LOADERS AND LINKERS

• 3 processes a system program performs →

1. Loading –
   bringing the object program into memory for execution

2. Relocation –
   modify the object program so that it can be located at a different location from the original one

3. Linking –
   combing 2 or more seperate object ~~and~~ programs and supplying information needed to allow reference b/w them.

Loaders – system program that perform loading function.
– Can also support linking & relocating.

Linker – seperate system program for linking operation.

LOADERS 3.1

→ Basic Loader Function or fundamental

- The most basic loader function is → bringing object program into memory and starting execution.

- Absolute Loader

  It is the most basic loader that just performs loading function.
  It performs all its functions in a single pass.
  - It checks Header record to verify that correct program is being loaded and that it ~~is~~ will fit in the memory space available
  - It reads each Text record and the object code is moved to its corresponding memory location
  - The End record ~~for~~ indicates end of object-code and gives address of location from where execution starts.

ALGORITHM :: Absolute loader

```
begin
    read Header record
    verify program name and length
    read first Text record
    while record type ≠ 'E' do
        begin
            {if object code is in character form, convert it
             internal representation}
            move object code to specified location in memory
            read next object program record.
        end
    jump to address specified in End record
end.
```

Note →

In the object program, each byte of assembled code is given using hexadecimal representation in character form.

eg- OPcode for STL → 14

It is represented using pair of characters '1' & '4'

So, when loader reads this, they occupy 2 bytes of memory. But, in the instruction loaded for execution it is to be stored as 1 byte represented by hexadecimal 14.

⇒ Each pair of byte from object prog record must be packed together into 1 byte during loading.

This method of representation is insufficient

So, object program can be stored in binary form → Each byte of object code stored in 1 byte of memory. but they aren't easy to read for humans.

- Simple Bootstrap Loader.
  - Special absolute loader that is loaded executed when computer is first started or restarted.
  - It loads the 1st program to be run on the computer, ie OS.

```
Line
0   BOOT    START   0       BOOTSTRAP LOADER FOR SIC/XE
1   .
2   . THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND
3   . ENTERS IT INTO MEMORY LOCATION STARTING FROM
4   . ADDRESS 80h. AFTER LOADING IS COMPLETE ~~CONTROL~~ JUMP
5   . TO 80h IS EXECUTED TO BEGIN EXECUTION OF PROGRAM.
6   . REGISTER X CONTAINS NEXT ADDRESS TO BE LOADED
7           CLEAR   A       CLEAR REG A TO 0
8           LDX     #128    INITIALIZE REG X TO 80h
9   LOOP    JSUB    GETC    READ HEX DIGIT FROM PROG
10          RMO     A,S     SAVE IN REG S.
11          SHIFTL  S,4     MOVE TO HIGH-ORDER 4 BITS
12          JSUB    GETC    GET NEXT HEX DIGIT
13          ADDR    S,A     COMBINE DIGITS TO 1 BYTE
14          STCH    0,X     STORE AT ADDR. IN X.
15          TIXR    X,X     ADD 1 TO MEMORY ADDRESS
16          J       LOOP    LOOP TILL EOF REACHED.
17  .
18  . SUBROUTINE TO READ FROM DEVICE AND CONVERT IT
19  . FROM ASCII TO HEXA DIGIT VALUE AND RETURN IT
20  . TO REG A. IF EOF ENCOUNTERED, CONTROL TRANSFERRED
21  . TO 80h
22  .
23  GETC    TD      INPUT   TEST INPUT DEVICE
24          JEQ     GETC    LOOP UNTIL READY
25          RD      INPUT   READ CHARACTER
26          COMP    #4      IF CHAR IS 04h (EOF)
27          JEQ     80      JUMP TO START OF PROG LOADED
28          COMP    #48     COMP TO 30h ('0')
29          JLT     GETC    SKIP CHAR < '0'
30          SUB     #48     SUBTRACT 30h FROM ASCII
31          COMP    #10     FOR 'A' TO 'F', RESULT < 10 THEN
32          JLT     RETURN. CONVERSION COMPLETE, ELSE.
            SUB     #7      SUBTRACT 7 MORE.
33  RETURN  RSUB            RETURN TO CALLER
34  INPUT   BYTE    X'F1'   INPUT DEVICE
35  .       END     LOOP
```

- The bootstrap begins at address 0 [Line 0]
- It loads the OS starting at address 80h by initializing register X (the pointer) to 80h [LINE 5]
- As this is the 1st prog to be loaded, its loading is simple.

The object program from device F1 is
- represented as 2 hexadecimal digit for 1 byte
- has no Header or End record or any other control information.

Here, the object code is loaded into consecutive bytes of memory starting at 80h.

- Subroutine GETC →
It reads 1 char. from device F1 and converts it from ASCII to the hexadecimal digit it represented.
When it encounters EOF, the control moves to 80h (i.e start of loaded program)

So in the program,
the main loop keeps track of the next memory location for loading and reads the 2 characters & stores it as 1 byte.
The subroutine reads the character and converts it from ASCII to represented hex value.

3.2

→ Disadvantage of absolute loader

- The program ~~has~~ needs absolute memory location for loading to be specified by the programmer.
But in large & advanced machine, multiple independent program run together shay memory.
Here predefining memory for loading is impossible.

