BHCS15B: System Programming

Linker and Loader

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Linking
   1. *Static vs Dynamic Linking*
   2. *Combining Object Modules*
   3. *Pass I of Linking*
   4. *Pass II of Linking*
2. Library Linking
3. Position Independent Code (PIC)
4. Shared Library Linking
5. Loader

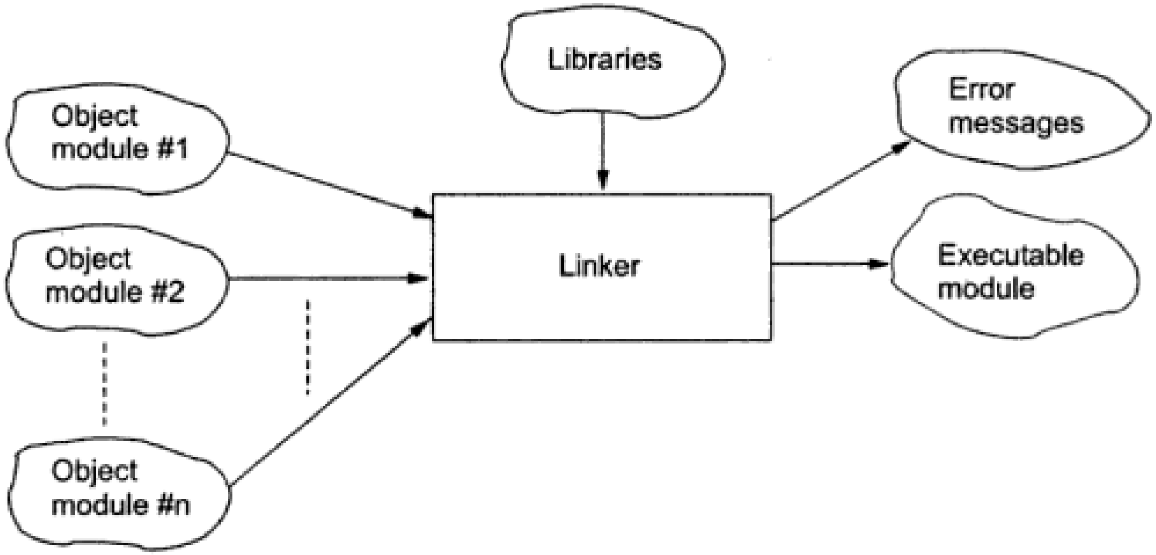
Linking

- Linking is the <u>process of combining different object modules into one executable file.</u>

- Assembler produces code assuming the start address of each section to be zero.
  - → When combined into one file, the offsets of the symbols in the program may need re-computation
  - → However after linking, the offsets should be calculated starting from the beginning of the program.

- Symbols defined in a section of some object module may be used as an external reference in some other module.
  - → Linker needs to fill up these blanks left by the assembler, from SYMTAB (external symbols, global symbols)

Linking – Input and output of a Linker



Linking – Input of a Linker (1)

1. Object files: the modules to be combined to create the executable version of the program.
   - → *Modules are specified in some special format.*

2. Static File: the standard pre-compiled libraries (archive files) containing individual object files for the library modules.
   - → *A recursive searching process, continues until a complete set of required library modules have been determined.*

3. Shared Library stubs: contains the common set of functions to be used by almost all the programs running in a system.
   - → *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).*

Linking – Output of a Linker (2)

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

2. External table: the shared library routines.

   → *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 reference for the loader to fill up and make the program fully ready for execution.*

Static vs. Dynamic Linking

- Static Linking : all addresses are resolved before the program is loaded into memory for execution.

