Linker

Introduction

A program execution consists of the following steps,

Translation: A program is translated into a target program

Linking: The code of a program is combined with codes of those programs and library routine that it calls.

Relocation: It is the action of changing memory addresses used in the code of the program so that it can execute correctly in the allocated memory area.

Loading: The program is loaded in a specific memory area for execution

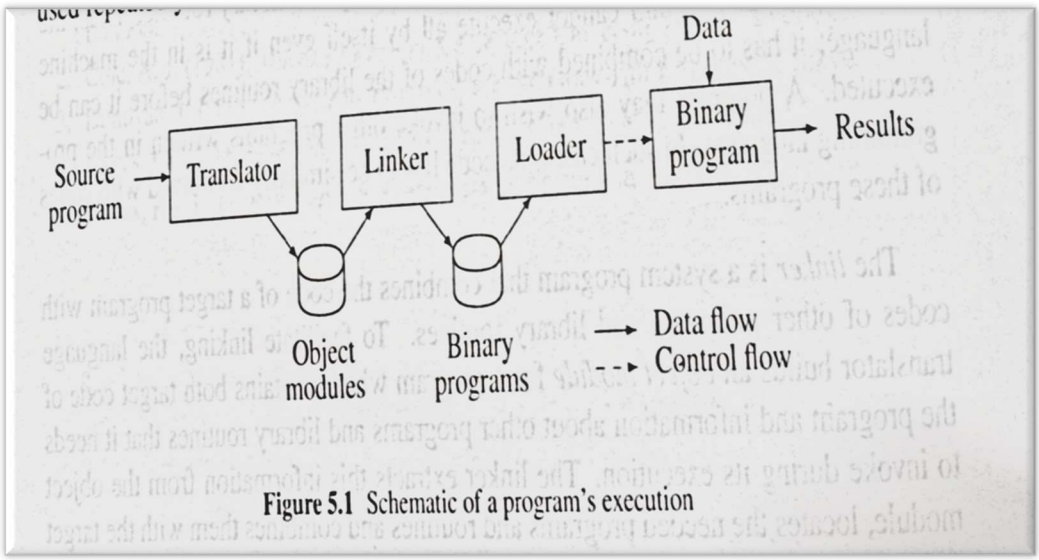


Figure 5.1 Schematic of a program's execution

The following terms are used related to the addresses

Translated address: Address assigned by the translator

Linked address: Address assigned by the linker

Load address: Address assigned by the loader

Translated origin: Address of the origin used by the translator. It is either the address specified by the programmer in ORIGIN or START statement or a default value

Linked origin: Address of the origin assigned by the linker while producing a binary program.

Load origin: Address of the origin assigned by the loader while loading the program in memory for execution.

Program Relocation

It is the action 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

Let AA be the set of Absolute Addresses (Address of instruction or data) of a program P.

An address sensitive program P, contains

> An address sensitive instruction: instruction address a, included in the set AA
>
> An address constant: data word address a, included in the set AA

If linked origin != translated origin => relocation must be performed by the linker