- The subroutines of libraries aren't used efficiently.
For efficient use only required subroutines should be loaded but this isn't possible with absolute addresses

MACHINE DEPENDENT LOADER FEATURES

→ In most modern computer, the loaders also perform the relocation and linking function, in addition to the basic loading function.

- Relocation
  - Loaders that allow relocation are called relocating loader or relative loader.
  - Methods for specifying relocation as part of object program

(i). Modification record is used to describe each part of object code that must be changed when program relocates
And the instructions whose value is affected by relocation are ones that use extended format.

The modification record specify the start address & length of fields to be altered. It then describes the modification to be performed

But, this method isn't suited for all machines
eg- In a SIC machine, there is no relative addressing & so almost all instructions need to be modified duy relocation. This leads to a lot of Modification record that dramatically increases object code size.

(ii) There is a relocation bit associated with ~~Text Record~~ each word of object code in Text Record.
It is used in machines that primarily use direct addressing & fixed instruction format.
In SIC machine, each instruction occupies 1 word, i.e one relocation bit per instruction
The relocation bits gathered togetha to a bit mask which is present in the Text Record following the ~~record~~ length indicator.

eg- T^001057,0A^800^100036^4C0000^F1^001000

If relocation bit correspondy to a word is
- 1 → modification required
prog's staty addr is to be added to this wood duny relocation
- 0 → no modification required.

If Text record has fewa than 12 words, then ~~correspondy~~ for the unused words correspondy wood relocation bit = 0.

eg- FFC (1111 1111 1100)
First 10 words need to be modified.

~~iii) Some~~

(iii) Some computa have hardware relocation capability that eliminates need of loada to relocate program.

The SIC/XE machine usually use the Modification record scheme for relocation.

ALGORITHM: SIC/XE relocation loader

```
begin.
    get PROGADDR from operaty system
    while not end of input do
        begin
            read next record
            while record type ≠ 'E' do
                begin
                    read next input record
                    while record type = 'T' then
                        begin.
                            more object code from a record to location ADDR
                            + specified address
                        end
                    while record type = 'M'
                        add PROGADDR at location
                        PROGADDR + specified address
                end.
        end
end.
```

The SIC machines usually use the modification bit scheme.

ALGORITHM: SIC relocation loader algorithm.

```
begin
    get PROGADDR from operating system
    while not end of input do
        begin
            read next record
            while record type ≠ 'E' do
            while record type = 'T'
                begin
                    get length = second data.
                    mask bits(M) as third data.

                    for (i=0, i<length, i++)
                        if M_i = 1 then
                            add PROGADDR at the
                            location PROGADDR + specified
                            address.
                        else
                            move object code from record
                            to location PROGADDR +
                            specified address
                    read next record.
                end
        end
end.
```

- Program Linking

  - programs made up of multiple control sections can be assembled in 2 ways
    - all control sections together
      i.e in same invocation of assembly
    - each independently.

  In both case they will appear as seperate segments of object code after assembly

  Assembler sees code only as control sections that are to be loaded, relocated & linked. It doesn't need to know which control sections

were assembled at same time

Consider 3 programs each control section :

Loc		Source statement		Object code
0000	PROGA	START	0	
		EXTDEF	LISTA, ENDA	
		EXTREF	LISTB, ENDB, LISTC, ENDC	
		.		
		.		
		.		
0020	REF1	LDA	LISTA	03201D
0023	REF2	+LDT	LISTB+4	77100004
0027	REF3	LDX	#ENDA-LISTA	050014
		.		
		.		
		.		
0040	LISTA	EQU	*	
		.		
		.		
0054	ENDA	EQU	*	
0054	REF4	WORD	ENDA-LISTA+LISTC	000014
0057	REF5	WORD	ENDC-LISTC-10	FFFFF6
005A	REF6	WORD	ENDC-LISTC+LISTA-1	00003F
005D	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000014
0060	REF8	WORD	LISTB-LISTA	FFFFC0
		END	REF1	

Loc		Source statement		Object code
0000	PROGB	START	0	
		EXTDEF	LISTB, ENDB	
		EXTREF	LISTA, ENDA, LISTC, ENDC	
		.		
		.		
		.		
0010	REF1	+LDA	LISTA	03100000
0014	REF2	LDT	LISTB+4	772027
0017	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0060	LISTB	EQU	*	
		.		
		.		
0070	ENDB	EQU	*	
0070	REF4	WORD	ENDA-LISTA+LISTC	000000
0073	REF5	WORD	ENDC-LISTC-10	FFFFF6
0076	REF6	WORD	ENDC-LISTC+LISTA-1	FFFFFF
0079	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	FFFFF0
007C	REF8	WORD	LISTB-LISTA	000050
		END		

Loc		Source statement		Object code
0000	PROGC	START	0	
		EXTDEF	LISTC, ENDC	
		EXTREF	LISTA, ENDA, LISTB, ENDB	
		.		
		.		
		.		
0018	REF1	+LDA	LISTA	03100000
001C	REF2	+LDT	LISTB+4	77100004
0020	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0030	LISTC	EQU	*	
		.		
		.		
0042	ENDC	EQU	*	
0042	REF4	WORD	ENDA-LISTA+LISTC	000030
0045	REF5	WORD	ENDC-LISTC-10	000008
0048	REF6	WORD	ENDC-LISTC+LISTA-1	000011
004B	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000000
004E	REF8	WORD	LISTB-LISTA	000000
		END		

- LISTA, LISTB, LISTC ⟶ list of items of each prog.
ENDA, ENDB, ENDC ⟶ marks end of lists
Reference to external symbol in
REF1 to REF3 ⟶ as instruction operands
REF4 to REF8 ⟶ values of data word.