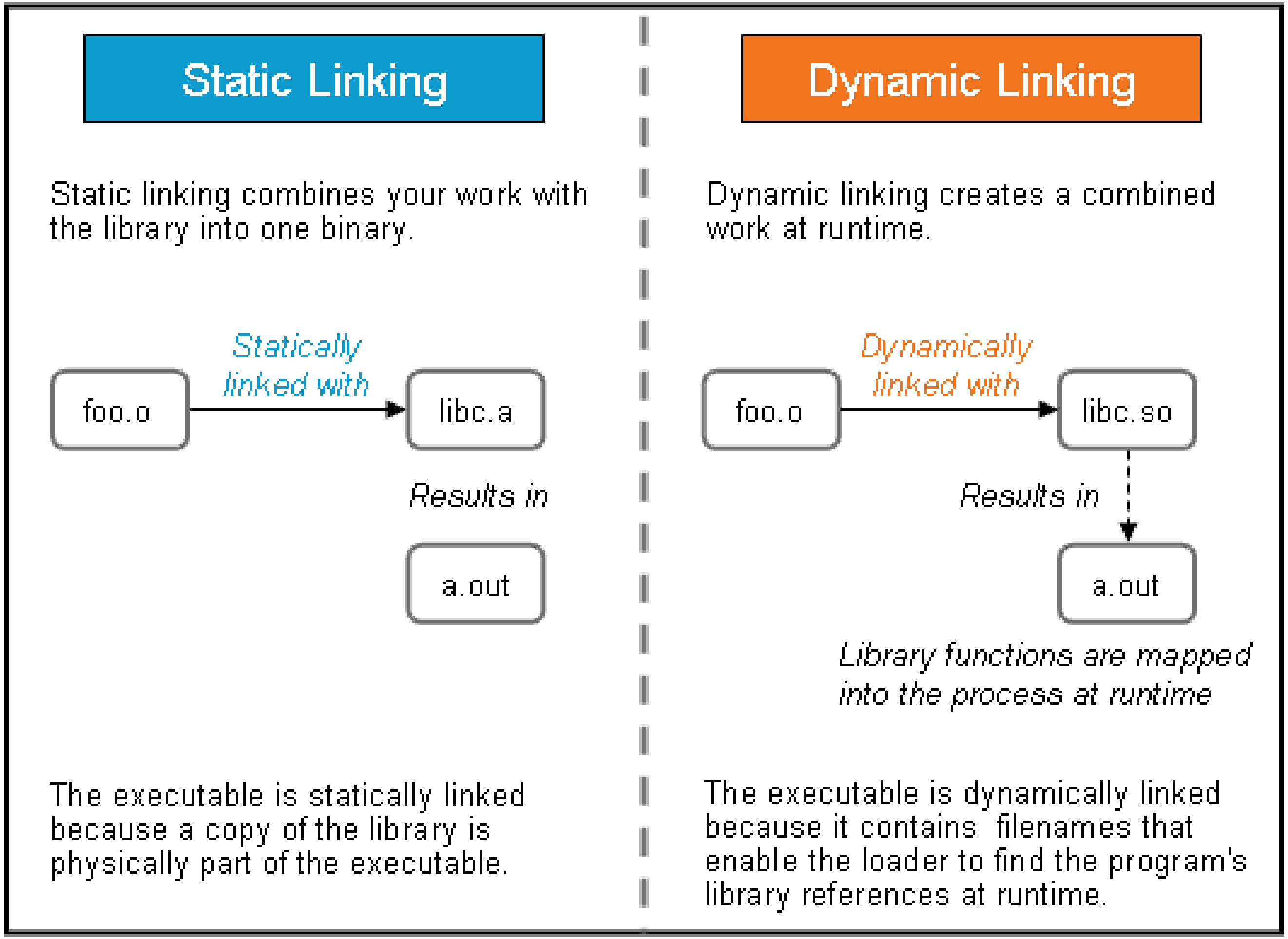
- Dynamic Linking (or Deferred Linking) : links modules on demand. In this case, when an address exception occurs, the exception handler is called which does the following:

  → 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.

Static vs. Dynamic Linking (2)



Combining Object Modules

- The *sections* of object modules are combined to produce a single executable module.

- The *SECTION* directive can have some associated attributes specified. All object file formats (elf, obj, etc.) have their corresponding similar set of attributes.

- For an *elf* object file format, *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.

  - 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. nowirte defines it as one which should not.

Combining Object Modules (2)

- progbits defines the section to be one with explicit contents stored in the object file: an ordinary code or data section. nobits defines the section to be one with no explicit contents given, such as BSS section.

- align= used with a trailing number, gives the alignment requirements of the section.
  - → align=x means that segment can start only at an address divisible by x

- Defaults assumed by NASM(if no above qualifiers are specified):

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

Combining Object Modules (3)

- 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 combining can be divided into two passes:

  1. *Pass I:* the relative position of all sections is computed, assuming the start offset of the file to be zero.

  2. *Pass II:* puts the code into the file.

Pass I of Linking

- It computes the start addresses of different sections in the input object modules and collects 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 structures are:

Section name	Start address	Size	Align

(a) Combined Section Table(CST)

Symbol name	Section name	Offset

(b) Public Definition Table (PDT)

Pass I of Linking (Flowchart)

