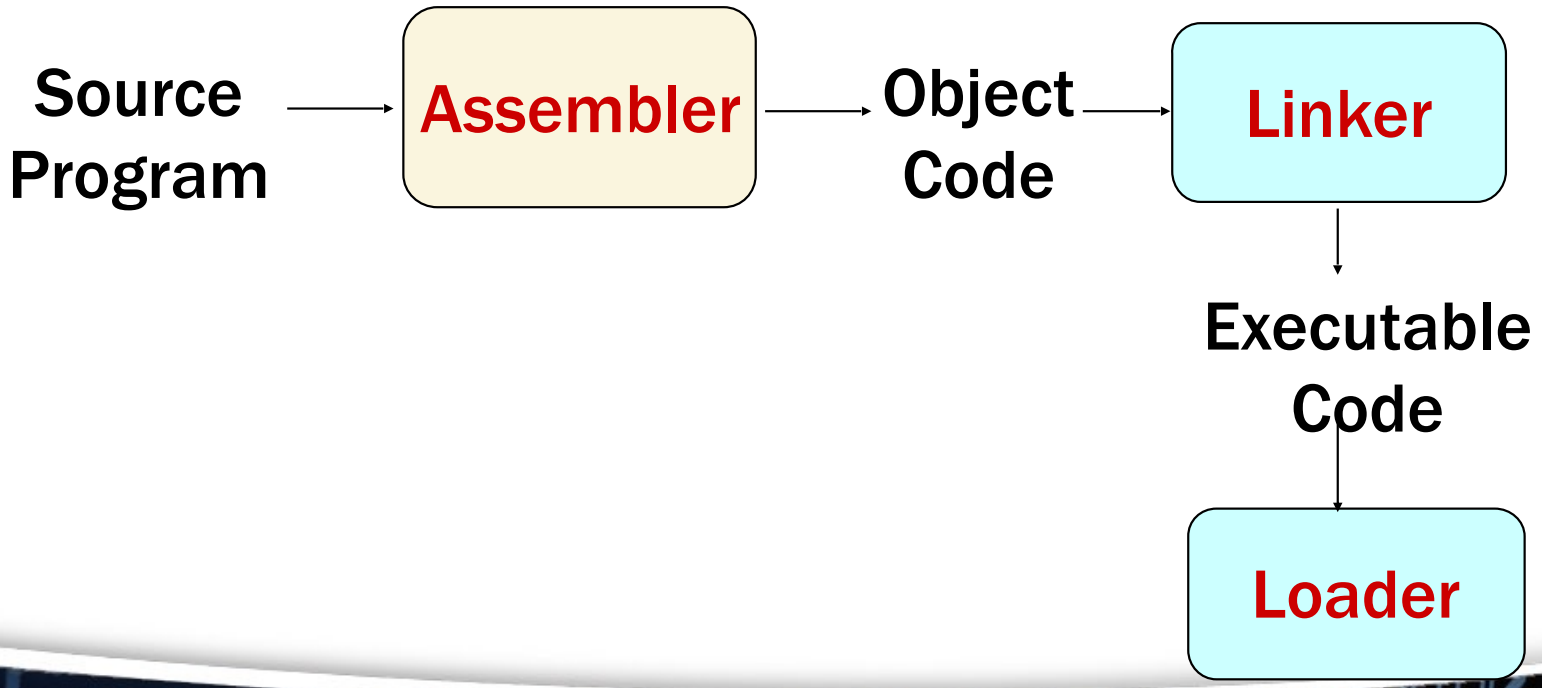
CC410: System Programming

Dr. Manal Helal – Fall 2014 – Lecture 10 –Linkers

Learning Objectives

- **Understand Linker Functions**
- **Differentiate between Machine Dependant and Independent Functions**



3.2.2 Program Linking

- **In Section 2.3.5 showed a program made up of three controls sections.**
  - **Assembled together or assembled independently?**
- Goal
  Resolve the problems with EXTREF and EXTDEF from different control sections
- Linking
  - 1. User, 2. Assembler, 3. Linking loader
- Example
  - Program in Fig.3.8 and object code in Fig.3.9
  - Use modification records for both relocation and linking
    - address constant
    - external reference

