTAB entry with type=PD name := symbol;
   f. linked_address := translated_address + relocation_factor; Enter(name,
      linked_address) in NTAB;
   g. Enter (object module name, program_linked_origin) in NTAB.
   h. Program_linked_origin:= Program_linked_origin+OM_size;
3. For each object module
   a. t_origin:=translated origin of the object module; program_linked_origin
      :=load_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 the linked address to
          the operand address in the word with the address address_in_work_area.
```

Self-Assessment Questions - 3

10. The relocation requirements of a program are influenced by ________ of the computer system on which it is to execute.
11. For linking, both ________ and ________ of the external symbol must be com-puted by the linker.

5. SELF RELOCATING PROGRAMS

The manner in which a program can be modified, or can modify itself, to execute from a given load origin can be used to classify programs into the following:

1. Non-relocatable programs,
2. Relocatable programs,
3. Self-relocatable programs.

A non-relocatable program is a program that cannot be executed in any memory area other than the area starting on its translated origin. Non-relocatability is the result of the address sensitivity of a program and the lack of information concerning the address-sensitive instructions in the program. The difference between a relocatable program and a non-relocatable program is the availability of information concerning the address-sensitive instructions in it. A relocatable program can be processed to relocate it to a desired area of memory. Representative examples of non-relocatable and relocatable programs are a hand-coded machine language program and an object module, respectively.

A self-relocating program is a program that can perform the relocation of its own address-sensitive instructions. It contains the following two provisions for this purpose:

1. A table of information concerning the address-sensitive instructions exists as a part of the program.
2. Code to perform the relocation of address-sensitive instructions also exists as a part of the program. This is called the relocating logic.

The start address of the relocating logic is specified as the execution start address of the program. Thus the relocating logic gains control when the program is loaded in memory for execution. It uses the load address and the information concerning address-sensitive instructions to perform its own relocation. Execution control is now transferred to the relocated program.

A self-relocating program can execute in any area of the memory. This is very important in time-sharing operating systems where the load address of a program is likely to be different for different executions.

6. LINKING FOR OVERLAYS

Overlay-DeFinition: An overlay is a part of a program (or software package) that has the same load origin as some other part(s) of the program.

Overlays are used to reduce the main memory requirement of a program.

Overlay structured programs

We refer to a program containing overlays as an overlay structured program. Such a program consists of

1. A permanently resident portion called the root
2. A set of overlays.

Execution of an overlay structured program proceeds as follows: To start with, the root is loaded in memory and given control for the purpose of execution. Other overlays are loaded as and when needed. Note that the loading of an overlay overwrites a previously loaded overlay with the same load origin. This reduces the memory requirement of a program. It also makes it possible to execute programs whose size exceeds the amount of memory that can be allocated to them.

The overlay structure of a program is designed by identifying mutually exclusive modules, that is, modules that do not call each other. Such modules do not need to reside simultane-ously in memory. Hence they are located in different overlays with the same load origin.

Example: Consider a program with 6 sections named init, read, trans_a, trans_b, trans_c, and print. Init perform some initialization and passes control to read. read reads one set of data and invokes one of trans_a, trans_b, trans_c depending on the values of the data. Print is called to print the results.

Trans_a, trans_b, and trans_c are mutually exclusive. Hence they can be made into separate over¬lays. Read and print are put at the root of the program since they are needed for each set of data. For simplicity, we put init also in the root, though it could be made into an overlay by itself.

Consider Figure 6.2, which shows the proposed structure of the program. The overlay structured program can execute in 40K bytes though it has a total size of 65K bytes. It is possible to overlay parts of trans_a against each other by analyzing its logic. This will further reduce the memory requirements of the program.

The overlay structure of an object program is specified in the linker command.

