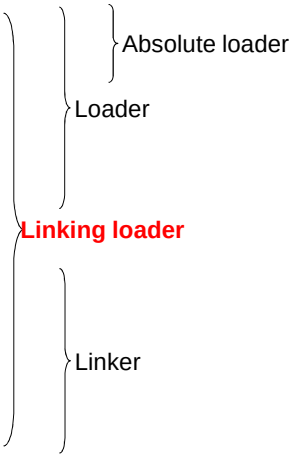
Processes to Run an Object Program

- Loading
  - Brings object program into memory
- Relocation
  - Modifies the object program where absolute addresses are specified
- Linking
  - Combines two or more separate object programs and supplies information needed to allow cross-references.



Program Linking

- A program is a logical entity that combines all of the related control sections.
- Control sections could be assembled together, or they could be assembled independently of one another.
- Control sections are to be linked, relocated, and loaded by loaders.
- External references among control sections can be assigned addresses after these control sections are loaded into memory by loaders.

LDA	00
LDT	74
LDX	04

Sample Program for Linking and Relocation

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	

Sample Program for Linking and Relocation

Loc		Source statement		Object code
0000	PROGB	START	0	
		EXTDEF	LISTB,ENDB	
		EXTREF	LISTA,ENDA,LISTC,ENDC	
		.		
		.		
		.		
0036	REF1	+LDA	LISTA	03100000
003A	REF2	LDT	LISTB+4	772027
003D	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	000060
		END		

Sample Program for Linking and Relocation

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		

Sample Program for Linking and Relocation

- Each control section defines a list:
  - Control section A: LISTA --- ENDA
  - Control section B: LISTB --- ENDB
  - Control section C: LISTC --- ENDC
- Each control section contains exactly the same set of references to these lists
  - REF1 through REF3: instruction operands
  - REF4 through REF8: values of data words
- After these control sections are linked, relocated, and loaded, each of REF4 through REF8 should have resulted in the same value in each of the three control sections. (but not for REF1 through REF3, why?)

Object Code of Control Section A

```
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
```

Object Code of Control Section B

```
HPROGB 000000^00007F
DLISTB 000060^ENDB  000070
RLISTA ENDA  LISTC ENDC
:
:
T000036^0B^03100000^77202705100000
:
:
T000070^0F^000000^FFFFF6^FFFFFFFFFFF0^000060
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
```

Object Code of Control Section C

```
HPROGC 000000^000051
DLISTC 000030^ENDC  000042
RLISTA ENDA  LISTB ENDB
:
:
T000018^0C^03100000^77100004^05100000
:
:
T000042^0F^000030^000008^000011^000000^000000
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
```

REF1 (LISTA)

- Control section A
  - LISTA is defined within the control section.
  - Its address is immediately available using PC-relative addressing.
  - No modification for relocation or linking is necessary.
- Control sections B and C
  - LISTA is an external reference.
  - Its address is not available thus an extended-format instruction with address field set to 00000 is used.
  - A modification record is inserted into the object code to instruct the loader to add the value of LISTA (once determined) to this address field.

REF2 (LISTB+4)

- Control sections A and C
  - REF2 is an external reference (LISTB) plus a constant.
  - The address of LISTB is not available thus an extended-format instruction with address field set to 00004 is used.
  - A modification record is inserted into the object code to instruct the loader to add the value of LISTB (once determined) to this address field.
- Control section B
  - LISTB is defined within the control section.
  - Its address is immediately available using PC-relative addressing.
  - No modification for relocation or linking is necessary.

