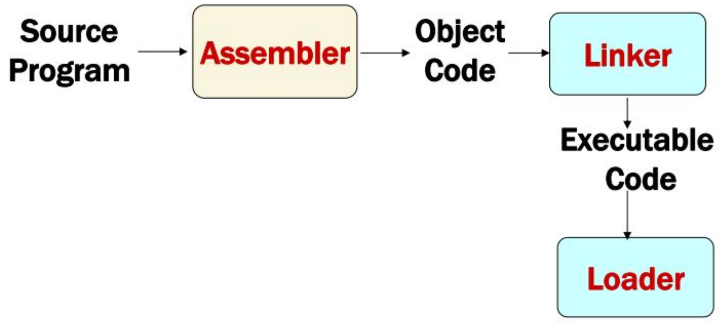
16/09/2020

Module II

Linkers

Steps of Execution of a Program

1. **Translation** of the program *Translator*
2. **Linking** of the program with other programs needed for its execution. *Linke*
3. **Relocation** of the program to execute from the specific memory are
4. **Loading** of purpose of execution.

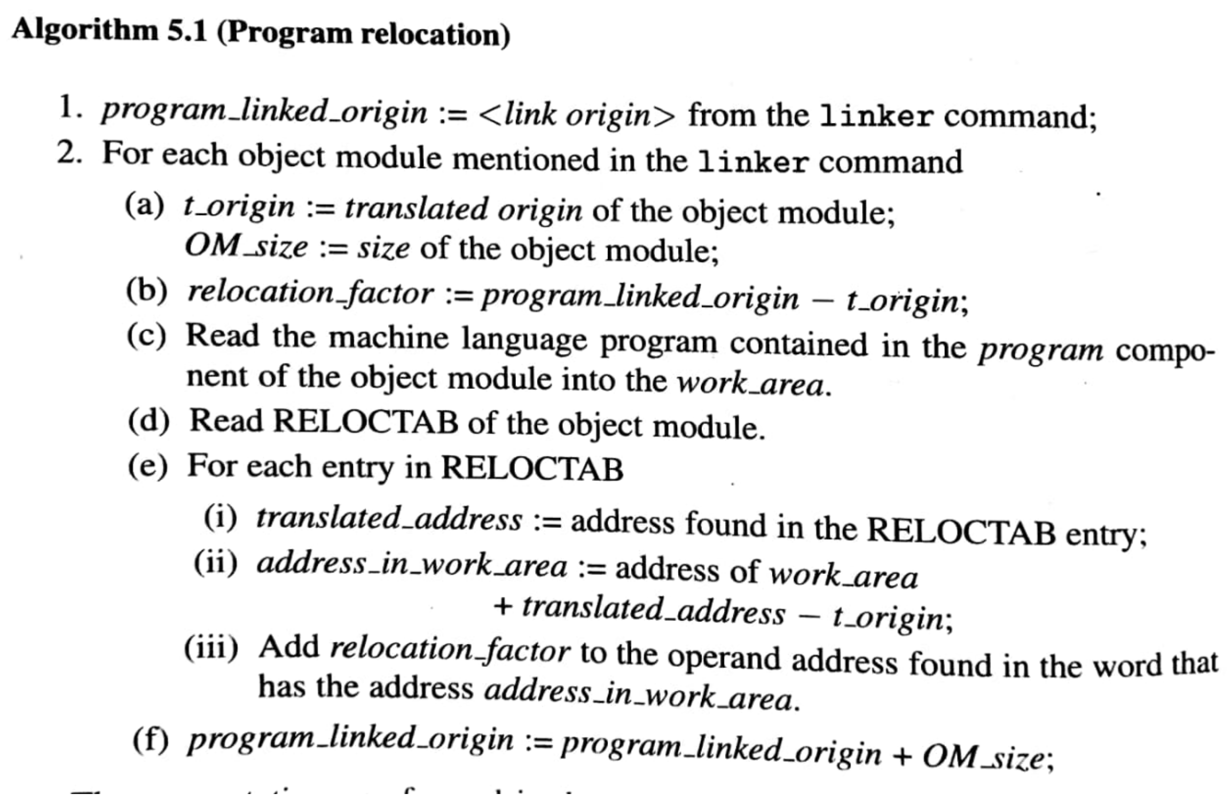


Object modules and ready-to-execute program forms can be stored in the form of files for repeated use