Lecture Outline

- Direct Linking Loader
  - The process (manually)
  - The Algorithm and the Data Structures
- Design options
  - linkage editors
  - dynamic linking
  - bootstrap loaders

3.2.2 Program Linking

- Consider the three programs in Fig. 3.8 and 3.9.
  - Each of which consists of a single control section.
  - A list of items, LISTA—ENDA, LISTB—ENDB, LISTC—ENDC.
  - Note that each program contains exactly the same set of references to these external symbols.
  - Instruction operands (REF1, REF2, REF3).
  - The values of data words (REF4 through REF8).
  - Not involved in the relocation and linking are omitted.

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	
		.		
		.		
		.		
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		

Figure 3.8 Sample programs illustrating 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		

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
.
.
T0000200A03201D7710000 4050014
.
.
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

Figure 3.9 Object programs corresponding to Fig. 3.8.

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
•
•
T0000360B031000007720270510000O
•
•
T000070OF000000FFFFF6FFFFFFFFFFF0000060
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
```

```
HPROGC ^000000000051
DLISTC ^000030^ENDC  ^000042
RLISTA ^ENDA  ^LISTB ^ENDB
:
:
T^0000180C^03100000^77100004^05100000
:
:
T^0000420F^00003000^00008^00001100^00000^000000
M^00001905^+LISTA
M^00001D05^+LISTB
M^00002105^+ENDA
M^00002105^-LISTA
M^00004206^+ENDA
M^00004206^-LISTA
M^00004206^+PROGC
M^00004806^+LISTA
M^00004B06^+ENDA
M^00004B06^-LISTA
M^00004B06^-ENDB
M^00004B06^+LISTB
