Chapter 4

Linker and Loader

Linking is the process of combining different object modules (developed in different files, probably by different people) into one executable file. As seen in the last chapter, an assembler produces code assuming the start address of each section to be zero. Now, when all these sections are put together into one file, the offsets of the symbols in the program may need recomputation. This is because at assembly time, all offsets were calculated from the start of the sections. However after linking, the offsets should be calculated starting from the beginning of the program. Apart from that, sections may have a set of attributes identifying their placement requirements in the memory. Symbols defined in a section of some object module may be used as an external reference in some other module. The assembler could not resolve these external references due to the non-availability of information about these symbols at assembly time. Now it is the responsibility of the linker to *fill up* these blanks left by the assembler. *Symbol* table produced by the assembler holds necessary information regarding the external symbols used in a module and the global symbols defined in it. Thus, the linker produces the executable version of the program. *Loader* is responsible for introduction of a program into physical memory or virtual memory. A major problem here is the conversion of logical program addresses to physical machine addresses.

4.1 LINKING

As pointed out earlier, the objectives of linking are as follows:

- To combine several object modules
- Resolve external references

The inputs of a linker are the following:

1. *Object files:* These are the modules to be combined to create the executable version of the program. As noted in Section 3.4, the object modules are specified in some standard format.
2. *Static libraries:* These are the standard pre-compiled libraries often distributed as an archive file containing individual object files for the library modules. Libraries are

handled by a recursive searching process. If an object module refers to an external symbol of a library module, then that object file for the library module is included in the set of files that need to be linked. In turn, other library modules containing symbols referred to by the first library module are also included. The process continues until a complete set of required library modules have been determined.

3. *Shared library stubs:* Shared libraries contain the common set of functions to be used by almost all the programs running in a system. Thus, instead of loading the same set of routines several times, they are loaded at a single place in the memory. These routines are written as *Position Independent Code (PIC)* to be explained later. Now, in the individual programs, some stub routines are placed that will call these routines in turn, if required.

The outputs of a linker are the following:

1. *Executable file:* The program is now ready for execution, that is, it may be loaded into the memory and start executing.

2. *External tables:* These correspond to the shared library routines. The linker cannot resolve these references as the position where these libraries are loaded is not known at the time of linking the program. Moreover, this position may vary from one execution of the program to another. Thus, the linker leaves a table of all such external references for the loader to fill up and make the program fully ready for execution.

Figure 4.1 shows the inputs and outputs of the linker.

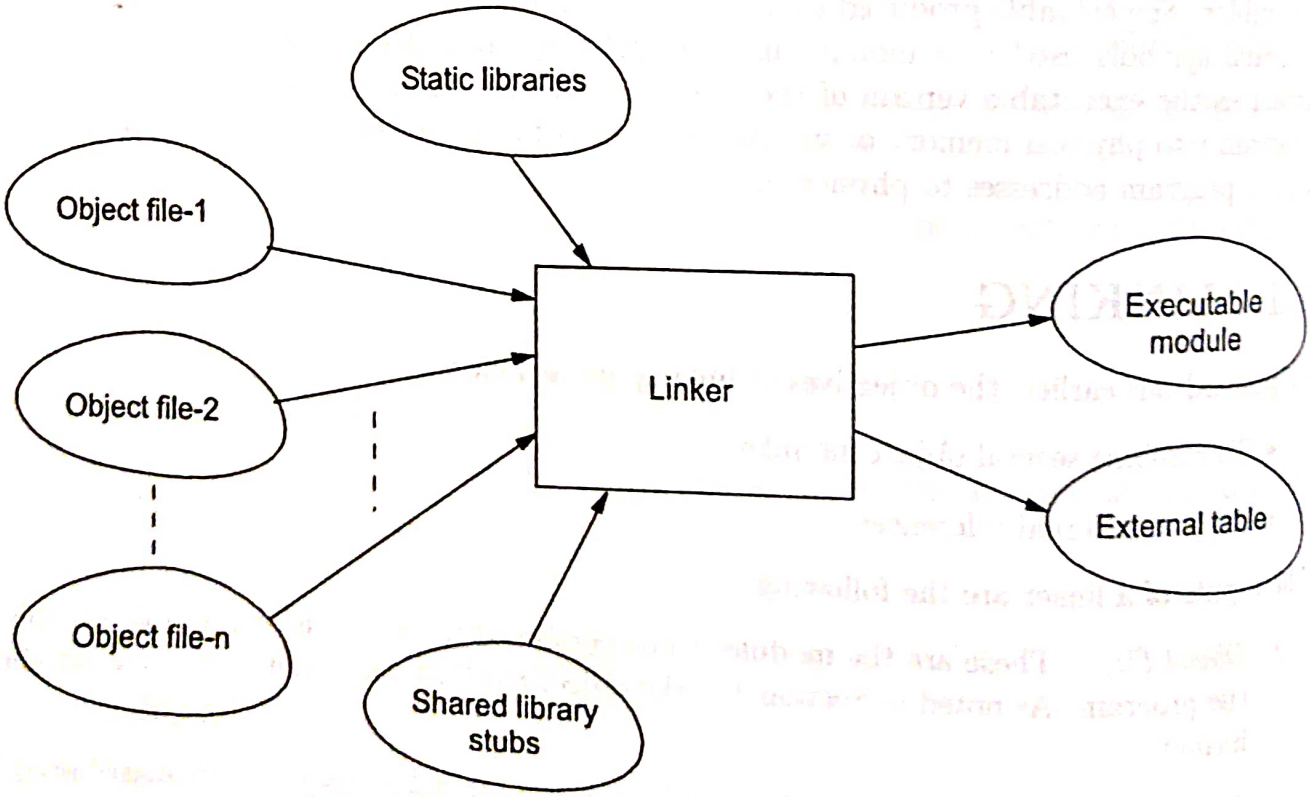

FIGURE 4.1 Inputs and outputs of a linker.

4.1.1 Static vs. Dynamic Linking

Most of the linking that we have discussed so far and that we will be discussing in this chapter pertains to *static linking*, in which all addresses are resolved before the program is loaded into the memory for execution. On the other hand, *dynamic linking* or *deferred linking* links modules on demand. In this case, when an address exception occurs, the *exception handler* is called which does the following.