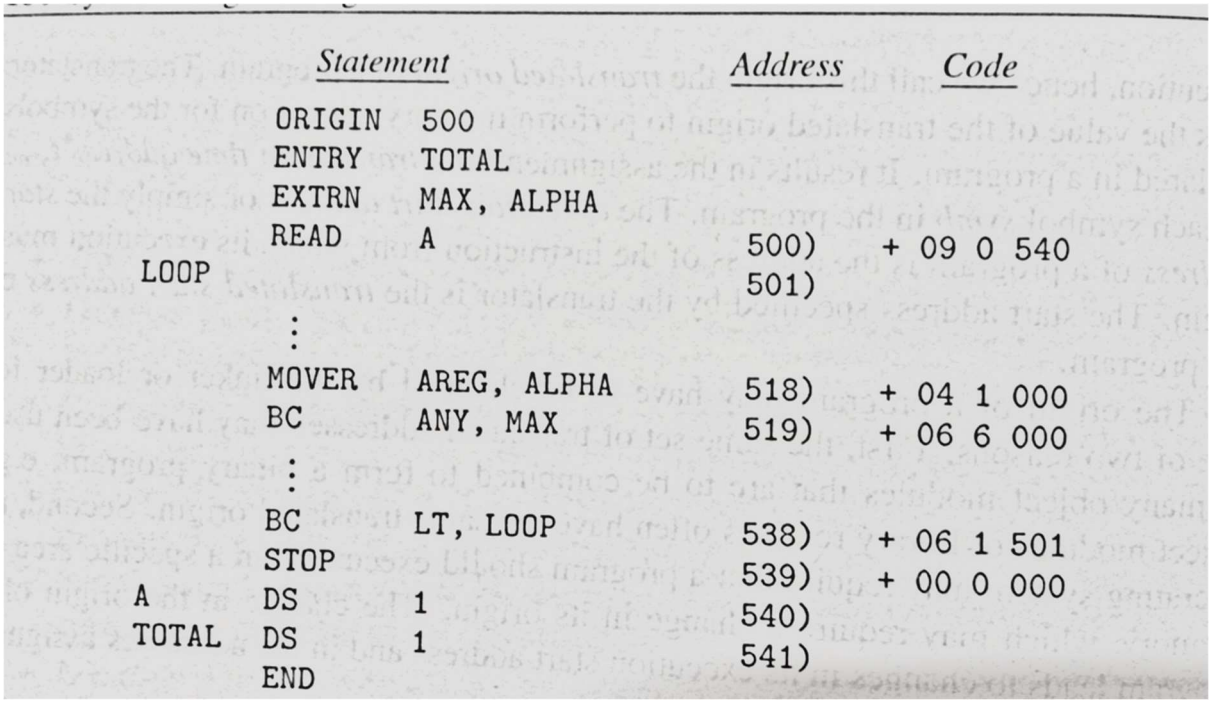
If load origin != linked origin => relocation must be performed by the loader

Generally the linker will perform relocation

Absolute loader will not perform relocation. (load origin = linked origin)

Relocation loader will perform relocation.

Correcting the addresses used in address sensitive instruction



	Statement		*Address*	*Code*
	ORIGIN	500		
	ENTRY	TOTAL		
	EXTRN	MAX, ALPHA		
	READ	A	500)	+ 09 0 540
LOOP			501)	
	⋮			
	MOVER	AREG, ALPHA	518)	+ 04 1 000
	BC	ANY, MAX	519)	+ 06 6 000
	⋮			
	BC	LT, LOOP	538)	+ 06 1 501
	STOP		539)	+ 00 0 000
A	DS	1	540)	
TOTAL	DS	1	541)	
	END			

500 is the translated origin.

If the program is loaded in the memory from the address

900, then it is linked/load origin

The READ instruction is address of A (540)

If 900 is the linked origin, then 940 is the linked address of A

Performing relocation

Let t_origin an l_origin be the translated and linked origin

Let t_symb and l_symb are the translated and linked address of a symbol symb

The relocation factor is defined as

Relocation_factor = l_origin – t_origin

t_symb = t_origin+d also l_symb = l_origin+d

So l_symb = t_symb + relocation_factor

From the previous example

Relocation_factor =900-500 = 400

The linked address of symbol A will be

L_symb = 540 + 400 = 940

Linking

A program unit may use another program unit during its execution. This is linking.

While creating a binary program the addresses of linked instruction/program unit to be added to the original program

For example an application consists of set of program units SP=Pi. Consider the program unit Pi requires to use another program unit pj during its execution. Binary program is formed by combining the addresses of linked instruction/program unit

To achieve this

Public definition of symbol: so that it can be used in other programs

External reference: a reference to a symbol that is not defined inside the program or defined in some either program

