UNIT III
LOADERS AND LINKERS

This Unit gives you…

Basic Loader Functions

Machine-Dependent Loader Features

Machine-Independent Loader Feature

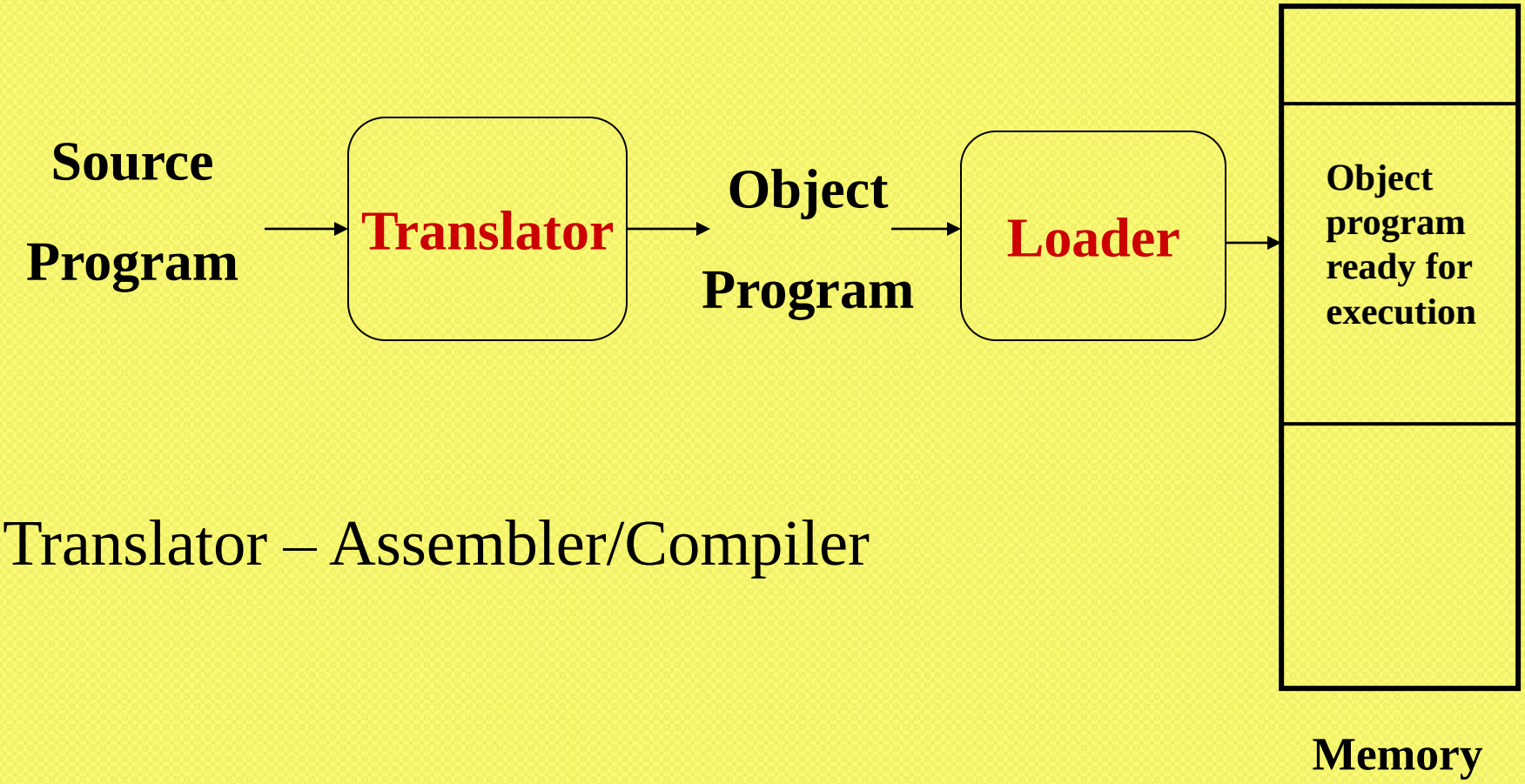
Loader Design Options

Implementation Examples

BASIC DEFINITION

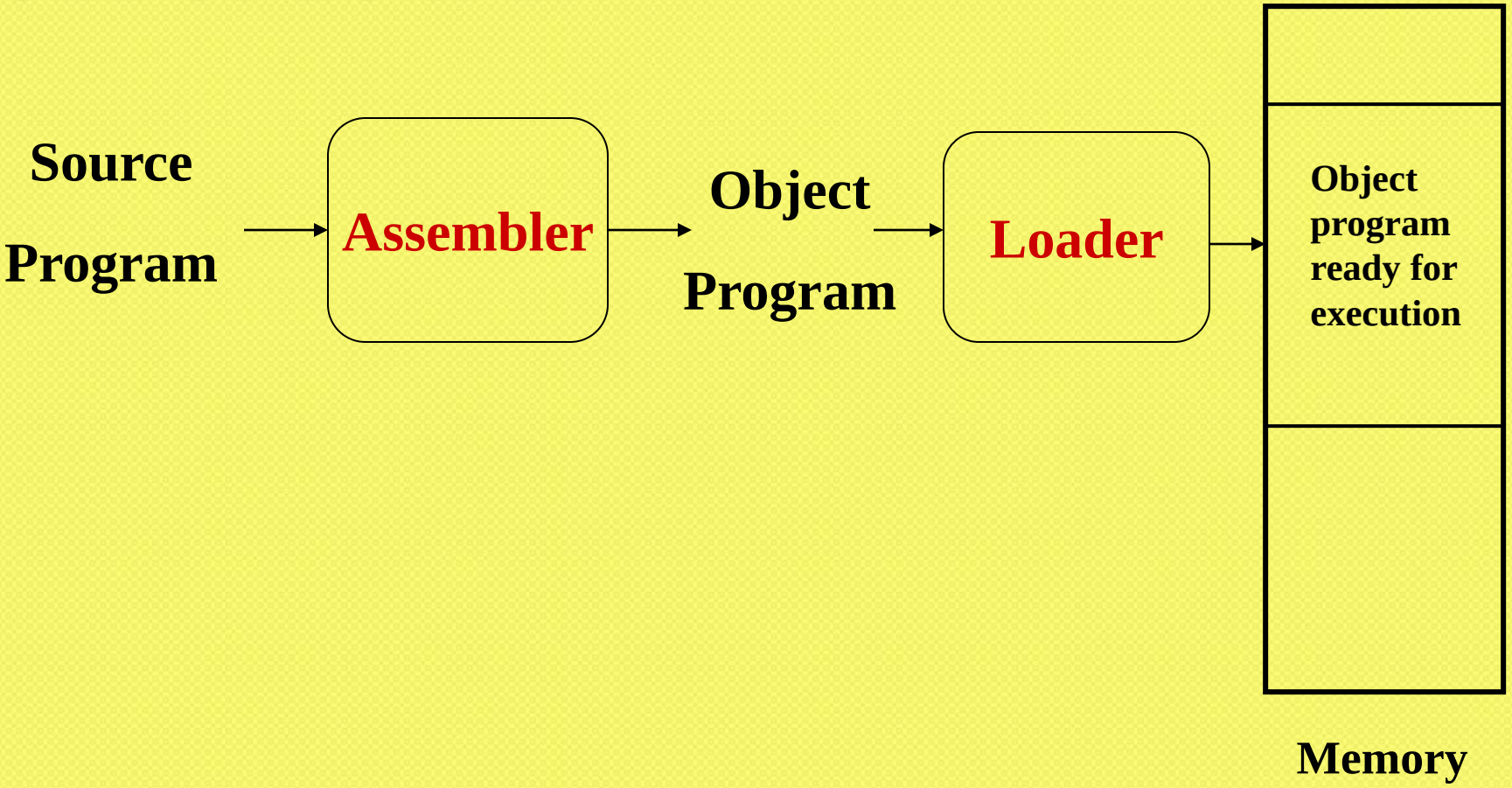
- **Loading** - which allocates memory location and brings the object program into memory for execution - (Loader)
- **Linking**- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)
- **Relocation** - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