REF1
In PROGA, REF1 is a reference to label within the program so, no modification for relocation or linking needed
In PROGB & PROGC → REF1 is reference to an external symbol so, assembler uses extended format instruction with address field → 0000 & Modification record required to tell loader that add value of LISTA is to be added after linking

REF2
Similar to REF1 but here PROGB has local reference & PROGA & PROG C have external symbol.

REF3
It is an immediate operand whose value is ENA-LISTA
In PROGA it can be directly computed but in the other 2 prog, the value is unknown
The expression is assembled as external reference & final result is an absolute value independent of location of where program is loaded.

General approach →
assembler evaluate as much of the expression as it can & remain is passed on to loader via Modification record.

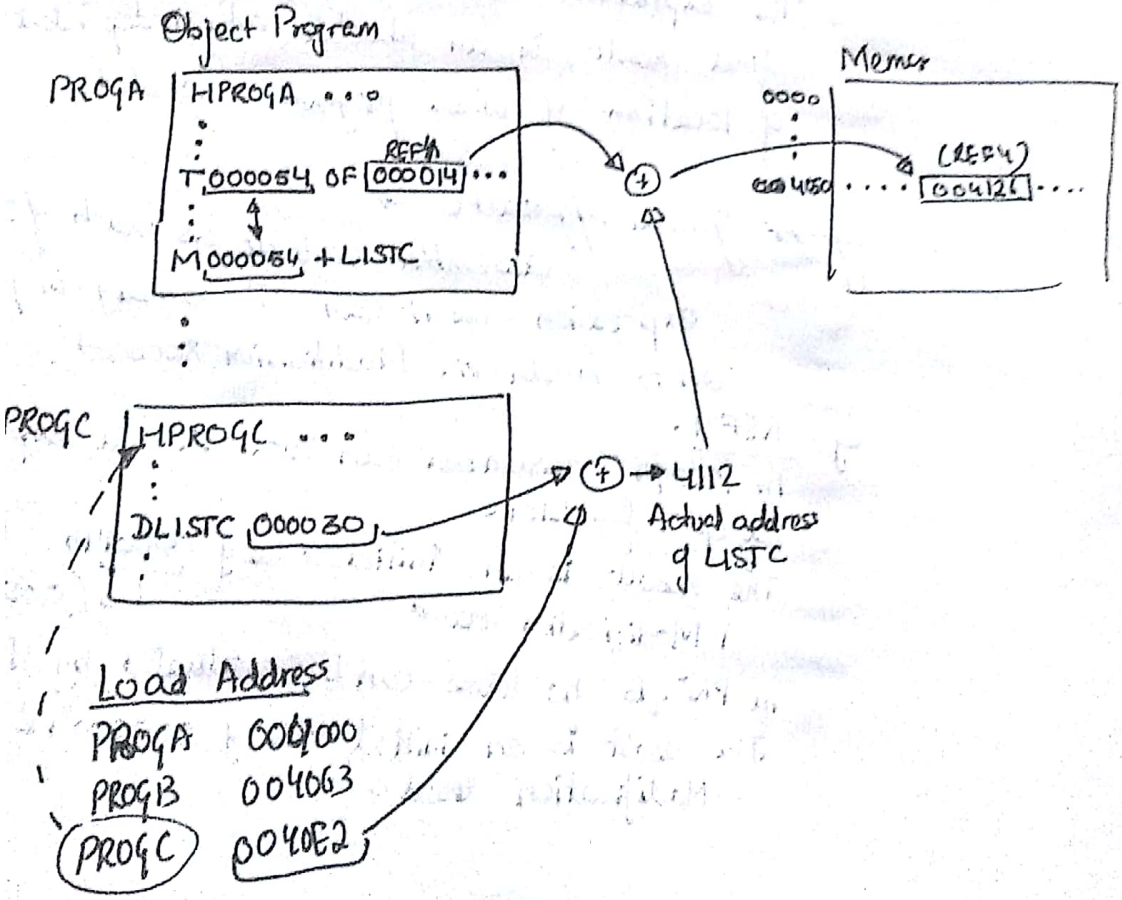
eg- REF4.
In PROGA assembler can evaluate all expression except for LISTC
The result is an initial value of 000014h and
↳ (0054-0040)
1 Modification record
In PROGB no terms can be evaluated by the assembler
The result is an initial value of 000000h & 3
Modification record -

In PROGC assembler can supply value of LISTC but rest is unknown.
Initial value is relative address of LISTC and Modification record tells to add value of ENDA & subtract value of LISTA.

— Consider the 3 progs have been loaded into memory with PROGA starts at address 4000, with PROGB & PROGC immediately following

* REF4 to REF8 will end up with same value in each of the 3 program after relocation and linking.

eg - value of reference REF4 in PROGA located at 4054 (4000 + relative address of REF4(0054))
Initial value of REF4 → 000014h (from the Text record)
To this we add address assigned to LISTC (4112) [beginning of PROGC + 30]
⇒ value in memory 4054
→ 000014 + 004112
= 004126.