EXTRN and ENTRY statements

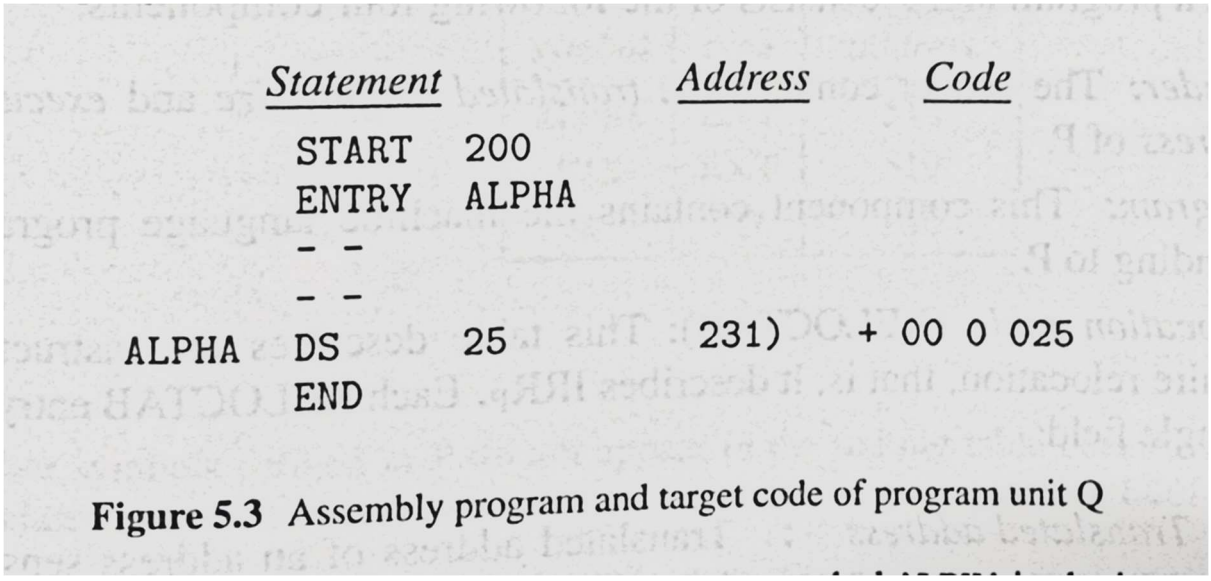
All public definitions of the program unit are listed by ENTRY statement

An EXTRN statement list the symbols to which external references are made.

Resolving external reference

Linking is the action of putting the correct linked address of those instructions in a program that contain external references.

An external reference in an instruction is said to be resolved when the correct linked address is added to the instructions: otherwise it is unresolved



Statement		*Address*	*Code*
	START 200		
	ENTRY ALPHA		
	- -		
	- -		
ALPHA	DS 25	231)	+ 00 0 025
	END		

Figure 5.3 Assembly program and target code of program unit Q

Constructing binary program

A binary program is a machine language program comprising a set of program units SP, such that every Pi is in SP

1.Pi has been relocated to the memory area whose starting address matches its linked origin and,

2.Each external reference in Pi is resolved

To form a binary program from the set of object modules, linker <link origin>, <object module> [,<execution start address>]

Object Module

The object module of a program unit contains all the information that would be needed to relocate and link the program unit with other program units

The object module contains 4 components

Header: contain translated origin, size and execution start address

Program: contains machine language program

Relocation Table (RELOCTAB): This table describes the instructions that require relocation

Translated address of an address sensitive instruction

Linking Table (LINKTAB): contain all public definition and external references. Each LINKTAB entry contains 3 fields

Symbol: Symbolic name

Type: PD and EXT indicate whether the entry is public definition or external reference

Translated address: It is address of the symbol

Example 5.6 (Object module) The object module of the program P of Figure 5.2 contains the following information:

1. The *header* contains the information *translated origin* = 500, *size* = 42, *execution start address* = 500.
2. The *program* component contains the machine language instructions shown in Figure 5.2.
3. *The relocation table* is as follows:

translated address
500
538

4. *The linking table* is as follows:

symbol	*type*	*translated address*
ALPHA	EXT	518
MAX	EXT	519
A	PD	540

Other symbols defined in P do not appear in the linking table because they are not declared as public definitions in ENTRY statements, e.g., the symbol LOOP.