ROLE OF LOADER

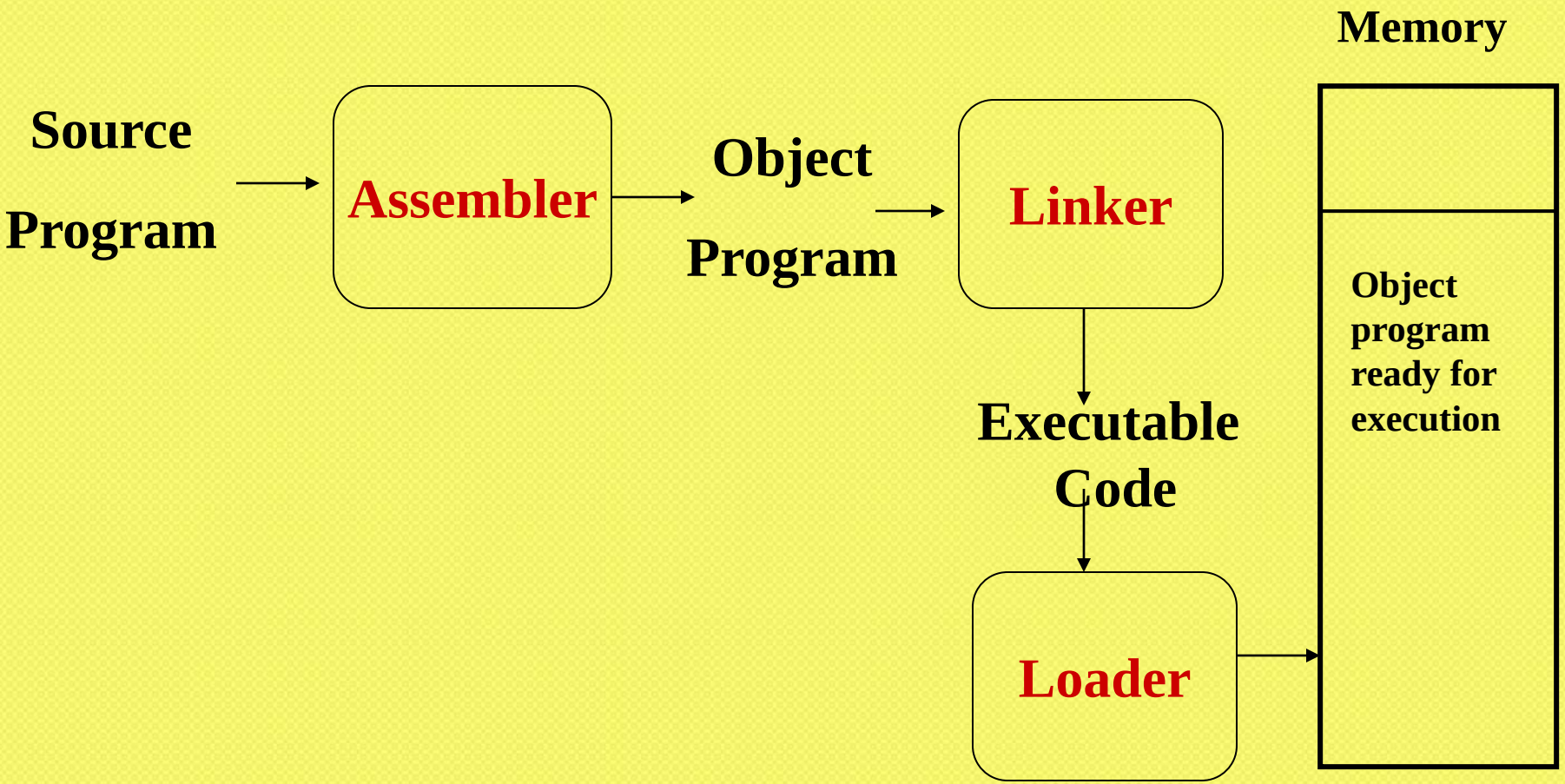


Translator – Assembler/Compiler

ROLE OF LOADER



ROLE OF LOADER AND LINKER



WE KNOW…

- Source Program – Assembly Language
- Object Program - From assembler
  - Contains translated instructions and data values from the source program
- Executable Code - From Linker
- Loader - Loads the executable code to the specified memory locations and code gets executed.