In PROGB for REF4

located at relative address 70
so, memory location (4063+70 → 40D3)
initial value → 000000
+ ENDA → 4064 (4000+54)
+ LISTC → 4112 (40E2+30)
+ LISTA → 4040 (4000+40)
= 004126

→ same as in PROGA

Similarly for PROGC, REF4 also results in 004126

* REF1 - REF3 → which are reference that are instruction operand, calculated values after loading aren't always equal as additional address calculation step involved in case of base or PC relative instructions

eg - REF1 →

In PROGA → target address 4040.
displacement 01D + PC (4023)

In PROGA → REF1 is extended format instruction with direct address which is 4040
(LISTA location → 4000+40 = 4040)

→ Algorithm & Data Structure for Linking Loader.

- Algorithm for linking & relocating loader that uses Modification record for relocation so that linking & relocation function are performed using same mechanism.
- i/p to loader is set of object programs that are to be linked together
Programs may contain external reference to symbol whose definition come later & so linking operation can't be performed till the external symbol is assigned an address.
∴ Linking loaders makes 2 passes over its i/p.
Pass1 → assigns address to all external symbols
Pass2 → performs actual loading, relocation & linking.

Data structure needed,

- ESTAB → external symbol table
  → it stores name & address of each external symbol in the set of control sections (programs) that are loaded
  → Hashed organization is used for this table.

Important variable needed,

- PROGADDR → program load address
- CSADDR → control section address

PROGADDR is the beginning address where linking program is to be loaded. Its value is supplied by the OS

CSADDR – starting address of control section currently being scanned by loader

Pass1

→ loader only concerned with Header & Define record types.

→ Value for PROGADDR is obtained from OS, which is the CSADDR for the 1st control section.

→ Control section name is obtained from Header record and is entered in ESTAB with its corresponding value given by CSADDR.

→ All external symbols that appear in Define record also entered in ESTAB. Their address is relative address + CSADDR.

→ When end record reached,
control section length (CSLTH) added to CSADDR → this is CSADDR for next section.

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR (for first control section)
while not end of input do
    begin
        read next input record (Header record for control section)
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag (duplicate external symbol)
        else
            enter control section name into ESTAB with value CSADDR
        while record type ≠ 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag (duplicate external symbol)
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end (for)
            end (while ≠ 'E')
        add CSLTH to CSADDR (starting address for next control section)
    end (while not EOF)
end (Pass 1)
```

Pass 2.

- Here actual loading, relocation & linking is done
- As Each Text Record is read, object code is moved to its specified address which is, relative address + CSADDR.
- When Modification Record is encountered, symbol required for modification is looked up in ESTAB & its value is added or subtracted from intended location.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record (Header record)
        set CSLTH to control section length
        while record type ≠ 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        (if object code is in character form, convert
                            into internal representation)
                        move object code from record to location
                            (CSADDR + specified address)
                    end (if 'T')
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end (if 'M')
            end (while ≠ 'E')
        if an address is specified (in End record) then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end (while not EOF)
jump to location given by EXECADDR (to start execution of loaded program)
end (Pass 2)
```

Last step performed by loader,
transfer of control to loaded program to Begin execution. The End record for each control section may contain address of 1st instruction in that control section to be executed.
If more than 1 control section specifies transfer address, loader uses the last one encountered
If no control section specifies transfer address, loader uses beginning of linked program (i.e PROGADDR).

→ Algorithm can be made more efficient if we use reference no for external symbol in Modification record, instead of the symbol name
Then we will need to add a Refer record that specifies the symbol & its reference no
eg- R^02 LISTB 03 ENDB 04 LISTC 05 ENDC.
→ Reference record in PROGA.
So the modification record will be of the form,
M^000024^05^+02

Advantage of this method →
- avoids multiple searches of ESTAB for same symbol while loading of control section
Now, only 1 lookup in ESTAB required for each external reference symbol.

MACHINE INDEPENDENT LOADER FUNCTION

- Automatic Library Search. →

– Many linking loaders can automatically incorporate subroutines from program libraries into the program being loaded.
– Some std. libraries are used in such a way, other libraries may be specified by control statement or by parameters to loaders.

- Subroutines called by program being loaded are automatically fetched from the library and linked to the program while loading.

This is known as.

→ Automatic library call (or) library search.

How is it done?

The linking loader that supports this must be able to keep track of external symbols used that aren't part of the input.

To do this the loader enters all external symbol it encounters into the ESTAB, ~~when it~~ if the symbol isn't already not present. When it encounters the external symbol's definition it complete its entry (if present) by filling in its address.

If at the end of Pass1, some unresolved symbols present in ESTAB then loader searches for them in the libraries.

It is possible that subroutines fetched from libraries may also contain external symbols so library search needs to be repeated till all external references have been resolved.

This process allows programmers to override the standard library's subroutines by providing our own subroutine as input to loader so when loader goes to search library for unresolved symbol reference, the overrided subroutine reference is already defined & resolved.

How libraries are searched?

The libraries themselves ~~are~~ have assembled or compiled version of subroutines. It is possible to search them using their Define records, but it is inefficient.

