R. P. Gohil

CHAPTER 13

Linkers

13.1 INTRODUCTION

A translator, such as a compiler and an assembler, produces an object module from the given input source program in a *virtual machine language* (VML). This object module is executable but not fully. The task of the linking process is to link different object modules distributed in one or more files into one executable program in the file/s. An executable program on a target machine is composed of one or more machine operation modules. This means that the linking process merges two/more object modules into one module. These modules are obtained after translation from different high-level or assembly-level language source programs for a particular target machine. It produces a single machine language program, i.e., an executable program, for the same target machine. This linking process performs via a system program known as a *linker*. Note that each object program is generated for the same target machine and the executable program is obtained for that target machine. In reality, the linker is also a translator and translates the program from a virtual machine language (object program) to a real machine language (target program). Such a program is termed as an *executable program*.

There are two different tasks to be accomplished to obtain an executable module from the object input modules. These tasks are *relocating relative addresses* during the combination of object modules in files or in libraries and *resolving external references*. In general, linking is the process of combining various pieces of code and data together to form a single executable module that can be loaded into the memory of the target machine. Linking can be done at various stages such as

(a) at the compile time (by the compilers) or translation time (by the translator),
(b) at the load time (by the loaders), and
(c) at run time (by the application programs).

The process of linking dates back to late 1940s when it was done manually. Now, we have linkers that support complex features, such as *dynamically linked shared libraries*.

In this chapter, we shall discuss the different aspects of the linking process, ranging from relocation and symbol resolution to supporting *position-independent* shared libraries. We consider different examples to keep things simple and more understandable. However, the basic concepts of the linking process remain unchanged, regardless of the operating system, processor architecture, or object file format being used.

It is observed in software literature that both linking and loading are performed in a single process through loaders, linking loaders, linkage editors, etc. We shall study these in Chapter 14 where we discuss loaders.

13.2 PRINCIPLES OF LINKING

The principles of linking can be studied in two different stages: (a) the linking operation and (b) the library linking.

13.2.1 The Linking Operation

The linking operation comprises the following two-step operations.

Step 1 Search the *memory map of the program* (MMP) for each externally referenced name in the *machine operation module* (MOM).

If found then

get the identification (Id) of the current MOM from the *external symbol dictionary* (ESD);

else communicate an error.

Enters its real address from the *memory map of the program* (MMP) into the *Local External referenced Symbol table* (LES), LES[Id].

Step 2 The entries in the LES are used to link the current MOM with other MOMs for references.

13.2.2 The Linking Process

The linking process performs the task of linking the operations on a given set of input object programs and different libraries (such as static library, dynamic and/or shared libraries) for the same target machine known as the *linker*. Here, the object programs are either in different files or in the same file or mixed (i.e., some object modules are on same file and some are on different files). The output of this process is a single executable program either in a single file or distributed over different files. It also produces some other optional files such as the *map file*. This operation is shown in Fig. 13.1.

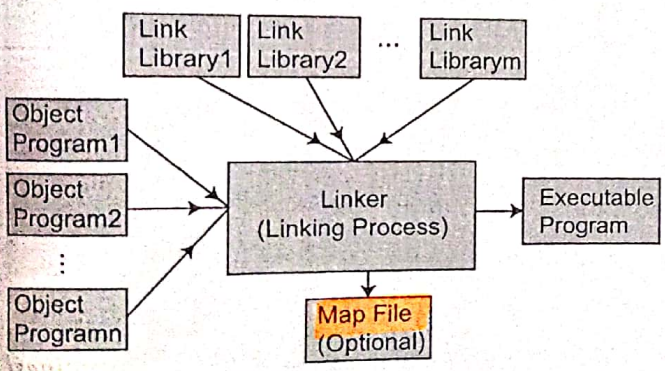

Fig. 13.1 The linking process: conversion of a set of object programs into an executable program

In Fig. 13.1, the linker program accepts n (≥ 1) number of object modules (files) and m (≥ 0) number of link libraries. These are combined and produce an executable program. An optional map file is also produced. This map file contains information, such as code segment, data segment, stack segment, etc., about the program. This executable program will directly run through the loader by the same target machine in which the object modules were produced.

13.2.3 Library Linking

A link library is a file that contains a number of object modules for different subprograms. One or more procedures of any link library may be called by the users' program. These called

modules are obtained from different libraries and are copied into the required position of the executable program during the process of linking.

Note that linking combines two or more separate object programs and supplies the information needed to allow references between them. So, the linker should take care of location-sensitive (i.e., address-sensitive) instructions in the object program during the process of linking (i.e., merging) to a single executable program in the machine language for the same target machine.

Note that the task of library linking is performed in different ways, such as static, dynamic, or shareable.

13.3 A SIMPLE LINKING PROCESS

A simple linking process combines independent translated program units into a single overall module, so that references between units indicate the proper location. This simple linking operation is shown in Fig. 13.2.

Illustration 13.1 Assume P, Q, and R are three independent program modules in which P is calling Q and Q is calling R. Figure 13.2(a) shows these independent program modules. Figure 13.2(b) shows the combined program module. The linking process establishes these links shown by the arrows.

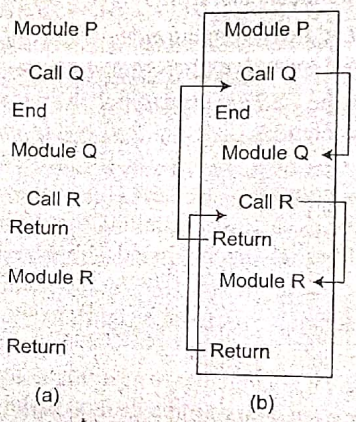

Fig. 13.2 A simple linking process: (a) independent object program modules and (b) single linked module

13.3.1 Different Linking Functions

Software is made up of instructions that are grouped in logical sections termed as *routines/subroutines* or *functions*. Most functions perform a specific task based on a set of input and output variables. These individual functions are able to use or call each other using a linking process.

Suppose a function is calling from another function, then the addresses or locations of each of the functions are known. Once this address is known, the calling function executes or passes control to the *called function*. When the called function completes execution, control is returned to the *calling function*.

Linking is simply the process of placing the address of a called function into the calling function's code. This is the fundamental concept of linking.

Subroutine Linkages An assembly main program (say, A) transfers control to a subprogram (say, B). In this case, the programmer writes an instruction in program A that transfers the

control to the subprogram B. The value of this reference is not known to the assembler during the assembly process. The assembler will declare it as an error (i.e., undefined symbol). A special mechanism is necessary to resolve it.

This mechanism is typically implemented with a relocating or a direct linking loader.

13.3.2 Simple Linking Operations

In the case of a simple linking, the linker performs the following operations:

- Combine object modules into a load module
- Relocate object modules as they are being loaded
- Link object modules together as they are being loaded
- Search libraries for external references not defined in the object modules

Some advantages of simple linking operations are stated below.

1. Suppose the source program of a module in a modular system is modified. Then it is required to recompile/reassemble only this modified module, i.e., no need to recompile/reassemble all the modules in the system.
2. The linkage editor or linker accepts all these independent object files/modules as input and combines them. Now, the linker produces an output which is an executable file.
3. The simple linker provides high flexibility for program development.
4. The linker combines/binds the standard machine operation modules, which can be obtained as a result of any (other) compiler/assembler.
5. The linker provides a complete inter-language communication mechanism.
6. The virtual machine language can be easily standardized; thus the linker becomes the only machine-dependent transformation of a source program towards its executable form.

13.4 BASIC LINKING TASKS AND PROCEDURE

In general, the basic tasks of a linker can be classified as follows.

Allocation Allocate memory for the input and output programs.

Resolution Resolve symbolic references between two/more object modules for linking.

Relocation Set the relocation flag for all address dependent locations in the executable program so that the loader can identify and adjust those addresses during loading into the primary memory for execution.

Loading Physically place the machine instructions and data into the primary memory for execution.

The linker uses the relocation information and the symbol table in each object module to find all the undefined labels. These labels are found in branch and jump instructions, and in data addresses. It finds the old addresses and replaces them with new addresses. It is faster to "patch" the code than recompile.

A linking procedure is shown in Algorithm 13.1.

ALGORITHM

Algorithm 13.1 Basic linking procedure

Input: Object modules distributed in one or more files.

Output: Executable modules.

