System Programming

(Chapter 3: Loaders and Linkers)

김지환교수

Office: AS관 713

Tel: 705-8924

Email: kimjihwan@sogang.ac.kr

Definitions of Loaders and Linkers

Loading : brings the object program into memory for execution.

Relocation : modifies the object program so that it can be loaded at an address different from the location originally specified.

Linking : combines two or more object programs.

Original Definition

Loader *:* a system program that performs the loading function.

Linker *:* a system program that performs the linking operation, thereby needs a separate loader.

In most cases, all the program translators (e.g., assemblers, compilers) on a particular system produce object programs in the same format. Thus one system loader or linker can be used regardless of the original source programming language.

Sometimes the definition of loader includes the linker functions.

An Absolute Loader

Brings an object program into memory and starts its execution without performing such functions as linking and program relocation.

All functions are accomplished in a single pass.

Algorithm for an 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 into
           internal representation}
        move object code to specified location in memory
        read next object program record
     end
   jump to address specified in End record
end
```

In text, all object codes are represented as character (8 bits) for educational purpose. However, the memory contents should be binary.

(e.g.) "1" "4" occupy two bytes in object program => Hexadecimal value 14 should be stored in a single byte.

Loading of an Absolute Program

```
HCOPY  001000001007A
T001000 1E 141033 48203900 103628 103030 101548 20613C 100300 102A 0C1039 00102D
T00101E 150C10 364820 61081033 4C0000 454F46 000003 000000
T002039 1E 041030 001030 E0205D 30203F D8205D 281030 302057 549039 2C205E 38203F
T002057 1C 101036 4C0000 F100100 041030 E02079 302064 509039 DC2079 2C1036
T002073 07 382064 4C0000 05
E001000
```

(a) Object program

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	No Text records
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
1000	14103348	20390010	36281030	30101548	COPY
1010	20613C10	0300102A	0C103900	102D0C10	
1020	36482061	0810334C	00000454F	46000003	
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0	
2040	205D3020	3FD8205D	28103030	20575490	
2050	392C205E	38203F10	10364C00	00F10010	
2060	00041030	E0207930	20645090	39DC2079	
2070	2C103638	20644C00	0005xxxx	xxxxxxxx	No Text Records
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

(b) Program loaded in memory

A Bootstrap Loader

A special type of absolute loader that is executed when a computer is first turned on or restarted.

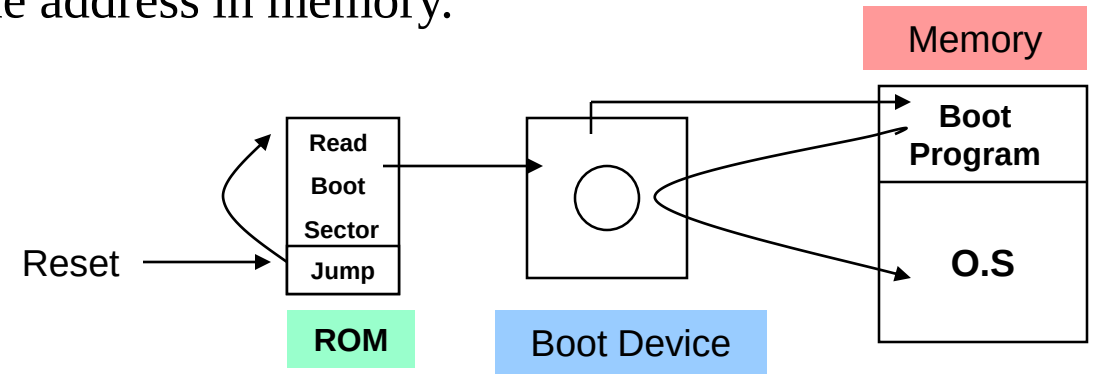
This bootstrap loads the first program to be run by the computer – usually an operating system.

How is the loader itself loaded into memory ?

Early computer : operators enter the object code on the computer console using switches.

Absolute loader is permanently resident in a read-only memory (ROM).

A built-in hardware function (or a very short ROM program) that reads a fixed-length record from some device into memory at fixed location. Transfer controls to the address in memory.



Bootstrap Loader for SIC/XE(강의제외 참조용)

```
BOOT     START   0           BOOTSTRAP LOADER FOR SIC/XE

.  THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
.  INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL).  AFTER ALL OF
.  THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
.  BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
.  THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
.  TO BE LOADED.