Special structure called directory used to search libraries. It contains name of each routine & a pointer to its address within the file.
If a subroutine referred to by multiple names, there is an entry for each name and all point to same location.

- This same technique applies to resolution of external reference to data items.

- Loader Options
  - Loaders allow options that modify standard processing of the loader. Many loaders have a special command language that is used to specify options. Sometimes there is a separate i/p file that contains such control statement, Sometimes the statements are embedded in the primary input stream b/w object programs & can be included in the source program.

  - On some systems options are specified as part of job control language that is processed by the OS. Here, OS incorporates the options specified into a control block that is made available to loader when its invoked

  - Some options -
    - to select alternative sources of i/p
      eg - INCLUDE program-name (lib-name)
      This directs loader to read object program from a library & treat it as primary loader i/p's part.

- to allow users to delete external symbols or entire sections
  DELETE csect-name
  deletes control section(s) from set of progs being loaded
- to change external reference within prog being loaded & linked
  CHANGE name1, name2
  name1 is changed to name2 wherever it appears in the object prog.

eg- Consider a main program say COPY that has 2 subprograms - RDREC : to read records
WRREC : to write records
Each has its own control section

Suppose utility subroutine available such that it contains subroutines READ & WRITE and it is more favorable for COPY to use them

As a temp measure, first we use some loader commands to make these changes without reassembling the program, to test the new routines

```
INCLUDE   READ(UTLIB)   } tells loader to include control section READ
INCLUDE   WRITE(UTLIB)  } & WRITE from UTLIB library
DELETE    RDREC, WRREC  → tells not to load RDREC & WREC
CHANGE    RDREC, READ   } → changes all external reference to
CHANGE    WRREC, WRITE  } RDREC to refer to READ & reference to WRREC to refer to WRITE.
```

- LIBRARY MYLIB
  → it automatically includes library routines to satisfy external reference
- NOCALL SYMBOLS
  → tells loader that these external references are to remain unresolved.

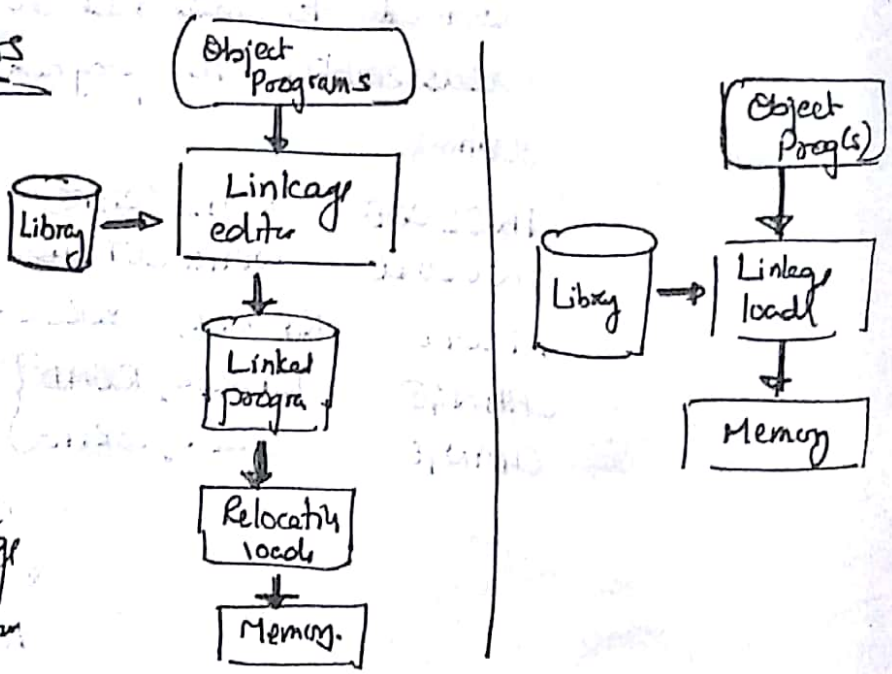
- option to specify that no external reference is to be resolved
  Usefull when programs are to be linked but not immediately executed
- option to specify where execution should begin
- option to control whether or not loader should execute program if error is detected during load

LOADER DESIGN OPTIONS

Organisation of loading function

- linking & relocation take place at load time (used by linking loader)
- linkage editors — linking is performed prior to load time
- dynamic linking — linking is performed at execution time

→ Linkage Editors



In linkage editor, the source program is first assembled or compiled

• Linkage Editor vs Linking Loader.

→ Linking Loader performs all linking & relocation function & loads linked program directly into memory for execution

– Linkage editor produces a linked version of program called load module or executable image, which is written into a file or library for later execution.

→ Linkage editor is useful for programs that need to be executed multiple times without reassembling everytime.
For execution, relocation loader loads program into memory. Only the object code modification required is getting the actual address for loading, rest is done during linking. So, now loading can be done in 1 Pass.

→ Linking Loader is better when program needs to be reassembled nearly for every execution.

• The linked program produced by linkage editor is in a form that is suitable for processing by relocating loader.
- All external references are resolved
- relocation is indicated by some mechanism like Modification record or bit mask.

Information about external references are often retained in the linked program as it allows subsequent relinking of program to replace control sections, modify external references, etc.