- *Please see the flowchart:* *Pass I* of Linking

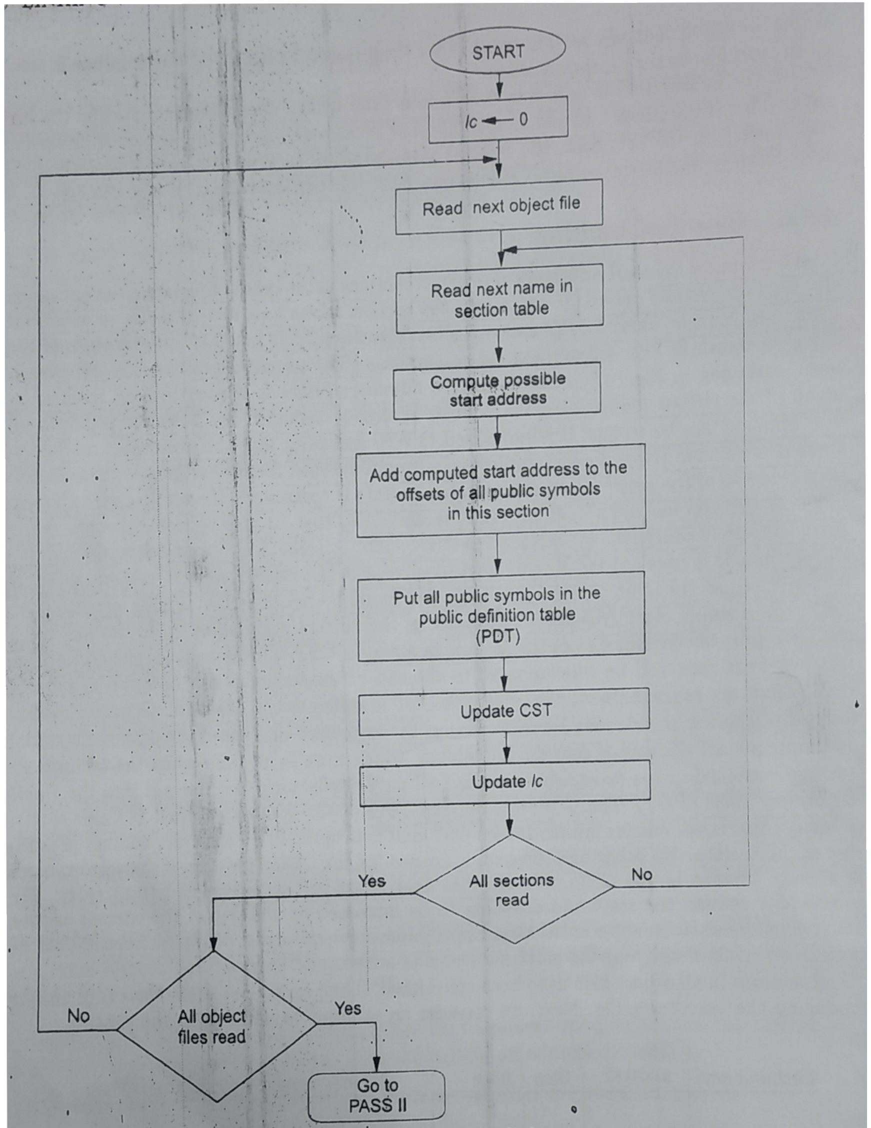


FIGURE 4.3 Pass I of the linker.

- 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. *lc* is initialized to zero.
- Next the sections are read from the object files one after the other.
- Next possible start address of the section in the executable module is computed.
- If the section is a new section(does not exist in CST) then 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.

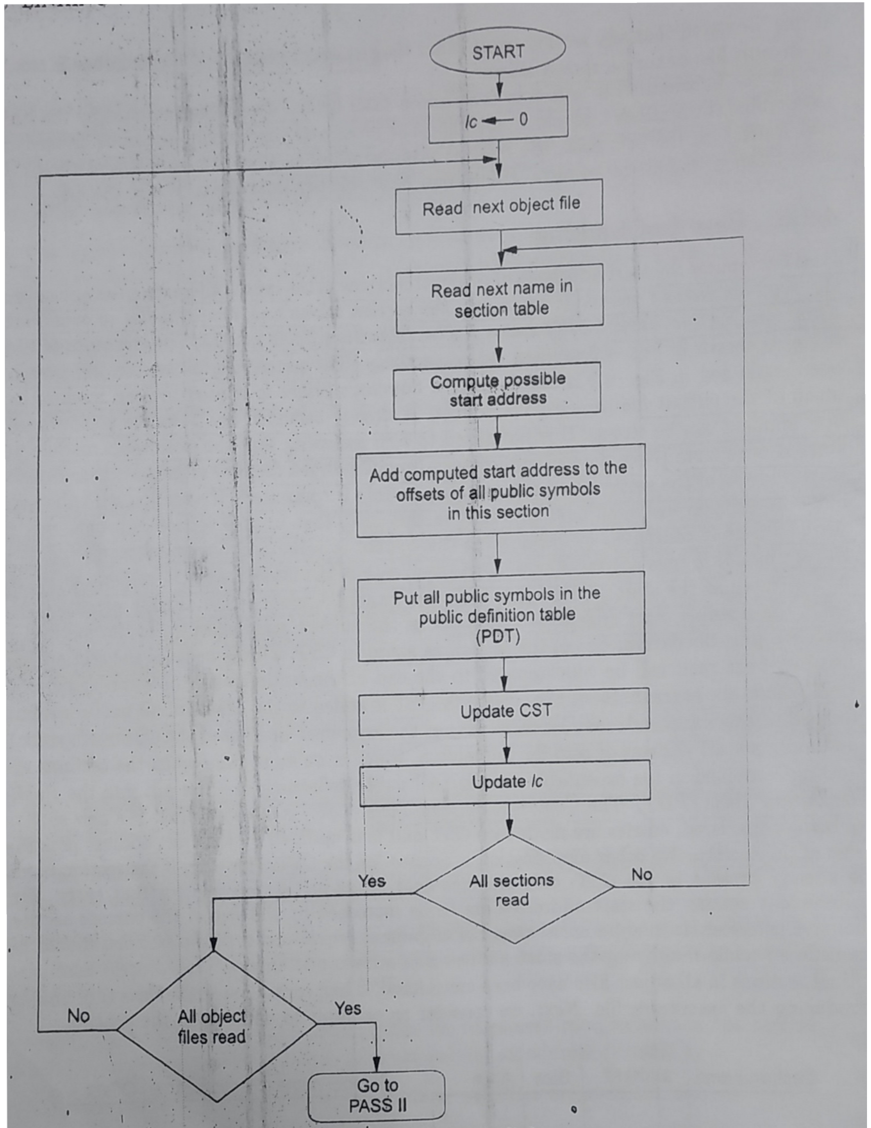


FIGURE 4.3 Pass I of the linker.

- If the section is a 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.

- Next the current size of the section in CST is added to the offsets of all public symbols defined in this section. All pubic definitions are now put into the PDT.

- The 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.