- The logical address is checked to determine if it refers to a routine or variable that must be dynamically linked. The information regarding the dynamically linkable objects are kept in a *link table*.
- If the address referred to is a valid one, the memory management state of the program is adjusted to reflect the allowed address range for the program.
- The instruction causing exception is then restarted.

4.1.2 Combining Object Modules

The *sections* of object modules are combined to produce a single executable module. In the discussion to follow, we assume the *flat* or *unsegmented* memory model, as this model is followed in almost all the modern operating systems. However, the *SECTION* directive can have some associated attributes specified. All object file formats like *elf*, *obj*, etc. discussed in the last chapter have their corresponding similar set of attributes. To carry out our discussion, we assume the *elf* format for the object files. *NASM* allows the following qualifiers to the SECTION declaration:

- **alloc** defines the section to be one which is loaded into memory when the program is run. **noalloc** defines it to be one which is not, such as an informational or comment section.
- **exec** defines the section to be one which should have execute permission when the program is run. **noexec** defines it as one which should not.
- **write** defines the section to be one which should be writable when the program is run. **nowrite** defines it as one which should not.
- **progbits** defines the section to be one with explicit contents stored in the object file: an ordinary code or data section, for example, **nobits** defines the section to be one with no explicit contents given, such as a BSS section.
- **align=** used with a trailing number, gives the alignment requirements of the section. *align* = x means that the segment can start only at an address divisible by x.

The defaults assumed by NASM if none of the above qualifiers are specified, are:

```
section     .text     progbits    alloc    exec      nowrite     align=16
section     .rodata   progbits    alloc    noexec    nowrite     align=4
section     .data     progbits    alloc    noexec    write       align=4
section     .bss      nobits      alloc    noexec    write       align=4
section     other     progbits    alloc    noexec    nowrite     align=1
```

It may be noted that any section name other than `.text`, `.rodata`, `.data` and `.bss` is treated by default like `other` as shown above.

All the information is added to the *section table* by the assembler and used by the linker to combine the sections into executable file. The process of this combining can be divided into two passes. In the first pass, the relative position of all sections is computed, assuming the start offset of the file to be zero. The second pass actually puts the code into the file.

4.1.3 Pass I of Linking

Pass I computes the start addresses of different sections in the input object modules and collects the symbols defined *public*. It looks into the *section* tables and *symbol* tables to construct a *Combined Section Table (CST)* and a *Public Definition Table (PDT)*. The structure of these tables is shown in Fig. 4.2 (a) and (b) respectively. The procedure followed in this pass has been explained in Fig. 4.3 in the form of a flowchart. Here *lc* is used to keep track of how much of the output file (executable module) is already occupied and thus the place where a new segment can be stored. It is initialized to zero assuming that the executable module will start at offset zero. Now the sections are read from the object files one after the other. Possible start address of the section in the executable module is computed. If a section with the same name as the one just read does not exist in the CST, that is, this section is a new one, its start address is computed by considering the current *lc* value and *alignment* requirements of the section. If *lc* does not satisfy the *align* specification, the immediate next address satisfying it is selected as the start address. On the other hand, if the section is already present in the CST, the possible start address is computed as the current start address (of the section) in the CST plus the current size of the section as noted in CST. This is because the section that has just been read will be concatenated to the end of the section with the same name as in CST. Next, the current size of the section in CST is added to the offsets of all *public* symbols defined in this section. It may be recalled that the assembler had noted the offsets of symbols assuming a start address of zero for the section, which is no more valid when the sections are getting combined in the executable module. All *public* definitions are now put into the *Public Definition Table (PDT)*. The *Combined Section Table (CST)* is updated next. If a new section is being considered, entries are made into CST and *lc* is updated by the start address plus the size of the section. No other modification is needed for the CST. However, if the section name is already present in the CST, only the *size* field needs to be increased for that entry. For subsequent entries, the start addresses are to be increased by the size of the current section being considered. It may be noted that the alignment requirements of the sections should be considered while modifying the start addresses of subsequent entries. The process continues till all sections in all object files have been considered. Then control passes to Pass II of linking producing the executable file. Next, we consider an example to understand the process.

Section name	Start address	Size	Align

(a)

Symbol name	Section name	Offset

(b)

FIGURE 4.2 (a) Combined section table (CST), and (b) Public definition table (PDT).