         CLEAR   A           CLEAR REGISTER A TO ZERO
         LDX     #128        INITIALIZE REGISTER X TO HEX 80
LOOP     JSUB    GETC        READ HEX DIGIT FROM PROGRAM BEING LOADED
         RMO     A,S         SAVE IN REGISTER S
         SHIFTL  S,4         MOVE TO HIGH-ORDER 4 BITS OF BYTE
         JSUB    GETC        GET NEXT HEX DIGIT
         ADDR    S,A         COMBINE DIGITS TO FORM ONE BYTE
         STCH    0,X         STORE AT ADDRESS IN REGISTER X
         TIXR    X           ADD 1 TO MEMORY ADDRESS BEING LOADED
         J       LOOP        LOOP UNTIL END OF INPUT IS REACHED

.  SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
.  CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE.  THE
.  CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A.  WHEN AN
.  END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
.  ADDRESS (HEX 80).
.
GETC     TD      INPUT       TEST INPUT DEVICE
         JEQ     GETC        LOOP UNTIL READY
         RD      INPUT       READ CHARACTER
         COMP    #4          IF CHARACTER IS HEX 04 (END OF FILE),
         JEQ     80              JUMP TO START OF PROGRAM JUST LOADED
         COMP    #48         COMPARE TO HEX 30 (CHARACTER '0')
         JLT     GETC        SKIP CHARACTERS LESS THAN '0'
         SUB     #48         SUBTRACT HEX 30 FROM ASCII CODE
         COMP    #10         IF RESULT IS LESS THAN 10, CONVERSION IS
         JLT     RETURN          COMPLETE.  OTHERWISE, SUBTRACT 7 MORE
         SUB     #7              (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN   RSUB                RETURN TO CALLER
INPUT    BYTE    X'F1'       CODE FOR INPUT DEVICE
         END     LOOP
```

Conversion

A Relocating Loader (Relative Loader)

Loaders that allow for program relocation.

Two ways for program relocation:

Use Modification Record

This is a convenient means for specifying program relocation, however, it is not well suited for use with all machine architectures.

For SIC machine, where the relative addressing is not used, almost all the instructions should be modified when the program is relocated.

Use Relocation Bit (강의제외 참조용)

This is for a machine that primarily uses direct addressing and has a fixed instruction format (e.g., SIC machine).

