R.P. Goyal

CHAPTER 14

Loaders

14.1 INTRODUCTION

The translated output of an assembler or a linker needs to be further translated or processed by a software that maps the virtual machine used by the assembler into the real target machine so that it will be available at the time of execution. The main task of this translating software is to load an object/executable program from the secondary memory into the primary memory of the target machine and make the program ready for execution. This software also passes the starting address of the program (or, the first instruction) to the computer hardware for execution. This translating software is also a system program, called a *loader*. Therefore, a loader accepts a sequence of machine operation code and data from an executable file and puts them in the primary memory of the computer for execution. In general, the loader must load, relocate, and/or link the object program.

14.2 PROGRAMS IN MEMORY

A loader loads an executable program from a file to the primary memory of the computer but the problem is in deciding where in the primary memory the programs need to be put. In an assembly program, the starting address is either at location zero or some value which modifies the location counter. This may be an acceptable solution for a small system but not a suitable solution for the development of a large system because the program code must be written without any knowledge of where it will eventually be run. The solution of this problem is to introduce a new concept known as *relocation mechanism* in this loading scheme. This mechanism allows the decision about where in the memory the program is to be put to be deferred until after the program has been assembled or compiled.

14.3 PRINCIPLES OF LOADING OPERATION

Loading is the process of making an executable program ready for execution after reading the executable file from secondary storage and appropriately write this content to the primary or virtual memory.

14.3.1 Loading Operations

The mechanism for loading the program uses *program load address* (PLA) as a pointer and stores the contents of the machine operation code (i.e., TEXT records in the executable file) in the memory at the address pointed to by the PLA.

The loader is a system program that performs the loading process for a given input program (executable) for the same target machine as it is obtained from the object modules of that target machine. The output of this process is that it is loaded to the allocated primary memory after proper relocation and to submit the starting allocated address to the hardware. Now the hardware starts to execute this program. Therefore, the basic principle of the loading operation is two fold as given below.

Make the program ready for execution To bring an object/executable program into primary memory with proper relocation.

Start execution The first instruction or the starting address of the program in the primary memory will pass to the computer hardware (i.e., target machine) for execution. This operation is clearly shown in Fig 14.1.

In Fig 14.1, the loader first reads the size of the executable program from the header record of the file and then requests to the memory manager of the operating system for the requisite space to load the executable program in the primary memory, when the loader and the operating system are both in the primary memory. The loader loads the executable program after proper relocation according to the relocation flag and the starting address. When the whole executable program is loaded into the primary memory from the secondary storage, i.e., disk, the loader sends the starting address to the MAR (memory address register) or program counter (PC) or instruction counter (IC) of the target computer hardware, so that computer can start executing the loaded program.

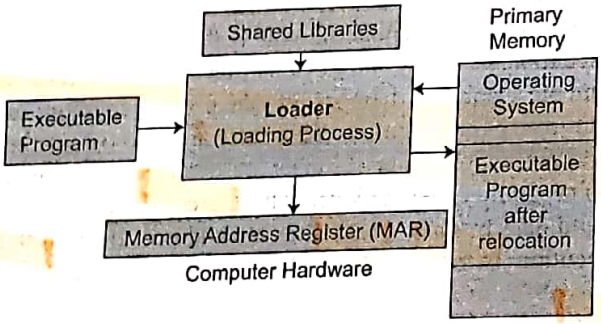


Fig. 14.1 A simple loading process

Note that the loader should take care of location-sensitive (i.e., address-sensitive) instructions in the executable program during the process of loading into the primary memory with respect to the allocated address.

The loader brings the object program into the memory for execution and relocation and modifies the object program so that it can be loaded at an address that may be different from 0.

In summary, we can say that a loader is a translator that translates a stored executable program in secondary memory into another version of executable program into the available primary memory for the same target machine.

14.3.2 Basic Loading Tasks

A different loader can perform different kinds of tasks. Suppose a loader can perform only the loading process. Then it performs the allocation, relocation, and loading tasks. Here, the linker and loader are separate processes. But it is observed that some loader performs both the linking and loading functions and then performs the allocation, linking, relocation, and loading tasks. These tasks are described below.

Allocation Allocate memory for the programs to execute the program.

Linking Resolve symbolic references between the object modules (if any) or run-time library of the system.

Relocation Adjust all address-dependent locations with respect to the corresponding allocated address while loading physically into the primary memory. This may be termed as *address binding*.

Loading Physically place each of the machine instructions of the program for execution and data from the secondary memory into the allocated primary memory.

Note that the primary objective of a loader is to bring a binary image from secondary storage to primary memory after binding the relocatable addresses to absolute addresses.

14.3.3 Loader Procedure

The loader procedure is a sequence of operations as described in the following steps.

Step 1 Reads the header of the executable file to find out the size of the text and data.

Step 2 Creates an address space which is large enough.

Step 3 Copies the instructions and data into primary memory.

Step 4 Copies parameters to the main program onto the stack.

Step 5 Initializes the machines registers and sets the stack pointer.

Step 6 Jumps to a start-up procedure that copies the parameters into the argument registers and calls the main procedure, say, *main*() in C/C++.

14.3.4 Loader Features

Loader features are of two types: machine dependent and machine independent.

The machine-dependent loader features include the task of relocation and linking. On the other hand, the machine-independent loader features are the automatic library search and the different loading options used by a loader.

14.4 LINKERS VS LOADERS

Linkers and loaders perform various related operations but conceptually different tasks.

Program Loading This is a common task performed by them. It is used to copy a program image from hard disk to the main memory in order to put the program in a *ready-to-run* state. In some cases, it performs the allocation of storage space or mapping virtual addresses to disk pages.

Relocation Compilers and assemblers generate the object code for each input module with a given starting address where default value is zero. The process of relocation assigns load addresses to different parts of the program by merging all sections of the same type into one section. The code and data sections of the program are also adjusted so that they point to the correct run-time addresses.

Symbol Resolution Suppose a program is made up of multiple modules, i.e., multiple subprograms, so that the reference of one subprogram to another is made through symbols. The task of a linker is to resolve the references by the following steps:

Step 1 Note the location of the symbol.

Step 2 Patch the object code of the *caller* program.

It is observed that a considerable overlap exists in between the functions of linkers and loaders. One way to think of this is that (a) the loader does the program loading and (b) the linker does the symbol resolution; and both of them can do the relocation.

Note There have been all-in-one linking loaders that do all three functions.

There is no distinct boundary between relocation and symbol resolution techniques because this partition largely depends upon their implementation. Suppose linkers can resolve references to symbols. One way to handle the code relocation is to assign a symbol to the base address of each part of the program and treat relocatable addresses as references to the base address symbols.

One important and unique powerful feature Both linkers and loaders patch object code. This feature is extremely machine specific. This task can lead to baffling bugs if done wrongly.

14.5 LOADING SHARED LIBRARIES FROM APPLICATIONS

Shared libraries can be loaded dynamically from applications even in the middle of their executions. An application program can request a dynamic linker to load and link shared libraries. It is to be executable even without linking those shared libraries at the linking time. *Linux*, *Solaris*, and other systems provide a series of function calls that can be used to dynamically load a shared object. Linux provides *system calls*, such as *dlopen*, *dlsym*, and *dlclose*, that can be used to load a shared object, to look up a symbol in that shared object and to close the shared object, respectively. On Windows, the *LoadLibrary* and *GetProcAddress* functions replace *dlopen* and *dlsym*, respectively.

14.6 DIFFERENT LOADING SCHEMES

There are a number of loading schemes that exist in the computer literature. Some of them are stated below:

1. Sequential and direct loaders
2. Compile and go loader
3. Absolute loaders
   (a) Load-and-go compiler
   (b) Bootstrap loaders
   (c) Boot loaders
4. Relocating loaders
   (a) Binary symbolic subroutine (BSS) loader
   (b) Practical relocating loaders
5. Linking loaders
   (a) Two-pass linking loader
   (b) One-pass linking loader
6. Relocating and linking loaders
7. Linkage editors
   (a) Static linkage editors
   (b) Dynamic linkage editors

8. Overlay
   (a) Overlay generator
9. Binders
10. Dynamic loader
11. Graphics loaders

14.6.1 Loader Design Options

The different options for loader design are given below.

Linking loaders The linking and relocation operations are to be done at the load time.

Linkage editors The linking operation is to be done prior to load time.

Dynamic linking The linking operation is performed at the execution time.

14.7 SEQUENTIAL AND DIRECT LOADERS

Sequential loader It examines sequentially the relocation flag attached to each instruction and data.

Direct loader It operates directly on the relocation and linking directory (RLD).

The differences between the sequential and direct loaders are shown in Table 14.1.

Table 14.1 The difference between sequential and direct loaders

Features	Sequential loader	Direct loader
Perform operations	Allocate, load, link, relocate	Allocate, load, link, relocate
ESD	Same as direct loader	Same as sequential loader
TEXT	Different from direct loader	Different from sequential loader
Structure of MOM	MOM=<ESD,TEXT,RLD>	MOM=<ESD, TEXT>

Note MOM = machine operation module, ESD = external symbol dictionary, and RLD = relocation and linking directory

14.8 COMPILE-AND-GO LOADERS

The *Compile-and-go* loading scheme is one of the simplest loaders. Suppose the assembler is in the primary memory of a computer and the assembled/translated program is directly placed into their assigned memory location of that computer. After the completion of the assembly/translation process, the assembler passes the starting address of the program to the computer hardware (e.g., memory address register (MAR), or program counter (PC)), i.e., the first instruction is sent to the hardware for execution. This kind of loader scheme is shown in Fig. 14.2 and it is known as *compile-and-go* or *assemble-and-go*, or more generally *translate-and-go* loader.