WE NEED…THREE PROCESSES

- Loading - which allocates memory location and brings the object program into memory for execution - Loader
- Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - Linker
- Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - Linking Loader

BASIC LOADER FUNCTIONS

- A Loader is a system program that performs the loading function

- It brings object program into memory and starts its execution

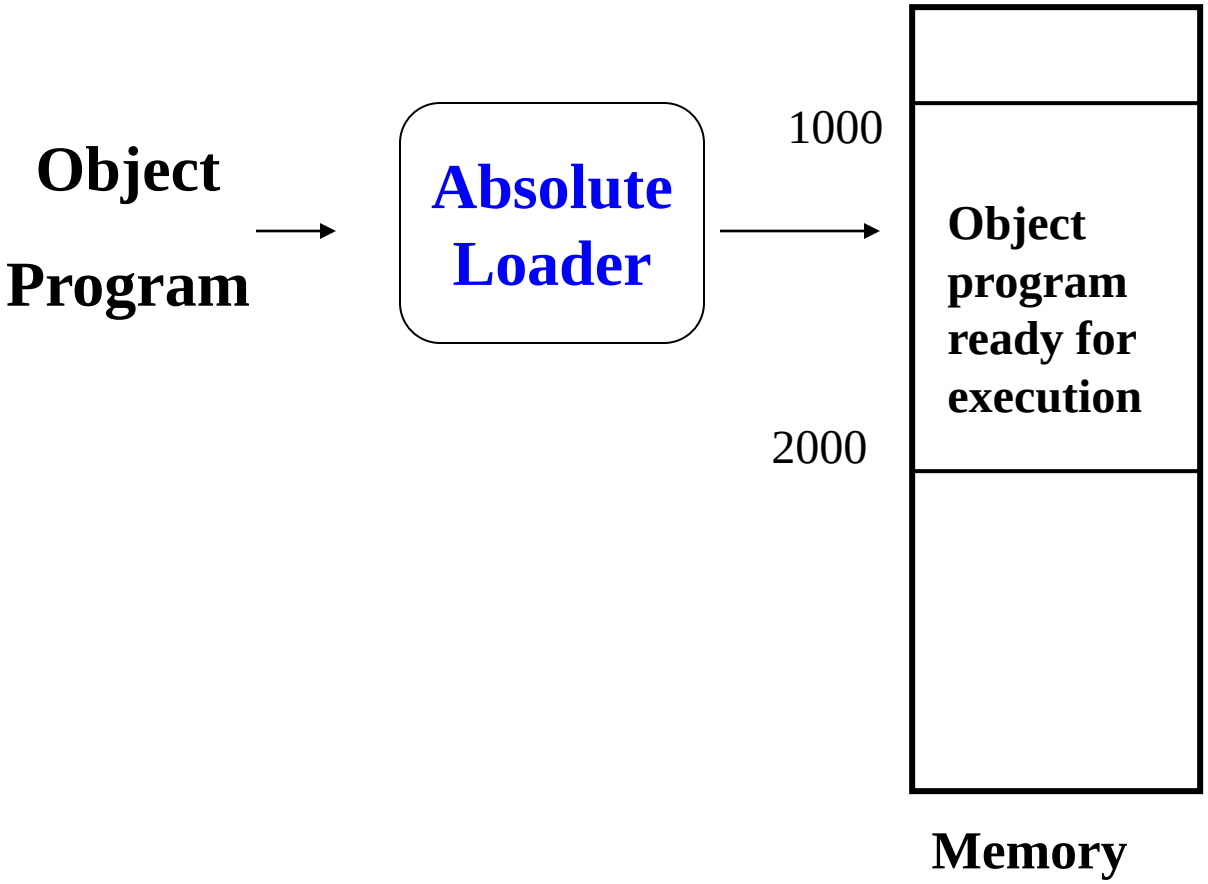
TYPE OF LOADERS

- absolute loader
- bootstrap loader
- relocating loader (relative loader)
- direct linking loader

ABSOLUTE LOADER

- Operation is very simple
- The object code is loaded to specified locations in the memory
- At the end the loader jumps to the specified address to begin execution of the loaded program