A *relocation bit* is associated with each word of object code.

The relocation bits are gathered together into a *bit mask* following the length indicator in each Text record.

If the relocation bit is set to 1, the program's starting address is to be added to this word when the program is relocated. A bit value of 0 indicates that no modification is necessary.

Example of a SIC/XE Program (From Fig 2.6)

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110					
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#4096	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER,X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195					
200			SUBROUTINE TO WRITE RECORD FROM BUFFER		
205					
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Need Relocation using Modification Record

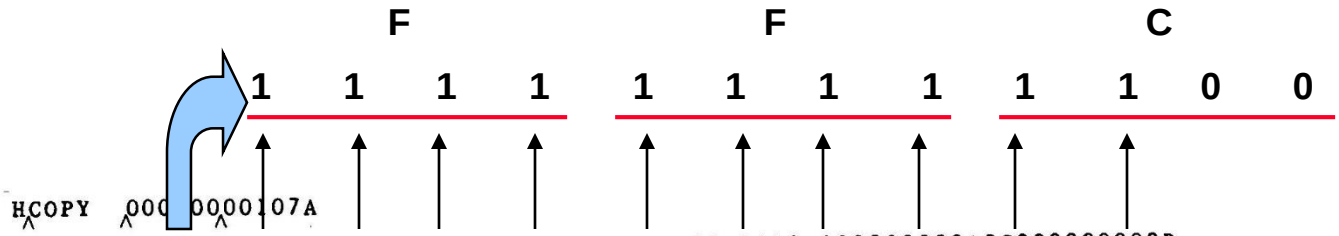
Example of a SIC Program (Re-locatable) (강의제외 참조용)

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061
40	0012		J	CLOOP	3C0003
45	0015	ENDFIL	LDA	EOF	00002A
50	0018		STA	BUFFER	0C0039
55	001B		LDA	THREE	00002D
60	001E		STA	LENGTH	0C0036
65	0021		JSUB	WRREC	481061
70	0024		LDL	RETADR	080033
75	0027		RSUB		4C0000
80	002A	EOF	BYTE	C'EOF'	454F46
85	002D	THREE	WORD	3	000003
90	0030	ZERO	WORD	0	000000
95	0033	RETADR	RESW	1	
100	0036	LENGTH	RESW	1	
105	0039	BUFFER	RESB	4096	
110					
115			SUBROUTINE TO READ RECORD INTO BUFFER		
120					
125	1039	RDREC	LDX	ZERO	040030
130	103C		LDA	ZERO	000030
135	103F	RLOOP	TD	INPUT	E0105D
140	1042		JEQ	RLOOP	30103F
145	1045		RD	INPUT	D8105D
150	1048		COMP	ZERO	280030
155	104B		JEQ	EXIT	301057
160	104E		STCH	BUFFER,X	548039
165	1051		TIX	MAXLEN	2C105E
170	1054		JLT	RLOOP	38103F
175	1057	EXIT	STX	LENGTH	100036
180	105A		RSUB		4C0000
185	105D	INPUT	BYTE	X'F1'	F1
190	105E	MAXLEN	WORD	4096	001000
195					
200			SUBROUTINE TO WRITE RECORD FROM BUFFER		
205					
210	1061	WRREC	LDX	ZERO	040030
215	1064	WLOOP	TD	OUTPUT	E01079
220	1067		JEQ	WLOOP	301064
225	106A		LDCH	BUFFER,X	508039
230	106D		WD	OUTPUT	DC1079
235	1070		TIX	LENGTH	2C0036
240	1073		JLT	LOOP	381064
245	1076		RSUB		4C0000
250	1079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Need Relocation Using Relocation Bit

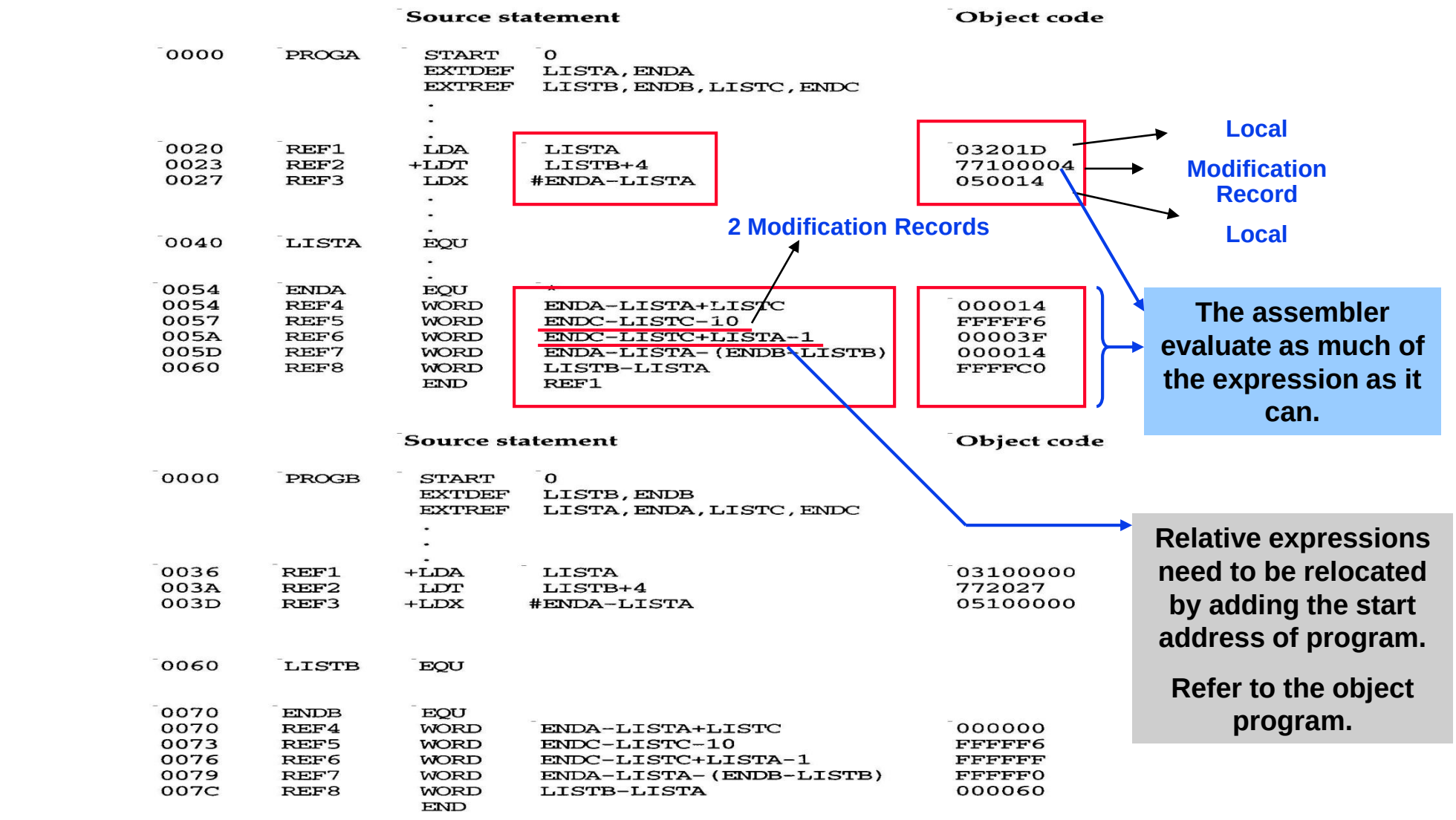
Modification Records and Relocation Bits