M^00004E06^+LISTB
M^00004E06^-LISTA
E
```

Figure 3.9 (*cont'd*)

3.2.2 Program Linking

- REF1 In the PROGA is simply a reference to a label
  - REF1 LDA LISTA 03201D
- REF1 In PROGB and PROGC is a reference to an external symbols.
  - Need to use extended format, Modification record.
  - REF1 LDA LISTA 03100000
- REF2 in PROGA.
  - REF2 LDT LISTB+4 77100004
- REF3 in PROGB
  - REF3 LDX #ENDA-LISTA 05100000

3.2.2 Program Linking

- REF4 through REF8,
  - WORD ENDA-LISTA+LISTC 000014+000000
- Figure 3.10(a) and 3.10(b)
  - Shows these three programs as they might appear in memory after loading and linking.
  - PROGA 004000, PROGB 004063, PROGC 0040E2.
  - REF4 through REF8 in the same value.
- For the references that are instruction operands, the calculated values after loading do not always appear to be equal.
  - because the operand is a Target address to an operation that is PC relative in most cases.
  - Example REF1 is a label in PROGA and operand LISTA is @ 4040 (relative to PC) loaded in A.

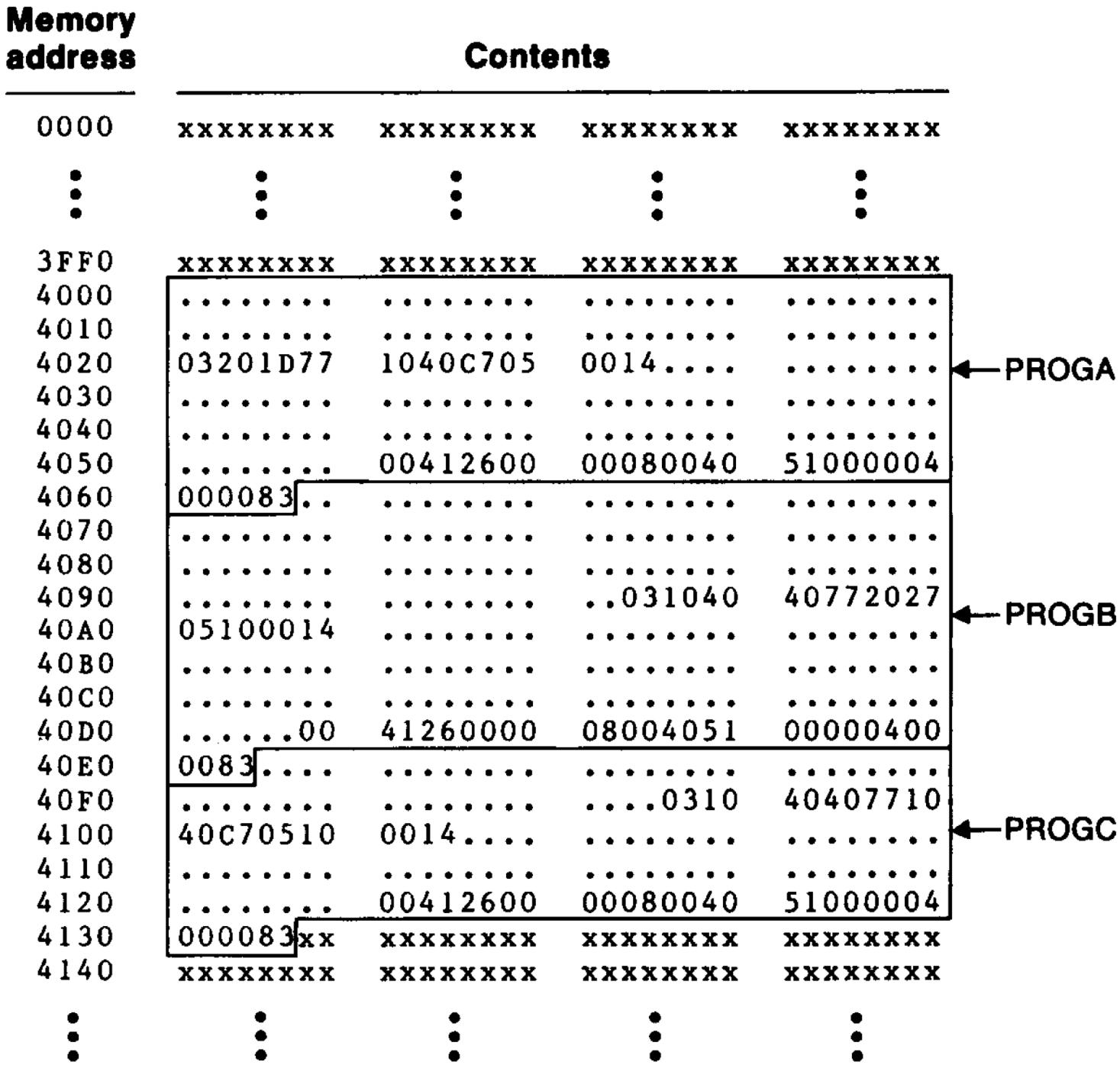


Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

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	

Program Linking Example

		Program A	Program B	Program C
Label	Expression	LISTA, ENDA	LISTB, ENDB	LISTC, ENDC
REF1	LISTA	local, R, PC	external	external
REF2	LISTB+4	external	local, R, PC	external
REF3	ENDA-LISTA	local, A	external	external
REF4	ENDA-LISTA+LISTC	local, A	external	local, R
REF5	ENDC-LISTC-10	external	external	local, A
REF6	ENDC-LISTC+LISTA-1	local, R	external	local, A
REF7	ENDA-LISTA-(ENDB-LISTB)	local, A	local, A	external