• If actual address for loading is known, then linkage editor can perform the relocation, i.e result is linked program that is exact image of way program will appear in memory.
But,
flexibility of loading program at any location is preferred over the reduction of overhead for performing relocation at run time

• Other useful functions →
→ modification of a linked program without having to process the entire program.
eg – Consider a program PLANNER that has multiple

subroutines. One of its subroine PROJECT had to be changed due to error a to improve efficiency. After new version of PROJECT is assembled or compiled, linkage editor can replace this subroutine in the linked version of PLANNER. using some linkage editor commands.

```
INCLUDE  PLANNER (PROGLIB)
DELETE   PROJECT
INCLUDE  PROJECT (NEWLIB)
REPLACE  PLANNER (PROGLIB)
```

→ linkage editor can be used to build packages of subroutines or other control sections that are generally used together.

This is useful while dealy with subroutine libraries that support high level programing lang.

eg- In a typical implementation of FORTAN, there are large number of subroutines that are used to handle formatted input & output. There are large no. of cross-reference b/w these subprograms because they are closely related

But, it is desirable to keep them as seperate modules for program modularity & maintability.

But, same set of cross-reference will be processed for almost every FORTAN program linked. This represents a substantial overhead.

We can use the linkage editor to combine the subroutines into a package using commands like,

```
INCLUDE  READR (FTNLIB)
INCLUDE  WRITER (FTNLIB)
INCLUDE  ENCODE (FTNLIB)
   .
   .
SAVE     FTN10 (SUBLIB)
```

The linked module FTN10 can be indexed in directory of SUBLIB under same names original subroutines. Thus, search of SUBLIB before FTNLIB would retrieve FTN10 instead of seperate routines.

And as FTN10 would already have all cross-reference b/w subroutines resolved, these linkage would not need to be reprocessed when user's program is linked.

→ linkage editor allow user to specify that external references are not to be resolved by automatic library search.

eg- If 100 FORTAN program using I/O routines in FTN10 are to be stored in a library, the library will store 100 copies of FTN10 if all external reference were resolved

This wastes a lot of library space

We can use commands to specify that no library search is to be performed during linkage editing and so they can only be resolved during execution.

This will require slightly more overhead due to 2 linkage operation but it results in large saving of library space.

- Linkage editors are in general more flexible than linking loader & also offer more control.

But they also are more complex and have greater overhead.

→ Dynamic Linking (or dynamic loading or load on call)

- Here the linking is performed during execution time.

i.e a subroutine is loaded & linked to rest of program when it is first called.

- It is used to allow several executing programs to share 1 copy of a subroutine or library.

eg- run-time support routines for a high level lang like C could be stored in a dynamic link library.

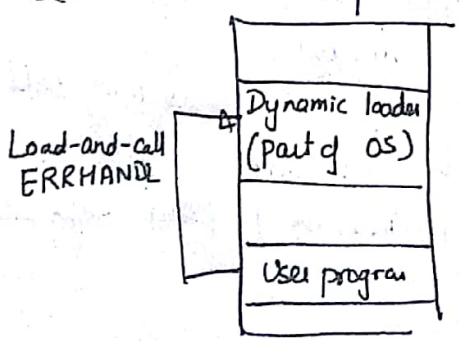
A single copy of the routines could be loaded into memory & all executing C programs could be linked to this copy instead of having seperate copy for each.

- In object-oriented system, dynamic linking is used for references to software objects.

This allows implementation of object & its methods to be determined during run-time.
The implementation can be changed anytime without affecting program that uses the object

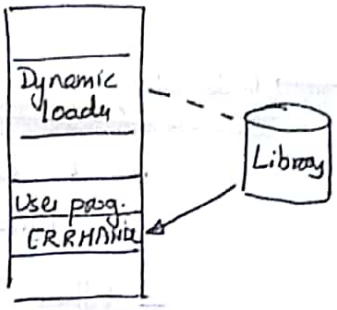
- Advantage of dynamic linking —
  - * it provides ability to load routine only when they are needed
    This results in saving of time & memory space
  - eg – consider program contains subroutines that correct or diagnose error in i/p data during execution. If no error occurs (which can be common) then these subroutines will not be used and so will not be loaded & linked.
  - If program has many subroutines but uses only a few depends on its input, then only the subroutines required can be loaded & linked during execution.

- How to accomplish loading & linking of called subroutine?
  - The routine that must be dynamically loaded must be called via OS service request, ie the request is to the part of the loader that is kept in memory.
  - ~~So a JSUB instruction that refers to an external symbol~~
  - So instead of executing a JSUB instruction that refers to an ~~great~~ external symbol, program makes a load & call request to OS with symbolic name of routine as the parameter.

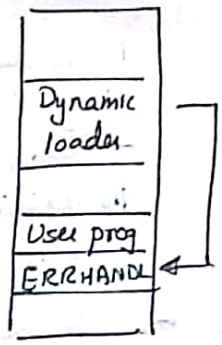


Here, the user program sends a load-and-call request for ERRHANDL subroutine.