```
HCOPY  000000001077
T0000001D17202D69202D4B10103603202629000033200 74B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```




```
HCOPY  000000107A
T0000001EFFC140033481039000036280030300015481061 3C000300002A0C0039000 02D
T00001E15E000C003648106108003 34C0000454F460000 03000000
T0010391EFFC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

Example for Program Linking – Fig 3.8 (1)



Example for Program Linking – Fig 3.8 (2)

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		

Object Programs Corresponding to Fig 3.8

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
:
:
T0000200A03201D77100004050014
:
:
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

LISTA is a relative symbol

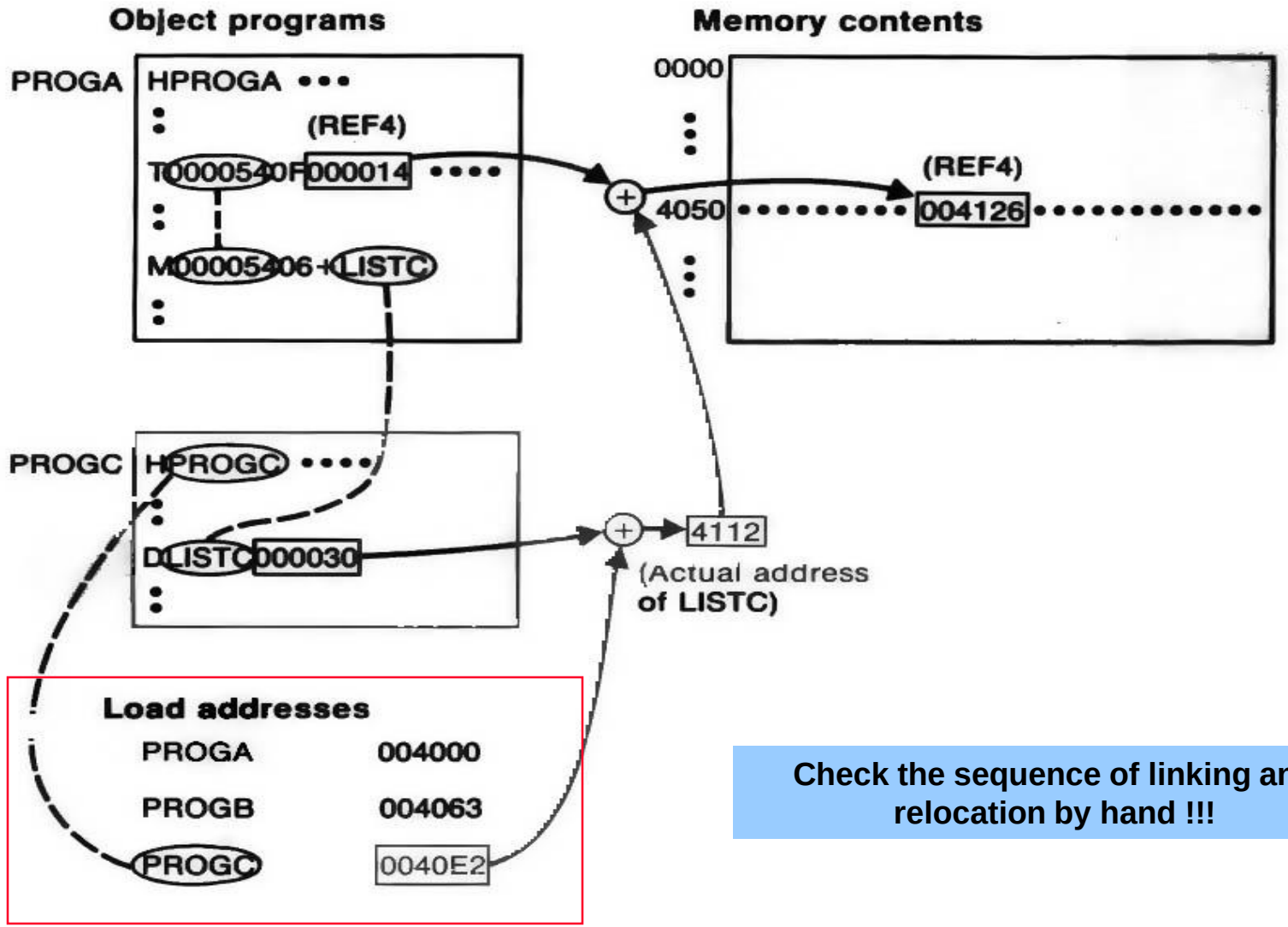
```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
:
:
T0000360B03100000772027105100000
:
:
T0000700F000000FFFFF6FFFFFEFFFFF0000060
M00003705+LISTA
M00003E05+ENDA
M00003E05-LISTA
M00007006+ENDA
M00007006-LISTA
M00007006+LISTC
M00007306+ENDC
M00007306-LISTC
M00007606+ENDC
M00007606-LISTC
M00007606+LISTA
M00007906+ENDA
M00007906-LISTA
M00007C06+PROGB
M00007C06-LISTA
E
```

LISTB is a relative symbol.

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
:
:
T0000180C03100000771000040510000O
:
:
T0000420F000030000008000011000000000000
M00001905+LISTA
M00001D05+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004B06+ENDA
M00004B06-LISTA
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB
M00004E06-LISTA
E
```