Step 1 **[Placement]** Place code and data symbolically in the memory.
Step 2 **[Searching]** Determine the addresses of data and instruction labels.
Step 3 **[Patching]** Patch both the internal and external references.
Step 4 **[Termination]** Stop.

13.5 MEMORY MANAGEMENT DURING LINKING

In this section, we discuss the memory management scheme during linking for the compilation of memory location. The details of the memory management schemes are discussed in Chapter 20.

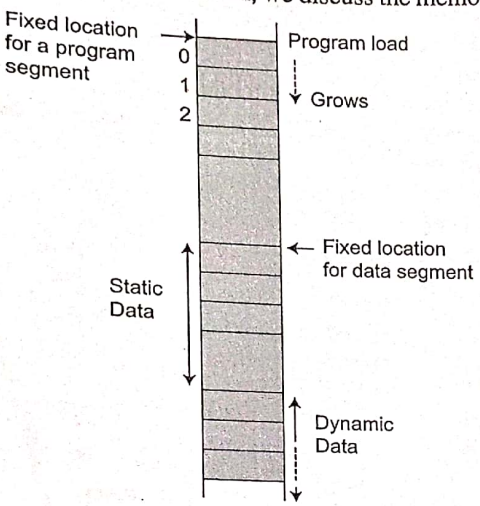

Fig. 13.3 Memory allocation

13.5.1 Computation of Memory Locations

The linker computes memory locations for the following situations.

1. The linker determines the memory location of all procedures and data when it resolves all the external references.
2. The assembler does not know the address where the code and data of a module will reside in the memory relative to other modules because each source file was assembled separately.
3. If a linker places a module in the primary memory then all absolute references used in the module and memory addresses (which are not relative to a register) must be relocated to their true addresses.

13.5.2 Memory Allocation Technique

A linker designer may design his/her own memory allocation scheme for the linker. A typical memory allocation technique for a linker (see Fig 13.3) is described below.

1. Assume a stack starts at the top and grows down, i.e., towards the data segment.
2. The program code starts at a fixed location.
3. The static data starts at a fixed location.
4. Dynamic data (i.e., data allocated by operation say, *new*) starts next after it.
5. The specific mark (chosen by the designer) is situated to make it easy to access the static data.

Illustration 13.2 **(Memory allocation by linking operation)** Consider the following assumptions for this illustration:

1. Program code starts at 0x40000.
2. Static data starts at 0x10000000
3. Specific mark = $ (say) to access the static data.

Suppose we have two hypothetical object program files: *File 1* and *File 2*. Here, *File 1* is for procedure PB and *File 2* for the procedure QA. The contents of *File 1* and *File 2* are shown in Tables 13.1 and 13.2, respectively. The linker links these two object files and produces an

Table 13.1 Object program in File 1 for procedure PB

Object File Header	Name	Procedure PB	
	Text size	200	
	Data size	30	
Text Segment	Address	Instruction	
	0	xxxxxxxx($)	
	4	Jal 0	
	...	...	
Data Segment	0	X	
Relocation Information	Address	Instruction type	Dependency
	0	Type 1	X
	4	Type 2	PB
Symbol Table	Label	Address	
	X	–	
	PB		
	...	...	

Table 13.2 Object program in File 2 for procedure QA

Object File Header	Name	Procedure QA	
	Text size	100	
	Data size	20	
Text Segment	Address	Instruction	
	0	xxxxxxxx($)	
	4		
	...	...	
Data Segment	0	Y	
	...	...	
Relocation Information	Address	Instruction type	Dependency
	0	Type 3	Y
	4	Type 2	PB
	...	...	
Symbol Table	Label	Address	
	Y	...	
	QA	...	
	...	...	

executable file as shown in Table 13.3. The object program in *File 1* for the procedure PB in Table 13.1 contains text 200 bytes (say) and data 30 bytes (say); other contents of this file are the text segment, data segment, relocation information, symbol table, etc. Similarly, the other object program in *File 2* for the procedure QA in Table 13.2 contains 100 bytes of text, 20 bytes of data, and others such as text segment, data segment, relocation information, symbol table, etc. The executable file in Table 13.3 contains a text size of 200 + 100 = 300 bytes and a data size of 30 + 20 = 50 bytes. The program code, i.e., text segment starts at the fixed location 0x40000 (say). Therefore, the first instruction of procedure PB is xxxxxxxx($), which is a 4 byte data that will be stored at 40000. The next instruction is stored at 40004 and so on. The static data segment starts at a fixed location 10000000.

Table 13.3 Executable File Structure

Executable File Header	Name	Procedure
	Text size	300
	Data size	50
Text Segment	Address	Instruction
	0040000	xxxxxxxx($)
	0040004	...
	...	...
	0040100	...
	0040104	...
Data Segment	Address	
	10000000	X
	...	...
	10000020	Y
	...	...
	14	

13.6 SYMBOLS AND SYMBOL RESOLUTION

Every relocatable object file has a symbol table and associated symbols. In the context of a linker, symbols are of the following categories.

Category 1 *Global symbols are defined by the module and referenced by other modules.* All non-static functions and global variables fall in this category.

Category 2 *Global symbols are referenced by the input module but defined elsewhere.* All functions and variables with external declaration fall in this category.

Category 3 *Local symbols are defined and referenced exclusively by the input module.*

All static functions and static variables fall in this category.

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Resolution of local symbols to a module is straightforward since a module cannot have multiple definitions of local symbols. Resolving references to global symbols is not straightforward. It is difficult and trickier. During compilation, the compiler exports each global symbol as either strong or weak (see Illustration 13.3). Functions and initialized global variables get strong weight, while global uninitialized variables are weak. Now, the linker resolves the symbols using the following rules:

Rule 1 Multiple strong symbols are not allowed.

Rule 2 Given a single strong symbol and multiple weak symbols, choose the strong symbol.

Rule 3 Given multiple weak symbols, choose any of the weak symbols.

Illustration 13.3 Linking the following two programs produces link time errors:

```
/* pqr.c */
int pqr () {
    return 0;
}
```

```
/* abc.c */
int pqr () {
    return 1;
}
int main () {
    pqr ();
}
```

The linker will generate an error message because pqr (strong symbol as its global function) is defined twice.

13.6.1 Symbol Resolution Using Static Libraries

During the process of symbol resolution using static libraries, the linker scans the relocatable object files and archives (i.e., libraries) from left to right as input on the command line. During this scan, the linker maintains a set of *relocatable object* files (called *O* files) that go into the executable, a set *U* of *unresolved symbols*, and a set D of *symbols defined* in the previous input modules.

ALGORITHM

Algorithm 13.2 Symbol resolution using static libraries

Step 1 Initially all three sets *O*, *U*, *D* are empty.

Step 2 For each input argument on the command line, the linker determines the following.

Step 2.1 If input is a relocatable object file, then perform
1. linker adds it to set O,
2. updates U and D, and
3. proceeds to next input file.

Step 2.2 If input is an archive then
it scans through the list of member modules that constitute the archive to match any unresolved symbols present in U.
If some archive member defines any unresolved symbol that archive member is added to the list O
then U and D are updated per symbols found in the archive member.
This process is iterated for all member object files.

Step 3 If U is empty then
it merges and relocates the object files in O to build the output executable file
otherwise
the linker prints an error report and terminates.

Step 4 **[Termination]** Stop.

This algorithm also explains that static libraries are placed at the end of the linker command. But in the case of cyclic dependencies between libraries, special care must be taken. Input libraries must be ordered in such a way that each symbol is referenced by a member of an archive and at least one definition of a symbol is followed by a reference to it on the command line. If an unresolved symbol is defined in more than one static library modules, the definition is picked from the first library found in the command line.

Illustration 13.4 This is an illustration for symbol resolution using a static library. Assume three object module with two libraries so that it reflects all the features described in Algorithm 13.2. Modify the algorithm to solve the cyclic dependencies.

13.7 BINDING

The term *bind* means the different program modules hold together to form a single executable module. Specifically, *binding* is a technique to do the task of holding different program modules together to form a single executable modile. *Binding* operations are the operations that do the binding tasks, such as combining modules and adjusting address. *A Binder* is a system program, which means that it is a tool that actually performs the *binding operation*. So the task of binding is in some way related to linking.

In general, *bind* means to choose a specific low-level implementation for a particular high-level semantic construct.

Different binding techniques are given below.