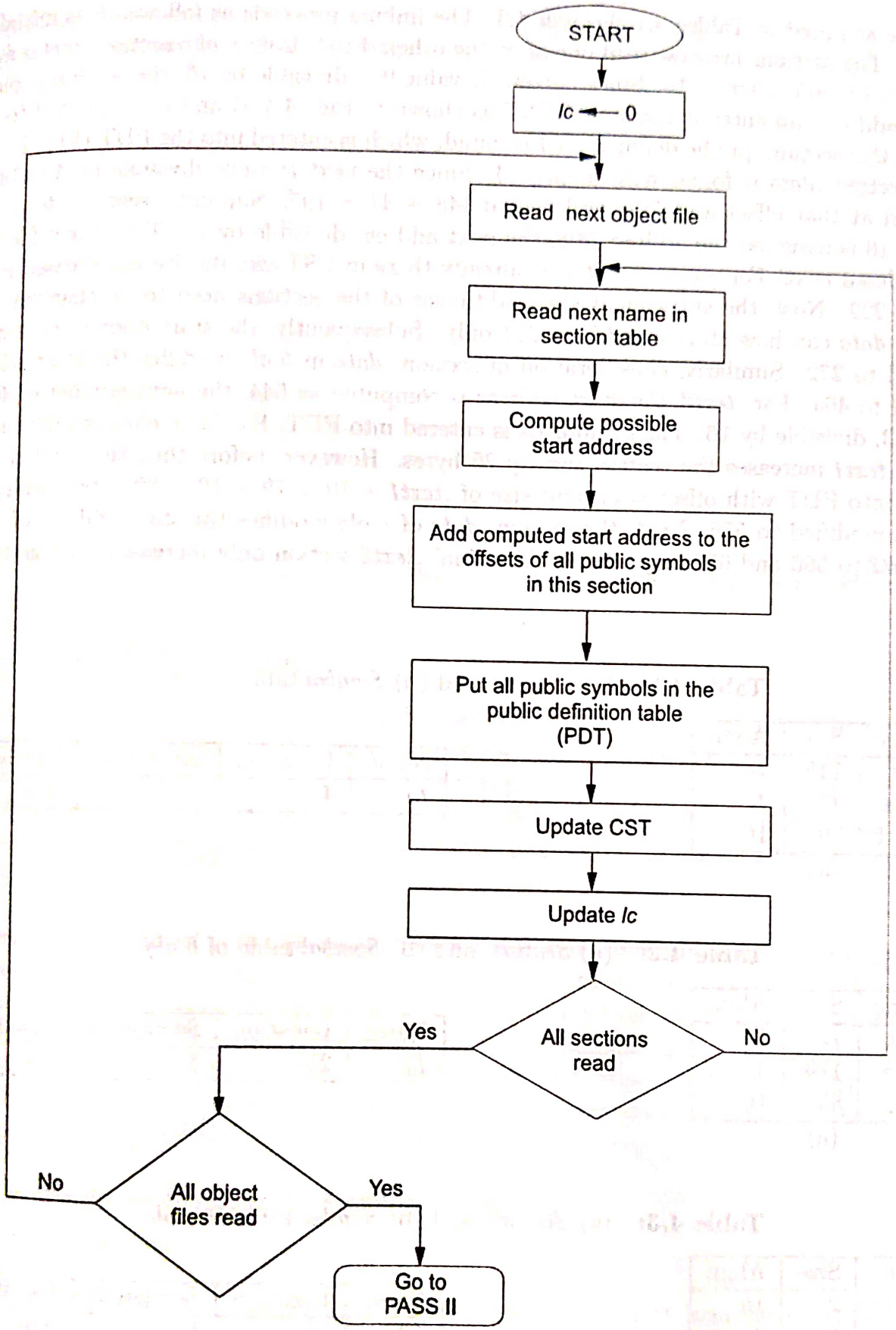

FIGURE 4.3 Pass I of the linker.

EXAMPLE 4.1 Let a set of object files named *a.obj*, *b.obj*, and *c.obj* be given to the linker to produce the executable file. The *section* and *symbol* tables (showing the *public* declarations

only) be as noted in Tables 4.1 through 4.3. The linking proceeds as follows. *lc* is initialized to zero. The sections are now read one after the other. First, from *a.obj*, section *.text* is found with size 145 and *align* = 16. Since current *lc* value 0 is divisible by 16, the section is placed at this address, an entry is made into CST as shown in Fig. 4.4(a) and *lc* is updated to 145. Also, in this section, public definition *L1* is found, which is entered into the PDT (Fig. 4.4(b)). Next, section *.data* is found with *align* = 4. Since the next *lc* value divisible by 4 is 148, it is placed at that offset and *lc* is updated to 148 + 47 = 195. Similarly, section *.text1* with *align* = 16 is assigned the address 208, the next address divisible by 16. The object file *b.obj* is considered next. For section *.text*, it is already there in CST. So, its size is increased to 145 + 77 = 222. Now, the subsequent start addresses of the sections need to be changed. The section *.data* can now start at address 224 only. Subsequently, the start address of *.text1* is modified to 272. Similarly, consideration of section *.data* in *b.obj* modifies the start address of *.text1* to 464. For *.text2*, the start address is computed as 544, the next number of 464 + 79 = 543, divisible by 16. The symbol *L2* is entered into PDT. For file *c.obj*, consideration of section *.text1* increases the section size by 25 bytes. However, before this, the symbol *L3* is entered into PDT with offset = current size of *.text1* + 10 = 79 + 10 = 89. Start address of *.text2* is modified to 576. Next, the section *.data* of *c.obj* modifies the start address of *.text1* and *.text2* to 560 and 672 respectively. The final *.text2* section only increases the size to 155 bytes.

Table 4.1: (a) *Section*, and (b) *Symbol* table of *a.obj*

Name	Size	Align
.text	145	16
.data	47	4
.text1	79	16

(a)

Name	Location	Section-id	Isglobal
L1	4	1	true

(b)

Table 4.2: (a) *Section*, and (b) *Symbol* table of *b.obj*

Name	Size	Align
.text	77	16
.data	179	4
.text2	85	16

(a)

Name	Location	Section-id	Isglobal
L2	23	3	true

(b)

Table 4.3: (a) *Section*, and (b) *Symbol* table of *c.obj*

Name	Size	Align
.text1	25	16
.data	99	4
.text2	70	16

(a)

Name	Location	Section-id	Isglobal
L3	10	1	true

(b)

Section name	Start address	Size	Align
.text	0	~~145~~ 222	16
.data	~~148~~ 224	~~47~~ ~~228~~ 325	4
.text1	~~208~~ ~~272~~ ~~464~~ 560	~~79~~ 104	16
.text2	~~544~~ ~~576~~ 672	~~85~~ 155	16

(a)

Symbol name	Section name	Offset
L1	.text	4
L2	.text2	23
L3	.text1	89

(b)

FIGURE 4.4 (a) Combined section table, and (b) public definition table.

4.1.4 Pass II of Linking

Pass II of linking is responsible for writing the final code into the executable file. The *Combined Section Table (CST)* contains information about the relative positions of sections within this final file to be created. Thus, the main task of this phase is to copy the object files into their corresponding locations. Another very important task to be accomplished in this phase is that of *offset correction* or *relocation*, which we discuss next before going into the algorithm for Pass II.

Relocation. As noted in the assembler design phase, the assembler, while generating object module, assumes the start addresses of individual sections to be zero. All references to variables in a program are translated to their distances from the beginning of the section containing them. However, later on, all the object modules are to be combined into one executable module and all offsets should be taken with respect to the start of the module. Thus, the code corresponding to instructions containing memory references needs to be corrected. This process is called *relocation*. It is useful for,

1. moving around object files during linking, and
2. loading a piece of code at a specified address.