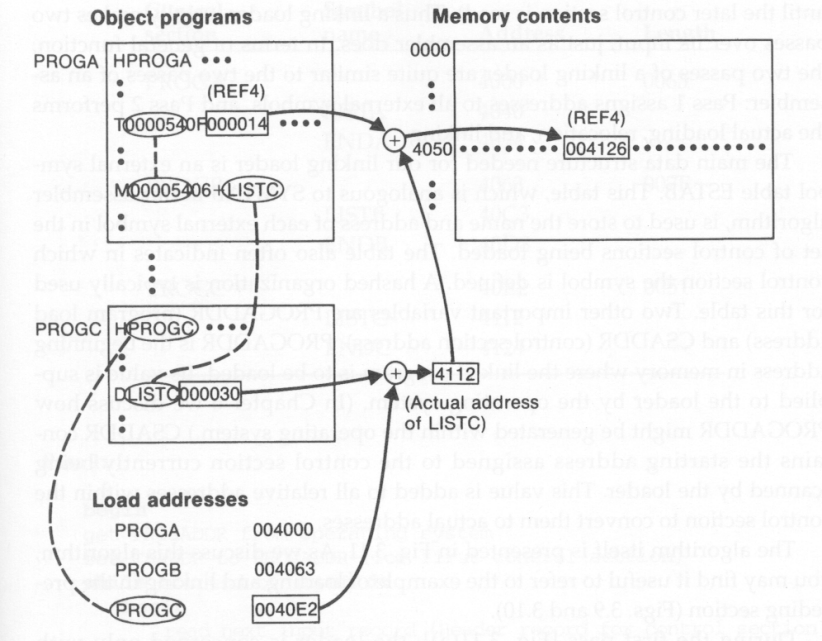
REF3 (#ENDA-LISTA)

- Control section A
  - ENDA and LISTA are defined within the control section.
  - The difference between ENDA and LISTA is immediately available.
  - No modification for relocation or linking is necessary.
- Control sections B and C
  - ENDA and LISTA are external references.
  - The difference between them is not available thus an extended-format instruction with address field set to 00000 is used.
  - Two modification records are inserted into the object code
    - +ENDA
    - -LISTA

REF4 (ENDA-LISTA+LISTC)

- Control section A
  - The values of ENDA and LISTA are known when assembled. Only the value of LISTC is unknown.
  - The address field is initialized as 000014 (ENDA-LISTA).
  - One Modification record is needed for LISTC:
    - +LISTC
- Control section B
  - ENDA, LISTA, and LISTC are all unknown.
  - The address field is initialized as 000000.
  - Three Modification records are needed:
    - +ENDA
    - -LISTA
    - +LISTC
- Control section C
  - LISTC is defined in this control section but ENDA and LISTA are unknown.
  - The address field is initialized as the relative address of LISTC (000030)
  - Three Modification records are needed:
    - +ENDA
    - -LISTA
    - +PROGC (for relocation)

Calculation of REF4 (ENDA-LISTA+LISTC)



Calculation of REF4 (ENDA-LISTA+LISTC)

- Control section A
  - The address of REF4 is 4054 (4000 + 54)
  - The value of REF4 is:

    000014 + 004112 = 004126

    (initial value) (address of LISTC)
  - The address of LISTC is:

    0040E2 + 000030 = 004112

    (starting address of PROGC) (relative address of LISTC in PROGC)
- Control section B
  - The address of REF4 is 40D3 (4063 + 70)
  - The value of REF4 is:

    000000 + 004054 - 004040 + 004112 = 004126

    (initial value) (address of ENDA) (address of LISTA) (address of LISTC)

REF2 (LISTB+4) in Control section A

- REF2 +LDT LISTB+4 77100004

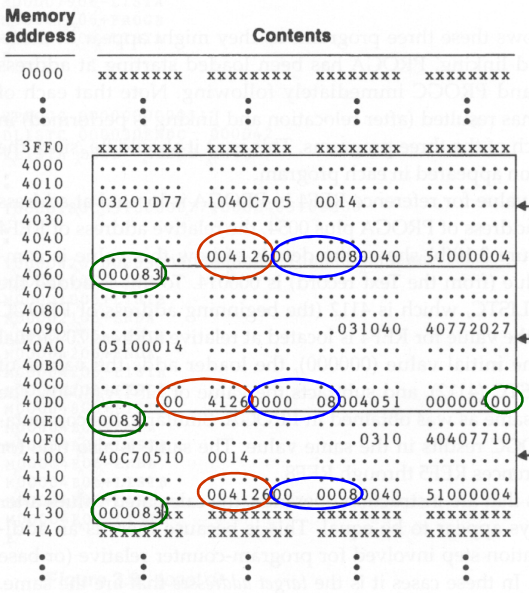
- Control section A
  - The address of REF2 is 4023 (4000 + 23)
  - The value of REF4 is:

    00004 + 040C3 = 040C7

    (initial value) (address of LISTB)
  - The address of LISTB is:

    04063 + 00060 = 040C3

Program in Memory after Linking and Loading



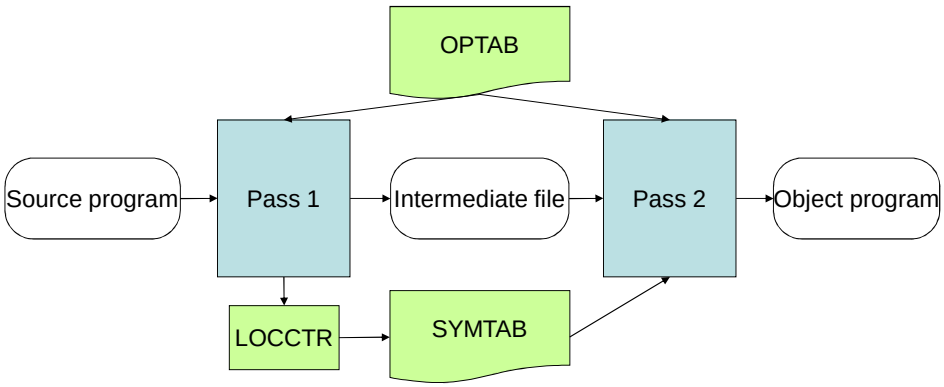
Values of REF4, REF5, …, REF8 in three places are all the same.

References in Instruction Operands

- For references that are instruction operands, the calculated values after loading do not always appear to be equal.
- This is because there is an additional address calculation step involved for PC (or base) relative instructions.
- In such cases, it is the target addresses that are the same.
- For example, in control section A, the reference REF1 is a PC relative instruction with displacement 01D. When this instruction is executed, the PC contains the value 4023. Therefore the resulting address is 4040. In control section B, because direct addressing is used, 4040 (4000 + 40) is stored in the loaded program for REF1.

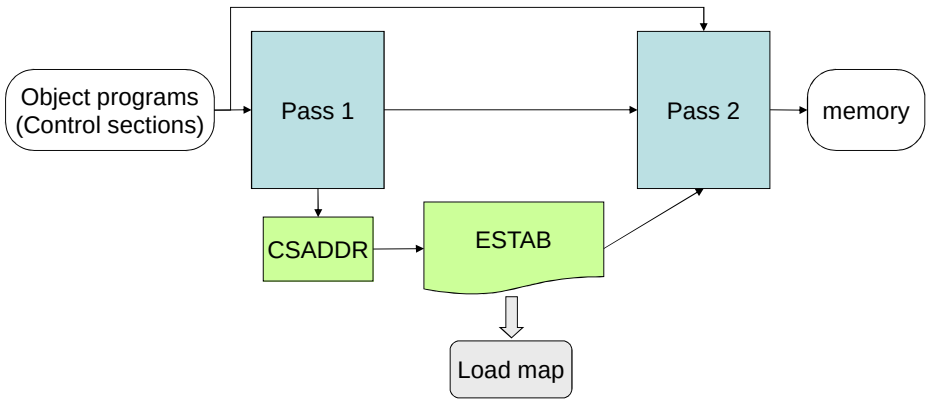
Recall:
Implementation of An Assembler

- Operation Code Table (OPTAB)
- Symbol Table (SYMTAB)
- Location Counter (LOCCTR)

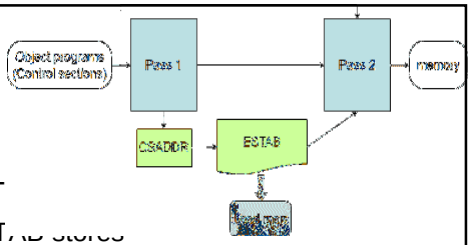


Implementation of a Linking Loader

- Two-pass process (similar to the Assembler):
  - Pass 1: assigns addresses to all external symbols
  - Pass 2: performs the actual loading, relocation, and linking



Data Structures



- External Symbol Table (EST
  - For each external symbol, EST[illegible]
    - its name
    - its address
    - in which control section the symbol is defined
  - Hashed organization
- Program Load Address (PROGADDR)
  - PROGADDR is the beginning address in memory where the linked program is to be loaded (supplied by OS).
- Control Section Address (CSADDR)
  - CSADDR is the starting address assigned to the control section currently being scanned by the loader.
  - CSADDR is added to all relative addresses within the control section.