Early binding This involves associating every variable with its abstract data type–enables type checking.

Late binding Here, the attributes' data types are not known until run time. Polymorphism allows selecting the implementation at run time.

Language binding and name binding They are related (i) to the linking of libraries to application programs and (ii) to the way symbols are handled by compilers.

Program time binding Some objects may be explicitly bound by the programmer.

Compile-time binding Some objects may be bound to specific memory locations at compile time.

Link-time binding Linking is a process of binding an external reference to the correct link time address. So, binding during linking is known as *link-time binding*.

Run-time binding The binding of some objects may be deferred, for example, until at the time the program is loaded in memory or even later.

Lazy binding It is related to dynamic linking with shared objects during the execution of a program. A run-time linker locates and includes referenced shared objects and then performs the task of relocation and actual linking operations to prepare the program for execution. This may be termed as lazy binding.

13.7.1 Working Principle of Lazy Binding

First, the referenced shared objects from the shared libraries need to be located and these must be included in the binding process. The linker performs relocation and linking operations to prepare the program for execution.

Step 1 During link-editing, *calls to globally defined procedures* are converted to references to a *procedure linkage table*.

Step 2 When a procedure is called for the first time, control is passed via this table to the run-time linker.

Step 3 The linker looks up the actual address of the *called procedure* and inserts it into the linkage table.

Step 4 Subsequent calls directly go to the *called procedure*.

Advantages The main advantage of binding is that programs are simpler. Every binding technique is developed based on their specific problem to solve.

Disadvantages The main disadvantage is that many type errors are not caught by it until the run time (if at all).

13.8 RESOLVING EXTERNAL REFERENCES

Suppose a C (or Java, or Pascal) program contains a *function call* `f(x)`. The function `f()` is compiled separately and produces an object module.

- When the source program is compiled, the compiler does not know the location of `f()` so there is no way they can supply the starting address.
- Instead of supplying a dummy address, a notational address needs to be filled in with the location of `f()`.
- The object module of `f()` contains a notation that `f` is being defined and gives the relative address of the definition. The linker can convert them to an absolute address.

13.9 RELOCATION AND CODE MODIFICATION

The heart of a linker's actions includes relocation and code modification. A compiler or an assembler generates an object file using the addresses of code that have not been relocated and the data defined within the file, usually zeros for code and data defined elsewhere. In the linking

process, the linker modifies the object code such that it reflects the actual addresses by suitable assignment.

Illustration 13.5 Suppose the program module $P1$ starts at 0 as shown in Fig. 13.4. Then we need to compute the address of the function f. Assume that the length of modules $P1$, $P2$, $P3$, $P4$, and $P5$ are $L1$, $L2$, $L3$, $L4$, and $L5$ respectively as shown in Fig. 13.4. Therefore, the base address of $P2$ is $0 + L1 = L1$. Similarly, the base addresses of $P3$, $P4$, and $P5$ are $L1 + L2$, $L1 + L2 + L3$, and $L1 + L2 + L3 + L4$, respectively. But the function f is in the module $P4$. Therefore, the relative address of f within $P4$ is *rel* (say).

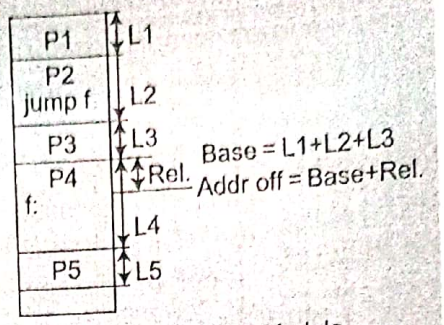

Fig. 13.4 Relocation Module

The address of f = Base address of $P4$ + Relative address of f

$$= L1 + L2 + L3 + rel$$

13.9.1 Types of Relocation

There are three types of relocations as given below.

Assembly-time relocation This involves the program to be reassembled in order to change the location where the program will be loaded in the primary memory.

Illustration 13.6 Sometimes the source code is compiled and the assembly code itself is viewed as something like an object code for communication between the compiler and a *load-and-go* assembler.

Run-time relocation Some machine languages allow a running program to be moved about or unloaded from memory and reloaded in a different location of the memory without harm. This run-time relocation is only possible if consistent use is made of base registers or relative addressing.

Binding-time relocation The object programs are bound to actual memory locations. We can therefore speak about the binding time of each object in the program.

13.9.2 Relocatability of Programs

Program codes are divided into two basic types.

Position-dependent code The machine code or the user data structures contain absolute memory addresses; moving the code and data to a different address would be very difficult. This kind of code is said to be *position-dependent code*.

Position-independent code The machine code or the user data structures contain relative memory addresses; moving the code and data to a different address would be simple. For example, all branch addresses and pointers are relative, i.e., expressed as displacements from the memory location containing the address to the memory location to which the address refers. We can move the entire block of data holding a program's code and data to a different memory address. Machines that allow programs to be written this way are said to support *position-independent code*.

Programs may be characterized in terms of relocation in the following three forms.

Non-relocatable programs A program is said to be *non-relocatable* if its addresses are bounded at translation time and the instructions can not be loaded into any section of the memory for execution. Note that these programs are address sensitive.

Relocatable programs A program is said to be relocatable if its address are not bounded at translation time and the instructions can be loaded into any section of the memory for execution.

Self-relocatable programs A program is said to be self-relocatable if its address sensitive portions can itself perform the relocation. These programs do not need an external agency such as the linkage editor to relocate it.

13.9.3 Relocation Mechanism

The relocation mechanism can be implemented either in hardware or in software. Instruction relocation can be implemented in hardware by a base-register loaded with the absolute address of the program at execution time. The operand addresses are displacements relative to the base register.

Special hardware Virtual memory hardware can hide the complexity of position independent programming from users by using special address translation hardware, which is also known as a special hardware for relocation. It performs run-time relocation. The position-independent programs are explicitly coded to be able to run at any memory address without the need to edit any memory addresses in the code.

13.9.4 Relocation Procedure

During linking, the linker resolves all the symbols used in the program and assigns a unique reference for each symbol. So, each symbol has exactly one definition with respect to reference when the linker starts the process of relocation; it involves a two-step process as described in Algorithm 13.3. After the completion of Step 1, every instruction and global variable in the program has a unique load time address. Then Step 2 performs the relocating of symbol reference within sections.

ALGORITHM

Algorithm 13.3 Process of relocation

Input: Object programs

Output: Relocation resolved program

Step 1 **[Relocating sections and symbol definitions]**

- Step 1.1 Linker merges all the sections of the same type into a new single section.
- Step 1.2 The linker then assigns runtime memory addresses to new combined sections to each section defined by the input module and also to each symbol.

Step 2 **[Relocating symbol reference within sections]**

- Step 2.1 Linker modifies every symbol reference in the code and data sections so they point to the correct load-time addresses.
- Step 2.2 Whenever the assembler encounters an unresolved symbol, it generates a relocation entry for that object code and places it in the relocating or text data sections.

Step 3 **[Termination]** Stop.

Illustration 13.7 The linker merges all the *data* sections of all the input relocatable object files into a single *data* section for the final executable module. A similar process is carried out for the *code* section.

A relocation entry contains information about how to resolve the reference.

Illustration 13.8 A typical ELF (Executable and Linking Format) relocation entry contains the following members.

- *Offset* In a section, an offset is a reference that needs to be relocated. For a relocatable file, this value is the byte offset from the beginning of the section to the storage unit affected by relocation.
- *Symbol* A symbol is the modified reference in a symbol table. This reference should point to the symbol. Basically, it is the symbol table index with respect to which the relocation must be made.
- *Type* The relocation type which is normally denoted as
  (i) R_386_PC32, that signifies PC-relative addressing and
  (ii) R_386_32 that signifies absolute addressing.

The linker iterates over all the relocation entries present in the relocatable object modules and relocates the unresolved symbols depending on the type.
For R_386_PC32, the relocating address is calculated as $S + A - P$; for R_386_32 type, the address is calculated as $S + A$.

where S = The value of the symbol from the relocation entry
P = The section offset or address of the storage unit being relocated (computed using the value of offset from relocation entry)
A = The address needed to compute the value of the relocatable field

13.9.5 Relocating Relative Addresses

Relative addresses are relocating from the given program module by the following rules.

Rule 1 Each object module is treated as if it will be loaded at location zero.