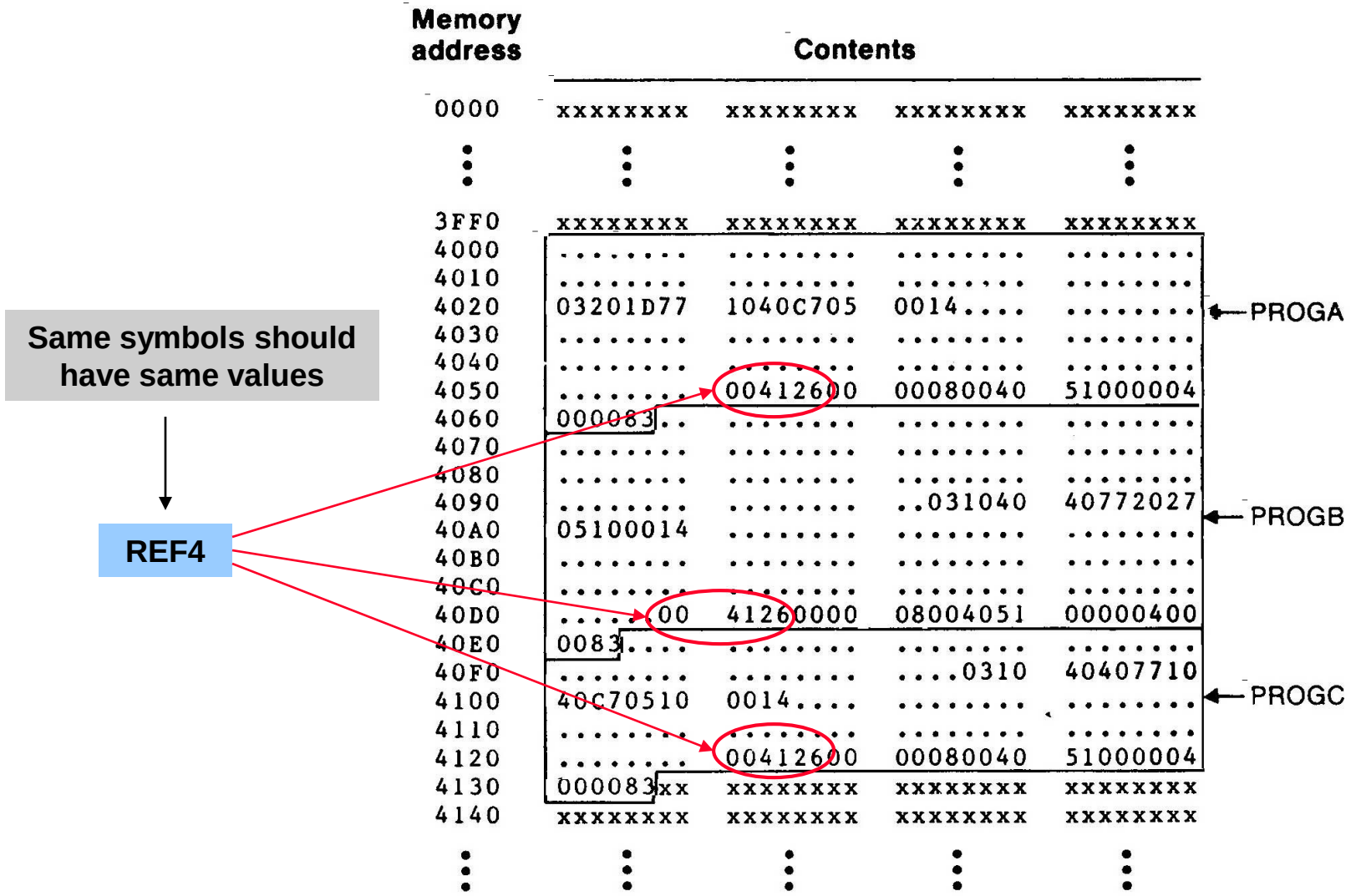
LISTC is a relative symbol.

Linking and Relocation Operations (REF4)



Check the sequence of linking and relocation by hand !!!

Programs (Fig 3.8) after Linking and Loading



Algorithm and Data Structure

Linking Loader usually makes two passes: -> It is possible for a control section to make an external reference to a symbol whose definition does not appear until later in this input stream.