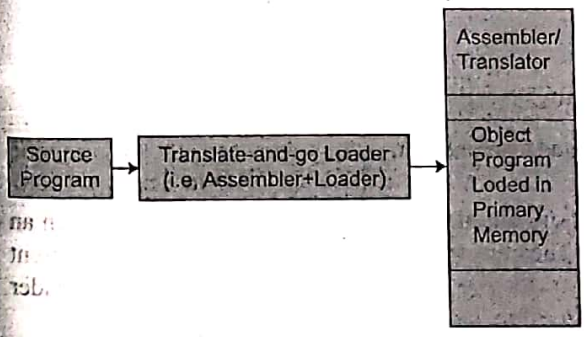


Fig. 14.2 Compile-and-go loader

The algorithm of the *Compile-and-go* loading scheme is described in Algorithm 14.1.

ALGORITHM

Algorithm 14.1 Compile-and-go loaders

Input: Start address

Output: Executable program in primary memory starting from start address

Step 1 **[Input]** Get the starting address of the allocated primary memory.

Step 2 Translate the source program into object program and store them into the primary memory of the machine, starting from the allocated address.

Step 3 Pass this starting address to the computer hardware for execution.

Step 4 **[Termination]** Stop.

Illustration 14.1 Suppose the starting address of the allocated primary memory is a. The source program Ps is translated into the object program Po by the compile-and-go loader and stored into the primary memory starting from a. At the end of the translation, the address a is stored into the memory address register (MAR) or program counter (PC) for execution.

Advantages Some advantages of compile-and-go loader are given below:

1. It is relatively easy to implement.
2. The loader is an additional instruction with the assembler. The purpose of this additional instruction is to transfer the starting address/instruction of the assembled program into the primary memory to the hardware.

Disadvantages Some disadvantages of compile-and-go loader are given below:

1. The assembler will occupy some space in the primary memory and it is unavailable to the object program.
2. For every execution, the translator translates the source program into executable/object program.
3. It is difficult to handle multiple source program segments in different languages.

Use WATFOR FORTRAN compiler

14.9 GENERAL LOADER SCHEMES

In any general loader scheme, the loader should perform the following functions:

1. The loader accepts the assembled/translated machine instructions, data, and other information present in the object program files.
2. The loader also places machine instructions and data in the allotted primary memory in an executable form for the target machine. The object program files are obtained from different high-level and assembly-level languages for the same target machine. A general loader scheme is shown in Fig. 14.3 and its algorithm is described in Algorithm 14.2.

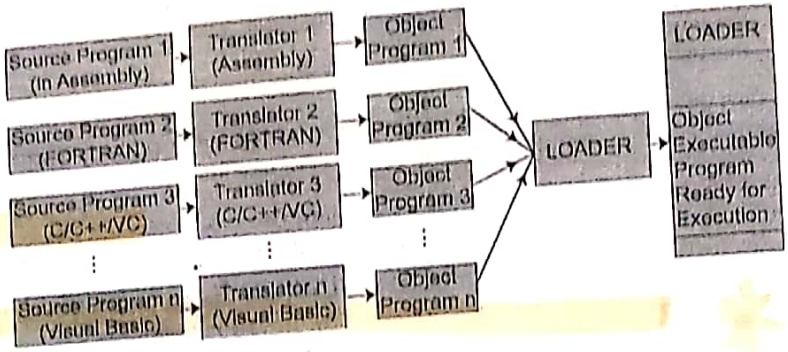


Fig. 14.3 General loader schemo

ALGORITHM

Algorithm 14.2 *General loader scheme*

Input: Linked object program, i.e., executable program file

Output: Executable program in primary memory

Step 1 **[Translation]** Translate ith source program S_i by the translator T_i and produces an object program O_i, for $i = 1, 2, \ldots, n$. Store these object programs in separate files.

Step 2 **[Input]** The loader accepts all the object files where each object file has header information, assembled instructions, and end records.

Step 3 **[Linking]** Resolve the unresolved references (if any) in the object modules.

Step 4 **[Relocation]** Adjust address dependent locations (if any).

Step 5 **[Loading]** Load all these instructions and data into the primary memory at the allocated address.

Step 6 **[Execution]** Pass the starting address of loaded program to the computer hardware for execution.

Step 7 **[Termination]** Stop.

Advantages Some advantages of general loader scheme are given below:

1. The loader software is much smaller than either assembler or any translators, so that more memory is available to the user.
2. Reassembly/recompilation of the programs is not necessary.
3. The translators of the source program produce compatible object program files.
4. It can handle source programs in several different languages.

Disadvantage The loader cannot handle different object modules obtained from different kinds of computers (such as SUN and VAX), i.e., hardware.

14.10 ABSOLUTE LOADERS

The absolute loader is the simplest type of sequential loader. It fits the general model of the loader scheme. In this scheme, the translator (e.g., assembler or compiler) outputs the machine language translation of the source program in almost the same form as in the *assemble-and-go* (or, *compile-and-go*) scheme. The loader accepts this translated output in the form of records (or images), each containing:

1. the text (*instructions, data, or both*) to be loaded,
2. the length of the text, and
3. the starting address.

The task of this loader is to place these records into the primary memory at the location prescribed by the assembler during the process of assembly.

Basic philosophy An absolute loader uses just one machine operation module (MOM) as its input data where TEXT is the only component in MOM, i.e., MOM = < TEXT>.

The allocation is performed either by the compiler generating the MOM or by the programmer. The loading is the only operation performed by the absolute loader.

Note that an absolute loader performs a loading operation, which is basically a loop. During loading, it destroys the contents by loading TEXT over the space occupied even by the loader. So, an absolute loader is a *self-destroyed loader*. Also, no relocatable constants are allowed in MOM. Hence, no relocation is needed. Instruction relocation is performed by *base-register*.

Working principles Assume that the object file structure for an absolute loader is as shown in Fig. 14.4. The first record of this object file is the number of records in the file, second record is the load address of the program, and the last record is also the load address which has to transfer to the hardware for execution. The records between the second and the last record are the actual object code for the program in the form of records.

Number of records
Load address of the program
Object codes for the program
Load address

Fig. 14.4 The file structure of an object file for an absolute loader

The absolute loader reads the first record length of the program. The second record gives the starting address. Now, the loader will load the instructions (i.e., executable code) into the specific and predetermined set of memory locations. The last record is the address to branch for execution.

An algorithm for a typical absolute loader scheme is described in Algorithm 14.3.

ALGORITHM

Algorithm 14.3 Absolute loader

Input: Executable program and program length

Output: Executable program in primary memory

Step 1 **[First Record]** Read the number of instruction, N.

Step 2 **[Second Record]** Read the *load-address*.

Step 3 Perform this step for *N* times.

1. Read the current machine instruction.
2. Store this instruction into the primary memory at the *load-address*.
3. Increment *load-address*.

Step 4 **[(N+3)th record]** Read the *execution address*.

Step 5 Load this *execution address* to the *program counter* (i.e., PC or MAR) of the computer for execution.

Step 6 **[Termination]** Stop.

Advantages Some advantages of an absolute loader are given below.

1. The absolute loader makes more memory available to the user since the assembler is not in the memory at the load time.
2. Absolute loaders are simple to implement.

Disadvantages Some disadvantages of an absolute loader are given below.

1. The programmer must specify the loading address in the assembly program. The loading address means where the program is to be loaded.
2. In the case of multiple subroutines, the programmer must remember the address of each subroutine. Assemblers use that absolute address explicitly in other subroutines to perform subroutine linkage.
3. Absolute loaders are not very common.

Example 14.1 An executable file is shown in Fig. 14.5 for a routine related to a personal computer. The content of this file is absolute in nature. Show the memory map for this program.

```
00 0E      ← Number of records
06 00      ← Load address of the program
FA
33 C0
8E D0
BC 00 76
8B F4
50
07
1F
FB
FC
BF 00 06
B9 00 01
F2
A5
06 00      ← Execution Address
```

Fig. 14.5 A file of an object program (The number of records in this file is 17 and this code is loaded at a fixed address starting from 0060_{16})

Solution All the contents of the file are absolute. So, an absolute loader is suitable for this file. We apply Algorithm 14.3 and the memory map for this program as shown in Fig. 14.6.

Primary Memory Address	Memory contents
0600	FA 33 C0 8E
0604	D0 BC 00 76
0608	8B F4 50 07
060C	1F FB FC BF
0610	00 06 B9 00
0614	01 F2 A5

Fig. 14.6 Memory map of the object file in Fig. 14.5

Illustration 14.2 **Absolute loader for the hypothetical machine**

Source Input To Absolute Assembler — Main program

```
HYPO   START   50
       BALR    5,0
       USING   HYPO+2,5
       :
       L       15, ASINE
*      Call    SINE
       BALR    14,15
       :
ASINE  DC      F'350'
*      Address SINE
       END
       SINE subroutine
SINE   START   350
       USING   *,15
       :  {Compute Sine}
       BR      14   Return
       END
```

(a)

Source Input In Memory To Absolute Assembler — Main program

```
50    BALR   5,0
52    :
70    L      15,142(0,5)
74    BALR   14,15
76    :
194   F'350'
198
      SINE subroutine
350
      :
      :
426   BCR    15,14
430   :
```

(b)

Fig. 14.7 An example of absolute loader: (a) source input to absolute assembler and (b) source input to memory

The programmer must be careful not to assign two subroutines to the same or overlapping memory locations.

In Fig. 14.7, the main program is assigned to locations 50–197 and the SINE subroutine is assigned locations 350–428 if changes were made to the HYPO program, so that it increased the length of the program to more than 300 bytes. The end of program HYPO (at 50 + 300 = 350) would overlap with the start of the SINE subroutine (at 350). It would then be necessary to assign SINE to a new location by changing its START pseudo-operation and reassembling it. Furthermore, it would also be necessary to modify all other subroutines that referred to the address of SINE. In some situations where a number (10 or 12) of subroutines are being used by the main program, this manual *shuffling* can be complex, tedious, and wasteful of primary memory.

Loader functions are accomplished as shown in Table 14.2 in an absolute loading scheme.

Table 14.2 Tasks of an absolute loader

	Task	Task performed
1.	Allocation	by programmer
2.	Linking	by programmer
3.	Relocation	by assembler
4.	Loading	by loader

14.10.1 Some Absolute Loaders

Absolute loaders are not very common. Some important absolute loaders are discussed below.