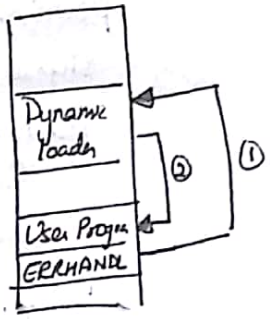
- The OS examines internal table to determine whether or not routine is already loaded
If not, routine is loaded from specified user or system library
[Load]



and then control is passed to the routine being called.
[Call]

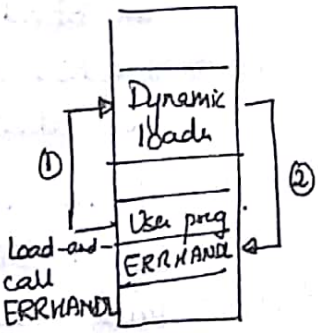


When subroutine completes its processing, it returns to its caller (i.e OS routine that handles load-and-call request).
The OS then returns the control to the user program.



After subroutine is completed, the memory that was allocated for loading may be released & used for other purpose
But, this isn't done immediately as if a 2nd call to it occurs, another load operation won't be required.
So, it is desirable to keep the subroutine till memory isn't required by any.

If subroutine called is still in memory, control is directly passed to it from the dynamic loader.



- In dynamic loading, binding of symbolic name to actual address is delayed from load time until execution time which results in greater flexibility

- But, this also requires more overhead as OS intervenes in the calling process

→ Bootstrap Loaders

In a idle computer with no program in memory, how do thing start?

- When machine is empty and idle there is no need for relocation, only absolute address for program beg 1st loaded is needed. (this program is usually the OS). For this we need an absolute loader loaded.
- Early computers required operator to enter into memory the object code of absolute loader using switches on computer console. But, this is too inconvenient & error-prone.
- In some computers, absolute loader program is permanently present in a ROM. When some hardware signal occurs indicating start up of the system, the machine begins executing this ROM program.
  On some computers, program is executed in the ROM
  on others, program is copied to main memory & executed.

  But, it is inconvenient to change the ROM program if modification necessary.
- Intermediate solution,
  have a built-in hardware function (is small ROM program) that reads fixed length records from some device into memory at fixed location

  After reading operation is complete, control is transferred to address in memory where record is stored. These records contain machine instructions that ~~absolute loader~~ loads the absolute program that follows.

  If the instructions can't be fit in 1 record, then record causes reading of other records & they in turn cause reading of more records.
  → hence the term bootstrap.

  1st record(s) → bootstrap loader.

This loader added to beginning of all object programs that

as to be loaded into empty & idle system.

IMPLEMENTATION EXAMPLE →

→ MS-DOS Linker for Pentium & other x86 system.

- Most MS-DOS compiler & assembler produce object modules, not executable machine language programs.
- These object modules have extension .OBJ and they contain binary image of translated instructions & data of program. It also describes structure of program.
- MS-DOS LINK – linkage editor that combines one or more object modules to produce a complete executable program.

The executable program have extension .EXE. LINK can also combine the translated program with other module from object code libraries.

- A typical MS-DOS object module,

Record Type	Description
THEADR	Translator Header
TYPDEF, PUBDEF, EXTDEF	External symbol & references
LNAMES, SEGDEF, GRPDEF	Segment defination and grouping
LEDATA, LIDATA	Translated instructions & data.
FIXUPP	Relocation & linking information
MODEND	End of object module.

similar to Header & End record of SIC/XE

- THEADR record – specifies name of object module
- MODEND record – marks end of module & contains reference to entry point of program

- PUBDEF record — contains list of external symbol called public names that are defined in the object module.

- EXTDEF record — contains list of external symbols that are refferred to in the object module.

Similiar to Define & Refer record of SIC/XE
Both PUBDEF & EXTDEF contain info abt data type designated by an external name.

- TYPEDEF record — defines the types

- SEGDEF record → describes segment in object module includy their name, length & alignment

GRPDEF record — specify how these segments are combined into groups

LNAMES record — contains list of all segment & class names used in program.

SEGDEF & GRPDEF refer to segment by giving the position of its name in the LNAMES records.

- ~~LE DATA~~
- LEDATA record — contains translated informations & date from source program
It is similiar to Text record of SIC/XE

LIDATA record — specify translated instructions & data that occur in repeating pattern.

- FIXUPP record — used to resolve external refernces & carry out address modifications that are associated with relocation & grouping of segment within the program.
It is similar to Modification record of SIC/XE
But FIXUPP records are more complicated.
A FIXUPP record must immediately follow the LEDATA & LIDATA record to which it applies.

- LINK performs its functions in two Passes.

Pass1 – computes starting address for each segment in the program
It constructs a symbol table that associates an address with each segment (using LNAMES, ~~SEGMENT~~ SEGDEF & GRPDEF records) and each external symbol (using EXTDEF & PUBDEF records).
If unresolved external symbols remain after all object modules are processed, LINK searches the specified libraries.

Pass 2 – LINK extracts translated instruction & data from object modules & build an image of executable program in memory.
This is because, executable program is organised by segment & not by order of object modules.
Building a memory image, most efficient way to handle rearrangements caused by combining & concatenating segment.
If enough memory isn't available, LINK uses temp disk file in addition.
Here LINK process each LEDATA & LIDATA record along with corresponding FIXUPP records & places binary data from LEDATA & LIDATA record into memory image at locations reflecting segment address computed during Pass1.
Relocation & resolving of external reference is done here. A table of segment fixups is maintained that is used to perform relocation that reflects actual segment address when program is executed.