Illustration 13.9 The machine instruction, say,

```
jump 100
```

is used to indicate a jump to location 100 of the current module.

Rule 2 (*To convert this relative address to an absolute address*) The linker adds the base address of the object module to the relative address. The base address is the address at which this module will be loaded.

Illustration 13.10 Module A is to be loaded starting at location 2300 and contains the instruction

```
jump 120
```

The linker changes this instruction to

```
jump 2420
```

Rule 3 The linker computes the starting location of the next object module.

Illustration 13.11 How does the linker know that Module *P5* (see Fig 13.5) is to be loaded starting at location 2300?

The starting address of *P5* (as shown in Fig 13.5) is computed by Algorithm 13.4(a).

If the first module *P1* is loaded at *B* (say), then module *P2* is loaded at $B + L1$ where *L1* is the length of *P1*. Similarly, module *P3* is loaded at $B + L1 + L2$, *P4* at $B + L1 + L2 + L3$, and *P5* at $B + L1 + L2 + L3 + L4$, where *L2*, *L3*, and *L4* are the lengths of modules *P2*, *P3*, and *P4*, respectively.

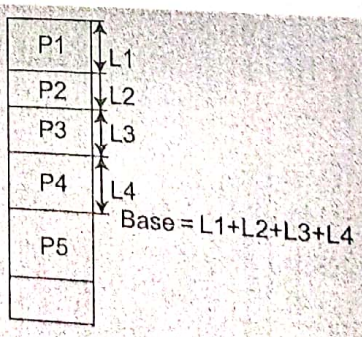

Fig. 13.5 Relocating Relative Addresses where relocation constant = *L1* + *L2* + *L3* + *L4*

ALGORITHM

Algorithm 13.4(a) Starting address computation
This algorithm is used to compute the start address of a module.

Input: Module length

Output: Starting address of next module

Step 1 **[Initialization]** Relocation constant, $B \leftarrow 0$.

Step 2 **[Module Input]** Get the next module M_i of length L_i.

Step 3 **[Process]** Process the module M_i with the help of relocation constant B. [see Algorithm 13.4(b)]

Step 4 **[Modify Relocation Constant]** $B \leftarrow B + L_i$.

Step 5 **[Increment** i**]** $i \leftarrow i+1$

Step 6 If $i <$ number of modules then goto Step 2.

Step 7 **[Termination]** Stop.

ALGORITHM

Algorithm 13.4(b) Process the module M_i.

Input: Module M_i and its base address

Output: Modified module M_i

Step 1 Get $t_{base} \leftarrow$ the translated base address of M_i.

Step 2 **[Compute Relocation Factor]** $rf \leftarrow B - t_{base}$

Step 3 Repeat

Search sequentially the relocation bit from M_i.

If found then

(1) get relative address t_{sym}.

(2) modify $l_{sym} \leftarrow t_{sym} + rf$.

Write the instruction/modified instruction in the output linked file

until end of the module.

Step 4 Return.

Illustration 13.12 Suppose we have four modules M_1, M_2, M_3, and M_4 as described in the table below.

Module Name	Module Length	Base Address During Translation	Relative Address
M_1	32	100	
M_2	128	0	20,32
M_3	16	1000	1004,1008
M_4	256	24	30

After relocation, the linker returns the following table. Assume that the base address of link module $B = 0$.

Module Name	Module Size	Base address of the module	Relocation factor *rf*	Relative Address
M_1	32	0	−100	
M_2	128	32	32	52,64
M_3	16	160	−840	164, 168
M_4	256	176	152	182

13.10 LINKING METHODS

In general, linking methods can be viewed as follows:

- Static linking
  - Linkage Editors
- Library Linking
  - Static Library Linking
  - Dynamic Library Linking
  - Shared/Dynamic Link Library Linking
- Dynamic linking
  - Implicit Dynamic Linking
  - Explicit Dynamic Linking
- Object Linking and Embedding (OLE)

13.11 STATIC LINKING

Static linking is used to combine multiple functions/modules into a single executable module. This linking process takes place once when the executable module is created. All internal references to the functions within a module are resolved at this time.

Advantages Some of the advantages of static linking are listed below:

1. It creates a self-contained independent program.
2. In some cases, an entire application is built out of a single statically linked executable module.
3. In this case, the resulting module is an autonomous unit that can execute without referencing any other modules.

Disadvantages Some of the disadvantages of static linking are listed below:

1. The resulting executable file becomes larger.
2. It consumes a larger amount of system resources.
3. It takes longer time to load into memory.

Illustration 13.13 Many DOS-based applications consist of a single executable module.

13.12 DYNAMIC LINKING

The task of dynamic linking is to link the modules that reference each other. It must be linked together dynamically when they are executed. Dynamic linking procedures are also termed as *segment systems*, *chaining* or *dynamic overlaps*, or *deferred linking*.

13.12.1 Outline

Dynamic linking is a linking process that allows one program module to execute a function program where this function is located in another program module stored in a hard disk. The

header record in the module file contains information about all the exported functions. Modules that wish to call these functions can either link to them implicitly or explicitly.

In implicit linking, the imported functions from the header fields of the module are identified and these functions are linked automatically.

In explicit linking: the application is responsible for dynamically finding the address of the exported functions.

Use Dynamic linking is a critical architectural component of Microsoft Windows. Almost all Windows-based applications execute functions found in external modules.

Advantages Some of the advantages of dynamic linking are listed below.

1. The memory and disk requirements for an application are reduced.
2. A portion of an application can be upgraded without replacing the entire application.

Disadvantages Some of the disadvantages of dynamic linking are listed below.

1. Problems occur when links are not completed.
2. The negative effects of modules occur due to some improper linking.

13.12.2 Principles of Dynamic Linking

In general, the dynamic linking process performs the following tasks.

Task 1 The module that contains the external function must be located.

Task 2 The address of the function within this module is searched.

Task 3 Once this address is found, the calling module can use this external function just as if it was linked statically.

Illustration 13.14 Almost all Microsoft Windows-based applications have references to external functions that must be resolved dynamically. In fact, all of the functions provided by the Windows API (application programming interface) are located in dynamically linkable modules. Thus, dynamic linking is a fundamental property of all Windows-based applications.

Exported Functions In dynamic linking, a function that is defined (or contained) in one module but can be used by other modules is called an *exported function*.

In order to *export/import* a function, the developer assigns a unique number to the function called its *ordinal*. In many cases, a unique function name is also assigned to the exported function. The function's ordinal number and name are the keys used to identify an exported/imported function.

Illustration 13.15 Suppose a module `XMOD.DLL` exports a single function, say, `XFUNCTION`. Assume that the developer is assigned the ordinal number 2 for the function name `XFUNCTION` when this module (`XMOD.DLL`) is created. This ordinal number and name would be placed in the header record in the module file. Another module could use either value to find the address of this exported function in `XMOD.DLL`.

Imported Functions The functions that are *called from* within one module but actually reside in another module are called *imported functions*.

Dynamic linking occurs when the ordinal number and name information are used to find the exported function's address. If the specified module is found, its headers are examined for the desired function ordinal number or name. If the specified function is also found, its address is dynamically placed in the calling module's code.

Thus, dynamic linking occurs when one module imports, or uses, a function which was exported, or made available, by another module.

Illustration 13.16 Suppose an application, XAPPL wants to use an exported function XFUNCTION in the module XMOD.DLL. In such a case, the application XAPPL must specify that (i) the imported function is in the module called XMOD.DLL and (2) the function name is XFUNCTION or the function ordinal number is 2. If the module and the function are both found, then XAPPL.EXE can call this function.

13.12.3 Methods of Dynamic Linking

There are two methods of dynamic linking:

- Implicit dynamic linking
  - Linking loaders
- Explicit dynamic linking
  - Dynamic linker/loader

Implicit dynamic linking An implicit dynamic linking can automatically handle the *process of identifying exported functions*. It uses imported modules that contain header record for each implicitly imported function. Each header record contains the *exporting module name* and the *exported function ordinal number* or *name*.

Illustration 13.17 Microsoft Windows loads a module. It checks the fields of module header record for imported functions. If any imported functions are supplied, the Windows operating system searches for the exporting module and loads it into memory, if necessary. Once the exporting module is loaded, Windows resolves the address of the imported functions automatically. This process occurs recursively until all implicit function references are resolved.