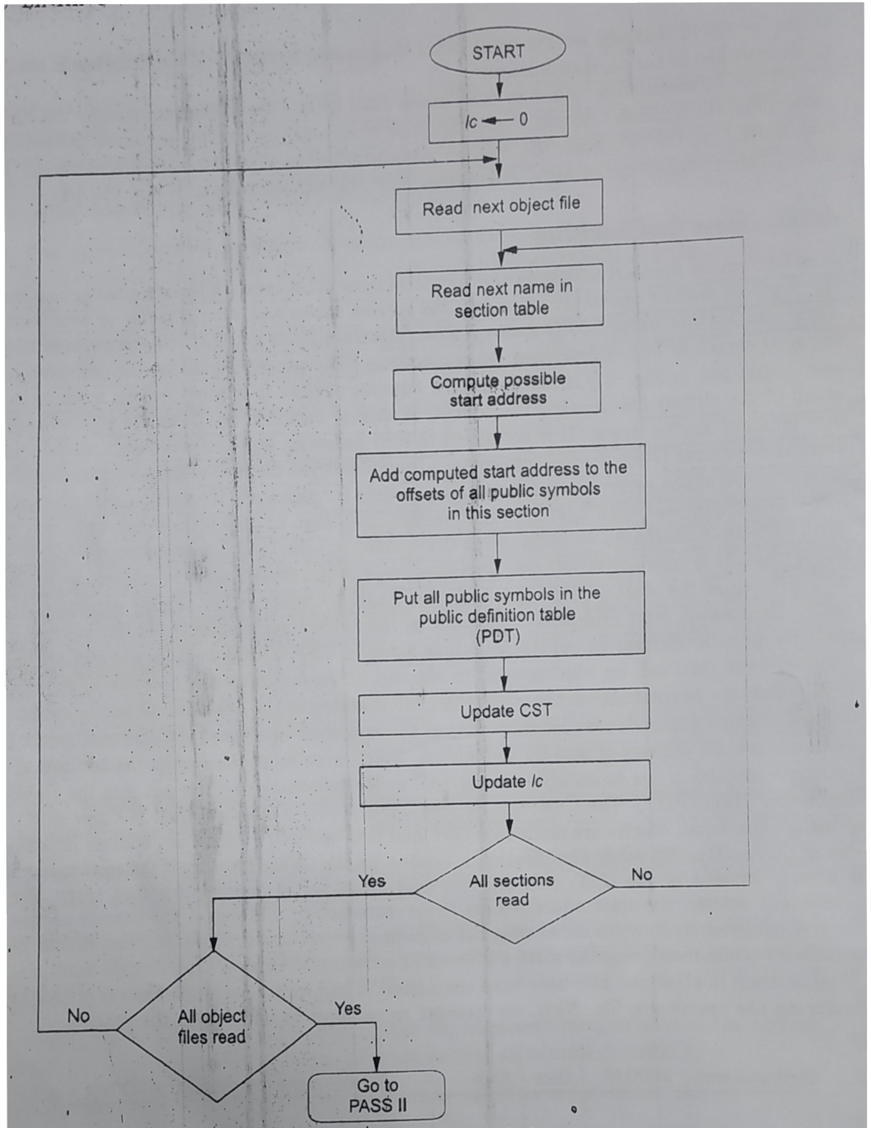


FIGURE 4.3 Pass I of the linker.

- 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.

- 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.

Pass I of Linking (Example)

- Lets assume 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) for all object files are shown in the next slide.

- The Combined Section Table(CST) and Public Definition Table(PDT) are shown in the next to next slide.

Pass I of Linking (Example)

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
*L*1	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
*L*2	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
*L*3	10	1	true

(b)

Pass I of Linking (Example)

Section name	Start address	Size	Align
.text	0	~~145~~ 222	16
.data	~~148~~ 224	~~47~~ ~~226~~ 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.

Pass II of Linking

- It 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.

- The important tasks of this phase is:

  1. *Copy the object files into their corresponding locations.*

  2. *Offset correction or Relocation*

Pass II of Linking - Relocation

- The assembler, while generating object code module, assumes the start addresses of individual sections to be zero.

  → *All references to variables are translated to their distances from the beginning of the section containing them.*

- However, while linking, 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, the 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.*

Pass II of Linking – Relocation (2)

- Relocation, needs to be carried out both at link-time and load-time.

- At link-time, relocation is needed <u>to arrange the object files into the executable module starting at offset zero</u>.

- At load-time, relocation is needed <u>for arranging shared libraries and the executable module into address space.</u>

- Link-time relocation is performed using direct editing that will modify the address sensitive locations within the code during concatenating object modules. Locations requiring relocation can be specified in two different ways:

  1. *Relocation bitmap*
  2. *Relocation table*

Pass II of Linking – Relocation (3)

1. *Relocation bitmap*

   → *For every instruction, the assembler associates a relocation bit.*

   → *If the instruction does not need any relocation (that is, instruction not involving memory address), the bit is set to zero, else the bit is set to one.*

   → *For cases with relocation **bit=1**, Pass II of linking does the necessary modification to the code.*