REF8	LISTB-LISTA	local, R	local, R	external

Ref No.	Symbol	Address
1	PROGA	4000
2	LISTB	40C3
3	ENDB	40D3
4	LISTC	4112
5	ENDC	4124

Ref No.	Symbol	Address
1	PROGB	4063
2	LISTA	4040
3	ENDA	4054
4	LISTC	4112
5	ENDC	4124

Ref No.	Symbol	Address
1	PROGC	40E2
2	LISTA	4040
3	ENDA	4054
4	LISTB	40C3
5	ENDB	40D3

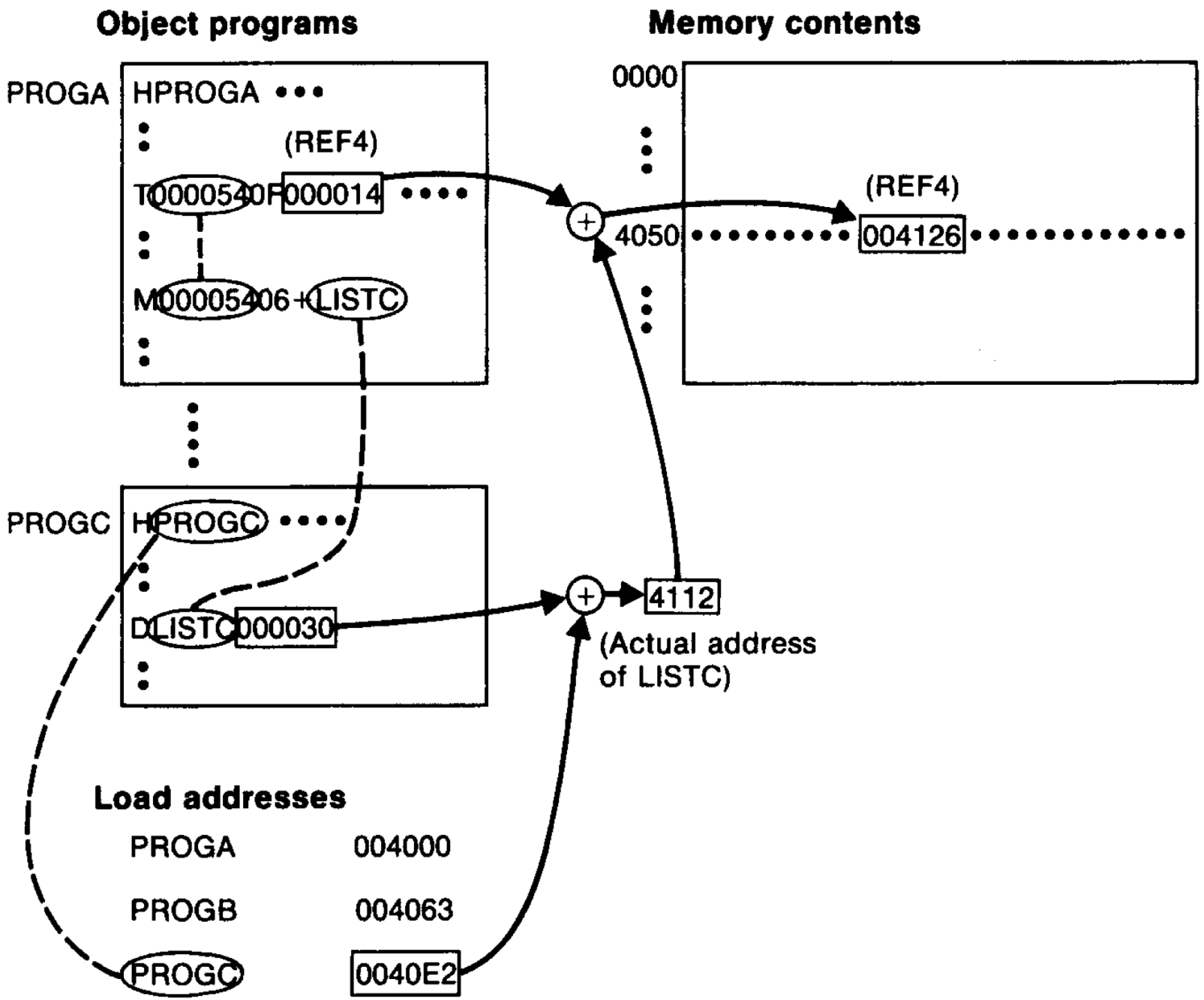


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

3.2.3 Algorithm and Data Structure for a Linking Loader

- A linking loader usually makes two passes
  - Pass 1 assigns addresses to all external symbols.
  - Pass 2 performs the actual loading, relocation, and linking.
  - The main data structure is ESTAB (hashing table).

3.2.3 Algorithm and Data Structure for a Linking Loader

- A linking loader usually makes two passes
  - ESTAB is used to store the name and address of each external symbol in the set of control sections being loaded.
  - Two variables PROGADDR and CSADDR.
  - PROGADDR is the beginning address in memory where the linked program is to be loaded.
  - CSADDR contains the starting address assigned to the control section currently being scanned by the loader.

3.2.3 Algorithm and Data Structure for a Linking Loader

- The linking loader algorithm, Fig 3.11(a) & (b).
  - In Pass 1, concerned only Header and Defined records.
  - CSADDR+CSLTH = the next CSADDR.
  - A load map is generated.
  - In Pass 2, as each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).
  - When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.
  - This value is then added to or subtracted from the indicated location in memory.

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}
```