ROLE OF ABSOLUTE LOADER



ABSOLUTE LOADER

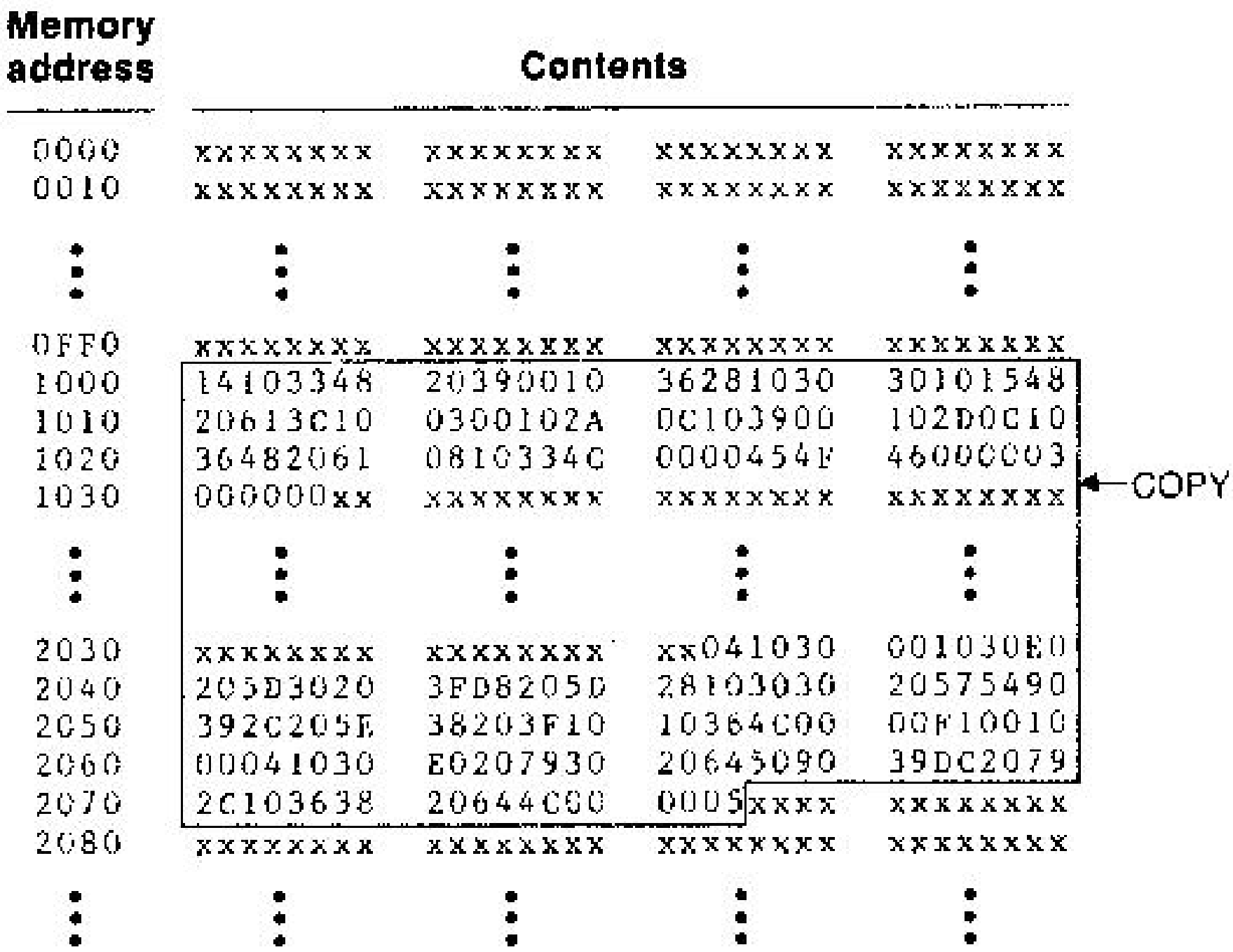
- Advantage
  - Simple and efficient
- Disadvantage
  - the need for programmer to specify the actual address
  - difficult to use subroutine libraries
- We have algorithm – next slide

Begin
read Header record
verify program name and length
read first Text record
while record type is <> ‘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

OBJECT PROGRAM

```
HCOPY  000000001007A
T00100001E14103348203900103628103030101548206130100300102A0C1039001020
T00101E150C10364820610810334C0000454F46000003000000
T00203918041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T00205710101364C0000F1001000041030E02079302064509039DC2079 2C1036
T00207307382064 4C000005
E001000
```