2. *Relocation table*

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

Table 4.4: Program with relocation bits

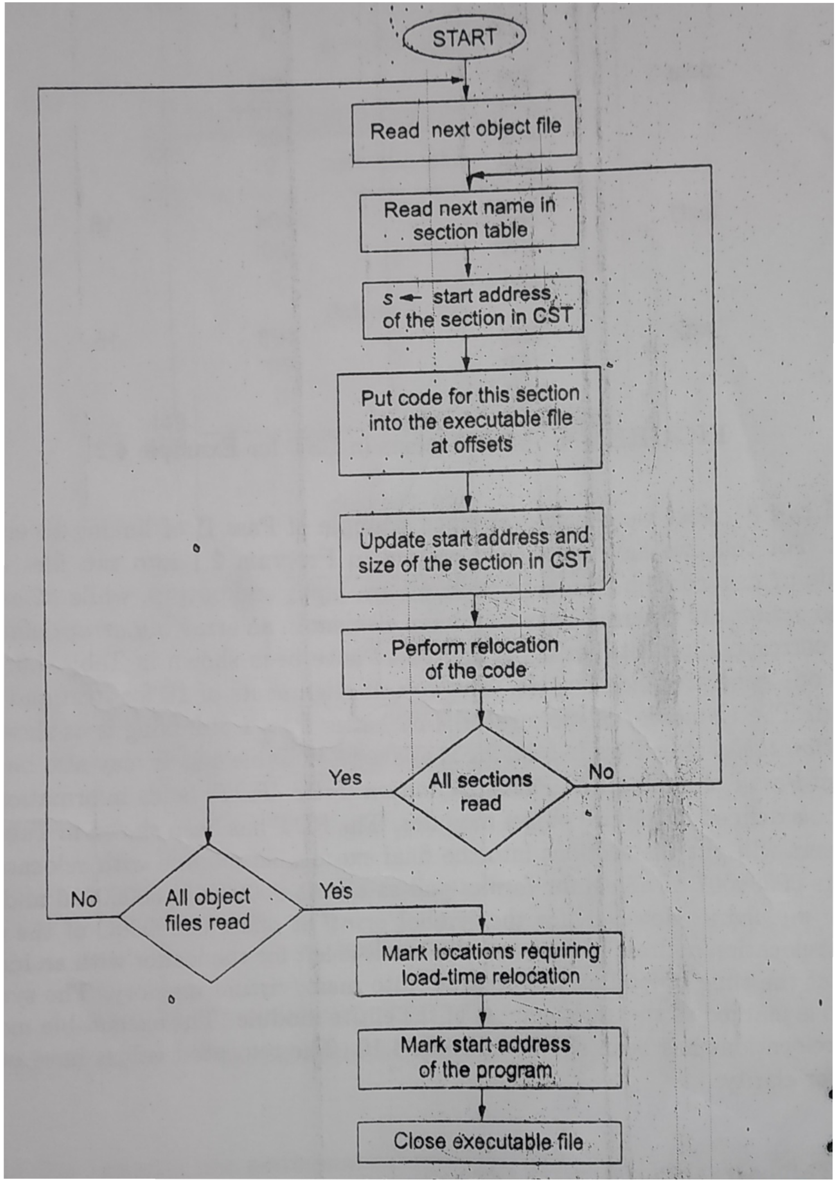
Line no.	Code	Source line	Relocation bit
1		global main	
2		extern printf	
3			
4		section .data	
5		my_array:	
6	00000000 0A000000140000001E-	dd 10, 20, 30, 100, 200, 56, 45,	0
7	00000009 00000064000000C800-	67, 89, 77	0
8	00000012 0000380000002D0000-		0
9	0000001B 004300000059000000-		0
10	00000024 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

Table 4.5: Relocation table

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

Algorithm for *Pass II* of Linking (Flowchart)