Design of a Linker

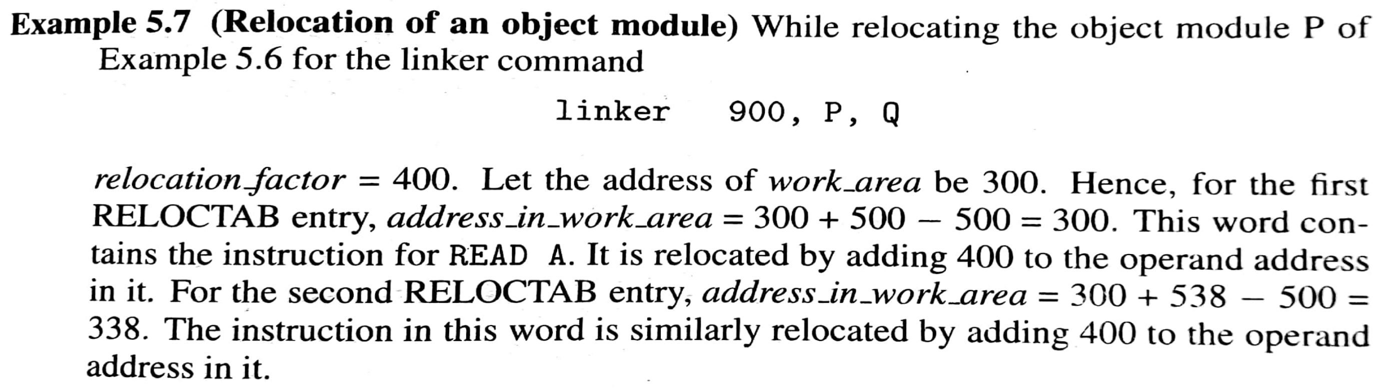
- We use separate algorithms for relocation and linking
- **Scheme for relocation**
  - *Work area* - linker uses this memory area to construct binary program
  - It loads the machine language program found in the program component of an object module into the work area and relocates the address sensitive instructions in it by processing entries of the RELOCTAB.
  - For each RELOCTAB entry, the linker determines the address of the word in the work area that contains the address sensitive instruction and relocates it.

Algorithm for Program Relocation



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 compo-
       nent 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;
```



Example 5.7 (Relocation of an object module) While relocating the object module P of Example 5.6 for the linker command

```
linker   900, P, Q
```

relocation_factor = 400. Let the address of *work_area* be 300. Hence, for the first RELOCTAB entry, *address_in_work_area* = 300 + 500 − 500 = 300. This word contains the instruction for `READ A`. It is relocated by adding 400 to the operand address in it. For the second RELOCTAB entry, *address_in_work_area* = 300 + 538 − 500 = 338. The instruction in this word is similarly relocated by adding 400 to the operand address in it.

Scheme for Linking

- The linker would process the linking tables(LINKTABs) of all object modules that are to be linked and copy the information about public definitions found in them into a table called the Name table(NTAB)
- *Fields of NTAB*

Symbol – symbolic name of an external reference or an object module

Linked_address – For a public definition, this field contains linked address of the symbol. For an object module, it contains the linked origin of the object module.

The linker performs relocation as follows: It obtains a LINKTAB entry that has EXT in the type field. This entry contains the address of an instruction that contains the external reference. To relocate this instruction it has to know the address of the word in the work area that contains this instruction. It computes this address by finding where the program component of the object module that contains the external reference has been loaded in the work area and adding to it the offset of the instruction that contains the external reference within this program

Algorithm for program linking

- **Two passes**
  - First pass- Step 2
    - It reads the program component of each object module into the work area, computes the linked origin of each object module and the linked address of each public definition in the object module and enters this information in the NTAB
  - Second pass – Step 3
    - It resolves each external reference by computing the address of the word in the work area that contains an instruction having an external reference and inserting the linked address of the symbol in it.

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 compo-
         nent 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.
```

Example 5.8 (Resolving an external reference) Let the `linker` command be

```
linker   900, P, Q
```

where P and Q are the program units of Example 5.2 and Example 5.5, respectively. The object module of program unit P was shown in Example 5.7. *program_linked-_origin* would have the values 900 and 942 while processing the object modules of P and Q. Accordingly, NTAB would contain the following information:

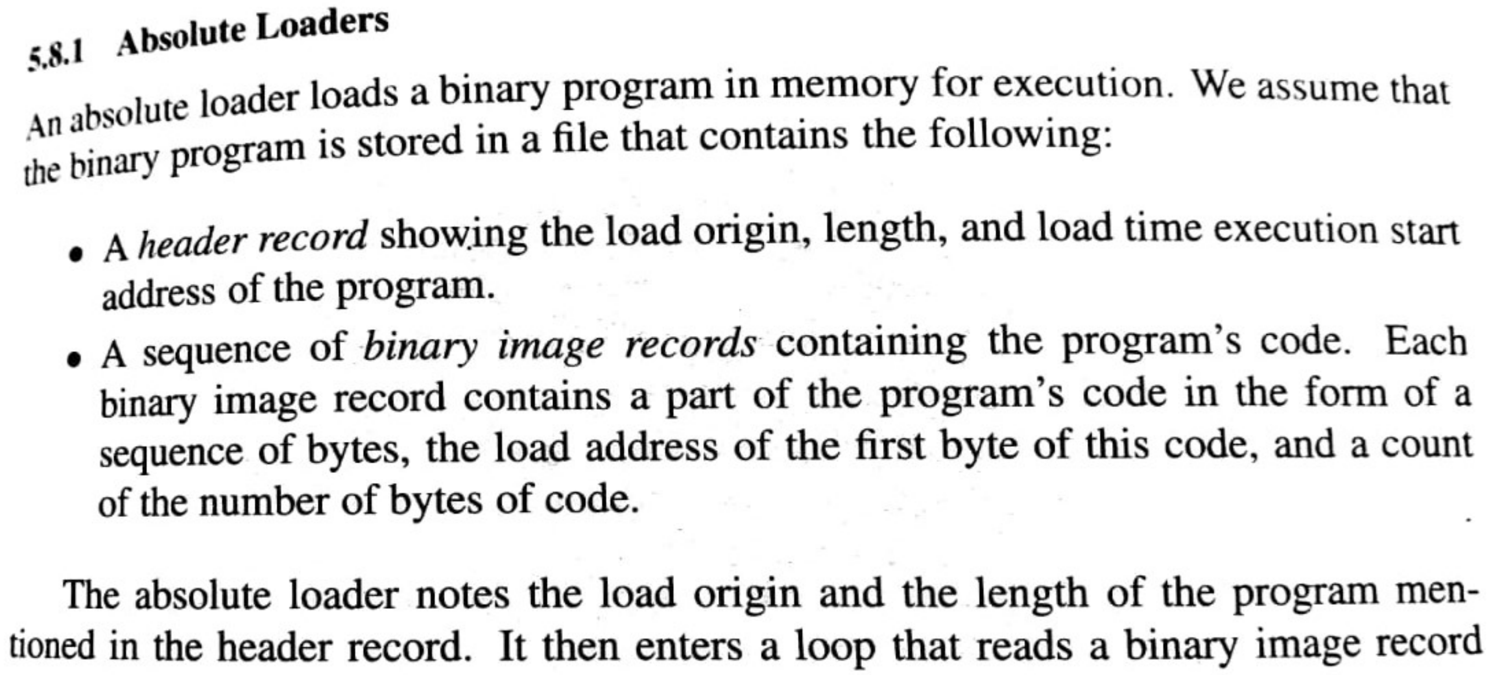
symbol	*linked address*
P	900
TOTAL	941
Q	942
ALPHA	973

Let the address of *work_area* be 300. When the LINKTAB entry of ALPHA is processed in Step 3, *address_in_work area* := 300 + 900 − 900 + 518 − 500, i.e., 318, because the LINKTAB entry shows the translated address of the external reference as 518 and *t_origin* is 500. Hence the linked address of ALPHA, i.e., 973, is copied from the NTAB entry of ALPHA and added to the word in address 318 (step 3(b)(ii)).

Loaders

- It brings the object program into the memory for execution.
- Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.