Relocation needs to be carried out both at link-time and at load-time. At link-time, relocation is needed to arrange the object files into the executable module starting at offset zero. At load-time, relocation is needed for arranging shared libraries and the executable module into the address space. For the time being, we concentrate on link-time relocation strategies only. Link-time relocation is performed using *direct editing* that will modify the address sensitive locations within the code during concatenating object

modules. There can be two different ways in which the locations requiring relocatio can be specified.

Relocation bitmap: Here, for every instruction, the assembler associates a relocation bi If the instruction does not need any relocation (that is, instructions not involving memor addresses), the bit is set to zero, else the bit is set to one. For cases with relocation bit = Pass II of linking does the necessary modifications to the code.

Relocation table: It is a table consisting of locations requiring corrections and a *Delta* valu that should be added to the offset of the symbol to get the address corrected.

Table 4.4 shows the relocation bits for the program of Program 2.1 reproduced in column 1, 2, and 3 of the table. Column 4 shows the relocation bits. For the same program, Table 4.5 shows the relocation table.

Algorithm for Pass II

In Pass II, the object files are read again per section and the corresponding codes with necessary relocation changes are put into the executable file. For each section that is read, its start address is looked for in the CST produced in Pass I. The code corresponding to this section is next put into the executable module. The CST entries for the section are updated by incrementing the *start address* of the section by the size of the current section read, and the *size* field in CST is decremented by the size of the section. This ensures that a subsequent occurrence of a section with the same name will get concatenated to the end of this section. Moreover, when all sections have been put into the executable module, the *size* field of all entries in CST will become zero. Next, the code written into the file is modified to sort out the relocation problems (if any). For each entry marked as requiring relocation, the start address of the section in which the symbol has been defined is added to the code. This ensures that all offsets are now with respect to the start address of the executable module. The locations referring to the *shared library* functions and/or variables cannot be corrected at this stage, as their addresses are not known. These relocations are to be performed at the time of loading the program into memory for execution. The entries should then point to the actual location of those shared elements at that time of execution. It may be noted that the addresses of those shared library elements can vary from execution to execution of the program, and thus be handled best by the loader in association with the memory management module of the operating system. The linker will produce a list of such locations requiring load-time relocations. Another important information passed by the linker to the loader is the *start address* of the program. The specification of the start address depends upon the underlying language. For example, the language *C* expects a unique function *main*, to be present in the executable module that defines the start address. In our examples in this book, we have used a global symbol *main* which is used by *gcc* linker to pass on the start address information to the loader.

EXAMPLE 4.2 Continuing Example 4.1, the Pass II algorithm first looks into the sections of *a.obj*. For section *.text*, it puts the code into the executable file as shown in Fig. 4.6, updates the *start address* in CST to 145 and *size* to 77. Next, *.data* is put into the file. *Start address* and *size* are updated to 271 and 278 respectively. The section *.text1* is next put into the file at offset 560. *Start address* and *size* for *.text1* in CST are updated to 639 and 25 respectively.

Table 4.4: Program with relocation bits

Line no.	Code	Source line	Relocation bit
1		global main	
2		extern printf	
3			
4		section .data	
5		my_array:	
6	00000000 0A000000140000001E-	dd 10, 20, 30, 100, 200, 56, 45,	0
7	00000009 00000064000000C800-	67, 89, 77	0
8	00000012 0000380000002D0000-		0
9	0000001B 004300000059000000-		0
10	00000024 4D000000		0
11		format:	
12	00000028 25640A00	db '%d', 10, 0	0
13		section .text	
14		main:	
15	00000000 B900000000	MOV ECX, 0	0
16	00000005 A1[00000000]	MOV EAX, [my_array]	1
17		L2:	
18	0000000A 41	INC ECX	0
19	0000000B 81F90A000000	CMP ECX, 10	0
20	00000011 7412	JZ over	0
21			
22	00000013 3B048D[00000000]	CMP EAX, [my_array + ECX*4]	1
23	0000001A 7D07	JGE L1	0
24	0000001C 8B048D[00000000]	MOV EAX, [my_array + ECX*4]	1
25		L1:	
26	00000023 EBE5	JMP L2	
27			
28		over:	
29	00000025 50	PUSH EAX	0
30	00000026 68[28000000]	PUSH dword format	1
31	0000002B E8(00000000)	CALL printf	1
32	00000030 81C408000000	ADD ESP, 8	0
33			
34			
35	00000036 C3	RET	0

For *b.obj*, section *.text* is put at address 145. *Start address* and *size* fields of CST are updated to 222 and 0 respectively. Next, section *.data* of *b.obj* is placed into the object file at offset 271, *start address* and *size* are updated to 450 and 99 respectively. The section *.text2* is put at offset 672. *Start address* and *size* in CST for *.text2* are updated to 757 and 70 respectively. From *c.obj*, section *.text1* is placed at offset 639. *Start address* and *size* are updated to 664 and 0 respectively. The section *.data* from *c.obj* is placed at offset 450. *Start address* and *size* are updated to 549 and 0 respectively. Finally, section *.text2* of *c.obj* is placed at offset 757. The

Table 4.5: Relocation table

Offset	Delta
00000005	.data
00000013	.data
0000001C	.data
00000026	.data
0000002B	External