- *Please see the flowchart:* *Pass II* of Linking

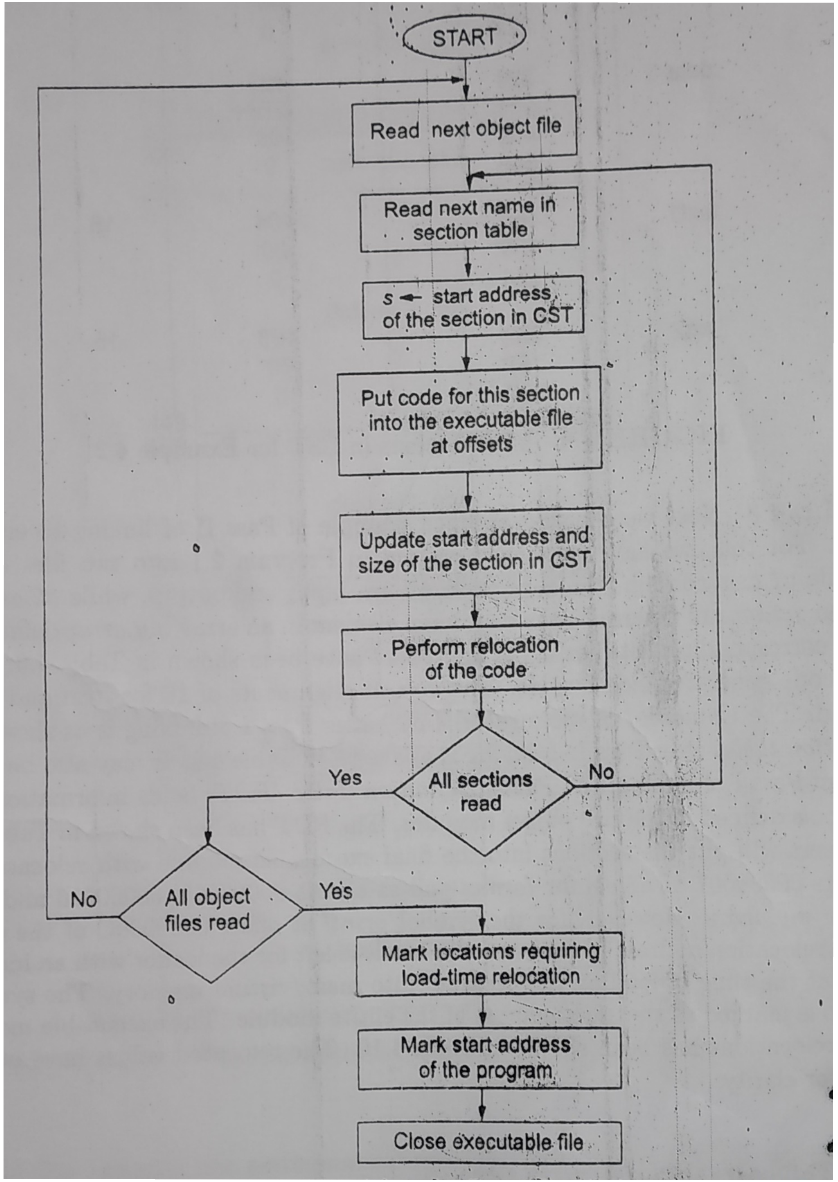


- In ***pass II***, the object files are read again per section and <u>the corresponding codes with necessary relocation changes are put into the executable file.</u>

- 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.

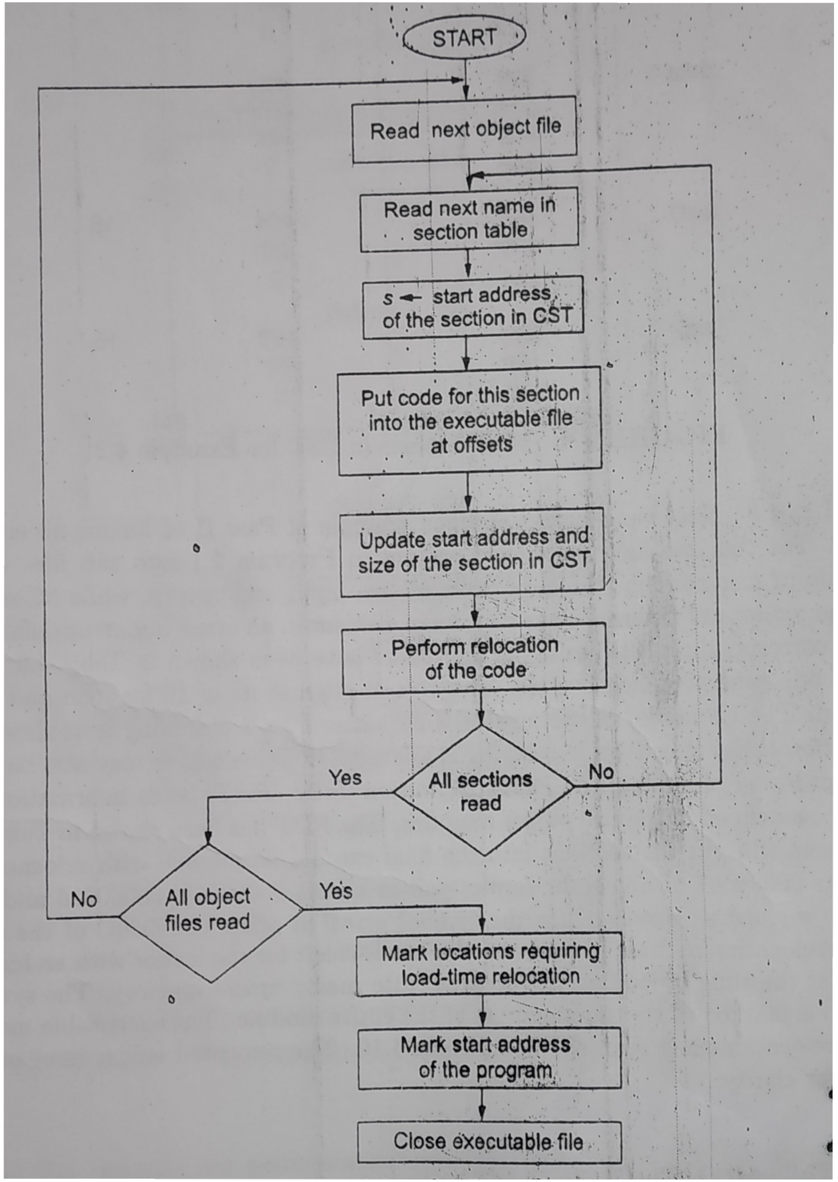
- The CST entries for the section are updated by incrementing the start address of the section by the size of the current section read, and the size field in CST is decremented by the size of the section.



- 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.* (ensures all offsets are with respect to the start address of the executable module)
  - → Load time relocations is done for shared libraray functions and/or variables.(The linker will produce a list of such locations)



- Another important information passed by the linker to the loader is the start address of the program.