Design of a linker

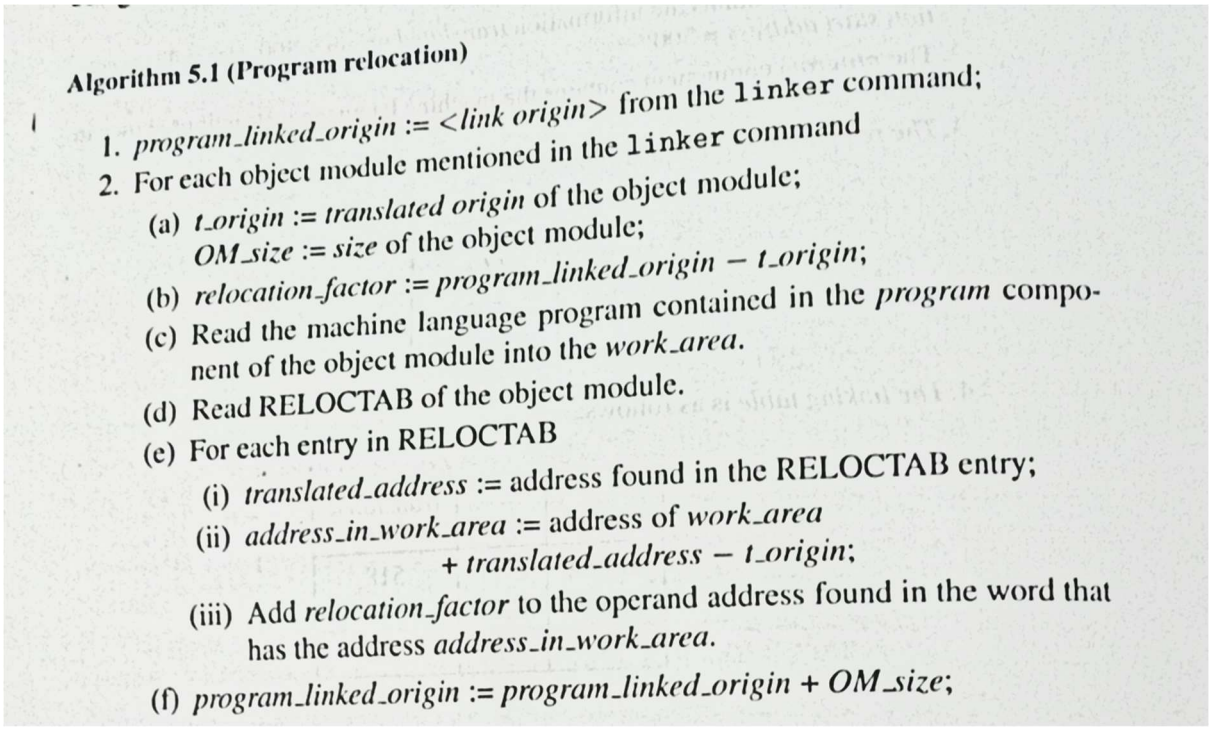
For simplicity we discuss separate algorithms for relocation and linking. These two algorithms can be combined to obtain an algorithm for linker

Scheme for relocation

The linker uses a memory called work area for constructing the binary program

It loads the machine language program found in the program component of an object module into the work area and

relocate the address sensitive instructions in it by processing RELOCTB



Algorithm 5.1 (Program relocation)

1. *program_linked_origin* := <*link origin*> from the `linker` command;
2. For each object module mentioned in the `linker` command
   (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 contained in the *program* component of the object module into the *work_area*.
   (d) Read RELOCTAB of the object module.
   (e) For each entry in RELOCTAB
      (i) *translated_address* := address found 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 found in the word that has the address *address_in_work_area*.
   (f) *program_linked_origin* := *program_linked_origin* + *OM_size*;

Scheme for Linking

The linker processes all LINKTABs of all object modules that are to be linked and copy the information from all public definitions into a table called NTAB(Name Table)

Each entry in NTAB contains

Symbol: Symbolic name of an external reference

Linked_address: For public definition it contains linked address of the symbol and for an object module it contains linked origin of the object module

Algorithm 5.2 (Program Linking)