Load-and-Go compilers These are compilers for the students. Here, the program is compiled directly and generates code in terms of the absolute locations where these addresses are reserved for student's use. Now this compiled program is then loaded into the primary memory starting from the given absolute address and immediately executed. This software can handle only small user's program. Every time reassembling or recompiling of the source program is to be done for execution. This is a costly method but it will be cost effective if the program runs infrequently.

Bootstrap loaders The term *bootstrap* is used for the following. If the loading process requires more instructions, then it can read in a single record. This is the first record and it causes the reading of others. These in turn can cause the reading of still more records. The first record (records) is generally termed as a *bootstrap loader*. It is a very small and simple loader. Its only job is to load the full loader, which in turn loads the other essential pieces of the system software. The bootstrap loader loads the full loader into fixed and known locations in primary memory. This absolute location is to specify by the system software designer. In the present architecture of computer, bootstrap loaders are always placed in *read-only-memory* (ROM). The process of initiating the execution of a bootstrap loader in ROM is usually accomplished by pushing a physical START button. This operation is known as *cold start* or *dead start*. It performs the following tasks:

1. The bootstrap loader loads the full loader.
2. The full loader loads the operating system (or, part of it).
3. The operating system then asks what should be done next and loads the appropriate software units.

Sometimes, it is observed that such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system. This includes the operating system itself and all stand-alone programs that are to be run without an operating system.

Boot loaders These are the pieces of system software placed in the ROM and startup operating systems when the computer has *power on*. Normally, boot loaders do their job automatically. However, when two or more operating systems are installed on the computer, it is necessary to configure the boot loader so that one can choose which operating system to load. This is often called *dual-booting*. Here, a boot loader gives an opportunity to pass information to the operating system when it boots up.

Boot loaders run before any operating system is loaded. They are independent of operating systems. Any boot loader should be able to boot any operating systems, such as Windows, Linux, and UNIX.

14.11 RELOCATING LOADERS

The address part of an instruction in a relocatable program will not bound by a relocating translator during the period of translation. These translated instructions can be loaded into any section of primary memory for execution with the help of another program known as *relocatable loader*. This relocatable loader is also known as *relocating loader* or *relative loader*.

Illustration 14.3 In this illustration, we describe the task performed by a relocating loader. Suppose the assembly code for a hypothetical machine is described by *U* and *V*, where the operation code (i.e. op code) for the machine operations *U* and *V* are hexadecimal #0 (4 bits or 1/2 bytes) and #00 (8 bits or 1 byte) memory maps starting at location #0000 and at #0100 as shown in Fig. 14.8. In this assembly program, LAB is a label variable and its location in the memory depends upon the starting address of the program. The effect of relocating a program is illustrated in Fig. 14.8.

Hypothetical Assembly code	Memory map loaded at #0000	Memory map loaded at #0100
LAB:U 10	0000: 0A	0100: 0A
V LAB	0001: 00	0101: **00** relocated
	0002: 00	0102: **10** relocated
V 0	0003: 00	0103: 00
	0004: 00	0104: 00
V #0100	0005: 00	0105: 00
	0006: 10	0106: 10
V LAB	0007: 00	0107: **00** relocated
	0008: 00	0108: **10** relocated

Fig. 14.8 The effect of relocating a program

In Illustration 14.3, the loader performs the final binding of code to specific memory locations. It adjusts or relocates all the pointers used in the program where the pointers are varying from one part of the loaded program to another. It is clear from Fig. 14.8 that adjustment is quite simple. Relocation involves (i) adding a constant, (ii) the relocation base, and (iii) each address that refers to another location in the same program. The relocation base is usually the same as the address at which the first byte of the program is loaded. Thus, if the program is loaded at address 0020, the relocation base is 0, and the addresses within the program are not modified. If the program is loaded at address 010020, the relocation base is 10020 and this constant is added to each address within the program.

As a result of discussion in Illustration 14.3, there are four different possible modes of loading scheme that can be identified.

Relocatable values may be stored in relocatable locations. It takes place when a branch instruction has a relocatable address for an operand, where that branch instruction itself will be stored in the relocatable part of the program.

Relocatable values may be stored in absolute locations. It takes place when a pointer to a relocatable part of the program must be stored in a fixed memory address, e.g., an input-output device interface location.

Absolute data may be stored in relocatable locations. It takes place when a constant or an operation code of an instruction is a part of the relocatable part for the program.

Absolute data may be stored in absolute locations. It must be done when a constant is stored in a dedicated memory location such as an input-output device interface.

Basic philosophy Relocatable loader accepts just one machine operation module (MOM) as the input. The MOM has all its three components—ESD, TEXT, and RLD. MOM is not pre-allocated. The loader/linker allocates, loads, and/or relocates the MOM. The only entry in the ESD is created for the program name.

Working principles Suppose the assembler translates the symbolic code in the assembly program not into *absolute addresses* but into *relative address*. The address will be relative to some known point, such as the start of the program. Assume that the default starting location is 0.

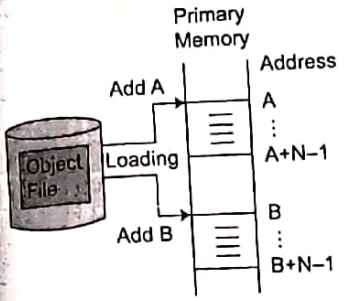


Fig. 14.9 Relocatable loader

A translator translates a source program into the corresponding N word machine language object program and is stored into the disk. This object program also contains relocatable addresses. Now, the relocatable loader will load this program from disk into the available physical primary memory at the start location A (say). This program will occupy the locations $A, A+1, \ldots, A+N-1$ as shown in Fig. 14.9. Here, the loader simply adds the constant A to each address part of the instruction as it is loaded into memory. If the program moves to memory location $B, B+1, \ldots, B+N-1$, simply add a constant B to each instruction instead of the value A.

Make a program relocatable The following two steps will make the program relocatable.

1. Both the location counter values and the address field(s) of all instructions must be assigned values relative to a fixed base point, typically the beginning of the program (say, location 0).
2. The assembler (translator) must append to each instruction a set of special flags, called *relocation bits*, telling how that instruction and its address are to be relocated at load time. The set of all relocation bits is called a *relocation dictionary*.

Advantages Some advantages of relocating loaders are given below:

1. Avoids possible reassembling of all subroutines when a single subroutine is changed
2. Performs the tasks of allocation and linking for the programmer

Disadvantages Some disadvantages of relocating loaders are given below:

1. Difficult to implement
2. Algorithm depends upon the file structure of the object program.
3. This relocating loader is slower than absolute loader since it checks the flag.

Algorithms Algorithm for the relocating loader is not unique. It depends on the relocatable object file structure, i.e., it depends on the translator of the program. Here, we discuss three algorithms.

R – Relative
B – Byte
W Word
A → Absolute

Algorithm I Assume that each record of the object file has four fields *ta*, *a*, *tv*, and *v* as shown in Fig. 14.10, where *ta* = tag on the address in character such that $ta \in \{R, A\}$, *a* = The address in numeric (decimal), *tv* = tag on the value in character such that $tv \in \{R, B, W\}$, and *v* = the value in hexadecimal.

Our objective is to design an algorithm that will read an instruction from the object file and relocate if necessary according to the given base address and then write it in the primary memory at the computed address.

ALGORITHM

Algorithm 14.4 Loader for a simple tagged textual object code.

Assume available memory is unlimited.

Input: Base address and executable program

Output: Executable program in the primary memory

Step 1 **[Input]** Get *base* the base address from the operating system.
Set *loadaddr* ← *base*
Step 2 Read a record (*ta*, *a*, *tv*, *v*)
Step 3 If *ta* = 'R' then set *addr* ← *a* + *base*
Step 4 If *ta* = 'A' then *addr* ← *a*
Step 5 If *tv* = 'R' then set *value* ← *v* + *base*;
Write value at the memory location *addr* and *addr*+1;
Set *addr* ← *addr* +2;
Step 6 If *tv* = 'B' then
Write *v* at the memory location *addr*, and
Set *addr* ← *addr* +1
Step 7 If *tv* = 'W' then
Write *v* at the memory location *addr*, and *addr* +1
Step 8 *Repeat* Step 2 to Step 7 *until* not end-of-file
Step 9 **[Load to the hardware]** *LC* ← *loadaddr*
Step 10 **[Termination]** Stop.

Note 1. When the primary memory of a computer is limited then the loader accepts the relocation base as a parameter as input. When the loading task finishes, it returns the address immediately after the highest relocatable location it loaded. Thus, the caller learns the size of the memory region used by the relocatable part of the loaded program. This feature is necessary for loading the multiple object/executable files consecutively in the primary memory. This will link together to make one large executable program.
2. This loader gives a pointer to the starting location in which the code is to be loaded and returns a pointer to the first location after the relocatable part of the loaded code.

We illustrate the procedure of Algorithm 14.4 in Illustration 14.8.

Illustration 14.8 Suppose an objective file of a program is shown in Fig. 14.10. The record structure of the object file is shown in Fig. 14.10. The memory map of the object program in Fig. 14.10 is shown in Fig. 14.11 when the base address is 0128 (say).

Object code in a file (only TEXT)

```
R0000 B05
R0001 R0003
P0003 W0103
A0100 B05
A0101 R0003
A0103 W0103
```

where R = Relative, B = Byte, W = Word, A = Absolute

Fig. 14.10 Relocatable assembly and a simple relocatable object code

Consider the first instruction R0000 B05 as shown in Fig. 14.10. Here, *ta* = R, *a* = 0000, *tv* = B, and *v* = 05 (as shown in Fig. 14.10). Therefore, the memory map of this instruction is as follows: This instruction will store at an address 0000 + 0128 = 0128. The content of the memory location is *B*05, i.e., it will occupy a byte of memory for 05_{16}. The next instruction R0001 R0003 will be stored at location 0128 + 0001 = 0129, where the content of the location 0129 is 00 and the content of the next location 0130 is 83. Since $(0128)_{10} + (0003)_{10} = (1000\ 0011)_2 = (0083)_{16}$. On the other hand, the 4th instruction 'A0100 B05' will be stored at location 0100 and the content of that location is $(05)_{16}$. Similarly, we can explain other instructions.