Figure 3.11(a) Algorithm for Pass 1 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}
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.

3.3 Machine-Independent Loader Features
3.3.1 Automatic Library Search

- Many linking loaders
  - Can automatically incorporate routines form a subprogram library into the program being loaded.
  - A standard system library
  - The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program, and loaded.

3.3.1 Automatic Library Search

- Automatic library call
  - At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.
  - The loader searches the library

3.3.2 Loader Options

- Many loaders allow the user to specify options that modify the standard processing.
  - Special command
  - Separate file
  - INCLUDE program-name(library-name)
  - DELETE csect-name
  - CHANGE name1, name2

```
INCLUDE     READ(UTLIB)
INCLUDE     WRITE(UTLIB)
DELETE      RDREC, WRREC
CHANGE      RDREC, READ
CHANGE      WRREC, WRITE
LIBRARY     MYLIB
NOCALL      STDEV, PLOT, CORREL
```

3.4 Loader Design Options
3.4.1 Linkage Editors

- Fig 3.13 shows the difference between linking loader and linkage editor.
  - The source program is first assembled or compiled, producing an OP.
- Linking loader
  - A linking loader performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.

» The essential difference between a linkage editor and a linking loader

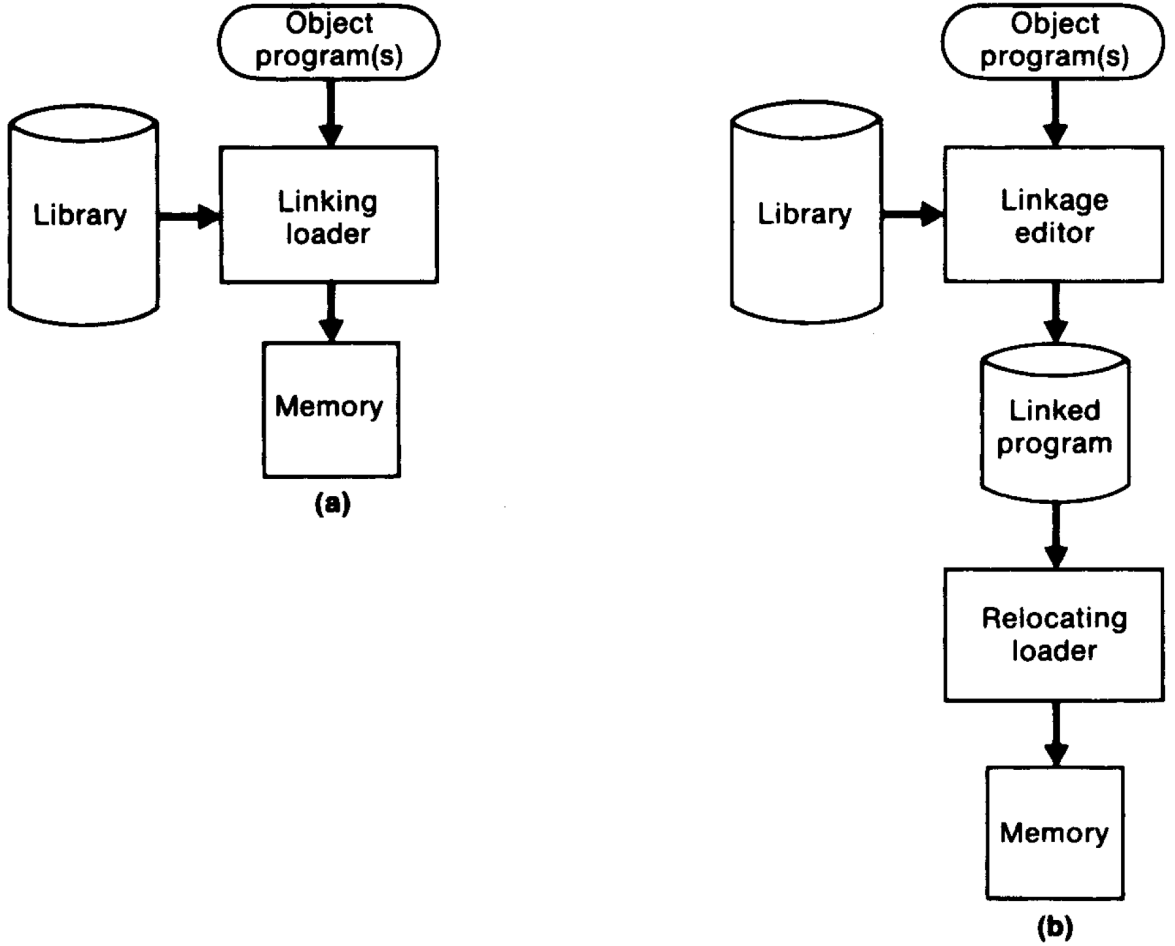


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

3.4.1 Linkage Editors

- Linkage editor
  - A linkage editor 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.
  - The LE performs relocation of all control sections relative to the start of the linked program.

3.4.1 Linkage Editors

- All items that need to be modified at load time have values that are relative to the start of the linked program.
- If a program is to be executed many times without being reassembled, the use of a LE substantially reduces the overhead required.
- LE can perform many useful functions besides simply preparing an OP for execution.

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

INCLUDE    READR(FTNLIB)
INCLUDE    WRITER(FTNLIB)