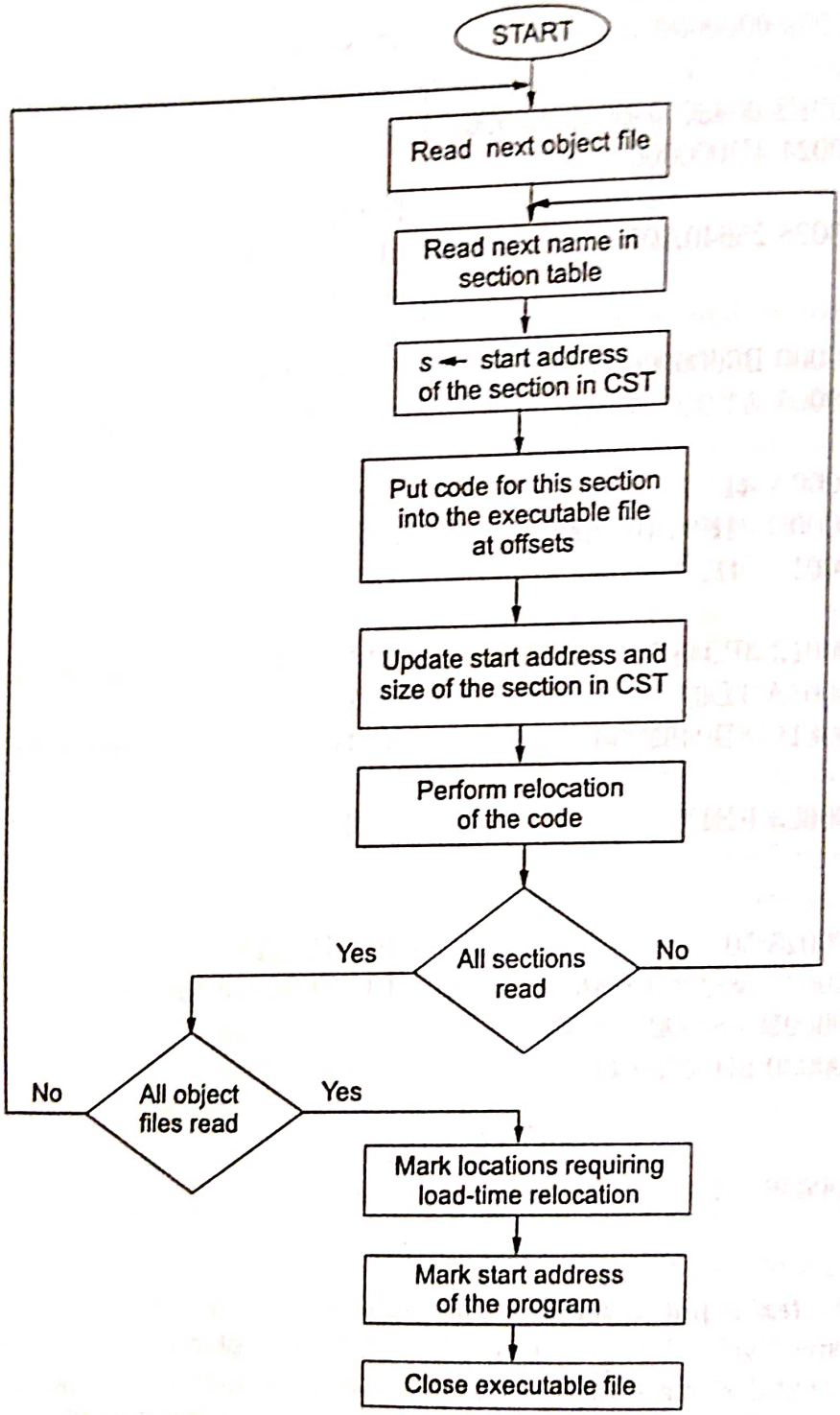

FIGURE 4.5 Algorithm for Pass II of linking.

start address and *size* fields are accordingly updated to 827 and 0 respectively. The layout of the sections in the executable file has been shown in Fig 4.7.

Section name	Start address	Size	Align
.text	~~0~~	~~222~~	16
	~~145~~	~~77~~	
	222	0	
.data	~~224~~	~~325~~	4
	~~271~~	~~278~~	
	~~450~~	~~99~~	
	549	0	
.text1	~~560~~	~~104~~	16
	~~639~~	~~25~~	
	664	0	
.text2	~~672~~	~~155~~	16
	~~757~~	~~70~~	
	827	0	

FIGURE 4.6 Modifications in CST for Example 4.2.

EXAMPLE 4.3 Next we consider another example of Pass II of linking to elaborate the relocation. For this, we have split the program in Program 2.1 into two files *a1.asm* and *a2.asm*. File *a1.asm* contains the data declarations, input and output, while *a2.asm* contains the function *get_max* to determine the maximum element in an array *my_array* defined external in it. The corresponding list files *a1.lst* and *a2.lst* have been shown in Table 4.6, which also shows the code generated side-by-side. The default alignments of 16 for *.text* and 4 for *.data* are assumed. The *Combined Section Table (CST)* after Pass I of linking is as shown in Table 4.7. *Relocation tables* after Pass I of linking is as shown in Table 4.8. It may also be noted that at the end of Pass I of linking, the *Public Definition Table (PDT)* holds information about all public symbols defined in the two object modules. The PDT has been shown in Table 4.9. The Pass II process will put the sections into the final executable module with relocation carried out at offsets 00000007 of *a1.obj* for *format* and at offsets 00000006, 00000016 and 0000001F of *a2.obj* for *my_array*. However, for the symbol *printf* at offset 0000000C of the executable module, the relocation address is not known. Thus it is left for the loader with an indication to be resolved at the time of loading the program into main/virtual memory. The symbol *main* at offset zero is marked as the *start address* of the entire module. The executable module after performing relocations has been shown in Table 4.10. The relocated values have been shown in boldface for clarity.

4.2 LIBRARY LINKING

As we have noted earlier, apart from the symbols defined by the user in different object modules, the program may also contain references to external symbols that are defined in