Advantages

Some advantages of implicit dynamic linking are:

1. It is an easy technique to follow.
2. It is the most common dynamic linking method.

Disadvantages

Some disadvantages of implicit dynamic linking are:

1. **[Extension Problem]** In the case of Windows, all implicitly linked exporting module files must use an extension of .DLL. The module name included in the import header record does not contain an extension value. By default, Windows appends the value .DLL to the module name provided.
2. **[Path Problem]** Windows must use its file search logic in order to find the desired module. The module name included in the import header record does not contain any path information. By default, Windows will search for implicitly linked modules using a pre-defined set of paths.
3. Windows must terminate an application that references an implicitly linked module or function that cannot be found.

Explicit dynamic linking In an explicit dynamic linking, the process of linking to exported functions is handled by the application itself.

Illustration 13.18 In the case of explicit dynamic linking, Windows provides functions that an application can use to explicitly load a module into memory. Once loaded, another Windows function can be used to extract an exported function's address using an ordinal number or name. If both of these functions are successful, the application can call the exported function.

Advantages Some of the advantages of explicit dynamic linking are listed below:

1. Explicitly loaded module files can use any extension, not just .DLL.
2. Requests for explicitly loaded module files can contain path information. These modules do not have to reside in Windows search directories.
3. The explicit dynamic linking functions return an error if either an exporting module or an exported function is not found. The application can handle either of these conditions without terminating.

Disadvantages One major disadvantage of explicit dynamic linking is from the implementation point of view. The explicit dynamic linking is more complicated than other linking procedures.

Features Some of the features of explicit dynamic linking are listed below:

1. Most commercial applications employ both implicit and explicit dynamic linking.
2. The choice of implicit or explicit dynamic linking technique depends on the design of the software. From the application user's point of view, once the modules are properly dynamically linked, there is no difference in the two techniques.

Implicit vs explicit dynamic linking Some comparisons between implicit and explicit dynamic linking procedures are discussed in Table 13.4.

Table 13.4 Comparison between implicit and explicit dynamic linking

Implicit Dynamic Linking	Explicit Dynamic Linking
(a) Process identifies exported functions.	(a) Process links to exported function.
(b) Tasks are done by the Windows OS.	(b) Tasks are done by the application itself.
(c) Applications can only find functions that are linked implicitly.	(c) Explicitly linked functions do not generate Windows module file header record that can be parsed by these applications.

Static vs dynamic linking Some comparisons between static and dynamic linking procedures are discussed in Table 13.5.

Table 13.5 Comparison between static and dynamic linking

Static Linking	Dynamic Linking
(a) All addresses are resolved before the program is loaded into the memory for execution.	(a) It links modules on demand during execution.
(b) No address exception occurs during execution.	(b) Address exception occurs.
(c) No address exception handler routine is necessary.	(c) Address exception handler routine is necessary.

Task of an address exception handler routine Suppose an address exception occurs. Then the *exception handler routine* is called and it does the following tasks.

1. It checks the logical address.
   (a) If it refers to a routine or a variable, then it must be dynamically linked and puts the information regarding the dynamically linkable objects in a table known as the *link table*.
   (b) If this referred address is valid, then the memory management state of the program is adjusted to the allowed address range for this program.
2. If the instruction causes any exception, then restart.

Dynamic linking of shared libraries The dynamic linking of shared libraries is basically used in the UNIX system. This has the following advantages:

1. It does not resolve unresolved symbols in a program until it runs. This reduces the time to build an image.
2. It helps save disk space since shared libraries do not have to be statically bounded into a program image.
3. It reduces memory cost since shared libraries can be shared between many executing images.

Dynamic linking in windows Static linked modules can contain references to functions that should be resolved during the execution. Window uses dynamic linking to provide a method for EXE modules to share common functions.

By placing commonly used functions in a DLL module, all Windows-based applications can access these functions without having to include them in their module file. This technique reduces the disk and memory usage of Windows-based applications.

Dynamic linking also allows functions found in DLL modules to be updated independently. Thus, if a developer makes an improvement to an exported function, changing the function's module file effectively updates all applications that link to the function.

13.13 LIBRARY LINKING

Library linking describes the inclusion of one or more of these software libraries into a new program. There are multiple types of linking.

1. Static Library Linking
2. Dynamic Library Linking
3. Dynamic Link Library (DLL) Linking or Shared Library Linking

Note that library files are described by the extension `.inc`, `.s`, `.so`, `.lib`, etc. in different environments.

Note First search the desired library before linking. This library searching can be done in different ways, which depend upon the existing system.

Illustration 13.19 One possible library order is described below:

1. The directory from which the application is loaded
2. The current directory
3. The system director
4. The 16-bit system directory
5. The Windows directory
6. The directories that are listed in the PATH environment variable

13.13.1 Static Library Linking

Static library linking is a linking technique in which the whole library or its required routines defined as external symbols are embedded into the program executable at compile time by a linker. A linker is a separate utility, which takes one or more libraries and object files (which are previously generated by a compiler or an assembler) and produces an actual executable file.

Disadvantages Some of the disadvantages of static library linking are as follows:

1. The resulting executable file becomes larger.
2. It consumes a larger amount of system resources.
3. It takes longer time to load into memory, and the behaviour of executable files cannot be changed without re-linking.

Illustration 13.20 The libraries that are traditionally designed to be statically linked include the ANSI C standard library and the ALIB assembler library.

13.13.2 Dynamic Library Linking

Dynamic library linking is also a linking technique in which a library is loaded separately by the loader of the operating system during the load time (or run time).

Load-time linking Most operating systems resolve external dependencies such as libraries (or imports) as a part of the loading process. For these systems, the executable programs contain a table called an *import directory*, which is a variable-length array of imports. Each element in the array contains a name of a library. The loader searches the hard disk for the required library, loads it into primary memory at an unpredictable location, and updates the executable with the location of the library in the memory. The executable program uses this information to call functions and access the data stored in the library. This type of dynamic linking is called *load-time linking*.

Disadvantages One disadvantage of load-time linking is that it is one of the most complex routines that the loader performs while loading an application.

Uses Load-time linking is used by most operating systems, including Windows and Linux.

Run-time linking The process of runtime linking is suitable for the operating system that resolves dependencies at run time. For these operating systems, the executable program calls an API (application programming interface) and also passes the name of a library file,

(i) a function number within the library and
(ii) the function's parameters.

The operating system resolves the import immediately and calls the appropriate function on behalf of the application. This type of dynamic linking is called *run-time linking*.

Disadvantages Some disadvantages of run-time linking are that (i) it is incredibly slow and (ii) it negatively affects the performance of the executable program.

Use Run-time linking is rarely used by modern operating systems.

13.13.3 Shared/Dynamic Link Library (DLL) Linking

A shared library contains a set of object modules for a class of common functions to be used by different programs currently running in a computer. But the shared library itself is developed based on two different concepts: (i) the sharing of code located on disk by programs that are not relocated and (ii) the sharing of code in memory. Suppose the library routines are loaded into the memory only once, although a different user's program calls a library routine several times during execution. These library programs are written using position-independent coding techniques. The generation of object code for a shared library is an independent task. In dynamic linking, the library (i.e., a *dynamic link library* (DLL) or *shared library*) is linked to the executable file and is stored separately on the hard disk.

Advantages Some advantages of shared/dynamic link library (DLL) linking are given below:

1. DLL is loaded only when needed by an application.
2. (*Stateless*) Multiple applications can use the same copy of the library at the same time.
3. There is no need for the operating system to load multiple instances of the library into the memory concurrently.

Disadvantages Some disadvantages of shared/dynamic link library (DLL) linking are given below:

1. Dynamic linking involves dependency of the executables on the separately stored libraries in order to function properly.
2. (*DLL-hell*) If the library is deleted, moved, renamed, or replaced with an incompatible version, the executable could malfunction.

Illustration 13.21 All Windows *.DLL files are dynamically linked libraries.

Shared library linking procedure Routines for each shared libraries are first compiled and then linked independently from the application. If the shared library is used in conjunction with the application, then one can observe the following problems.

1. Offset of various routines can change.
2. This process of linking cannot take care of the shared libraries for this application due to the lack of information about the location of the shared library during execution of the application.
3. The corresponding loader will handle the situation. If the shared library routines are written in position-independent code, then it provides a good solution.

Suppose an application program calls a shared routine *dtoc*, which converts a date to a character string. In such a case, it performs Algorithm 13.5 for shared library linking.