The task of the loader for a simple tagged textual object code can be written in an algorithmic form as shown in Algorithm 14.4.

128_{10} ⇒
+3
131
1000 0000
11
1000 0011
83_H

Memory Address	Memory Map (in hexadecimal)
0100	05
0101	00
0102	83
0103	01
0104	03
⋮	⋮
0128	05
0129	00
0130	83
0131	01
0132	03

Fig. 14.11 Object code in primary memory when base address=0128

Algorithm II Another algorithm for relocating loader is designed based on the object file structure, which is described in Format 2 (see Fig. 5.14) of Section 5.6.3. The first record indicates the file description and the last record is the execution address of the program. The remaining records starting from second to last but one has the same structures.

Here, the relocation bits (e.g., *a*, *r*, *e*) are used in the object program with the following three values 0, 1, and 2. The meaning of these values is described in Table 14.3.

Table 14.3 Values of relocation bits and their meaning

Values	Meanings
0	Do not relocate this address. The logical address in the program is either a fixed constant or an absolute address that should not change when the program is loaded.
1	The logical address is given as a relative value. Relocate this address relative to the program origin point, *a*. *Physical address = Program address+a*
2	The logical address is a relative value. Relocate this address relative to the data origin point, *d*. *Physical data address = Data address+d*

Now, the algorithm based on this object file is shown in Algorithm 14.5

ALGORITHM

Algorithm 14.5 A typical relocating loader

Input: Program section length, data section length, and executable program file.

Output: Executable program in primary memory.

Step 1 **[Get header]** Read Program Section Length (PSL) and Data Section Length (DSL) from the object file.

Step 2 **[Memory allocation]** *AllocateMemory*(*x*,*y*) ← Allocate memory of size *x* and it returns the starting address *y* as

AllocateMemory(PSL,*ProgAddr*)

AllocateMemory(DSL,*DataAddr*)

P ← ProgAddr

D ← DataAddr

Step 3 Repeat Step 4 to Step 6 until all instructions have been processed.

Step 4 Read LC, Instruction, *r*, *a*

Step 5 **[Binding phase]**

Case r of

0: no action

1: AddressField ← AddressField + ProgAddr

2: AddressField ← AddressField + DataAddr

End

Step 6 **[Loading phase]**

Case a of

0: **Error** -- Illegal absolute references

1: Store the instruction in memory location LC+P

Increment P: P ← P + 1

2: Store the instruction in memory location LC+D

Increment D: D ← D + 1

End

Step 7 **[Execution phase]** Read execution address, *e*

Case e of

0: PC ← execution address

1: PC ← execution address+ ProgAddr

2: PC ← execution address+ DataAddr

End

Step 8 **[Termination]** Stop.

Illustration 14.9 Consider the following object file according to the file structure described in Table 14.3.

0105		0000	
0000	33	1	0
0001	DB	1	0
0002	B8	1	0
0003	00 01	1	0
0005	03	1	0
0006	C3	1	0

0007	E9	1	0
0008	01 00	1	1
...	...	...	...
0100	8B	1	0
0101	D8	1	0
0102	E9	1	0
0103	00 02	1	1
0000	1		

Suppose the operating system supplies the starting load address and it is 0600_{16} (say), then the memory map by Algorithm II for this object file is as follows.

```
0600 33 DB B8 00
0604 01 03 C3 E9
0608 07 00
...
0700 8B D8 E9 07
0704 02
```

And the execution address in LC is $0000_{16} + 0600_{16} = 0600_{16}$.

Algorithm III Algorithm III for relocating loader is designed based on the object file structure described in Format 3 (see Fig 5.15) of Section 5.6.3. The first record is the start address and the last record is the execution address of the program. But the fields of the remaining records (say, the ith record) depends on that of neighboring records (such as $(i-1)$th or $(i+1)$th record) as described in Fig. 5.15. Based on this object file structure, an algorithm for a relocating loader is shown in Algorithm 14.6.

ALGORITHM

Algorithm 14.6 A typical relocating loader

Input: Program length, start address, and executable program file

Output: Executable program in primary memory

Step 1 **[Get header]** Read Program Length (PL)

Step 2 Get start memory address *Addr* of size PL

Step 3 $LC_0 \leftarrow Addr$

Step 4 for $n = 1$ to PL do

Read LC_n, d_n, B_n, T_n

$LC \leftarrow LC_n + Addr$

$k \leftarrow d_n$

for j = 1 to k do

if $b_{nj} = 1$ then $(hh)_{nj} \leftarrow (hh)_{nj} + Addr$

Store modified T_n at LC in the primary memory

Step 5 Read execution address

PC ← execution address + *Addr*

Step 6 **[Termination]** Stop.

Illustration 14.10 Consider the following object file according to the file structure described in Format 3.

```
0105
0000    2    00     33 DB
0002    3    000    B8 00 01
0005    2    00     03 C3
0007    3    011    E9 01 00
...                 ...
0100    2    00     8B D8
0102    3    011    E9 00 02
0000
```

Suppose the operating system supplies the starting load address and it is 0600_{16} (say), then the memory map by Algorithm III for this object file is as Illustration 14.9.
The execution address in LC is $0000_{16} + 0600_{16} = 0600_{16}$.

14.12 PRACTICAL RELOCATING LOADERS

A number of standard relocating loaders are available in system software. We shall discuss some popular relocating loaders, of which some may be obsolete today.

Binary symbolic subroutine (BSS) loader It is a relocating loader scheme that was used in the IBM 7049, IBM 1130, GE 635, and UNIVAC 1108.

The output of a relocating assembler using a BSS scheme is an object file corresponding to the source assembly programs. This object file contains both object code and information about all other programs it references. In addition, information like relocation and the loader information (e.g., the length of the program and the transfer vector) is necessary to locate them in this program. Also, this information needs to be changed if it is to be loaded in an arbitrary place in the primary memory, which comprises of the locations which are dependent on the memory allocation. In this context, we define the term '*transfer vector*'.

Transfer vector It is a vector of subroutine names and its corresponding address referred by the source program.

Subroutine Name	Sub1	Sub2	...	Sub*n*
Address	Addr1	Addr2	...	Addr*n*

Fig. 14.12 Transfer vector

The typical structure of a transfer vector is shown in Fig. 14.12. It is a two-dimensional array in which one cell of an element contains the subroutine names and the other cell contains its address.
For each source program, the assembler generates an output that is a text (machine translation of the program) prefixed by a transfer vector, where the transfer vector consists of addresses containing names of the subroutines referenced by the source program. The BSS loader loads (i) the text, (ii) the transfer vector, and (iii) each subroutine specified in the transfer vector.

During loading, the loader identifies and relocates the address portion of each instruction. This problem is often solved by the use of *relocation bits*. The assembler associates a bit with each instruction and/or address field.

Advantages Some advantages of the BSS loader are given below:

1. Four functions (such as allocation, linking, relocation, and loading) of the loader are performed automatically using the transfer vector.

2. The relocation bits are used to solve the problem of relocations.
3. The transfer vector is used to solve the problem of linking.
4. The length of the program is used to solve the problem of allocation.

Disadvantages Some disadvantages of BSS loader are given below:

1. The transfer vector linkage is only useful for transfers, but it is not well-suited for loading or storing external data (data located in another procedure segment).
2. The transfer vector increases the size of the object program in memory.
3. The BSS loader does not facilitate access to data segments that can be shared.

Relocating loaders for modern computers Most modern computers have a relocating loader. It is a primary tool used by the computer when a user requests to the operating system of the computer that a program to be execute (i.e., run). Suppose a user types a command: *RUN X* from the keyboard, or double clicks on the icon for an executable file X. The operating system accepts this request and runs the loader program with the file X as input before control is transferred to the starting address of the loaded program. There are many variations on the practical loading process.

When good hardware facilities are available for run-time relocation, a simple (non-relocating) loader may be used to run user programs. Other systems may require separate tools to load a program and to run that program. There is also a broad range of syntactic alternatives in the command language.

In the case of the UNIX operating system, the command interpreter shell encounters a command that is not listed in its symbol table of built-in commands. Then it assumes that the command name is the name of an object file and the loader attempts to load that object file and run it. Actually, there are many different UNIX shells: (i) the *Bourne shell* (*sh*), (ii) the *C shell* (*csh*), (iii) the *Bourne-again shell* (*bash*), etc., but all of them have this characteristic. Most of the common UNIX commands are not built into the shell, but rather, are simply the names of object files found in the directory */usr/bin* or other directories in the search path, the list of directories that the shell checks when it tries to open a file.

14.13 LINKING LOADERS

Conceptually, the *linking* and *relocation functions* are different but these two functions are often implemented concurrently in the design of loader/linker. Here, each segment is read for linking/ loading functions and both the functions require inspection of the relocation bits. Now, a clever programming technique can avoid this duplication of effort if linking can be deferred until loading. Therefore, a loader that first loads, links and then relocates segments is known as *linking loader*.

At present, a linking loader is one of the most popular loading schemes used in computers. It is also known as a *direct-linking loader* or a *general relocatable loader*. Some advantages of the linking loader are given below:

1. It allows the programmer multiple procedure segments and multiple data segments.
2. It gives complete freedom in referencing data or instruction contained in other segments.
3. It provides flexible inter-segment referencing and accessing ability, while at the same time allowing independent translation of programs.

Disadvantages Some disadvantages of the linking loader are given below:

1. It is necessary for allocation, relocation, linking, and loading.
2. All the subroutines are required each time in order to execute a program.
3. It is time consuming.

14.13.1 Two-Pass Linking Loader

A two-pass linking loader performs the loading task in two passes, which means that the loader has to read/scan the input object program twice. In this section, we shall design a two-pass linking loader algorithm.