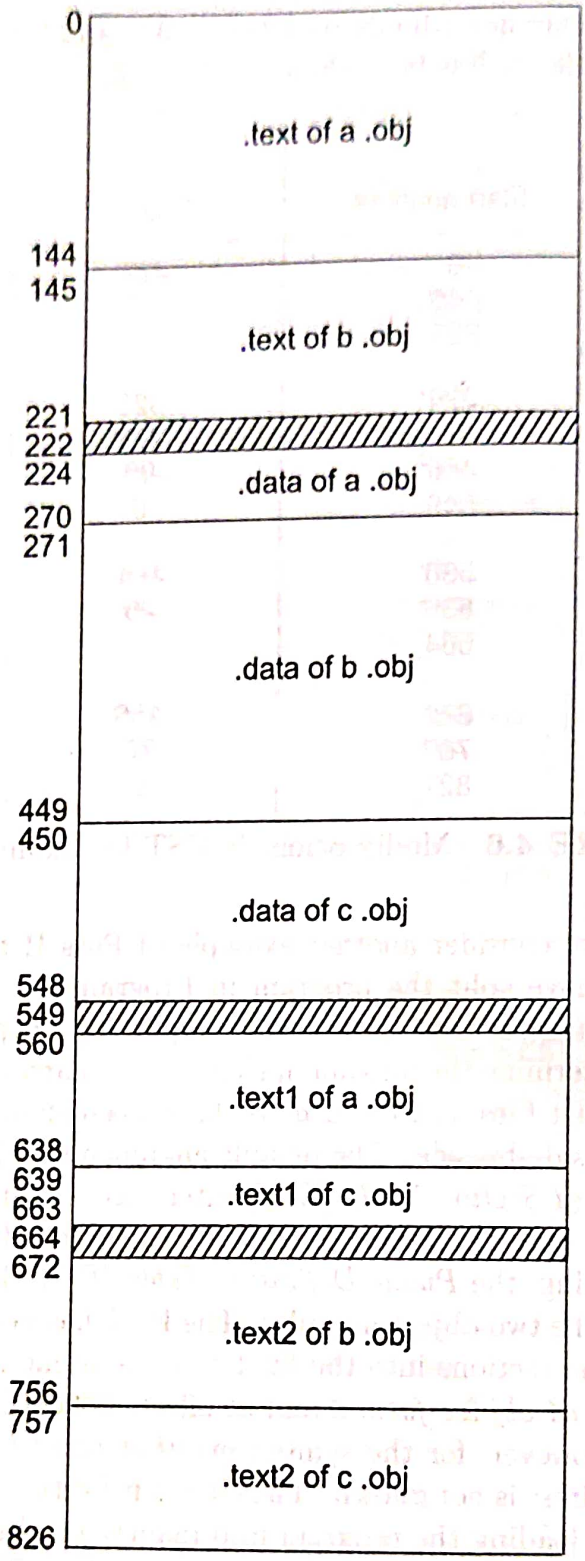

FIGURE 4.7 Layout of the executable file for Example 4.2.

library modules. For example, the mathematical routines are clubbed together into a library, string functions may be kept together into another one. This facilitates the users to refer to a particular library and include it along with other object modules for complete execution of programs. External libraries are usually provided in two forms: *static libraries* and *shared libraries*. In Linux, for example, static libraries have extension ".s" in their file names, whereas

Table 4.6: List files for Example 4.3

a1.lst		*a2.lst*	
	global main, my_array		extern my_array
	extern printf, get_max		global get_max
	section .text		section .text
	main:		get_max:
00000000 E8(00000000)	CALL get_max	00000000 B900000000	MOV ECX, 0
00000005 50	PUSH EAX	00000005 A1[00000000]	MOV EAX, [my_array]
00000006 68[28000000]	PUSH dword format		L2:
0000000B E8(00000000)	CALL printf	0000000A 41	INC ECX
00000010 81C408000000	ADD ESP, 8	0000000B 81F90A000000	CMP ECX, 10
00000016 C3	RET	00000011 7412	JZ over
		00000013 3B048D[00000000]	CMP EAX, [my_array
	section .data		+ ECX*4]
	my_array:	0000001A 7D07	JGE L1
00000000 0A000000140000001E-	dd 10, 20, 30, 100, 200	0000001C 8B048D[00000000]	MOV EAX, [my_array
00000009 00000064000000C800-			+ ECX*4]
00000012 0000			L1:
00000014 380000002D00000043-	dd 56, 45, 67, 89, 77	00000023 EBE5	JMP L2
0000001D 00000059000004D00-			
00000026 0000			over:
	format:	00000025 C3	RET
00000028 25640A00	db '%d', 10, 0		

Table 4.7: Combined section table for Example 4.3.

Section name	Start address	Size	Align
.text	0	61	16
.data	64	43	4

Table 4.8: Relocation tables for Example 4.3

a1.obj

Offset	Delta	Symbol
00000001	External	*get_max*
00000007	*.data*	*format*
0000000C	External	*printf*

a2.obj

Offset	Delta	Symbol
00000006	External	*my_array*
00000016	External	*my_array*
0000001F	External	*my_array*

Table 4.9: Public definition table for Example 4.3

Symbol name	Section name	Offset
main	*.text*	0
my_array	*.data*	64
get_max	*.text*	23

Table 4.10: Executable module for Example 4.3.

Address	Code	Comments
00000000	E8(17000000)	
00000005	50	
00000006	68[68000000]	.text of *a1.obj*
0000000B	E8(00000000)	
00000010	81C408000000	
00000016	C3	
00000017	B900000000	
0000001C	A1[40000000]	
00000021	41	
00000022	81F90A000000	
00000028	7412	
0000002A	3B048D[40000000]	.text of *a2.obj*
00000031	7D07	
00000033	8B048D[40000000]	
0000003A	EBE5	
0000003C	C3	
00000040	0A000000140000001E	
00000049	00000064000000C800	
00000052	0000	.data of *a1.obj*
00000054	380000002D00000043	
0000005D	000000590000004D00	
00000066	0000	
00000068	25640A00	

for shared libraries, the extension is ".so". When a program is linked against a static librar for each of the external functions used by the program, the corresponding machine code copied from the object file containing the library. The extracted code is attached with othe modules to create the overall executable module. However, in some linkers, instead of includin the particular function, the entire library object module is included in the code. In turn, an library module containing symbols referred to by the first library module are also include The process continues until a complete set of required library modules have been determine and included in the code.