→ *For example, the language C expects a unique function* ***main****, to be present in the executable module that defines the start address.*

→ *In our example, we have use global symbol* ***main*** *which is used by* ***gcc*** *linker to pass on the start address information to the loader.*

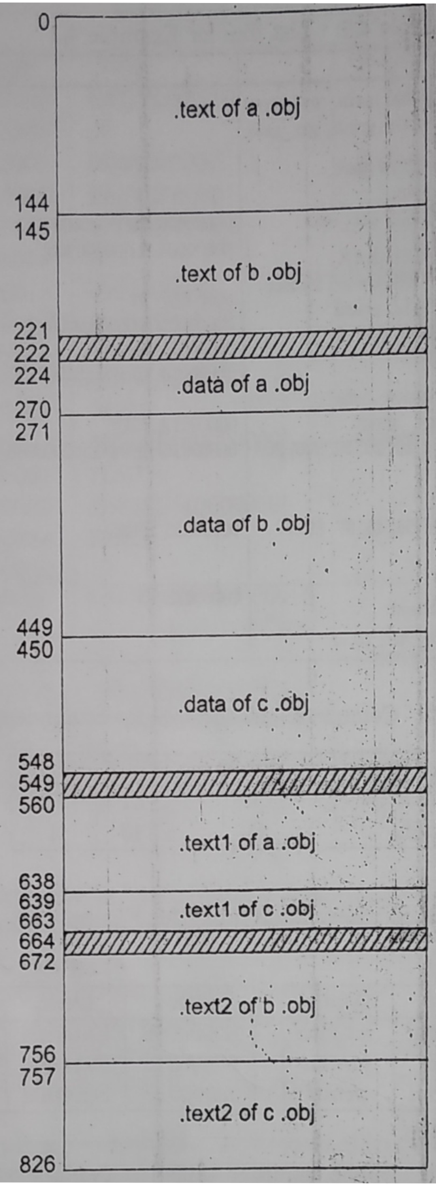
Pass II of Linking (Example)

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

Modification in CST

Pass II of Linking (Layout of Executable)

- *Please see the layout of Executable:* (*Pass II* of Linking example)



0
.text of a .obj
144
145
.text of b .obj
221
222
224
.data of a .obj
270
271
.data of b .obj
449
450
.data of c .obj
548
549
560
.text1 of a .obj
638
639
.text1 of c .obj
663
664
672
.text2 of b .obj
756
757
.text2 of c .obj
826

Library Linking

- Object modules contains symbols defined by the user and external symbols defined in library modules

  → *For example mathematical routines are grouped together into a library, string manipulation functions may be in another one.*

  → *This facilitates the user to refer to a particular library and include it along with other object modules for complete execution*

- External libraries are usually provided in two forms:

  1. *Static libraries*
  2. *Shared libraries*

- In Linux, static libraries have extension **“.s”** whereas for shared libraries, the extension is **“.so”**.

Library Linking (2)

- Static Library
  - → *For each of the external functions used by the program, the corresponding machine code is copied from the object file containing the library.*
  - → *The extracted code is attached with other modules to create the overall executable module.*
  - → *However, in some linkers, instead of including the particular function, the entire library object module is included in the code.*

- Shared Library
  - → *It can be the sharing of code located on disk by unrelated programs.*
  - → *It can also be sharing of code in memory. (the programs execute the same piece of code loaded at the same physical page of the memory, but mapped to the address spaces of the programs)*
  - → *Main memory sharing can be accomplished by using Position Independent Code(PIC) as in UNIX.*

Library Linking (3)

- *By using various techniques(such as pre-mapping the address space) reserving slots for the shared library modules.(DLL in windows)*

- *In most modern OS's shared libraries can be of the same format as the regular executables.*

- *Two main advantages:*
  - *(1) it requires making only one loader for both of them.*
  - *(2) It allows the executables also to be used as DLLs, if they have 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.*

Position Independent Code (PIC)

- 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.

- An important property for building shared libraries.

- PIC is achieved via the 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 addresses cannot be used.*

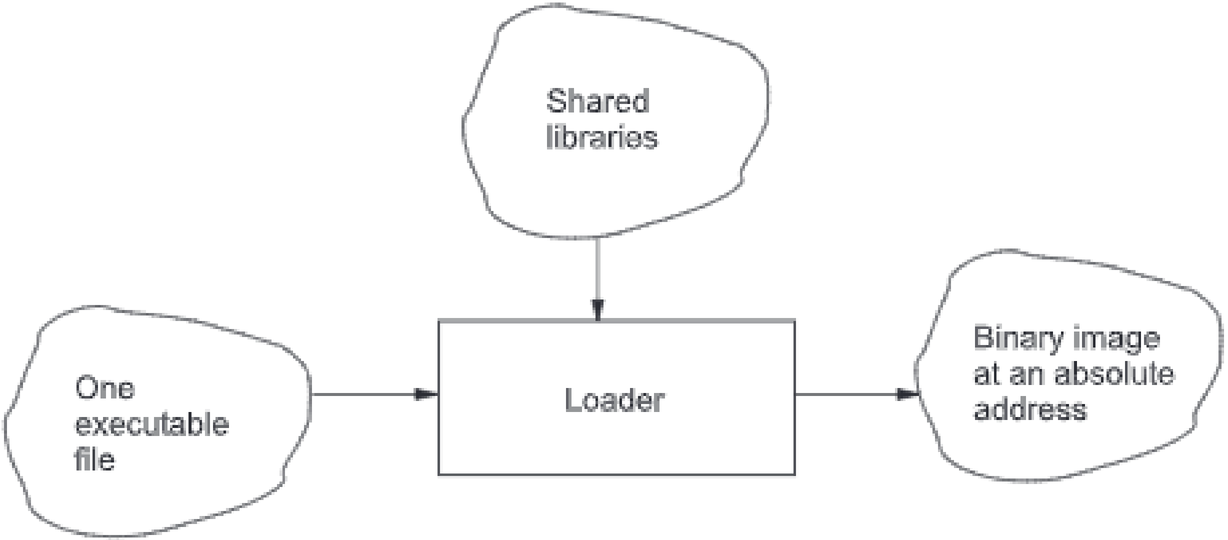
Position Independent Code (PIC)