INCLUDE   BLOCK(FTNLIB)
INCLUDE   DEBLOCK(FTNLIB)
INCLUDE   ENCODE(FTNLIB)
INCLUDE   DECODE(FTNLIB)
.
.
.
SAVE      FTNIO(SUBLIB)
```

3.4.2 Dynamic Linking

- Linking loaders perform these same operations at load time.
- Linkage editors perform linking operations before the program is loaded for execution.

3.4.2 Dynamic Linking

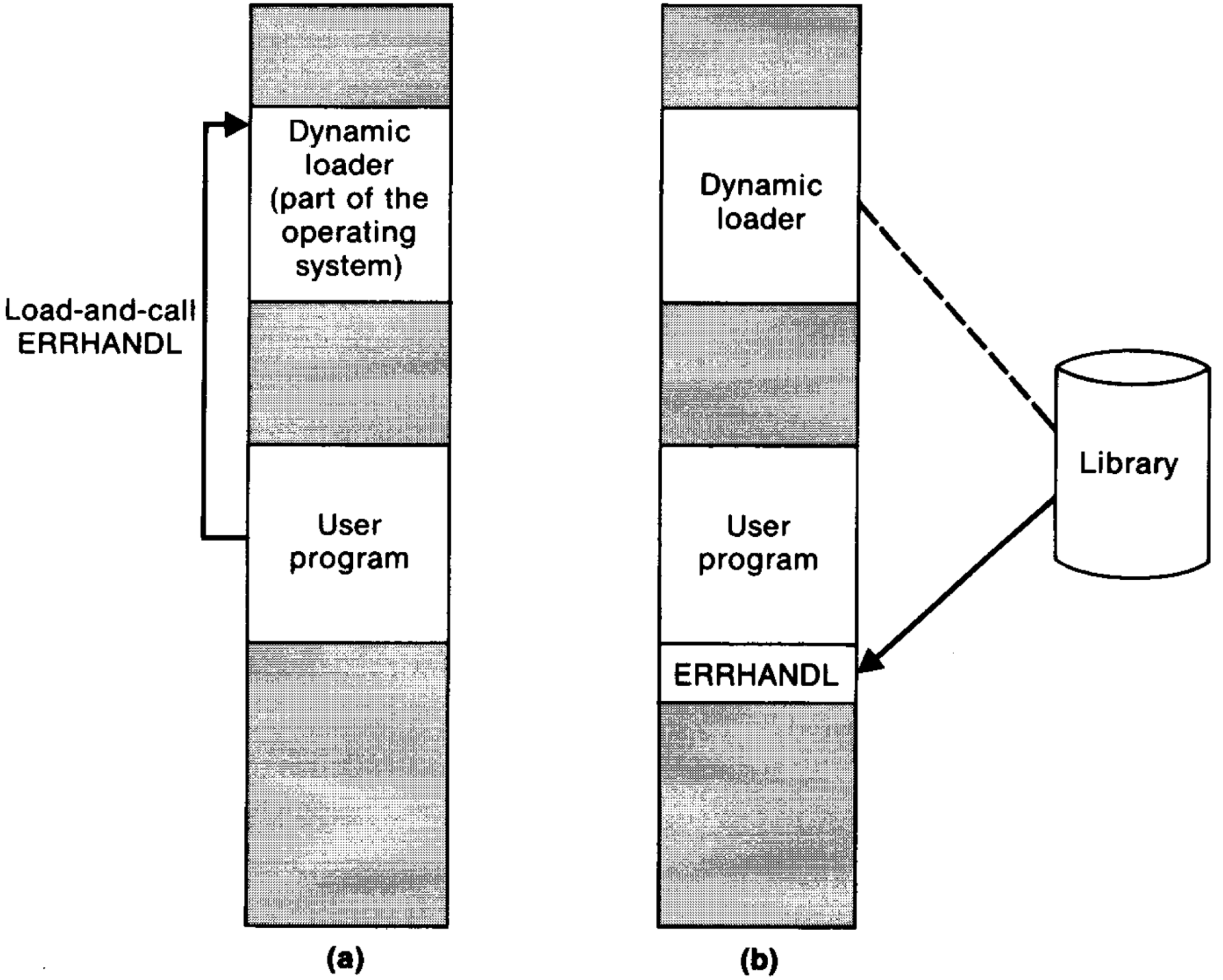
- Dynamic linking (dynamic loading, load on call)
  - Postpones the linking function until execution time.
  - A subroutine is loaded and linked to the rest of the program when is first loaded.
  - Dynamic linking is often used to allow several executing program to share one copy of a subroutine or library.
    - Run-time library (C language), dynamic link library
    - A single copy of the routines in this library could be loaded into the memory of the computer.

3.4.2 Dynamic Linking

- Dynamic linking provides the ability to load the routines only when (and if) they are needed.