The term *shared library* is a bit ambiguous, as it covers at least two different concepts. Firs it is the sharing of code located on disk by unrelated programs. On the other hand, it can b sharing of code in memory. In this case, the programs execute the same piece of code loade at the same physical page of the memory, but mapped to the address spaces of the program Obviously, the second approach has got a lot of advantages. For example, the application may now be only a few hundred kilobytes in size and loaded instantly—the majority of th code being in the libraries that has been loaded for other purposes by the operating syste However, the major price paid is that the shared code must be specifically written in a multi tasking environment, and this has effects on performance. Main memory sharing can b accomplished by using *Position Independent Code (PIC)* as in UNIX, leading to a comple

but flexible architecture, or by using normal (that is, non-PIC) code as in Windows and OS/2. These systems make sure, by various tricks such as pre-mapping the address space, reserving slots for the shared library modules. In Windows environment, such modules are called *DLL (Dynamic Link Library)*. It may be noted that Windows DLLs are not shared libraries in the UNIX sense. In most modern operating systems including Windows, shared libraries can be of the same format as the *regular* executables. This provides two main advantages: first, it requires making only one loader for both of them, rather than two. The added complexity of the one loader is considered well worth the cost. Secondly, it allows the executables also to be used as DLLs, if they have a symbol table. Most dynamic library systems link a symbol table with blank addresses into the program at compile/assemble time. All references to code or data in the library pass through this table, the *import directory*. At load time, the table is modified with the location of the library code/data by the linker/loader. The biggest disadvantage of this shared library is that the executables depend on the separately stored libraries in order to function properly. If the library is deleted, moved, or renamed, or if an incompatible version of the DLL is copied to a place that is earlier in the search, the executable could malfunction or even fail to load. This impedes the performance and stability of the operating system also. On Windows, this is commonly known as *DLL hell*.

4.3 POSITION INDEPENDENT CODE (PIC)

Position independent code (PIC) is a form of absolute object code that does not contain any absolute addresses and therefore does not depend on where it is loaded in the process's virtual address space. This is an important property for building shared libraries. Most modern compilers can be asked to generate PIC (for example, *-fpic* option for *gcc*). Position independence is achieved via the following two mechanisms:

1. **IP-relative** addressing is used wherever possible for branches within modules.

2. **Indirect addressing** is used for all accesses to global variables, or for intermodule procedure calls and other branches and literal accesses where *IP-relative* addressing cannot be used.

For example, the following is a coding technique that may be used to implement function-calls in a position independent manner.

```
        ...
        MOV EAX, ESI[format]
        CALL EAX
        ...
format:
        ...
```

Here, if ESI is preloaded with the start address of the module containing this piece of code, the statement CALL EAX will call the function *format* indirectly. In general, a shared library will contain a number of related functions. As noted earlier, the object code for such library module will also contain a symbol table containing all the functions and variables that may be accessed by other modules. Consider a shared library for the *C* language containing the

functions *printf*, *format*, and *atoi*. In the module, let *printf* be defined at offset 0, *format* at offset 1000, and *atoi* at offset 1100. Then the symbol table will look as shown in Table 4.11.

Table 4.11: Symbol table

printf	0
format	1000
atoi	1100

Any program using calls to these functions should first locate the position of the library module either in the main memory or in the virtual memory and use indirect addressing (as shown earlier for the function *format*) to access them. A program named *Program-A* can be like the following.

```
Initialize LIBC_VECTOR
MOV ESI, LIBC_VECTOR
MOV EAX, ESI[printf]
CALL EAX
```

Whereas another program named *Program-B* can use the same library similarly, as shown below.

```
Initialize LIBC_VECTOR
MOV ESI, LIBC_VECTOR
MOV EAX, ESI[atoi]
CALL EAX
```

For *Program-A*, if the virtual address at which the library has been attached is 50000, the *Initialize LIBC_VECTOR* should get this value, so that ESI is adjusted accordingly. On the other hand, for *Program-B*, the virtual address may be 20000. Thus, the initialization should get this value to set ESI. The virtual memory manager may identify that the virtual pages at address 50000 for *Program-A* and at address 20000 for *Program-B* are referring to the same code, and thus decide to load the corresponding pages only once in the main memory with *shared*-bit turned *ON* for the pages. Thus, the processes maintain implicitly some table of addresses for the shared library functions and call the routines via this table. The variables in the shared library can be accessed in a similar manner by maintaining another table, may be named as *Global Offset Table* and using another register to hold the address of this table. All accesses to shared variables thus need two memory references—one to read the offset from the table and the other one to actually access the variable.

To summarize, PIC enjoys the following advantages over the ordinary executables.

- No need to relocate.
- Library is shared on disk.
- Library is shared in primary memory as well. All the page tables of various processes can share the main memory frames for the library.

On the other hand, it has the following drawbacks.

- We sacrifice one register to hold pointer to the indirect table.
- Each method invocation needs two memory accesses.
- We have to use another register to access the variables in the shared library.
- Each access to variables in the shared library requires two memory accesses.

4.4 SHARED LIBRARY LINKING

Shared libraries are compiled and linked independently from the application. However, when used in conjunction with the application, the following problems come up.

- Offset of various routine can now change.
- The linking process of the application cannot take care of the shared libraries, because it has no idea where the shared library is actually located at the time of execution. Moreover, the position may vary from one execution to another.
- Loader has to take care of the situation. PIC provides a nice solution. However, the scheme is improved further by using *stub* routines in the individual processes.

For example, an application program may have a call to a shared library routine *atoi* like,

```
PUSH 25
CALL atoi
```

The application is augmented with a dummy (stub) *atoi* function, which, in turn, calls the *real_atoi* function residing in the shared library as follows.

```
atoi:   MOV ESI, LIBC_VECTOR
        MOV EAX, ESI[real_atoi]
        JMP EAX
```

On the other hand, the shared library may be located in another file *Libc.so* containing the actual routines, that is *real_atoi*, etc. Let the file contain three such routines with their offsets as shown in Table 4.12. This table is called *Procedure Linkage Table (PLT)*. As is obvious, for correct execution of the the application, LIBC_VECTOR should contain the start address of the memory at which *Libc.so* has been loaded.

Table 4.12: Symbol table for the library

real_printf	0
real_format	1000
real_atoi	1100

With such a scheme, object code generation for the application and library can continue independently. Linker resolves the undefined entries in the object modules of the application with PLT entries of the shared library. During loading, the value of LIBC_VECTOR is known (since either the library is already present in the memory, or it is loaded at this time). Thus, the loader can resolve this reference ensuring a smooth execution of the application.