Once memory image is complete LINK writes it to .EXE file, which also contains a header that contains table of segment fixups & information about memory requirement & entry points & also initial contents of CS & SP registers

→ SunOS Linkers for SPARC system.

- SunOS provides 2 different linkers
  - run-time linker
  - link-editor.
- Link-editor is most commonly used in process of compiling a program.
  It takes 1 or more object module produced by assemblers & compilers & combines them to produce a single o/p module.
- Types of output module →
  1. Relocatable object module.
     It is suitable for further link-edit
  2. Static executable
     It has all symbolic references bound & ready to run
  3. Dynamic executable
     It has some symbolic references that are to be bound at run-time
  4. Shared Object
     It provides services that can be bound at run time to 1 or more dynamic executables.
- Object module contain multiple sections which represent instructions & data areas from source program.
  These sections have a set of attributes such as "executable", "writable".
  Object modules also include list of relocation & linking operations that need to be performed & a symbol table that describes the symbols used.
- Sun-OS link-editor reads the object modules that are given to it to process. Sections that have same attributes are concatenated to form new section in o/p file.

- Symbol table from i/p files are processed to match symbol definations & references, and relocation & linking operations are performed within o/p file.
  Linking generates new symbol table & new set of relocation instruction in output file. They represent symbols that need to be bound at run-time & relocations that need to be performed during loading.

- Relocation & linking operation are specified using set of processor-specific code
  The codes reflect instruction format & addressing modes that are found in the machine as they describe the size of the field to be modified & calculations that need to be performed.

- Symbolic references from i/p file that aren't resolved are processed by referring to archives & shared objects
  archives → collection of relocatable object modules.
  Directory within archives associate symbol name with object module that contains its definition. & selected module from archive is included to resolve the reference

~~Shared~~
Shared object is an indivisible unit that was generated by some previous link-edit operation.
If reference symbol is defined in a shared object, entire content of shared object becomes logical part of o/p file.
Link-editor records dependency to shared object, actual inclusion of the shared object happens at run time

- SunOS run-time linker,
  used to build dynamic executable & shared objects at execution time
  It determines what shared objects are required by dynamic executable & ensures that they are included.
  It also resolves any additional depencies from other shared objects.

After locating & including necessary objects, linker performs relocation & linking to prepare program for execution.

They bind symbol to actual memory address to which segment is loaded &. Then control is passed to executable program after binding data reference.

Binding of procedure call is done during execution. During link-edit, calls to globally defined procedure is converted to reference to a procedure linkage table. When procedure is called for the 1st time, control is passed to run-time linker via the table. The linker looks up the actual address of the procedure & includes it to linkage table.

So, subsequent call will directly go to called procedure → lazy binding.

- Run-time linker provides flexibility.

During execution, prog can dynamically bind to new shared objects; ~~by~~ this allows prog to choose b/w no. of shared objects.

If a shared object isn't needed it isn't ~~binded~~ binded.

→ Cray MPP Linker : for Cray T3E system.

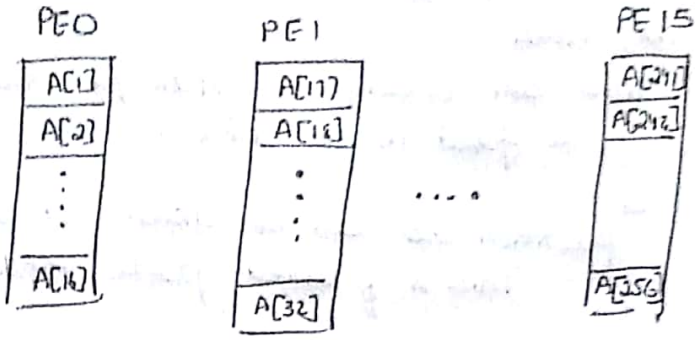
- T3E system contain large no. of processing elements (PEs).

Each PE has its own local memory & can access memory of all other PEs.

- An application program on a T3E system is allocated a partition that consist of several PEs. (to take advantage of parallel architecture of machine)

- Work to be done is divided into b/w the PEs

eg - partition Contain consists of 16 PEs, & elements of a 1D array is distributed



If prog contains loop that process all 256 elements, PE0 can execute loop for A[0] to A[15] PE1 can execute loop for A[17] to A[32] & so on.

- Shared data → data that is divided among no. of PEs.

Private data → data that isn't shared by dividing it, each PE contains a copy of the data.
Or PE has private data that exists only in its own local memory

- When program is loaded,
  each PE gets a copy of executable code, its private data & its portion of shared data.
- MPP linker organizes blocks of code or data from object program into lists.
  The blocks on a given list all share some same property.
  The blocks on each list is collected, address is assigned to each block & relocation and linking operations are performed.
  The linker then writes a executable file that contains relocated & linked blocks. It also specifies no. of PEs required & other control information.

- Distribution of shared data depends on no. of PEs.

If no. of PEs is specified at compile time, it can't be overridden later.

If no, either
linker can create executable file that targets for a fixed no. of PEs
or
partition size can be chosen at run time
This is called plastic executable

Plastic executable is often larger than one targeted for fixed no. of PEs as,
it must contain copy of all relocatable object module & all linker directives that are needed to produce final executable.