ALGORITHM

Algorithm 13.5 Shared library linking

Input: Shared library, application program.

Output: linked program

Step 1 **[Input]** Get a line from the application program.

Step 2 **[Searching]** Scan the line to search a called routine from any shared library.

Step 2.1 If search is successful then

Step 2.2 get the name of the routine (say, *dtoc*). Its real routine (say, *real_dtoc*) is in the shared library (say, *MyShareLib.so*) file (this is different from program file). These real routines are listed in a table which is known as *Procedure Linkage Table* (PLTab) with their offsets for this shared library. A vector contains the start address of the memory at which the shared library (i.e., *MyShareLib.so*) has been loaded.

Step 2.3 the linker resolves the undefined symbol in the object modules of the application with the help of PLTab entries of the shared library.

Step 3 If not end of application program then goto Step 1

Step 4 **[Termination]** Stop

Note 1. While loading the application program, the start address of the memory at which the shared library (i.e., *MyShareLib.so*) has been loaded is known since the shared library is already present in the memory or it is loaded at this time.
2. The entries in the procedure linkage table (PLTab) is a fixed table. So, it may be implemented by using hash data structure (design of a perfect hash function is also possible).

Practical problem A computer may need several versions of the same library because some programs use the older versions and some are newer versions. Also the versions of the shared libraries are not always downward compatible due to the alteration of program code for the library routines. So, this alteration causes a change of offset in its PLTab. Hence, the PLTab is different for different versions of the same library. This is a serious and common problem for the correct execution of all applications in the system. This problem may be termed as the *versioning problem*.

Remedy A possible solution to this versioning problem may be either (i) to rename the shared library everytime but this overhead is on the programmer obviously it loses the beauty of sharing, or (2) to integrate a numbering scheme in the file name.

13.14 OBJECT LINKING AND EMBEDDING (OLE)

A system allows data or images originally created in one application to be copied into and linked or embedded into a secondary document or image file created by another application. Linking creates a link to the original document in such a way that it automatically updates the linked secondary document. Embedded objects are not automatically updated, but their source document and the application by which they were created can be accessed from the secondary document.

13.15 MODULE LINKING

In this section, we discuss different module linking features.

Module scanning The module scanner scans the module file headers looking for implicitly linked functions for an application. These applications use this information to determine relationships between modules.

Limitations In practice, it is impossible to know which exported functions and modules are used in a module by scanning/examining the module file.

Module types In dynamic linking, there are two types of modules:

1. Executable (EXE) modules
2. Dynamic link library (DLL) modules

Executable (EXE) modules The main portion of an application program is an EXE module. A new application starts under Windows by loading the .EXE file associated with the application. Once loaded, each EXE module performs a distinct task under windows that competes for CPU time with all other loaded EXE modules.

Dynamic link library (DLL) modules The purpose of the DLL module is to provide support for loaded applications. A DLL module does not get scheduled to run like an EXE module. Rather, it provides external functions that can be *called by applications* that are currently active.

EXE modules vs DLL modules EXE and DLL modules are compared in Table 13.6.

Table 13.6 Comparison between EXE and DLL Modules

EXE Modules	DLL Modules
(a) These are the main portion of an application program.	(a) Provides support for loaded applications.
(b) Execute for an application.	(b) Do not execute like EXE module.
(c) Most EXE modules only import functions from DLL modules.	(c) DLL modules can export functions as well as import functions from other DLL modules.

13.16 DESIGN OF LINKERS

A linker translates a given set of object programs or modules from a target machine to an executable program for the same target machine. This translation process generates a symbol table and a segment table from these object programs and also uses these tables for the translation.

Different linkers are available for use. These are mostly classified as two-pass, one-pass, and multi-pass (more than 2) linkers.

13.16.1 Two-Pass Linkers

A linker is fundamentally a two-pass process. A linker accepts input as a set of input object files, libraries, control, and command files. It produces output as an executable file and other ancillary information such as a load map or a file containing debugger symbols, etc.

Basic philosophy The philosophy behind the principle of the design of a two-pass linker is as follows.

Pass I (Table creation) In the first pass, it reads the sequence of machine operation (i.e., object) modules as individual loading units and resolves their allocation requirements.

Pass II (Translates the constructs) In the second pass, it translates the constructs in the virtual machine language, i.e., it reads the sequence of machine operation modules, loads and links them according to the result of Pass I, and then relocates them according to their relocation information.

Working principles Each input file contains a set of segments, contiguous chunks of code or data to be placed in the output file. Each input file also contains at least one symbol table. Some symbols are exported and defined within the file for use in other files. These are generally the names of routines within the file that can be called from elsewhere. Other symbols are imported and used in the file but not defined. This generally includes the names of routines called from but not present in the file.

When a linker runs, its first task is to scan each input file and find out the size of each segment, and also collect the definition of each segments and references of all of the symbols. It creates a segment table for listing all the segments defined in the input files, and also creates a global symbol table with all of the symbols imported or exported (see Fig. 13.6).

At the end of the first pass, the linker assigns numeric locations to the symbol in the global symbol table, determines the sizes and location of the segments for the output address space, and also figures out where everything goes in the output file.

In the second pass, the linker uses the information collected in the first pass to control the actual linking process (see Fig. 13.6). It reads and relocates the object code, substituting numeric addresses for symbol references, and adjusting memory addresses in the code and data to reflect

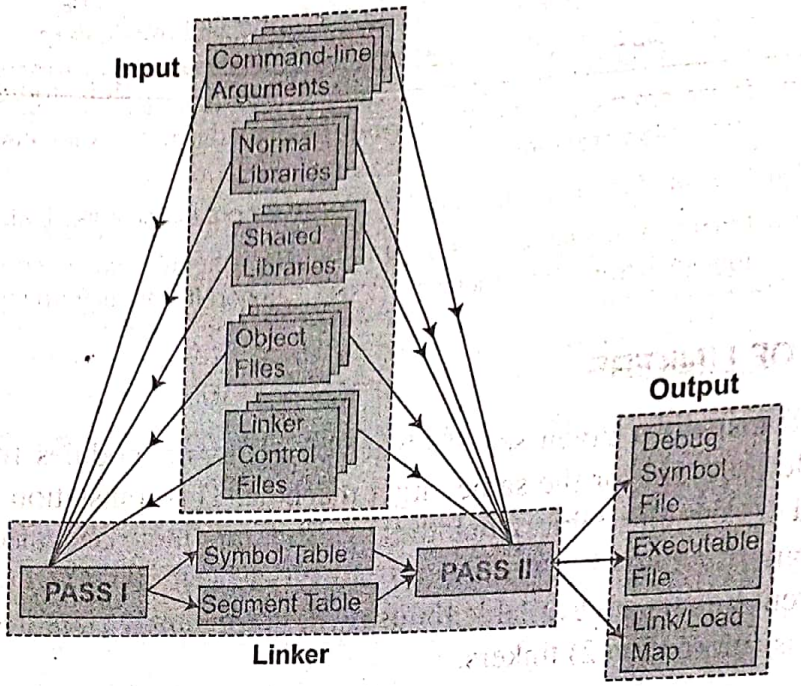

Fig. 13.6 A two-pass linker with different input files, intermediate tables, and output files

relocated segment addresses. Then it writes the relocated code to the output file. At the end, writes the output file to the disk, generally with header information, the relocated segments, ar symbol table information.

Notes 1. If the program uses dynamic linking, the symbol table contains the information about the run time linker. This will need to resolve dynamic symbols. In many cases, the linker itself will generate sma amounts of code or data in the output file, such as *glue code* used to call routines in overlays or dynamicall linked libraries, or an array of pointers to initialization routines that need to be called at program startup time.

2. The program may or may not use dynamic linking but the output file may contain the information abou relinking or debugging in the symbol table. This information is not used by the program itself, but may be used by other programs that deal with this output file.

3. Some object file formats are 'relinkable', which means that the output file from one linker run can be used as the input to a subsequent linker run. This requires (i) the output file to contain a symbol table like one in an input file and (ii) all of the other auxiliary information present in an input file.

4. All object file formats have provision for debugging symbols, when the program is run under the control of a debugger. The debugger can use these symbols when the programmer controls the program in terms of the line numbers and names used in the source program. Depending upon the object format, the debugging symbols may be intermixed in a single symbol table with symbols needed by the linker, or there may be two separate tables: one for the linker and other for the debugger.