A Load Map

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

1. loader gets the PROGADDR from operating system.
2. This PROGADDR becomes the starting address for the first control section.
3. The control section name from the **H**eader record is entered into ESTAB with the value give by CSADDR.
4. All the external symbols in the **D**efine record for the control section also entered into ESTAB.
5. When the End record is read , the control section length CSLTH is added to CSADDR to obtain the starting address for the next control section.
6. At end of the pass-1 ,ESTAB contains all the external symbols with value.

PASS-2

1. pass2 performs the actual loading ,relocation and linking of the program.
2. As each **T**ext record is read, the object code is moved to **the specified address plus the current value of CSADDR.**
3. When **M**odification record is encountered, the ESTAB is searched to find the address of the symbol specified is either added or subtracted from the indicated location in memory .
4. The loader then starts the execution from the address specified in the End record of the control section.

Algorithm for Linking Loader

Have a Try …

Algorithm

Pass 1: (only Header and Define records are concerned)

```
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}
```

Algorithm

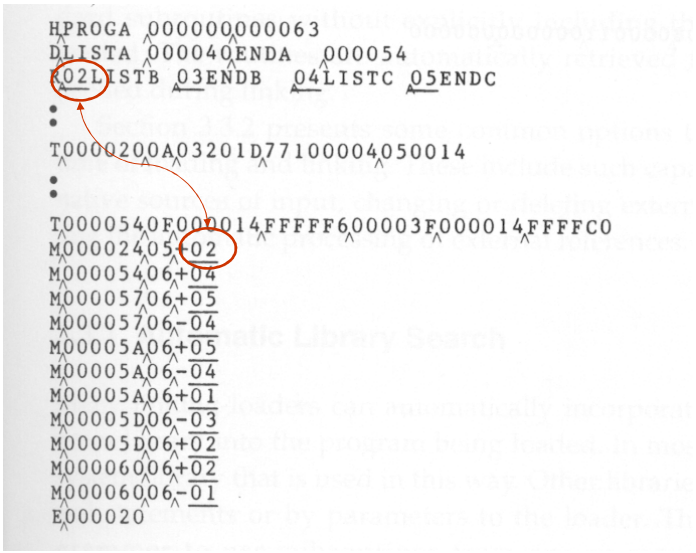
Pass 2:

```
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}
```

Enhance the Algorithm

- We can make the Assembler more efficient by storing search information in the intermediate file and avoiding the search of OPTAB in Pass 2.
- We can make the linking loader algorithm more efficient by:
  - assigning a reference number to each external symbol referred to in a control section
    - Control section name: 01
    - Other external reference symbols (stored in the Refer records): 02symname, 03symname, …
  - using this reference number (instead of the symbol name) in Modification records
  - avoiding multiple searches of ESTAB for the same symbol during the loading of a control section.
    - Search of ESTAB for each external symbol can be performed once and the result is stored in a table indexed by the reference number.
    - The values for code modification can then be obtained by simply indexing into the table.

Examples of Using Reference Numbers



Examples of Using Reference Numbers

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
:
T0000360B031000007720270510000O
:
T0000700F000000FFFFF6FFFFFFFFFF0000060
M00003705+02
M00003E05+03
M00003E05-02
M000070O6+03
M000070O6-02
M000070O6+04
M000073O6+05
M000073O6-04
M000076O6+05
M000076O6-04
M000076O6+02
M000079O6+03
M000079O6-02
M00007C06+01
M00007C06-02
E
```

Examples of Using Reference Numbers

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
:
T0000180C031000007710000405100000
:
T0000420F000030000008000011000000000000
M000019O5+02
M00001DO5+04
M000021O5+03
M000021O5-02
M000042O6+03
M000042O6-02
M000042O6+01
M000048O6+02
M00004BO6+03
M00004BO6-02
M00004BO6-05
M00004BO6+04
M00004EO6+04
M00004EO6-02
E
```

Machine-Independent Loader Features

Automatic Library Search
Loader Options

Machine-Independent Loader Features

- Automatic library search for handling external references
  - allows a programmer to use standard subroutines which are automatically retrieved from a library as they are needed during linking
- Options of linking and loading
  - Specifying alternative sources of input
  - Changing or deleting external references
  - Controlling the automatic processing of external references

Automatic Library Search

- Goal:
  - automatically incorporate routines from subprogram libraries into the program being loaded
    - Standard system library
    - Libraries specified by control statements or by parameters to the loader
- The subroutines called by the program are automatically fetched from the library, linked with the main program, and loaded.
- The programmer does not need to take any action beyond mentioning the subroutine names as external references in the source program

Implementation of Automatic Library Search

- Linking loader must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.
  - Enter symbols from each Refer record into ESTAB and mark them undefined unless these symbols are already present
  - Make the symbol defined when its definition is encountered
  - At the end of Pass1, the symbols in ESTAB that remain undefined represent unresolved external references.
- The loader searches the libraries for routines that contain the definitions of these unresolved symbols, and processes the subroutines found by this search process exactly as if they had been part of the primary input stream.
- Repeat the above library search process until all references are resolved, because the fetched subroutines may themselves contain external references.