Working principle A two-pass linking loader is shown in Fig. 14.13. In this, the Pass I of the two-pass linking loader accepts object files and *program load addresses* (PLA) (initial as well as other) from the operating system as an input. It produces a global external symbol table (GEST) and the copies of the object programs as output. This output as well as PLAs are the input to the Pass II of the two-pass linking loader. During the Pass II, the object programs are loaded to the primary memory after proper translation of addresses with the help of a scratch pad which is known as *local external symbol array.*

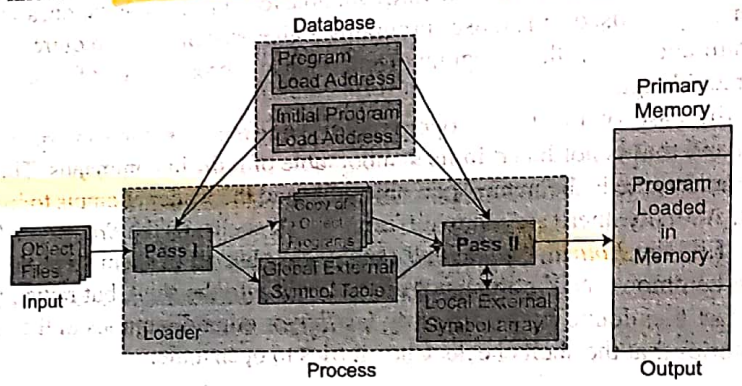


Fig. 14.13 Two-pass linking loader

Outline of Algorithm

Pass I Builds the external symbol table (EST) by combining the external definition tables (EDT) of the modules.

Pass I of a two-pass linking loader performs the following tasks:

1. Allocates storage
2. Determines the relocation constants
3. Relocates global relative symbols
4. Builds the global symbol table

It can defer inspection of program texts and use tables.

Pass II It resolves the external symbols, performs relocation and loading, and it also starts execution. The Pass II of this two-pass linking loader performs the following tasks.

1. Loads the TEXT into the primary memory
2. Relocates and loads relative symbols as it proceeds
3. Adjusts and relocates fields that include externally defined symbols

Design of algorithm Input is the object codes for each module where the file structure of this object module is shown in Fig. 14.14.

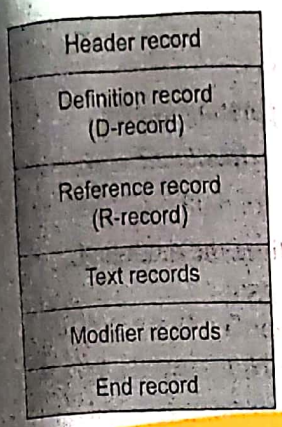


Fig. 14.14 File structure for object modules

The fields of these records (see Fig. 14.14) are given below.

Header record Program name, Start address, and Program length.

Definition record External Definition Table (EDT) where each entry of EDT contains a symbol and its (relative) address.

Reference record External Reference Table (ERT) for which each entry of ERT contains a symbol and the (relative) addresses where the address of the symbol is needed.

Text records Each record must have its start address, length, object code, and checksum.

Modifier records:

1. The start location of the address field is to be modified relative to the beginning of the program.
2. The length of the address field is to be modified in half-byte (in hexadecimal form).

End record Start location of the first module.

Assumptions We consider the following assumptions for the design of a two-pass loader.

1. Only one module specifies a starting address in its end record.
2. Modules are loaded successively with no gaps.
3. This algorithm ignores reference records.

Notations are described in Table 14.4 and these are used in Algorithm 14.7.

Table 14.4 Notations

Notations	Description
EST	External Symbol Table
X	Load address variable
Saddr	Start address
M	Module name
L	Length of the module
D-record	It contains a symbol and its (relative) address
Y	A symbol in the record
R	Relative address
S	Start address specified in the record
N	Number of bytes in the record.
Q	Modifier symbol
A	Address of Q specified in the record.
Z	Address of symbol Q from EST
B	Address specified in the end record

— ALGORITHM —

Algorithm 14.7 Two-Pass Linking Loader

Pass I

Step 1 **[Initialization]** Set EST ← empty.

Step 2 **[Input]** Get load address X for the first module from the operating system.
Set *Saddr* ← X.

Step 3 Get the next module.

Step 4 Read header record and obtain module name, M, and length of the module, L.

Step 5 Check multiple defined symbol and Insert (M, *Saddr*) into EST.

Step 6 for each record of module M
do
If (it is a D-record) then
(a) The symbol, Y and its relative address, R is in the D-record.
(b) Check for multiple defined symbol and
Insert (Y, *Saddr*+R) into EST.
enddo

Step 7 Compute start address for the next module
Saddr ← *Saddr* + L

Step 8 Repeat Step 3 to Step 7 until all the modules are processed.

Step 9 Go to Pass II.

Pass II

Assume that the variable *LoadAddr* gives the address at which the execution of the module must begin.

Step 1 **[Input]** Get the load address X from Pass I and
Set *Saddr* ← X.

Step 2 Get the next module

Step 3 Read header record to get module name, M and its length, L.

Step 4 for each record of module M
do
If (it is a text record) then
(a) S ← the start address specified in the record and
N ← the number of bytes in the record.
(b) Load the N bytes in the record starting from address *Saddr* + S.
else
if (it is a modifier record) then
(a) Q ← the modifier symbol and A ← the address of Q specified in the reco
(b) Find the address Z of symbol Q from EST.
check flag and then Add (or subtract) Z to (from) the value stored
at the address *Saddr* + A.

ALGORITHM

Algorithm 14.7 Two-Pass Linking Loader

Design of algorithm Input is the object codes for each module where the file structure of this object module is shown in Fig. 14.14.

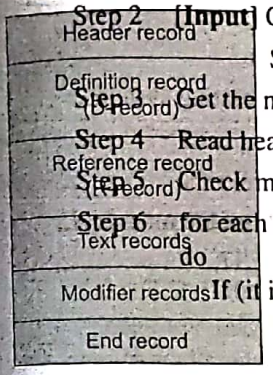


Fig. 14.14 File structure for object module

Each of these records (see Fig. 14.14) are given below.

Header record Program name, Start address, and Program length.

Definition record External Definition Table (EDT) where each entry of EDT contains a symbol and its (relative) address.

Reference record External Reference Table (ERT) for which each entry of ERT contains a symbol and the (relative) addresses where the address of the symbol is needed.

Text records Each record must have its start address, length, object code, and checksum.

Modifier records :

1. The start location of the address field is to be modified relative to the beginning of the program.
2. The length of the address field is to be modified in half-byte (in hexadecimal form).

End record Start location of the first module.

Assumptions We consider the following assumptions for the design of a two-pass loader.

1. Only one module specifies a starting address in its end record.
2. Modules are loaded *absolutely* with the address at which the execution of the module must begin.
3. This algorithm ignores reference records.

Notations are described in Table 14.4 and these are used in Algorithm 14.7.

Table 14.4 Notations

Notations	Description
EST	External Symbol Table
X	Load address variable
Saddr	Start address
M	Module name
L	Length of the module
D-record	It contains a symbol and its (relative) address
Y	A symbol in the record
R	Relative address
S	Start address specified in the record
N	Number of bytes in the record
Q	Modifier symbol
A	Address of Q specified in the record.
Z	Address of symbol Q from EST
B	Address specified in the end record

Pass I

Assumptions are made that *Saddr* is given with the address at which the execution of the module must begin.

Step 1 **[Input]** Get the load address X from Pass I and
Set *Saddr* ← X

Step 2 Get the next module.

Step 3 Read header record to get module name, M and its length, L.

Step 4 for each record of module M
do
If (it is a Text record) then
(a) S ← the start address specified in the record and
N ← the number of bytes in the record.
(b) Load the N bytes in the record starting from address *Saddr* + S.
else
if (it is a modifier record) then
(a) Q ← the modifier symbol and A ← the address of Q specified in the record.
(b) Find the address Z of symbol Q from EST.
check flag and then Add (or subtract) Z to (from) the value stored at the address *Saddr* + A.

Step 5 Check multiple defined symbol and Insert (Y, *Saddr*+R) into EST.

Step 6 for each record of module M
do
If (it is a D-record) then
(a) The symbol, Y and its relative address, R is in the D-record.
(b) Check for multiple defined symbol and
Insert (Y, *Saddr*+R) into EST.

Step 7 Compute start address for the next module
Saddr ← *Saddr* + L

Step 8 Repeat Step 3 to Step 7 until all the modules are processed.

Step 9 Go to Pass II

Pass II

Assumptions are made that *Saddr* is given with the address at which the execution of the module must begin.

Step 1 **[Input]** Get the load address X from Pass I and
Set *Saddr* ← X

Step 2 Get the next module.

Step 3 Read header record to get module name, M and its length, L.

Step 4 for each record of module M
do
If (it is a Text record) then
(a) S ← the start address specified in the record and
N ← the number of bytes in the record.
(b) Load the N bytes in the record starting from address *Saddr* + S.
else
if (it is a modifier record) then
(a) Q ← the modifier symbol and A ← the address of Q specified in the record.
(b) Find the address Z of symbol Q from EST.
check flag and then Add (or subtract) Z to (from) the value stored at the address *Saddr* + A.

Step 5 Check multiple defined symbol and Insert (Y, *Saddr*+R) into EST.

Step 6 for each record of module M
do
If (it is a D-record) then
(a) The symbol, Y and its relative address, R is in the D-record.
(b) Check for multiple defined symbol and
Insert (Y, *Saddr*+R) into EST.

Step 7 Compute start address for the next module
Saddr ← *Saddr* + L

Step 8 Repeat Step 3 to Step 7 until all the modules are processed.

Step 9 Go to Pass II

else if (it is an end record) then

(a) If an address B is specified in the end record then

Set *LoadAddr* ← *Saddr* + B.

Saddr ← *Saddr* + L.

enddo

Step 5 **[Ready for Execution]**

Start executing the program from address *LoadAddr*.

Step 6 **[Termination]** Stop.

14.13.2 One-Pass Linking Loader

A one-pass linking loader performs the loading task in one pass, which means that the loader has to read/scan the input object program only once. In this section, we shall provide an outline for the design of an one-pass linking loader.

Working principle The one-pass linking loader performs all operations that are specified in a two-pass linking loader in one single pass. In this context, the one pass linking loader uses a cleaver programming technique which is augmented by *following-chains* of forward references.