Types of Loaders

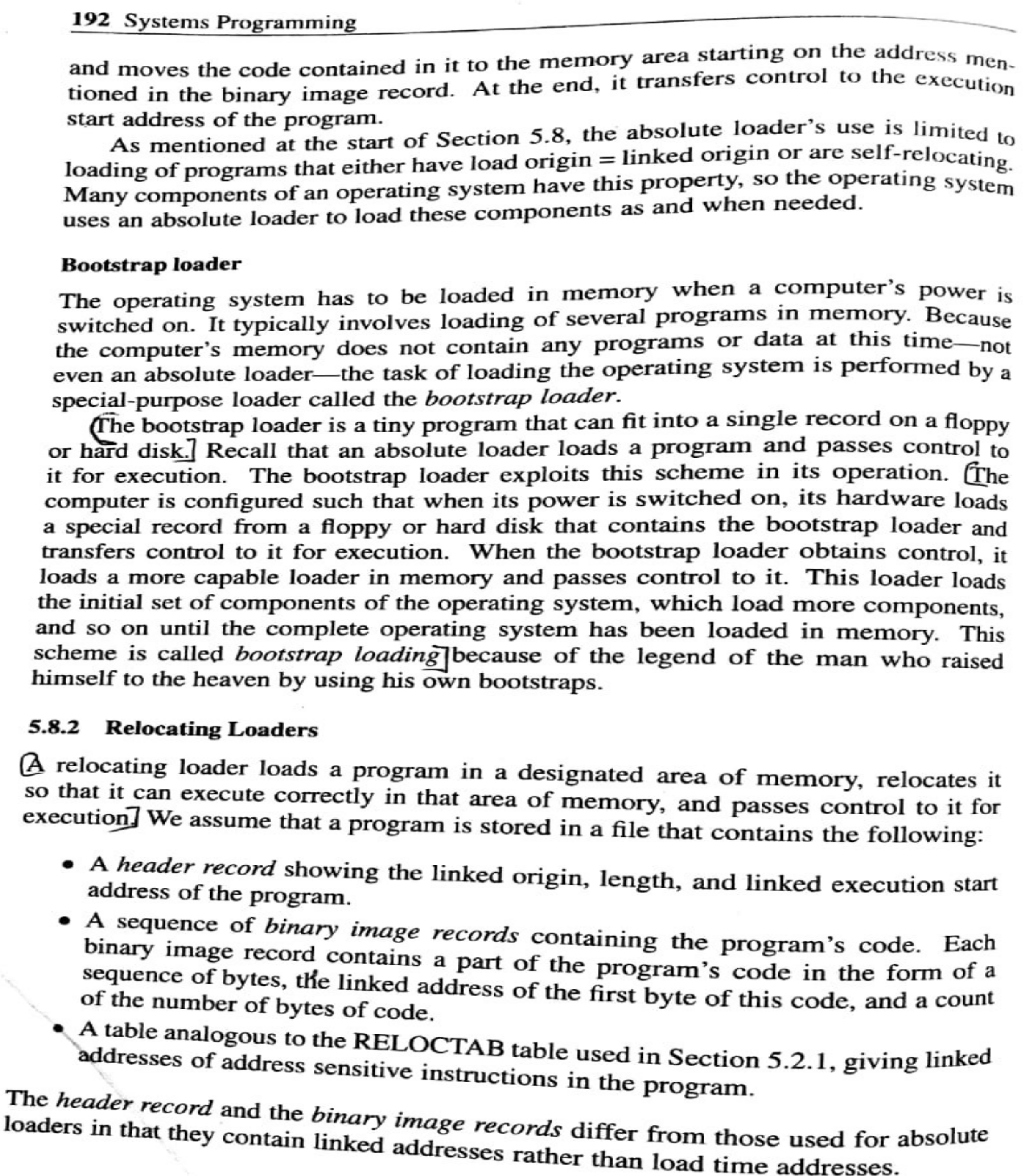


5.8.1 Absolute Loaders

An absolute loader loads a binary program in memory for execution. We assume that the binary program is stored in a file that contains the following:

- A *header record* showing the load origin, length, and load time execution start address of the program.
- A sequence of *binary image records* containing the program's code. Each binary image record contains a part of the program's code in the form of a sequence of bytes, the load address of the first byte of this code, and a count of the number of bytes of code.

The absolute loader notes the load origin and the length of the program mentioned in the header record. It then enters a loop that reads a binary image record



and moves the code contained in it to the memory area starting on the address mentioned in the binary image record. At the end, it transfers control to the execution start address of the program.

As mentioned at the start of Section 5.8, the absolute loader's use is limited to loading of programs that either have load origin = linked origin or are self-relocating. Many components of an operating system have this property, so the operating system uses an absolute loader to load these components as and when needed.

Bootstrap loader

The operating system has to be loaded in memory when a computer's power is switched on. It typically involves loading of several programs in memory. Because the computer's memory does not contain any programs or data at this time—not even an absolute loader—the task of loading the operating system is performed by a special-purpose loader called the *bootstrap loader*.

The bootstrap loader is a tiny program that can fit into a single record on a floppy or hard disk. Recall that an absolute loader loads a program and passes control to it for execution. The bootstrap loader exploits this scheme in its operation. The computer is configured such that when its power is switched on, its hardware loads a special record from a floppy or hard disk that contains the bootstrap loader and transfers control to it for execution. When the bootstrap loader obtains control, it loads a more capable loader in memory and passes control to it. This loader loads the initial set of components of the operating system, which load more components, and so on until the complete operating system has been loaded in memory. This scheme is called *bootstrap loading* because of the legend of the man who raised himself to the heaven by using his own bootstraps.

5.8.2 Relocating Loaders

A relocating loader loads a program in a designated area of memory, relocates it so that it can execute correctly in that area of memory, and passes control to it for execution. We assume that a program is stored in a file that contains the following:

- A *header record* showing the linked origin, length, and linked execution start address of the program.
- A sequence of *binary image records* containing the program's code. Each binary image record contains a part of the program's code in the form of a sequence of bytes, the linked address of the first byte of this code, and a count of the number of bytes of code.
- A table analogous to the RELOCTAB table used in Section 5.2.1, giving linked addresses of address sensitive instructions in the program.

The *header record* and the *binary image records* differ from those used for absolute loaders in that they contain linked addresses rather than load time addresses.