Discussions of Automatic Library Search

- The presented linking loader allows the programmer to override the standard subroutines by supplying his/her own routines.
- Directory
  - The loader searches for the subroutines by scanning the Define records for all of the object programs.
  - It can be more efficient by searching a directory giving the name and address of each routine.
- The same techniques applies equally well to the resolution of external references to data items.

Loader Options

- Users can specify options that modify the standard processing of the loader.
- How to specify options (usually using a special command language):
  - Separate input file
  - Statements embedded in the primary input stream between object programs
  - Loader control statements included in the source program

Loader Options

- Typical loader options:
  - Selection of alternative sources of input

    INCLUDE program-name(library-name)
    - Direct the loader to read the designated object program from a library
  - Deletion of external symbols

    DELETE csect-name
    - Instruct the loader to delete the named control sections from the set of programs being loaded
  - Change the external symbols

    CHANGE name1, name2
    - Cause the external symbol *name1* to be changed to *name2* wherever it appears in the program
  - Automatic inclusion of library routines

    LIBRARY MYLIB
    - Specify alternative libraries to be searched
    - User-specified libraries are normally searched before the standard system libraries.

    NOCALL name
    - Specify the external reference *name* are to remain unresolved.

Example of Using Loader Options

- In the COPY program, two subroutines, RDREC and WRREC, are used to read/write records.
- Suppose that the utility subroutines contain READ and WRITE that perform the same functions of RDREC and WRREC.
- Without reassembling the program, a sequence of loader commands could be used to use READ and WRITE instead of RDREC and WRREC:

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

Loader Design Options

Linkage Editors
Dynamic Linking
Bootstrap Loaders

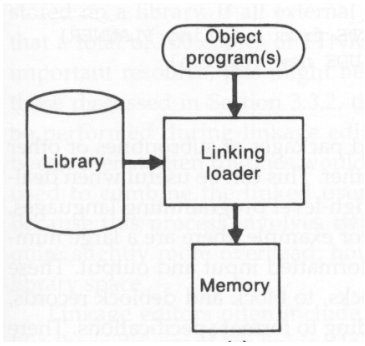
Linkage Editors

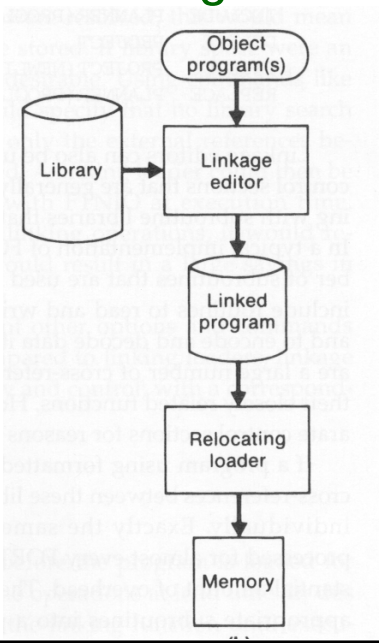
- Difference between a linkage editor and a linking loader:
  - Linking loader
    - performs all linking and relocation operations, including automatic library search, and loads the linked program into memory for execution.
  - Linkage editor
    - produces a linked version of the program, which is normally written to a file or library for later execution.
      - A simple relocating loader (one pass) can be used to load the program into memory for execution.
      - The linkage editor performs relocation of all control sections relative to the start of the linked program.
      - The only object code modification necessary is the addition of an actual load address to relative values within the program.

Linkage Editors

- Difference between a linkage editor and a linking loader:
  - Linking loader
    - Suitable when a program is reassembled for nearly every execution
      - In a program development and testing environment
      - When a program is used so infrequently that it is not worthwhile to store the assembled and linked version.
  - Linkage editor
    - Suitable when a program is to be executed many times without being reassembled because resolution of external references and library searching are only performed once.