Versioning. A typical problem arises if several versions of the same library are present in the system simultaneously. Some programs use the older version, while some others use newer ones. Versions are not downward compatible, mainly because due to the addition and deletion of code, the offsets of the routines within the library will vary from one version to another. Ensuring correct execution of all applications in such a situation is really a challenge. The situation is the *DLL Hell* as mentioned earlier.

Among the probable solutions, the following may be tried out.

1. *Rename the shared library every time.* In this case, the system does not provide any support for versioning. The burden is entirely on the programmer. A variant of this puts a copy of the library in each program directory. However, in this case, all forms of sharing are lost!

2. *Integrate numbering scheme in the file name.* For example, *libc.so.5.1.* Here, 5 is the *major revision number* and 1 is the *minor revision number.* Major upgrade increments the major version number. It is better than renaming, however, still rely on basic convention.

4.5 LOADER

Loading is the process of making a program ready for execution by copying the file from secondary storage to the primary or virtual memory. It is often a part of the operating system, and thus not visible to the system user directly. The major objectives of a loader are as follows.

- Bring a binary image into memory.
- Bind relocatable addresses to absolute addresses.

As pointed out earlier, at the end of linking, we are left with a single executable module which needs to be further coupled with the shared libraries that may or may not be already loaded. The task of the loader is now to locate the position of the shared library, correct the appropriate entries in the executable referring to shared library routines and variables, and hence create the binary image that is ready for execution from all angles. Thus, the inputs to the loader are the following.

- One executable
- Zero or more shared libraries

The output of the loader is:

- A usable binary image at an absolute address in the virtual address space. The virtual memory manager takes care of the image now for its correct execution.

The inputs and outputs of a loader have been shown in Fig. 4.8.

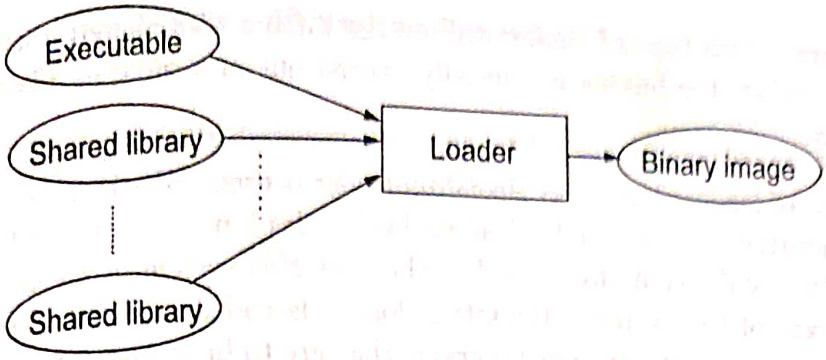

FIGURE 4.8 Input-Output of the loader.

4.5.1 Binary Image

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

1. *A header.* It indicates the type of the image. An image may be an executable file, some library, etc. Some of the images are to be loaded at some preassigned address, or it may be determined by consulting the memory management routine.

2. *Text of the program.* It consists of the actual piece of code. The code may be in some specific format, like *Intel Hex format*, *elf*, etc. It holds the object files, static libraries, and stub tables for shared libraries.

3. *List of shared libraries.* The shared library routines that are called by the module. The loader needs to resolve the addresses accordingly.

The loader checks the file header to determine how to load the program. A virtual memory area is added for the text of the program. The virtual memory area is backed up by the file containing the code. When a page fault occurs, the virtual memory subsystem goes to the backing store (the file) and maps the missing page. A complete discussion on virtual memory manager to load and manage the executable modules is beyond the scope of this book. Interested readers are requested to refer to some book on *Operating Systems*.

4.5.2 Types of Loaders

There are three main categories of loaders, namely:

1. Absolute loaders
2. Relocating loaders
3. Linking loaders

Absolute loaders. This is the simplistic type of situation in which the assembler generates code and writes instructions in a file together with their load addresses. The loader reads the file and places the code at the absolute address given in the file.

Relocating loaders. Here the assembler produces code and the relocation information. The loader, while loading the program performs relocation as well.

Linking loaders. This type of loaders will do the linking with shared libraries as well. As discussed earlier, the linking is normally carried out in a separate phase to reduce the complexity.

Another class of loaders known as *Bootstrap loader* is executed when the computer is first turned on or restarted. It is a simple absolute loader. Its function is to load the first system program to be run by the computer, which is the operating system or a more complex loader that loads the rest of the system. Bootstrap loader is coded as a fixed-length record and added to the beginning of the system programs that are to be loaded into an empty system. A built-in hardware or a very simple program in ROM reads this record into the memory and transfers control to it. When it is executed, it loads the program which is either the operating system itself or other system programs to be run without the operating system.

4.6 CONCLUSION

In this chapter, we have seen how the object modules created by an assembler or compiler can be combined together using the process of linking. The task often requires relocation to be done to rectify the address references within the module. Techniques to create *position-independent-code* have been detailed that can be used for shared-library development. Loading process has also been discussed. However, the details like how a program be loaded into virtual memory has been skipped, as it is covered rigorously in books on *operating systems*. In the next chapter, we will see another feature used in both assembly- and high-level-language programs to reduce the program size. The technique is called *macro*. Processing of these macros is often done in a separate preprocessing phase known as *macroprocessing*.

EXERCISES

4.1 Explain the responsibilities of the linker and the loader in program development.

4.2 What is the difference between static and shared libraries? Can a system have all its libraries only as static or only as dynamic? Is it advisable?

4.3 What advantage does dynamic linking provide over static linking?

4.4 Explain the linking process for the object file x, y, and z for which the *section tables* are shown in Fig. 4.9. Also, show the layout of the final executable module.

Name	Size	Align
.text	305	16
.data	59	4
.data1	65	4

Section table for x

Name	Size	Align
.text	200	16
.data	175	4
.data2	300	4

Section table for y

Name	Size	Align
.text	75	4
.data	89	4
.data2	76	16

Section table for z

FIGURE 4.9 Section tables for *Exercise 4.4*.