Outline of algorithm The one-pass linking loader performs the following tasks:

1. Allocates storage
2. Determines the relocation constants
3. Relocates global relative symbols and builds the global symbol table
4. Loads the TEXT
5. Relocating load relative symbols as it proceeds
6. Adjusts and relocates fields that include externally defined symbols using global symbol table, otherwise puts a link to the symbol in the text as well as in the table

14.14 RELOCATING LINKING LOADERS

File of object generated (FOG) can be defined as a sequence of different *machine operation modules* (MOM). Each MOM is a combination of three tuples such as *external symbol directory* (ESD), text part of the program (TEXT) and *relocation linking directory* (RLD). This can be written as

$$FOG = < MOM_1, MOM_2, \ldots, MOM_n > \text{ and}$$

$$MOM_i = <ESD,TEXT,RLD>, \text{ for } i = 1, 2, \ldots, n$$

The algorithm for general loading scheme can also be used as the specification and the implementation of relocating linking loaders.

The relocating linking loaders are classified based on the nature of the produced output.

1. When a loader generates an executable image in the primary memory, then it is termed as a *relocating linking loader.*
2. A loader generates a file on which all machine operation modules are allocated and linked together into an executable image on a disk. This is termed as *linkage editor.*

14.14.1 Linkage Editors

A linkage editor accepts binary symbolic output as input and produces a relocatable binary output. Then, a relocating loader accepts this relocatable binary output as input and produces absolute binary output. Also, the linking loader accepts the binary symbolic output of the translator as input and produces an absolute binary output. This absolute binary output is directly executable by the target machine.

Linkage editors can be divided into two major categories depending upon the time and the mechanism used to perform four operations such as allocation, resolution, relocation and linking—static linkage editor and dynamic linkage editor.

Static linkage editor A static linkage editor performs the following two-step operations.

Step 1 It performs completely all the operations provided in the modules processed by it.

Step 2 It generates a machine operation module (MOM) as required by an absolute loader, i.e., it generates an executable program.

The result of a static linkage editor can be loaded for execution by an absolute loader. This loading approach is particularly convenient in a computer with virtual memories.

Advantages Some of the advantages of direct linkage editors are given below:

1. It performs one linking and the most of relocation for many executions.
2. There exists a possibility to create subroutines libraries.
3. The actual loading is very simple.

Disadvantages Some of the disadvantages of direct linkage editors are given below:

1. During the development of a program, recompiling is necessary for almost every execution.
2. [*Storage waste*] This method leads to a waste of storage due to infrequently used programs.

Dynamic linkage editor A dynamic linkage editor performs the following two-step operations.

Step 1 **[Called editing]** It performs the allocation and linking tasks. Thus, it edits the program in terms of its machine operation module (MOM) components and results a relocatable MOM.

Step 2 The tasks of a dynamic linkage editor are performed by a routine of the operating system when the program is scheduled for execution. This routine performs both loading and relocation as done by a relocating loader.

Dynamic linkage editors are used particularly for overlay programming.

Comparisons The differences between relocating linking loader and linkage editor are discussed in Table 14.5.

Table 14.5 Relocating linking loader vs linkage editor

Relocating linking loader	Linkage editor
The linking loader accepts object programs and library routines from the secondary storage device, then links them and produces an executable module for execution in the primary memory itself.	A linkage editor produces a linked version of the program (load module), which is written to a file or library for later execution.

14.15 OVERLAY

Suppose a program contains a number of program segments. The total storage requirement of the program (i.e., main program and subprograms or routines) is more than the available primary memory in a computer. In this situation, it is difficult to run this program. This problem can be solved by a new mechanism known as *overlay*.

The philosophy behind the solution to this overlay problem is very simple. It prepares the load module in such a way that during execution not all of its program segments need to be resident concurrently in main memory. Therefore, the concept of overlays is used in linking operations.

Let us consider the following illustrations to understand the concept of overlay and overlay structure. Then, we shall state the formal definition for different related terms such as overlay, overlay structure, static overlay generator and dynamic loading of overlays.

Illustration 14.12 Suppose a program contains five subprograms and their subroutine calls within the program are given in Table 14.6. Here, the subprogram Math1 of size 30K calls the following subprograms: Math2 of size 30K, Math4 of size 20K, and Math5 of size 30K. Similarly, Math2 is calling Math3 and Math5; Math4 is calling Math5. Now, our objective is to construct an overlay structure to optimize the storage requirement. Also, it provides the memory map.

Table 14.6 Subprograms, their size, and corresponding calling routines

Subprogram	Size	Calling subprograms
Math1	30K	Math2, Math4, Math5
Math2	30K	Math3, Math5
Math3	40K	
Math4	20K	Math5
Math5	30K	

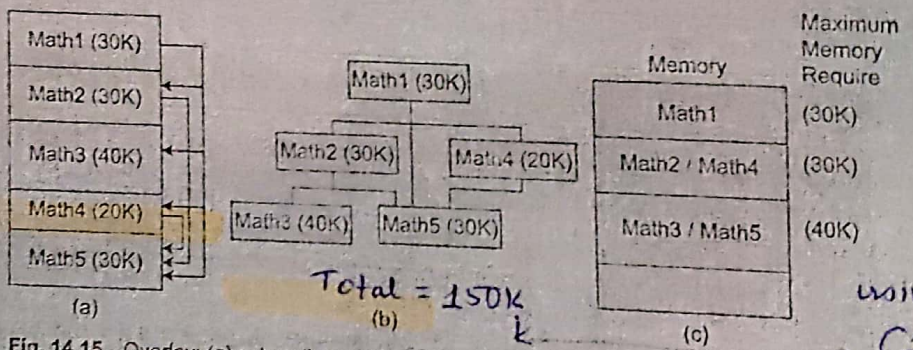
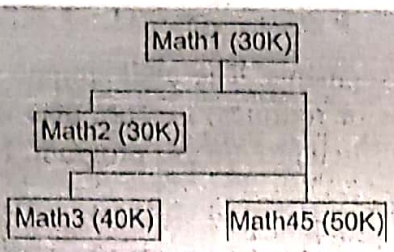


Fig. 14.15 Overlay: (a) subroutine calls between the procedures, (b) overlay structure, and (c) memory allocation for each procedure

Further assume that Math5 is calling Math4. The overlay structure for the subroutine calls between the procedures, the memory allocation for each procedure are shown in Fig. 14.15. Here, the total memory requirement is 30K + 30K + 40K + 20K + 30K = 150K, but the overlay structure needs only 100K with 30K (in the first layer), 30K (i.e., *maximum*(30K, 20K) in the second layer) and 40K (i.e., maximum(40K, 30K) in the third layer). Hence, we are saving 30K of memory. One question that

Table 14.7 Subprograms, their size, and corresponding calling routines

Subprogram	Size	Calling subprograms
Math1	30K	Math2, Math45
Math2	30K	Math3, Math45
Math3	40K	
Math45	50K	



arises here is–what will be its overlay structure? Math4 and Math5 are calling each other. It is recursive in nature. In this case, Math4 and Math5 are combined to a single model, Math45 (say), then the subroutine calls between the procedures does not change. If we remodel the problem then we consider Table 14.7 and obtain an overlay structure as shown in Fig. 14.16.

Fig. 14.16 Overlay structure

Illustration 14.13 A program contains six subprograms and their subroutine calls within the program are shown in Table 14.7. Now, our objective is to construct the overlay structure to optimize the storage requirement. It also provides the memory map.

Table 14.8 Subprograms, size, and their calling routines

Subprogram	Size	Calling subprograms
Sub0	10K	Sub1
Sub1	5K	Sub5
Sub2	20K	
Sub3	15K	
Sub4	10K	
Sub5	5K	Sub2, Sub3, Sub5

The solution to this problem is shown in Fig. 14.17.

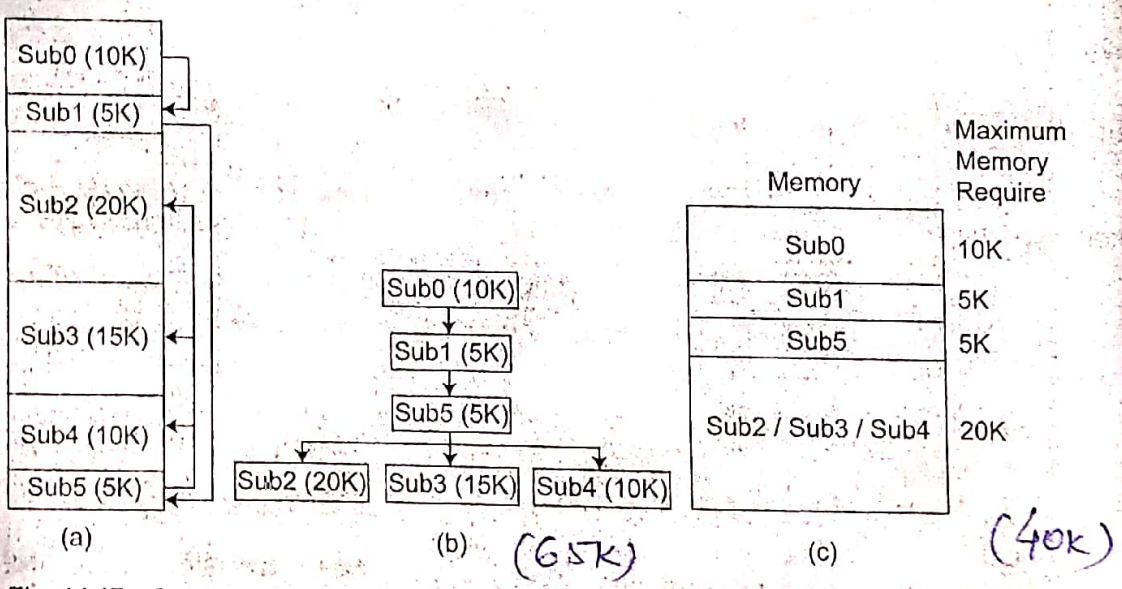


Fig. 14.17 Overlay: (a) subroutine calls between the procedures, (b) overlay structure, and (c) memory allocation for each procedure

Illustration 14.14 Suppose a program has six subprograms and their subroutine calls within the program are given in Table 14.9. Construct an overlay structure to optimize the storage requirement. Also, provide the memory map.