Linking Loader vs. Linkage Editor





Additional Functions of Linkage Editors

- Replacement of subroutines in the linked program
  - For example:

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

- Construction of a package for subroutines generally used together
  - There are a large number of cross-references between these subroutines due to their closely related functions.
  - For example:

```
INCLUDE    READR(FTNLIB)
INCLUDE    WRITER(FTNLIB)
    :
SAVE         FTNIO(SUBLIB)
```

- Specification of external references not to be resolved by automatic library search
  - Can avoid multiple storage of common libraries in programs.
  - Need a linking loader to combine the common libraries at execution time.

Address Binding

- Address Binding:
  - Symbolic Address (label) ➔ Machine Address
- Address Binding:
  - Assembling Time: 8051
  - Load Time: 8086
  - Run Time: Dynamic Linking Library
- Address Binding
  - Complexity, Flexibility

Linking Time

- Linkage editors: before load time
- Linking loaders: at load time
- Dynamic linking: after load time
  - A scheme that postpones the linking function until execution time.
  - A subroutine is loaded and linked to the test of the program when it is first called.
  - Other names: *dynamic loading*, *load on call*

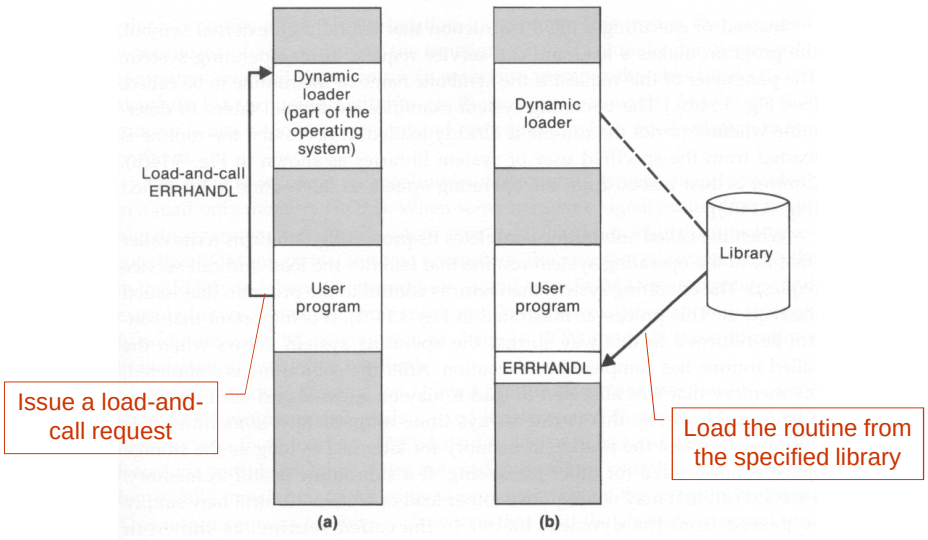
Dynamic Linking Application

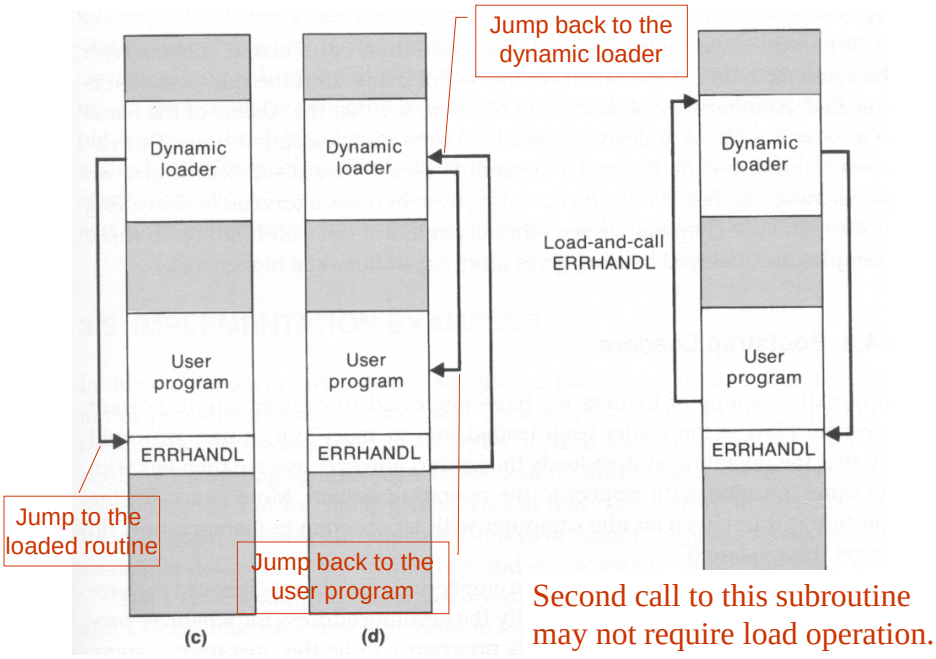
- Allows several executing programs to share one copy of a subroutine or library.
- Allows the implementation of the shared object and its methods to be determined at execution time in an object-oriented system
- Provides the ability to load the routines only when (and if) they are needed.
  - This can result in substantial savings of load time and memory space.
  - For example, error handling routines.