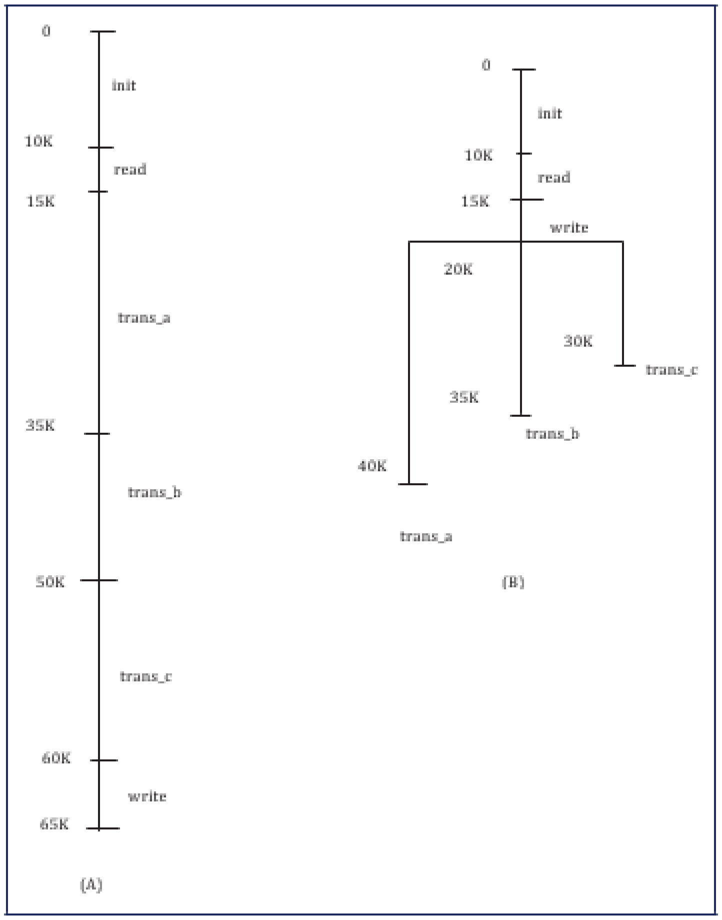

Fig 2: An overlay tree

Execution of an overlay structured program

For linking and execution of an overlay structured program, the linker produces a single executable file at the output, which contains two provisions to support overlays. First, an overlay manager module is included in the executable file. This module is responsible for loading the overlays when needed. Second, all calls that cross overlay boundaries are replaced by an interrupt producing instruction. To start with, the overlay manager receives control and loads the root. A procedure call that crosses overlay boundaries leads to an interrupt. This inter¬rupt is processed by the overlay manager and the appropriate overlay is loaded into memory. When each overlay is structured into a separate binary program, a call that crosses overlay boundaries leads to an interrupt which is attended by the OS (Operating System) kernel. Con¬trol is now transferred to the OS loader to load the appropriate binary program.

Usage of Overlays

Constructing an overlay program involves manually dividing a program into self-contained object code blocks called overlays laid out in a tree structure. Sibling segments, those at the same depth level, and sharing the same memory is called overlay region or destination region. An overlay manager, either part of the operating system or part of the overlay program, loads the required overlay from external memory into its destination region when it is needed. Often linkers provide support for overlays.

Self-Assessment Questions - 4

12. Part of a program that has the same load origin as some other part of the pro-gram is called _________ .
13. Code to perform the relocation of address sensitive instructions exists as a part of the program is called_________ .
14. Program which can be processed to relocate it to a desired area of memory is called _________ .
15. Program which can perform the relocation of its own address sensitive instructions is called ________ .

7. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- Linker is a program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code.
- This process involves resolving references between the modules and fixing the relocation information used by the operating system kernel when loading the file into memory to run it.
- Program relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.
- A self-relocating program is a program that can perform the relocation of its address-sensitive instructions.
- Overlays are used to reduce the main memory requirement of a program.

8. GLOSSARY

Binary program: Linker processes a set of object modules to produce a ready-to-execute program form.

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

Link editing: Makes a single program from several files of relocatable machine code.

Object module: The translator outputs a program form called an object module

Source program: A computer program that must be compiled, assembled, or otherwise translated to be executed by a computer. In contrast with the object program

Register: A small, high-speed memory circuit within a microprocessor that holds addresses and values of internal operations

9. TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Q1. Describe briefly about linker and linking of a program.

Q2. Explain program relocation concepts.

Q3. Explain the design of a linker.

Q4. Describe briefly self-relocating programs.

Q5. What are Overlays? Explain the process of linking for Overlays.

10. ANSWERS

SELF ASSESSMENT QUESTIONS

1. Linker or Link editor
2. Linker Objects
3. Defined or exported symbols
4. Undefined or imported symbols
5. Binary program
6. Program relocation
7. Public definition
8. External reference
9. Linking
10. Addressing structure
11. Segment base address and offset
12. Overlays
13. Relocating logic
14. Relocatable programs
15. Self-relocating program.

TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Answer 1: Linker is a program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code. (Refer to section 2 for detail).

Answer 2: Program relocation is the process of modifying the addresses used in the address-sensitive instructions of a program such that the program can execute correctly from the des¬ignated area of memory. (Refer section 3 for detail).

Answer 3: The relocation requirements of a program are influenced by the addressing structure of the computer system on which it is to execute. The use of the segmented addressing structure reduces the relocation requirements of the program. (Refer section 6.4 for detail).

Answer 4: A self-relocating program is a program that can perform the relocation of its own address-sensitive instructions. (Refer section 5 for detail)

Answer 5: An overlay is a part of a program that has the same load origin as some other part of the program. Overlays are used to reduce the main memory requirement of a program. (Refer section 6 for detail)

11. SUGGESTED BOOKS AND E-REFERENCES

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