Pass 1 : Assigns addresses to all external symbols (Fig 3.11 (a)).

Pass 2 : Performs the actual loading, relocation, and linking (Fig 3.11 (b)).

Data Structures

ESTAB (External Symbol Table) – Hash Table.

Stores the name and addresses of each external symbol in the set of control sections.

Indicates in which control section the symbol is defined.

PROGADDR (Program Load Address)

Beginning address in memory where the linked program is to be loaded. Its value is supplied by the operating system.

CSADDR (Control Section Address)

Starting address assigned to the control section currently being scanned by the loader.

Pass 1 of Linking Loader

Concerned only with H (Header) and D (Define) records.

The beginning load address (PROGADDR) is obtained from the OS.

This becomes the starting address for the first control section (CSADDR).

The control section name from the *H record* is entered into ESTAB, with value given by CSADDR.

All external symbols in the *D record* are also entered into ESTAB.

Their addresses are obtained by adding the value specified in the D record to CSADDR.

When the *E* (End) *record* is read, the control section length CSLTH (which was saved from the H record) is added to CSADDR.

This calculation gives the starting address for the next control section.

At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.

Many loader also generates a *load map* (printout of ESTAB) for debugging.

Algorithm for Pass 1 of a Linking Loader

Control section	Symbol name	Address	Length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	0051
	LISTC	4112	
	ENDC	4124	

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 of Linking Loader

Performs the actual loading, relocation, and linking of the program.

CSADDR is used in the same way it was in Pass 1.

As each *T* (Text) *record* is read, the object code is moved to the specified address (plus the current value of CSADDR).

When a *M* (Modification) *record* is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.

This value is then added or subtracted from the indicated location in memory.

When the *E* (End) *record* is read, get the address of the first instruction and transfer control for execution.

If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.

Normally a transfer address would be placed in the E record for a main program, but not for a subroutine.