Implementation of Dynamic Linking

- Subroutines to be dynamically loaded must be called via an operating system service request, e.g., load-and-call, instead of executing a JSUB instruction.
- The service request will be handled by a part of the OS, called the dynamic loader, which is kept in memory during execution of the program.
- The parameter of this request is the symbolic name of the routine to be called..

Example of Dynamic Linking





Dynamic Linking

- **Advantages:**
  - Several executing programs can share one copy of a subroutine or library (Xwindows, C support routines).
  - The implementation of the method can be determined or even changed at the time the program is run.
  - The subroutine name might simply be treated as an input item (!)
- **Additional advantages:**
  - Applications with rarely used subroutines (f.e. errors handling)
  - Application with a lot of possible services (f.e. mathematic packages)

Implementation Example -- MS-DOS

- MS-DOS assembler (MASM) produce object modules (.OBJ)
- MS-DOS LINK is a linkage editor that combines one or more modules to produce a complete executable program (.EXE)
- MS-DOS object module
  - THEADER similar to Header record in SIC/XE
  - MODEND similar to End record in SIC/XE

MS-DOS object module

- TYPDEF data type
- PUBDEF similar to Define record in SIC/XE
- EXTDEF similar to Reference record in SIC/XE
- LNAMES contain a list of segments and class names
- SEGDEF segment define
- GRPDEF specify how segments are grouped
- LEDATA similar to Text Record in SIC/XE
- LIDATA specify repeated instructions
- FIXUPP similar to Modification record in SIC/XE

SunOS Linkers(3/1)

- SunOS actually provides two different linkers, called the link-editor and the run-time linker
- The link-editor takes one or more object modules produced by assemblers and compilers, and combines them to produce a single output module
- Different types of output modules: relocatable object module, static executable, dynamic executable, shared object

SunOS Linkers(3/2)

- The object module includes a list of the relocation and linking operations that need to be performed Symbolic references from the input files that do not have matching definitions are processed by referring to archives and shared objects
- An archive is a collection of relocatable object modules Selected modules from an archive are automatically included to resolve symbolic references A shared object is an indivisible unit that was generated by a previous link-edit operation

SunOS Linkers(3/3)

- The SunOS run-time linker is used to bind dynamic executable and shared objects at execution time
- When a procedure is called for the first time, control is passed via the linkage table to the run-time linker
- The linker looks up the actual address of the called procedure and inserts it into the linkage table, thus subsequent calls will go directly to the called procedure
- This process is sometimes referred to as *lazy binding*
- During execution, a program can dynamically bind to new shared objects depending on the exact services required

Cray MPP Linker

Cray MPP (massively parallel processing) Linker is developed for Cray T3E systems.

A T3E system contains large number of parallel processing elements (PEs) – Each PE has local memory and has access to remote memory (memory of other PEs).

Cray MPP Linker

The processing is divided among PEs - contains shared data and private data.

The loaded program gets copy of the executable code, its private data and its portion of the shared data.

The MPP linker organizes blocks containing executable code, private data and shared data

Cray MPP Linker

The linker then writes an executable file that contains the relocated and linked blocks. The executable file also specifies the number of PEs required and other control information

Cray MPP Linker

The linker can create an executable file that is targeted for a fixed number of PEs, or one that allows the partition size to be chosen at run time. Latter type is called plastic executable