- Advantages:
  - → *No need to relocate.*
  - → *Library is shared on disk.*
  - → *Library is shared on primary memory as well. All the page tables of various processes can share the main memory frames for the library.*
- Disadvantages:
  - → *One register is used to hold pointer to the indirect table.*
  - → *Each method invocation needs two memory accesses.*
  - → *Another register is used to access the variables in the shared library.*
  - → *Each access to variable in the shared library requires tow memory accesses.*

Loader

- Loading is <u>the process of making a program ready for execution</u> by copying the file from secondary storage to primary or virtual memory.
  - → *It is often a part of operating system, and thus not visible to the system user directly.*
- The major objectives of a loader are:
  1. *Bring a binary image into memory.*
  2. *Bind relocatable addresses to absolute addresses.*
- Linker gives 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 loader is now:
  (1) *to locate the position of the shared library*,
  (2) *correct the appropriate entries in the executable referring to shared library routines and variables*,
  (3) *create the binary image that is ready for execution.*

Input-Output of the Loader



Binary Image

- The binary image of a program consists of the following components:

1. *A header:*
   - → *Indicates the type of the image (an exe. file, some library etc.).*
   - → *Loaded at some preassigned address, or it may be determined by consulting memory management routine.*

2. *Text of the program*
   - → *It consists of the actual piece of code (may be in some specific formats).*
   - → *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 loaders needs to resolve the addresses accordingly.*

Types of Loaders

- There are three categories of loaders, namely:

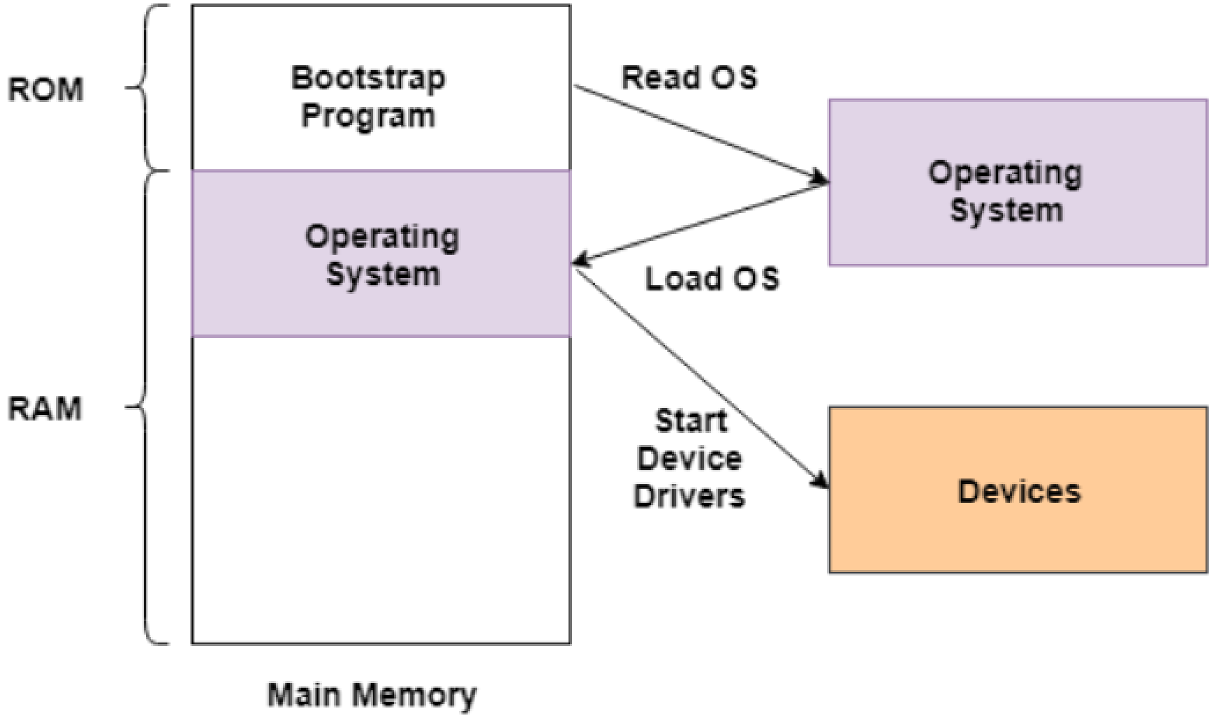
1. *Absolute loader*
   - → *The assembler generates code and writes instructions in a file together with their load address.*
   - → *The loader reads the file and places the code at the absolute address given in the file.*
2. *Relocating loaders*
   - → *The assembler generates code and the relocation information.*
   - → *The loader, while loading the program performs relocation as well.*
3. *Linking Loaders*
   - → *This type of loaders will do the linking with shared libraries as well.*

Types of Loaders (2)

- Bootstrap loader
  - → *It 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, i.e., 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 memory and transfers control to it.*
  - → *When it is executed, it loads the program which is either the OS itself or other system programs to be run without the OS.*

Types of Loaders (3)

- Bootstrap loader



References

R Reference for this topic

- **Book:** Systems Programming, Santanu Chattopadhyaya, PHI, 2011.

- **PPT:** Hsung-Pin Chang, Department of Computer Science, National Chung Hsing University, **Chapter 3: Loaders and Linkers**(for additional reading)