The overlay structure of this problem is shown in Fig. 14.18. Here, the system has two isolated structures. Basically, it consists of two independent programs.

Table 14.9 Subprograms, their, size and corresponding calling routines

Subprogram	Size	Calling subprograms	Remark
Seg1	8K	Seg2, Seg3	root
Seg2	7K		
Seg3	4K	Seg4, Seg6, Seg9	not root
Seg4	10K		
Seg5	8K	Seg7, Seg8	root
Seg6	8K		
Seg7	6K		
Seg8	9K		
Seg9	6K		

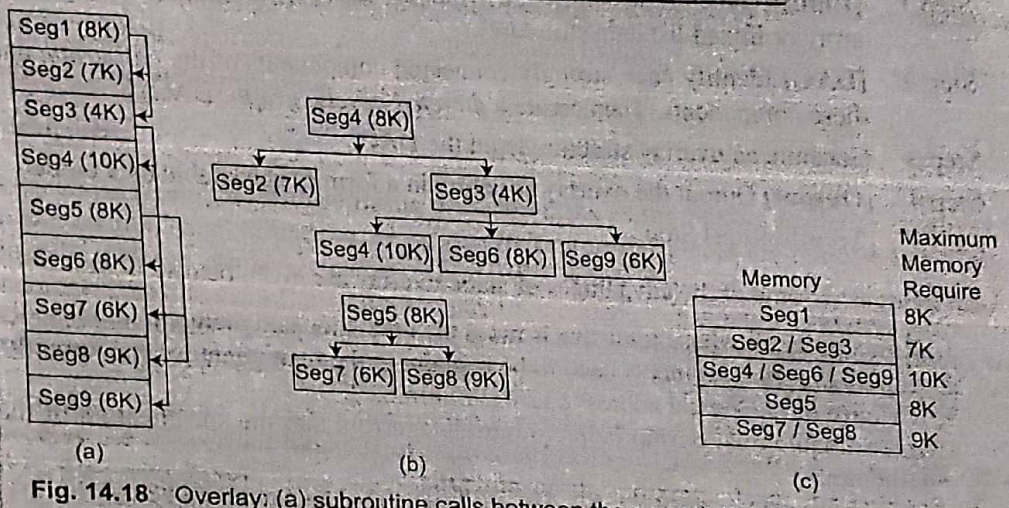


Fig. 14.18 Overlay: (a) subroutine calls between the procedures, (b) overlay structure, and (c) memory allocation for each procedure

Overlay An overlay is a section of a program that has the same load address where the origin is certain and it may be in other sections of the program.

Overlay structure An overlay structure is obtained from a given program on the basis of the mutual exclusiveness of program modules. It is a static feature.

Overlay supervisor/Overlay manager It is a system program that is used to manage the overlays during linking and executing.

Working principle The working principle of overlay structure is described as follows.

1. All overlays are programs. They are stored in a program library.
2. Execution starts after loading one overlay in the main memory of the target computer.
3. Other overlays are loaded as and when necessary. During loading, the overlay previously in the storage is overwritten.

This makes it possible to execute programs whose total size is larger than the storage capacity of the computer.

Static overlay generator Static overlay is generated in two ways: (i) manually and (ii) automatically.

Manual method In manual methods, we have to study all possible subroutine calls and create a call graph (Fig. 14.17(a)). We shall construct a hierarchical structure of subroutine calls (Fig. 14.17(b)). This structure will tend to be an overlay structure.

Automatic method The overlay structure is designed by an automatic method. The computational algorithm for the determination of the overlay structure is given in Algorithm 14.8.

ALGORITHM

Algorithm 14.8 Generate an overlay structure

Input: Subroutine call list

Output: Overlay structure

Step 1 **[Input]** Generate a *call graph* from the given subroutine/subprogram calls (using array or linked list data structure).

Step 2 **[DAG]** Identify each strongly connected components of the *call graph* and collapse these components. Then create a *directed acyclic graph* (DAG).

Step 3 Generate an overlay structure from the DAG.

Step 4 **[Output]** Output the overlay structure in a form of acceptable by a binder.

Step 5 **[Termination]** Stop.

Advantages Some of the advantages of static overlay are given below:

1. Run a program whose total size is more than the allocated memory.
2. Not all program segments need to be resident in the main memory concurrently.
3. The size of the logical address space is minimized.
4. Inter-module relationship helps to introduce errors into the specification of the overlay structure.

Disadvantages Some of the disadvantages of static overlay are given below:

1. Execution of an overlay program is always delayed due to loading a segment on demand basis.
2. Overall efficiency is poor.

14.15.1 Linking a Program Containing Overlays

If a program contains an overlay, then the linker performs the following operations described in Algorithm 14.9 to produce a single executable file.

ALGORITHM

Algorithm 14.9 Linking procedure with overlay

Input: Overlay manager module and executable file

Output: Linked executable file

Step 1 Include an overlay manager module in the executable file being generated.

Step 2 Replace all inter-segment calls that cross overlay boundaries by an interrupt processing instruction.

Step 3 **[Termination]** Stop.

14.15.2 Loading a Program Containing Overlays

An executable program with overlay structure after linking, loads into the primary memory for execution. This execution procedure performs the steps described in Algorithm 14.10.

ALGORITHM

Algorithm 14.10 Load/execution procedure

Input: Executable program file with overlay structure

Output: Executable program in primary memory.

Step 1 Control is handed over to the overlay manager.

Step 2 **[Load root]** Overlay manager loads the root segment to the primary memory.

Step 3 **[Generate interrupt]** A procedure call generates an interrupt.

Step 4 **[Process interrupt]** This generated interrupt processed by the overlay manager.

Step 5 **[Load]** Appropriate overlay is loaded into memory.

Step 6 **[Termination]** Stop.

14.16 BINDER

A binder is a program that performs the same tasks as the linking loader (i.e., *direct linking loader*) with a small difference. The direct linking loader places the relocated and linked text directly into the primary memory. The binder writes these outputs instead of the primary memory. This file is in a formal form, ready for the loader. This output is known as *load module*. The module loader loads the output of the binder. The binder performs the following tasks: (i) *allocation*, (ii) *relocation*, and (iii) *linking*.

Working principle The working principle of a binder is shown in Fig. 14.19. Here, the binder accepts the *unoverlaid* modules of a program as input. The binder also accepts the overlay structure of the program. With the help of this *overlay structure* and unoverlaid modules, the binder produces overlay modules for execution.

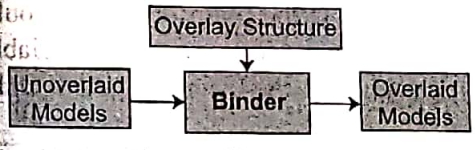


Fig. 14.19 The working principle of binder

Classification *Binders* are classified into two types: (i) the core image binder and (ii) the linkage editor.

Core Image binder This binder produces a core image module, which is an actual 'image' or 'snap shot' of a section of the primary memory. This means that it will load always from the particular location as an absolute loader.

Linkage editor It keeps track of the relocation information. So, the resulting load module can be loaded anywhere in the primary memory.

Comparison A comparison between the core image binder and linkage editor is discussed in Table 14.10.

Table 14.10 Core image editor vs linkage editor

Core Image editor	Linkage editor
1. It is relatively simple and fast.	1. It is complex.
2. Allocation and loading scheme is fixed.	2. Allocation and loading scheme is flexible.

14.17 DYNAMIC LOADER

In this section, we shall discuss dynamic loading schemes.

Relocation during binding Generally, binding is done at any time such as (i) in between run time, when it is appropriate to speak of position-independent code, (ii) assembly time, and (iii) compile time. The object code modification technique is to allow a distinction between the values that are already determined and not yet bound. These determined values represent either constants or addresses of objects that are already bound to specific memory addresses.

Dynamic linking Suppose a subroutine of a program is referenced at different locations in the program but never executed in some context. In this case, the linker/loader would still incur the overhead of linking/loading the subroutine. This is a problem. In this context, our objective is to design a loading scheme so that it improves the performance of the loading/linking scheme.

Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called. It is not possible to write programs such as

```
⋮
READ PROCNAME, PARAMETERS
ANSWER = PROCNAME(PARAMETERS)
PRINT ANSWER
⋮
```

where the name of the subroutine PROCNAME and its parameters PARAMETERS are the inputs. Here, the subroutine name PROCNAME is a variable like other data read. This is not allowed even in the dynamic loading scheme.

A very general type of loading scheme is called *dynamic linking*. In this mechanism, the loading and linking of external references are postponed until execution ends. In other words, the assembler produces text binding and relocation information from a source language program.

Working principle The loader loads only the main program. If the main program should execute a transfer instruction to an external address, or should reference an external variable (i.e., a variable that has not been defined in this procedure segment), the loader is called. Only then is the segment containing the external reference loaded.

Advantages Some advantages of dynamic linking loader are stated below:

1. No overhead is incurred unless the procedure to be called or referenced is actually used.
2. The system can be dynamically reconfigured.

Disadvantages Some disadvantages of dynamic linking loader are stated below:

1. Considerable overhead and complexity incurred,
2. It has postponed most of the binding process until stop execution.

14.18 AUTOMATIC LIBRARY SEARCH

The library subroutines called by the program being loaded are given in the following steps.

Step 1 Automatically fetched from the library.

Step 2 Linked with the main program.

Step 3 Loaded into the primary memory.

The linking loader must perform the following steps:

Step 1 It keeps track of the unresolved external references.

Step 2 It searches the specified libraries that contain the definitions of these symbols.

Step 3 It processes the subroutines found by this search by the regular way.

Step 4 The last step can provide additional unresolved external references; so this whole process must be iterative.

Note This process allows the programmer to override the standard subroutines in the library by supplying his/her own routines.

14.19 COMPLEXITY

The complexity of a loader algorithm is determined based on the following three features:

1. The number of machine operation modules in the executable file
2. The complexity of the information contained in the components of each machine operation modules taken as data
3. The number of operations required to be executed on each machine operation module (MOM)

Time complexity Time requirement can be reduced after imposing some restrictions on

1. the relationship between machine operation modules and
2. the components within machine operation module.