  - For example, that a program contains subroutines that correct or clearly diagnose error in the input data during execution.
  - If such error are rare, the correction and diagnostic routines may not be used at all during most execution of the program.
  - However, if the program were completely linked before execution, these subroutines need to be loaded and linked every time.

3.4.2 Dynamic Linking

- Dynamic linking avoids the necessity of loading the entire library for each execution.
- Fig. 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an operating system (OS) service request.


Dynamic loader (part of the operating system)
Load-and-call ERRHANDL
User program
(a)
Dynamic loader
User program
ERRHANDL
Library
(b)

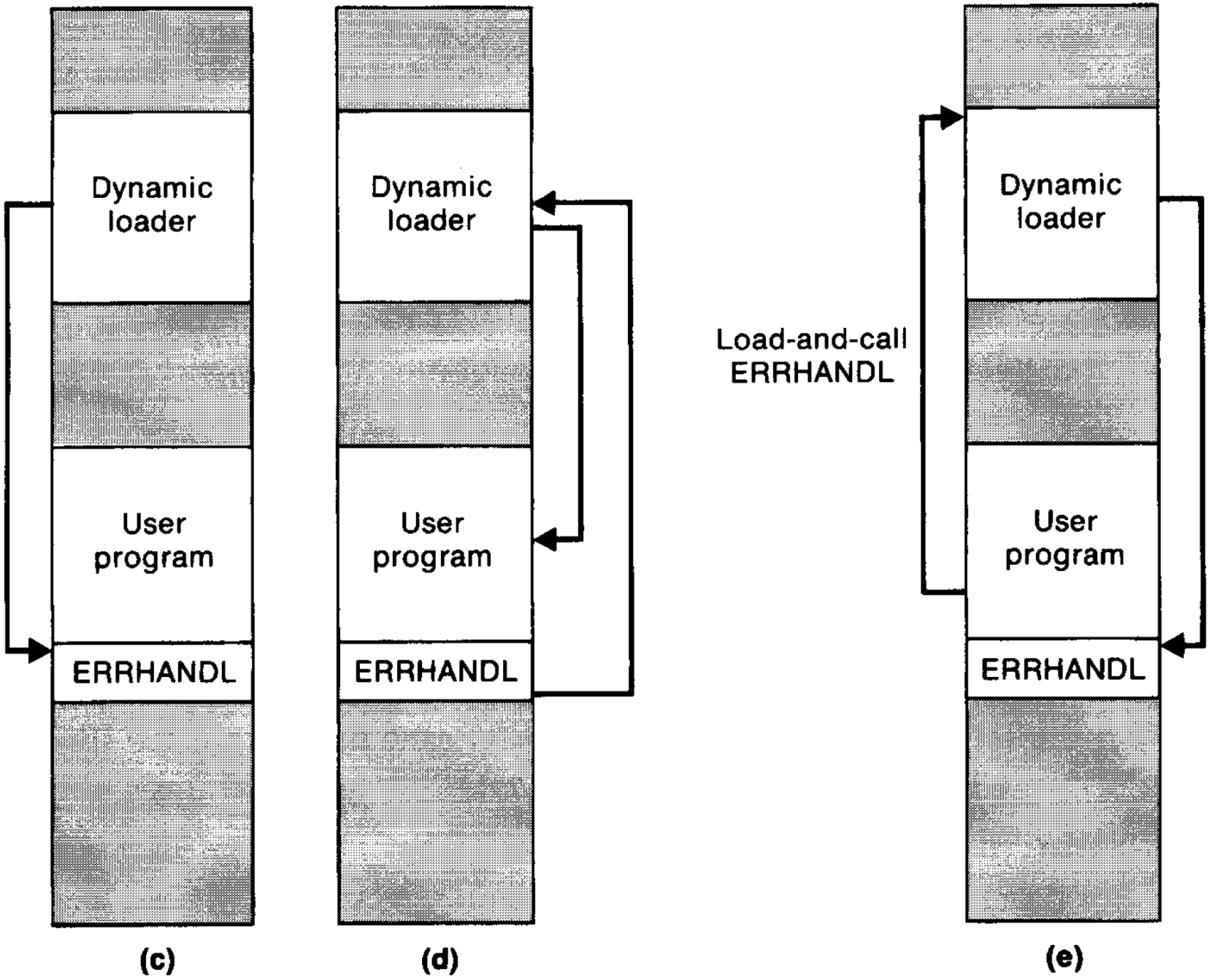


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

3.4.2 Dynamic Linking

- The program makes a load-on-call service request to OS. The parameter of this request is the symbolic name of the routine to be loaded.
- OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded form the specified user or system libraries.
- Control id then passed form OS to the routine being called.
- When the called subroutine completes its processing, OS then returns control to the program that issued the request.
- If a subroutine is still in memory, a second call to it may not require another load operation.

3.4.3 Bootstrap Loaders

- An absolute loader program is permanently resident in a read-only memory (ROM)
  - Hardware signal occurs
- The program is executed directly in the ROM
- The program is copied from ROM to main memory and executed there.

3.4.3 Bootstrap Loaders

- Bootstrap and bootstrap loader
  - Reads a fixed-length record form some device into memory at a fixed location.
  - After the read operation is complete, control is automatically transferred to the address in memory.
  - If the loading process requires more instructions than can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of more records.