Content of input files Each input file contains the following:

1. A set of segments
2. Each segment contains code and/or data to be placed in the output file
3. At least one symbol table
4. Some symbols are exported, i.e., define within the file for use in other files
5. Some (remaining) symbols are imported, which means that they are used in the file but not defined

Outline of a two-pass linker Pass I of a two-pass linker constructs a global symbol table and segment tables after merging the definition tables of the several segments into one. During Pass II, the linker updates the address field related to Pass I. Pass II of the linker involves patching

the external references. The object code is copied after the necessary update. The two-pass linker will perform the following steps in Pass I and Pass II.

Pass I

1. Scan all the input files.
2. Find the sizes of the segments.
3. Collect the definitions and references of all symbols.
4. Create a segment table listing from the segments of the input file.
5. Create a symbol table with import and export symbols in the program.
6. Assign address to symbols.
7. Determine the sizes and location of the segments in the output address.

Pass II

1. Read the object code.
2. Relocate the object code.
3. Substitute numeric address for symbol references.
4. Adjust memory addresses in code and data to reflect relocated segment address.
5. Write the relocated code to the output file.
6. Write header information, the relocated segments and symbol table information.

A two-pass linking method is described in Algorithm 13.6. Also, an alternative method of a two-pass linker algorithm is described in Algorithm 13.7.

ALGORITHM

Algorithm 13.6 Two-Pass Linker

Input: Object modules with symbol tables and segments.

Output: Executable or load module

Step 1 **[Initialization]** Create an empty load module and an empty symbol table.

Pass I

Step 2 **[Input]** Read the next object module or library name from the command line.

Step 3 If it is an object module, then

- (a) insert it into the load module,
- (b) relocate it and its symbols,
- (c) merge its symbol table into the global symbol table,
- (d) for each undefined external reference in the object module's symbol table:
  - (i) if the symbol is already in the global symbol table, then copy the value to the object module and
  - (ii) if not, then insert it (as undefined) into the global symbol table and make a note to fix up the symbol later (next pass, i.e., Pass II).
- (e) For each defined symbol in the object module, fix up all previous references to the symbol (in object modules loaded earlier).

Step 4 If it is a library, then

- (a) find each undefined symbol in the global symbol table and
- (b) if the symbol is defined in a module in this library, then load the object module.

Step 5 **[Loop]** Repeat from Step 2 to Step 4 until end of the command line.

Pass II

Step 6 Repeat from Step 2 to Step 5 from the beginning of the command line.

Step 7 **[Termination]** Stop.

ALGORITHM

Algorithm 13.7 Two-Pass Linker (Alternate Approach)

The linker brings all modules one-by-one referenced in the program and combines them int single executable unit. To do this, it must construct a *global external symbol directory* (GES This data structure contains the same information found in the local ESDs (*external sym directory*), but its values are given in terms of a *global location counter value* (GLC). The G represents the single relative base point for all relative addresses in the entire module. T individual relative address of each module 0, 1, ..., M; 0, 1, ..., N; 0, 1, ..., P; ... are conver by the linker, using the GLC, into global relative addresses 0, 1, 2, 3, ..., ($M + N + P + \ldots$

Input: Modules

Output: Load module

Pass I

Step 1 **[Initialization]** The entire unit will be relative to location 0, i.e., Set GLC = 0.

Step 2 **[Bring the *next* module for linking]** The units automatically included in eve translation of this language, as well as the program units explicitly specified by the use

Step 3 Enter all external definitions from the local ESD of this module into the GESD. Set th value field of the symbol to *local ESD value* + *GLC*.

Step 4 Repeat this step for each external reference in the ESD of the current module whos definition has already been entered into the GESD.

1. Use the pointer field of the ESD to access the chain of instructions that referenc the external symbol.
2. Replace the pointer in the address field of each instruction in the chain with th value field of that symbol stored in the GESD.

Keep going until it comes to the end of the chain of references.

Step 5 Set GLC = GLC +length of the last program module.

Step 6 Repeat Steps 2 through 5 until all modules referenced in this program have bee processed.

Pass II

Step 7 **[Process Forward References]** Make a second pass through the local ESD of al modules just proceed after looking for the external references that were not satisfied or the first pass. (This is necessary to handle the case in which a reference to a module is made before the module itself is encountered. This is called a *forward reference.*) If any forward references are found, apply Step 4 and satisfy all references.

Step 8 **[Linker Error]** If any unsatisfied external references remain, then it is a *linker error.* The user has referred to a nonexistent program unit. Typically, most linkers place the address of an operating system error routine in place of the non-existent reference.

Step 9 **[Termination]** Stop.

13.16.2 One-Pass Linkers

Sometimes, a few two-pass linkers appear to work in one pass. This is an implementation trick that uses a buffering technique. Here, some or all of the contents of the input file are put in the memory or disk during the linking process. Then the buffered material is read later. This technique does not fundamentally change the two-pass nature of the linking.

An independent one-pass linker is not found in the computer literature. Alternatively, a true one-pass linker issues to the loader a directive to effect the patch. It may be a very restrictive kind of linkers—not much use in practice.

Philosophy If the references among machine operation modules (MOM) in object file are always backward, then the linker can perform its task in just one pass.

Outline

1. The allocation of all modules referenced in the current MOM is performed.
2. Symbols involved in linking and relocation are either undefined or defined in the previous MOMs and are in the memory map constructed so far.

Illustration 13.22 The load-and-go assembler combines one full pass with a number of mini-passes.

13.16.3 Three-Pass Linking

Truly speaking, an independent, three-pass linker is not available in systems programming literature, even though different kind of linking are performed to obtain an executable program from a set of object programs at different stages. This is explained in Illustration 13.23.

Illustration 13.23 The compiler driver is invoked with a special option

```
gcc -shared -fPIC -o libfoo.so a.o b.o
```

This command tells the compiler driver to generate a shared library, `libfoo.so`, comprised of the object modules `a.o` and `b.o`. The `-fPIC` option tells the compiler to generate *position independent code* (PIC). Now, suppose the main object module is `bar.o`, which has dependencies on `a.o` and `b.o`. In this case, the linker is invoked with

```
gcc bar.o ./libfoo.so
```

This command creates an executable file, `a.out`. This `a.out` file can be linked to `libfoo.so` at load time. Here, `a.out` does not contain the object modules `a.o` and `b.o`. The executable file simply contains some relocation information and symbol table. They allow references to code and data in `libfoo.so` to be resolved at run time. Thus, `a.out` here is a partially executable file that still has its dependency in `libfoo.so`. The executable also contains `a.interp` section that contains the name of the dynamic linker, which itself is a shared object on Linux systems (`ld-linux.so`). When the executable file `a.out` is loaded into memory, the loader passes control to the dynamic linker as shown in Fig. 13.7. The dynamic linker contains some start-up code that maps the shared libraries to the program's address space. It then does the following:

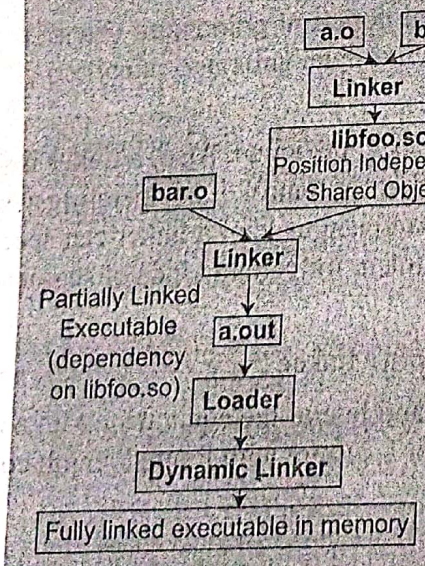

Fig. 13.7 An executable program development using different linkers at three stages

1. Relocates the text and data of `libfoo.so` into the memory segment
2. Relocates any references in `a.out` to symbols defined by `libfoo.so`

Finally, the dynamic linker produces a fully linked executable program in memory.

13.16.4 Complexity

The complexity of a linker algorithm is determined by the following three parameters:

1. The number of machine operation modules in the object 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 process each machine operation module

Time complexity It can be reduced after imposing some restrictions on the relationship between machine operation modules and components within machine operation module.