Space complexity Space requirement can be reduced by the following:

1. Ordering MOM components according to the order between the operations performed by the loader:

   Allocation → *Loading* → *Linking* → *Relocation*

2. The information used by allocation is contained in the external symbol dictionary (ESD) component of the MOM. The ESD needs the first component of the MOM.
3. The information used by loading and linking is in TEXT and ESD. The TEXT needs the second component of the MOM.
4. The information used by relocation is in the RLD component of the MOM. Hence, the relocation and linking directory (RLD) needs the third component of the MOM.

14.20 SOME POPULAR LOADERS

In this section, we discuss some popular loaders used in practice such as boot loaders, UNIX loaders, and binary image loaders.

14.20.1 Boot Loaders

A boot loader is a special kind of loader that needs to load the system/OS in a machine.

Basic concept of boot loader A boot loader consists of a number of features. Some of them are listed below:

1. *The memory requirement of installation* ROM (read only memory), EEPROM (electrically erasable programmable read-only memory), or Flash.
2. *The boot-loader software* Blob, bootldr, Redboot, Angel, etc.

3. *The device/mechanism used to control the loader* None, serial port, switches, etc.
4. *The storage device from which the kernel image is downloaded* Host PC, disk, internet, etc.
5. *The device and protocol by which the image is transmitted* Serial, USB (universal serial bus), ethernet, IDE (integrated development environment)
6. The 'host-side' software used to transmit the image, when it is coming from an external device/machine.
7. Whether the boot loader is single stage or multistage. Booting from disk is a two- or three-stage process. Some devices have a native boot loader/OS that can be easy to replace. It is used to fetch and boot the Linux kernel.
8. The CPU type '*Sleep*' resets.

Based on the features, the task of the boot loaders varies with their types. The generalization of a boot loader is very difficult. In practice, there are a number of common features that exist in a boot loader, which means that the common features are covered in most cases. Also, commonalities are found in many situations and devices, which means that many boot loaders can run on a range of platforms.

The boot sequence on a standard PC The steps of boot sequence on a standard personal computer (PC), i.e., IBM PC compatible, are as follows.

Step 1 **[Initialization]** At the start point of booting, the CPU runs an instruction located at the memory register FFFF0h in the BIOS (*Basic Input/Output System*). This memory register location is at the end of system memory. This location contains a jump instruction that moves execution to the location of the BIOS start-up program.

Step 2 **[Testing]** This BIOS start-up program runs a *Power-On Self Test* (POST) program. The task of this test program is to check the devices of the computer for reliable functioning. Also, it initializes the connected devices.

Step 3 **[Working tasks]**

(a) The BIOS tests a preconfigured list of devices

until it finds one that is bootable.

(b) *If* not finds *then*

(i) generates an error, and

(ii) stops the boot process.

(c) *If* the BIOS find a bootable device *then*

it loads and executes its *master boot record* (MBR).

Note that in most cases, the MBR checks the partition table for an active partition.

If one is found *then*

the MBR loads the partition's boot sector and runs it.

This boot sector is operating system (OS) specific, however in most kernel w continues startup.

Other variants of the boot mode include the following:

(a) Serial mode boot
(b) Parallel mode boot
(c) HPI (Host Port Interface) boot
(d) Warm boot

The boot sequence on Linux The Linux boot loader has to load the following two things into the primary memory:

1. The Linux kernel (popularly known as the *kernel*)
2. A root file system (popularly known as the *root*) for it to use

It is possible to link the kernel and boot loader together into one file. It makes the boot-loading part simpler and the linking significantly more complicated.
The major advantage of a Linux boot loader is that it can only load one file and there is no easy way to change it.

Note The file system usually takes the form of an initial RAMdisk (or *initrd*), which is a gzip-compressed normal *ext2* file system image, but it can also be a *ramfs*, *cramfs*, or *jffs2* image.

Modes of operation Most boot loaders are capable of two distinct modes of operation—*bootloading* and *downloading*.

In principle, there is no big difference between these two modes, but in practice it is useful to consider them separately.

Boot loading In this mode of operation, the target device operates autonomously, loading its kernel and root file system from an internal device. This is a normal booting procedure.

Downloading In this mode of operation, the target device down-loads the kernel and the root file system from an *external device*, often under the control of that device. This is the mode that is used to first install the kernel and root, and for subsequent updates.

Dual mode A boot loader may be used for both *boot loading* and *downloading* tasks. There has to be some mechanism to change the behaviour between these two modes. This mechanism works as follows:

(a) A link on the board or
(b) the reception of a character on a serial link which interrupts the normal boot and
(c) puts the boot loader into a mode where it will take commands.

14.20.2 UNIX Loaders

In this section, we shall discuss the UNIX loader and the various types of file formats that it works with.

Using early *a.out* format The operating system loads a simple two-segment file in early *a.out* format in the primary memory. The loading process is shown in Fig. 14.20. The working principle is described in Algorithm 14.11.

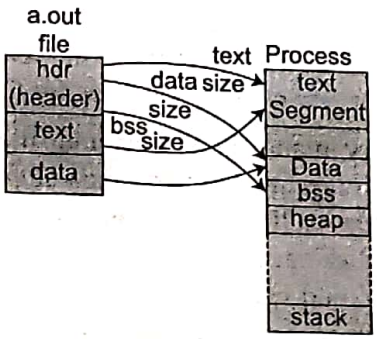


Fig. 14.20 Loading an early *a.out* file (NMAGIC) into the primary memory for a process and segments with arrows pointing out data flows

14.20.4 Loaders: Running a PE Executable

Starting a PE executable process is a relatively straightforward procedure. This is described in Algorithm 14.12.

ALGORITHM

Algorithm 14.12 The UNIX loader with PE format

Input: Executable program file.

Output: Executable program in primary memory

Step 1 **[Input]** Read in the first page of the file with the DOS header, PE header, and section headers.

Step 2 Determine whether the target area of the address space is available, if not allocate another area.

Step 3 Using the information in the section headers, map all of the sections of the file to the appropriate place in the allocated address space.

Step 4 If the file is not loaded into its target address then apply fix-ups.

Step 5 Check the list of DLLs in the imports section for the files not loaded in the memory.

Step 6 Load the files not yet loaded in the memory.

Step 7 Repeat Steps 5 to 6 until complete loading

Step 8 Resolve all the imported symbols in the imports section.

Step 9 Create the initial stack and heap using values from the PE header.

Step 10 Create the initial thread and start the process.

Step 11 **[Termination]** Stop.

14.20.5 Binary Image Loader

The binary image of a program consists of the following three components.

A header It contains the type of the image, such as executable file/library, and loading information, such as pre-assigned address or not.

Text of the program It contains actual pieces of code in some specified formats such as Intel hexa format, and elf. It also contains object files, static libraries, and information about shared libraries.

List of shared libraries These are the shared library routines that are called by the module.

Loading principle The binary image loader checks the file header to determine the loading strategy. A virtual memory is used for the text and it is also used for backup by the code file. The virtual memory management scheme will take care of the page faults, mapping for missing pages, etc.

Loading principle of MS-DOS .COM file A.COM file contains binary codes. When the operating system runs a .COM file, it merely loads the contents of the file into the memory starting at a fixed offset 0x100, where 0-FF are reserved for PSP, (*Program Segment Prefix*) with command line arguments and other parameters. Also, it sets the x86 segment registers all to point to the PSP, the SP (*stack pointer*) register to the end of the segment, since the stack grows downward, and jumps to the beginning of the loaded program. The segmented

architecture of the x86 makes this task. Since all x86 program addresses are interpreted relative to the base of the current segment and the segment registers all point to base of the segment, the program is always loaded at a segment-relative location 0x100. Hence, for a program that fits in a single segment, no fix-ups are needed since segment-relative addresses can be determined at link time. When the programs are not fit in a single segment then the fix-ups are the problem to the programmer. There are indeed programs that start out by fetching one of their segment registers, and adding their contents to stored segment values elsewhere in the program. This is a tedious job where linkers and loaders are intended to automate this task. MS-DOS solves this problem with .EXE files.

Loading principle of MS-DOS .EXE file Loading an .EXE file is only slightly more complicated than loading a .COM file. The loading principle of MS-DOS .EXE file is described in Algorithm 14.13.

ALGORITHM

Algorithm 14.13 Loading principle of MS-DOS .EXE file

Input: MS-DOS .EXE file

Output: File loaded into primary memory for execution

Step 1 **[Input]** Read the header, check the magic number for validity.

Step 2 Find a suitable area of memory. The minalloc and maxalloc fields decide the minimum and maximum number of extra paragraphs of memory to allocate beyond the end of the loaded program. (Linkers invariably default the minimum to the size of the program's BSS-like uninitialized data, and the maximum to 0xFFFF.)

Step 3 Create a PSP, the control area at the head of the program.

Step 4 Read in the program code immediately after the PSP. The *nblocks* and *lastsize* fields define the length of the code.

Step 5 Start reading *nreloc* fix-ups at *relocpos*. For each fix-up, add the base address of the program code to the segment number in the fix-up, then use the relocated fix-up as a pointer to a program address to which to add the base address of the program code.

Step 6 Set the stack pointer to sp, relocated, and jump to ip, relocated, to start the program.

Step 7 **[Termination]** Stop.

Notes

1. EXE files do not support 286 protected modes in the system.
2. Each segment of code/data in the executable file is loaded into a separate segment whose numbers are not consecutive.
3. Each protected mode executable has a table near the beginning listing all of the segments that the program will require.
4. The system makes a table of actual segment numbers corresponding to each segment in the executable.
5. When processing fix-ups, the system looks up the logical segment number in that table and replaces it with the actual segment number, a process more akin to symbol binding than to relocation.

14.21 GRAPHICS LOADERS

The graphics loaders are to load graphics files into the machine/document. JPEG (*joint photographic experts group*) is the industry standard method of compressing 24-bit pictures, achieving compression ratios of up to 10:1. The JPEG loader will allow these files to drop directly into their documents, loading the image as a 24-bit sprite.