Algorithm for Pass 2 of a Linking Loader

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}
```

Reference Number

Pass 2 algorithm can be more efficient by using a *reference number* in the object format.

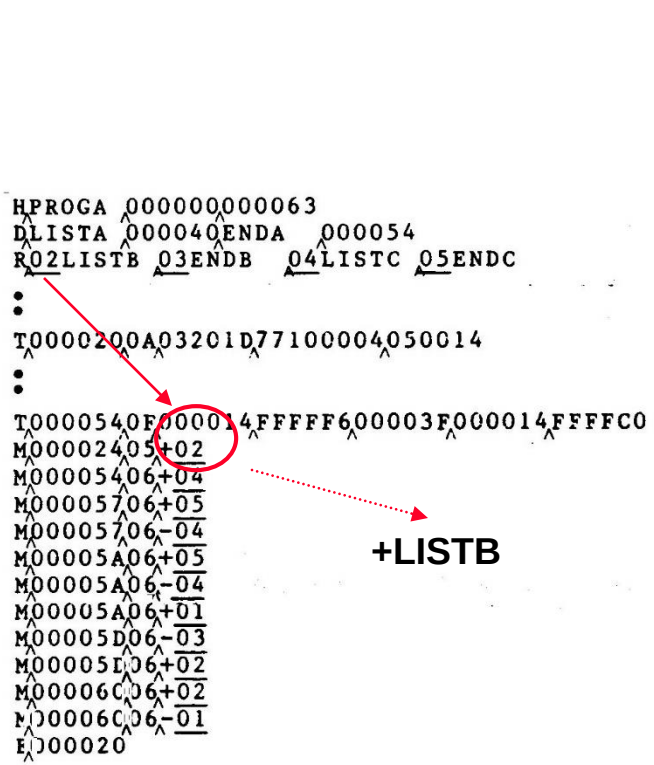
A reference number is assigned to each external symbol referred to in a control section (e.g., R records).

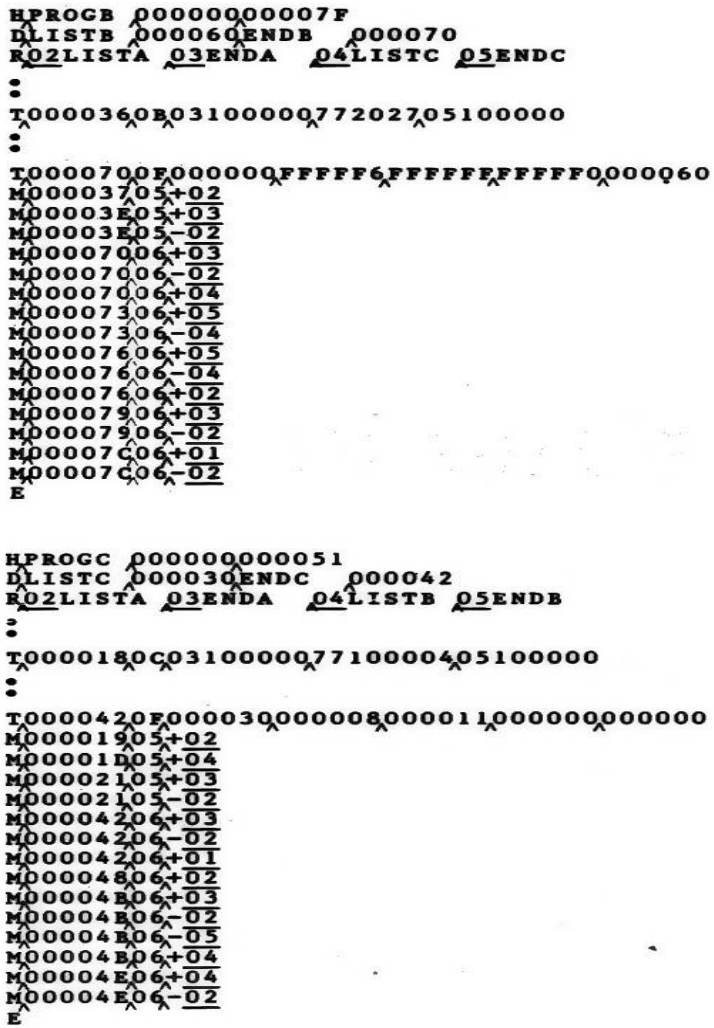
This reference number is used (instead of the symbol name) in Modification records.

This avoids multiple searches of ESTAB for the same symbol during the loading of a control section.

The values for code modification can then be obtained by simply indexing into an array of these values.

Example of Reference Number





Automatic Library Search (1)

The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program, and loaded => *automatic library call* or *automatic library search*.

Most standard libraries are used in this way.

Other libraries may be specified by control statements or by parameters to the loader (e.g., -l option in C compiler).

Linking loaders must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

At the end of Pass 1, the symbols in ESTAB that remain undefined represent *unresolved* external references and the loader searches the library for linking.

Automatic Library Search (2)

If the subroutines fetched from a library contain other external references, repeat the search until all references are resolved.

If unresolved external references remain after the search, these must be errors.

Programmers can *override* the standard subroutines in the library by supplying his or her own routines (User defined symbols have higher priorities).

Libraries are often object programs (assembled or compiled version of the subroutines) and we need to scan D records for all of the object programs on the libraries => inefficient.

A special file structure called a *directory* is used for this process.

A directory contains the name of each routine and a pointer to its address within the file.

The directory for commonly used libraries can be put in memory permanently to expedite the search process.

Loader Options (1) (강의제외 참조용)

Many loaders allow the user to specify options using a special command language that modify the standard processing.

Types of the option specification

Using a separate input file.

Embedded in the primary input stream between object programs.

Included in the source program.

Job control language that is processed by the operating system.

Loader commands

INCLUDE program-name(library-name)

Direct the loader to read the designated object program from a library.

DELETE csect-name

Instruct the loader to delete the named control section(s) from the set of programs being loaded.

Loader Options (2) (강의제외 참조용)

Loader commands (continued)

CHANGE name1, name2

Cause the external symbol name1 to be changed to name2 wherever it appears in the object programs.

Example