Space complexity It can be reduced by ordering MOM components according to the order between the operations performed by the loader

$$Allocation \rightarrow Loading \rightarrow Linking \rightarrow Relocation$$

The information used during allocation is contained in the external symbol dictionary (ESD) component of the MOM. Thus, ESD needs to be the first component of the MOM. The information used during Loading and Linking is in TEXT and ESD. Thus, TEXT needs to be the second component of the MOM. The information used by relocation is in the RLD component of the MOM. Hence, the relocation and linking directory (RLD) needs to be the third component of the MOM.

13.17 LINKER COMMAND LANGUAGES

A linker controls the linking process via some command. A command line may accept various options. At least the linker needs the list of *object files* and *libraries* (if any) to link. Generally, there is a long list of possible options such as

1. whether to keep debugging symbols,
2. whether to use shared or unshared libraries, and
3. which of several possible output formats to use.

A linker specifies the address at which the linked code is to be bound. Also, a linker is used to link a system kernel or other program that does not run under control of an operating system. It supports multiple code and data segments. The command language of a linker can specify the order in which segments are to be linked, special treatment for certain kinds of segments, and other application-specific options.

There are four common techniques to pass commands to a linker.

Command line Most linking systems have a command line or the equivalent with a mixture of file names and switches. A command file is used for a system with limited length command lines.

Illustration 13.24 UNIX and Windows

Intermixed with object files This is an obsolete linking technique for IBM mainframe linkers which accept alternating object files and linker commands in a single input file via punched card decks.

Illustration 13.25 IBM mainframe linker

Embedded in object files Some object file format allow to process linker command. These commands are embedded inside object files. This permits a compiler to pass any options needed to link an object file in the file itself.

Illustration 13.26 The C compiler passes commands to search the standard C library.

Separate configuration language A few linkers have a full fledged configuration language to control linking. This is a complex control language. A programmer may be used to specify (1) the order in which segments should be linked, (2) rules for combining similar segments, (3) segment addresses, and (4) a wide range of other options. Other linkers have less complex languages to handle specific features such as programmer-defined overlays.

Illustration 13.27 The GNU linker can handle an enormous range of object file formats, machine architectures, and address space conventions.

13.18 AUTOMATIC LIBRARY SEARCHING

In this section, we discuss different library searching techniques in different environments such as during static linking, dynamic linking, remote searching, etc.

Library searching in static linking In the linking process, libraries are handled by a recursive searching technique. For example, an object module refers to an external symbol of a library module. Then the object file for this library module (related to that external symbol) is included in the set of file that needs to be linked. If this library module refers to other symbols, then the related object files for the libraries are also included in the set of file that need to be linked. This process of library file inclusion in the set continued until it exhausts all the referred symbols to resolve.

Note The searching process allows overriding the standard library routines by the user's defined routines (if any) in the program.

Library searching in dynamic linking The order of library searching in dynamic linking in a particular operating environment depends upon some mode of operation.

Illustration 13.28 In the Windows environment, *SetDllDirectory* function is used to set the mode as either enables or disable *SafeDllSearchMode*. This mode will guide the order of library searching in dynamic linking.

If *SafeDllSearchMode* is enabled, then the order of library searching in dynamic linking is as follows.

1. The directory from which the application is loaded
2. The system directory
3. The 16-bit system directory
4. The Windows directory
5. The current directory
6. The directories that are listed in the PATH environment variable

If *SafeDllSearchMode* is disabled, then the order of library searching in dynamic linking is as given in Illustration 13.19.

Remote library searching Remote file linking, which is also known as hot linking, is a common cause of high bandwidth usage. There are several ways to combat remote linking.

13.19 SOME POPULAR LINKERS

In this section, we discuss some popularly used linkers such as the MS-DOS linker and SUN OS linker.

13.19.1 MS-DOS Linker

The format for the MS-DOS linker is as follows:

LINKER *<Object module names>,<executable file name>, <Load origin>, <List of library files>*

Illustration 13.29 Suppose the object modules *sum.obj, mul.obj,* and *div.obj* are to be linked with the libraries *stdio.lib* and *math.lib*. The executable output file is *calculator.exe*. Assume that the load origin of the executable program is 1024. The following DOS LINKER will perform this task:

```
C>LINKER sum+mul+div,calculator,1024,stdio.lib,math.lib
```

Working principle The MS-DOS LINKER performs the linking task in two passes. In the first pass, the linker scans each object module listed in the command line and collect information concerning unresolved external symbols. In the second pass, the linker performs both relocation and linking tasks to form an executable program.

We assume that each object module follows the MS-DOS object file format. Here, an object module is a sequence of object records where each object record is related to some information as described in the MS-DOS object file format in Section 5.7.1.

ALGORITHM

Algorithm 13.8 MS-DOS Linker

Input: Object modules

Output: Executable modules

Pass I Construct NTAB (Name Table) from object records.

Step 1 Get the linker command line then extract the *load_origin* and list of object modules from it.

If *load_origin* found in the command line then

linked_origin ← *load_origin*

else

linked_origin ← *default value.*

Step 2 Repeat Step 3 to Step 4 for all object modules to be linked.

Step 3 Get an object module to be linked from the list of object modules.

Step 4 Process all the object records in the object module.

Step 4.1 If object record is LNAME then
Get the names from *namelist* field
Put these names in NAMELIST.

Step 4.2 If object record is SEGDEF then
k ← *name index* from *name index* field,
segment_name ← NAMELIST[*k*],
segment_Addr ← start address from *attributes* field which also indicate the nature absolute or relocatable.
If the *segment_name* is an absolute segment then
put (*segment_name, segment_Addr*) in NTAB.
If the *segment_name* is a relocatable segment and not combined with others then
1. align the address using *linked_origin* on
the next word or paragraph as indicated in the attributes field.
2. Put (*segment_name, linked_origin*) in NTAB
3. Update *linked_origin*
linked_origin ← *linked_origin*+ *segement length*

Step 4.3 If object record is PUBDEF then
j ← *base* field
segment_name ← NAMELIST[*j*]
symbol ← *name* field
segment_Addr ← load address of *segment_name* in NTAB.
symbol_Addr ← *segment_Addr* + *offset* field.
Put (*symbol, symbol_Addr*) in NTAB

Step 5 Goto Pass II.

Pass II Construct the executable program in *work-area* and write into the specified file in the LINKER command line.

Step 1 Get the linker command line then extract *load_origin*, the list of object modules and the name of the output file from it.
Object_Module_List ← list of object modules in the command line.

Step 2 Repeat Step 3 to Step 4 for all object modules to be linked until *Object_Module_List* becomes empty.

Step 3 Get an object module from *Object_Module_List*.

Step 4 Process all the object records in this object module.

Step 4.1 If the object record is LNAME then
Get the names from *namelist* field
Put these names in NAMELIST.

Step 4.2 If the object record is SEGDEF then
k ← *name index* from *name index* field,
segment_name ← NAMELIST[*k*],
load_Addr ← *load_Addr* from NTAB corresponding to *segment_name*
Put (*segment_name, load_Addr*) in SEGTAB.

Step 4.3 If object record is EXTDEF then
external_name ← name from EXTDEF record;
If *external_name* is not found in NTAB then
Search the *external_name* as a segment or Public definition from the object modules in the Library.
If search is unsuccessful then linking error and jump to Step 6.
else
Get the name of object module from the library.
Add the name of the object module into the *Object_Module_List*.
Perform Pass I for the new object module.
load_Addr ← *load_Addr* from NTAB corresponding to *external_name*
Put (*external_name, load_Addr*) in EXTTAB.

Step 4.4 If object record is LEDAT then
k ← *segment index* field,
d ← *data offset*,
Progload_origin ← SEGTAB[*k*].*load address*,
WorkArea_Addr ← *address of work_area* + *Progload_origin* –*load_origin* + *d*.
Move data from LEDATA into the memory area starting at the address *WorkArea_Addr*.

Step 4.5 If object record is FIXUPP then
For each FIXUPP specification do
begin
s ← *offset* from *locat* field,
fixAddr ← *WorkArea_Addr* + *s*,
Fix up addresses using (i) load address from SEGTAB or EXTTAB and (ii) the value of code in *locat* and *fix dat*.
end

Step 4.6 If object record is MODEND then
If *start addr* is specified then
Compute the corresponding load address and
Store it in the executable work area

Step 5 Copy the work area in the output file as given in the command line.

Step 6 **[Termination]** Stop.