1. *program_linked_origin* := <*link origin*> from the `linker` command.
2. For each object module mentioned in the `linker` command
   (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 contained in the *program* component of the object module into the *work_area*.
   (d) Read LINKTAB of the object module.
   (e) Enter (object module name, *program_linked_origin*) in NTAB.
   (f) For each LINKTAB entry with *type* = PD
   name := *symbol* field of the LINKTAB entry;
   linked_address := *translated_address* + *relocation_factor*;
   Enter (*name*, *linked_address*) in a new entry of the NTAB.
   (g) *program_linked_origin* := *program_linked_origin* + *OM_size*;
3. For each object module mentioned in the `linker` command
   (a) *t_origin* := translated origin of the object module;
   program_linked_origin := *linked_address* from NTAB;
   (b) For each LINKTAB entry with *type* = EXT
   (i) *address_in_work_area* := address of *work_area* +
   program_linked_origin − <*link origin*> in `linker` command
   + *translated address* − *t_origin*;
   (ii) Search the symbol found in the *symbol* field of the LINKTAB entry in NTAB and note its linked address. Copy this address into the operand address field in the word that has the address *address_in_work_area*.

Self - relocating programs

There are 3 ways in which program may be relocated

Non relocatable program: The program cannot be executed in any memory area other than the area starting from its translated origin.

Relocatable program: The program can be relocated by a linker or loader to have a linked address or load address that matches the start address of the specified area of memory

Self-Relocatable program: The program can be loaded in any area of memory for execution. At the start of execution, it would perform its own relocation so that it can execute correctly in that memory area.

A self-relocating program has the following 2 components along with machine language instructions

1. Table of information related to the address sensitive instructions.
2. Code to perform the relocation of address sensitive instructions. This code is called the relocating logic

Linking of overlay structured programs

An overlay is a part of the program that has same load origin as some other parts of the program

A program contains overlay is called overlay structured program

It consists of

1.A permanently resident part, called the root.

2.A set of overlays that would be loaded in memory when needed

Overlays are handled by overlay manager .It loads the required overlay which replaces the previoulsly loaded overlay with the same load origin.

Example of an overlay structured program is an assembler.

The passes of the assembler are different overlays and the data structures that are shared by them are in the root.

Example 5.15 (Design of an overlay structured program) Consider a program with 6 sections named `init`, `read`, `function_a`, `function_b`, `function_c`, and `print`. `init` performs some initializations and passes control to `read`. `read` reads one set of data and invokes one of `function_a`, `function_b` or `function_c` depending on the values of the data. `print` is called to print the results.

`function_a`, `function_b` and `function_c` are mutually exclusive. Hence they can be made into separate overlays. `read` and `print` are put in 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. Figure 5.9 shows the proposed structure of the program. The overlay structured program can execute in 40 K bytes though it has a total size of 65 K bytes. It may be possible to overlay parts of `trans_a` against each other by analyzing its logic. It would further reduce the memory requirements of the program.

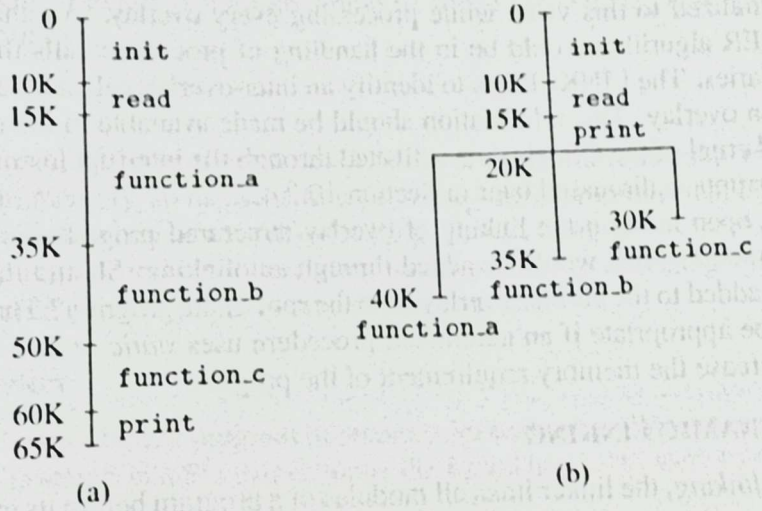


Figure 5.9 An overlay tree

Dynamic linking

If a linker links all modules of a program before its execution is called static linking

Linking process during execution of a program is called dynamic linking