```
INCLUDE     READ(UTLIB)
INCLUDE     WRITE(UTLIB)
DELETE      RDREC, WRREC
CHANGE      RDREC, READ
CHANGE      WRREC, WRITE
```

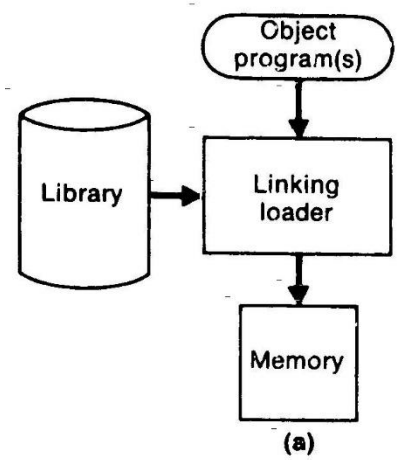
LIBRARY MYLIB

Allow the user to specify alternative libraries to be searched before the standard libraries.

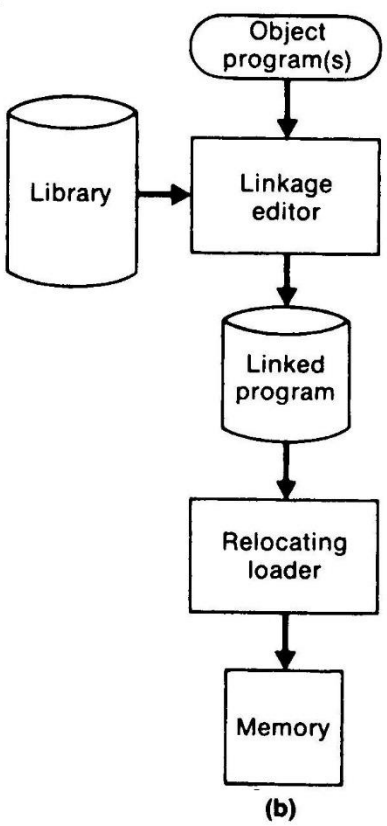
NOCALL STDDEV, PLOT, CORREL

Instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and save memory space.

Linking Loaders vs. Linkage Editors



Linking Loader



Linkage Editor

Linkage Editors (1)

A linkage editors, unlike the linking loaders, produces a linked version of the program (*load module* or *executable image*), which is written to a file or library for later execution.

When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory.

The only object code modification necessary is the addition of an actual load address to relative values within the program.

This means that the loading can be accomplished in one pass with no external symbol table required.

If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.

Linkage Editors (2)

Resolution of external references and library searching are only performed once. In contrast, a linking loader searches libraries and resolves external references every time the program is executed.

For testing environment (a program is reassembled for nearly every execution), it is more efficient to use a linking loader.

If the actual address at which the program will be loaded is known in advance, the linkage editor can perform all of the needed relocation.

The result is a linked program that is an exact image of the way the program will appear in memory during execution. => absolute program.

Normally, the added flexibility of being able to load the program at any location is worthy.

Useful Functions by Linkage Editors (1) (강의제외 참조용)

Case 1: When one subroutine used by the program is changed to correct an error or to improve efficiency, only that subroutine can be assembled and replaced into the linked version of the program.

```
INCLUDE     PLANNER(PROGLIB)
DELETE      PROJECT      {DELETE from existing PLANNER}
INCLUDE     PROJECT(NEWLIB)   {INCLUDE new version}
REPLACE     PLANNER(PROGLIB)
```

Case 2: Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This saves a substantial amount of linking overhead.

```
INCLUDE     READER(FTNLIB)
INCLUDE     WRITER(FTNLIB)
INCLUDE     BLOCK(FTNLIB)
INCLUDE     DEBLOCK(FTNLIB)
…..
SAVE        FTNIO(SUBLIB)  ---→ Make a package FTNIO
```

Useful Functions by Linkage Editors (2) (강의제외 참조용)

Case 3: Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search.

If 100 programs that use the I/O routines in case 2 above were to be stored on a library, 100 copies of the package (e.g., FTNIO) would be stored. -> highly undesirable.

The user can specify that no library search be performed during linkage editing.

Thus only the external references between user-written routines will be resolved and the linking loader can combine the linked user routines with the package (e.g., FTNIO) at execution time.

As a result, this process involves two separate linking operations.

Although this requires slightly more overhead, it will result in a large savings in library space.

Compared to linking loaders, linkage editors tend to offer more flexibility and control with a corresponding increase in complexity and overhead.

Dynamic Linking

When the linking is performed:

Linkage editor: before the program is loaded for execution.

Linking loader: at load time.

Dynamic Linking: A scheme that postpones the linking function until execution time (or, the address *binding* is delayed until execution time) - a subroutine is loaded and linked to the rest of the program when it is first called. Also called *dynamic loading* or *load on call*.

Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library (e.g., *dynamic link library* (DLL) in C language) – save memory space.

The implementation can be changed at any time, without affecting the program that makes use of the dynamic link library.

Examples of Dynamic Linking

Case 1: A program contains subroutines that correct or diagnose errors during execution (e.g., rarely used subroutines).

If such errors are rare, the correction and diagnostic routines may not be used at all during most executions of the program.

If the program were completely linked before execution, these subroutines would need to be loaded and linked every time the program is run.

Dynamic linking provides the ability to load the routines only when they are needed.

This can result in substantial savings of time and memory space if the subroutines involved are large or have many external references.

Case 2: A program uses only a few of a large number of possible subroutines, but the exact routines needed can not be predicted until the program examines its input (e.g., subroutines needed but not used at the same time)

Loading and Linking Steps for Dynamic Linking

Instead of executing or referencing external symbol, the program makes a *load-and-call* service request to the OS (assume the dynamic loader is a part of operating system).

The OS examines its internal tables to determine whether or not the routine is already loaded.

If not loaded, the routine is loaded from the specified user or system libraries.

Control is then passed from the OS to the routine being called.

When the called subroutine completes its processing, it returns to its caller (e.g., OS) and the OS returns control to the program that issued the request.

After the subroutine is completed, the allocated memory can be reused or the routine remains in the memory for later use.

Loading and Calling using Dynamic Linking