(a) Object program



(b) Program loaded in memory

OBJECT CODE REPRESENTATION

- Each byte of assembled code is given using its hexadecimal representation in character form
- Easy to read by human beings
- Each byte of object code is stored as a single byte
- Most machine store object programs in a binary form
- We must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters

A SIMPLE BOOTSTRAP LOADER

- When a computer is first tuned on or restarted, a special type of absolute loader, called bootstrap loader is executed
- This bootstrap loads the first program to be run by the computer -- usually an operating system

EXAMPLE (SIC BOOTSTRAP LOADER)

- The bootstrap itself begins at address 0
- It loads the OS starting address 0x80
- No header record or control information, the object code is consecutive bytes of memory

Begin

X=0x80 (the address of the next memory location to be loaded

Loop

A←GETC (and convert it from the ASCII character code to the value of the hexadecimal digit)

save the value in the high-order 4 bits of S

A←GETC

combine the value to form one byte A← (A+S)

store the value (in A) to the address in register X

X←X+1

End

SUBROUTINE GETC

```
GETC    A←read one character
            if A=0x04 then jump to 0x80
            if A<48 then GETC
            A ← A-48 (0x30)
            if A<10 then return
            A ← A-7
            return
```

MACHINE-DEPENDENT LOADER FEATURES

Absolute Loader – Simple and efficient

Disadvantage is – programmer has to specify the starting address

One program to run – no problem – not for several

Difficult to use subroutine libraries efficiently

RELOCATION

Execution of the object program using any part of the available and sufficient memory

The object program is loaded into memory wherever there is room for it

The actual starting address of the object program is not known until load time

RELOCATING LOADERS

- Efficient sharing of the machine with larger memory and when several independent programs are to be run together
- Support the use of subroutine libraries efficiently

METHODS FOR SPECIFYING RELOCATION

- Use of modification record Refer Figure
- Use of relocation bit Refer Figure
  - Each instruction is associated with one relocation bit
  - These relocation bits in a Text record is gathered into bit masks

MODIFICATION RECORD

Modification record
- col 1: M
- col 2-7: relocation address
- col 8-9: length (halfbyte)
- col 10: flag (+/-)
- col 11-17: segment name

- For complex machines
- Also called RLD specification
  - Relocation and Linkage Directory

H^COPY ^000000 001077

T^000000
^1D^17202D^69202D^48101036^…^4B105D^3F2FEC^032010

T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F4
6

T^001035
^1D^B410^B400^B440^75101000^…^332008^57C003^B850

T^001053^1D^3B2FEA^134000^4F0000^F1^..^53C003^DF2008^B8
50

T^00070^07^3B2FEF^4F0000^05

M^000007^05+COPY

M^000014^05+COPY

M^000027^05+COPY

E^000000

Figure

Object program with relocation by Modification records

RELOCATION BIT

- For simple machines
- Relocation bit
  - 0: no modification is necessary
  - 1: modification is needed

RELOCATION BIT

- For simple machines
- Relocation bit
  - 0: no modification is necessary
  - 1: modification is needed

Twelve-bit mask is used in each Text record - col:10-12 – relocation bits

- since each text record contains less than 12 words

- unused words are set to 0

- any value that is to be modified during relocation must coincide with one of these 3-byte segments - line 210

H^COPY ^000000 00107A

T^000000^1E^FFC^140033^481039^000036^280030^300015^…^3C0003^…

T^00001E^15^E00^0C0036^481061^080033^4C0000^…^000003^000000

T^001039^1E^FFC^040030^000030^…^30103F^D8105D^280030^...

T^001057^0A^ 800^100036^4C0000^F1^001000

T^001061^19^FE0^040030^E01079^…^508039^DC1079^2C0036^...

E^000000

Object program with relocation by bit mask

- FFC - all ten words are to be modified
- E00 - first three records are to be modified

PROGRAM LINKING

- Goal - Resolve the problems with EXTREF and EXTDEF from different control sections
- Example - Program in Fig. 3.8 and object code in Figure (Refer)
- Use modification records for both relocation and linking
  - - address constant
  - - external reference

EXTDEF (EXTERNAL DEFINITION

- The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this control section and may be used by other sections
- **Ex: EXTDEF BUFFER, BUFFEND, LENGTH (Refer Figure)**
- **EXTDEF LISTA, ENDA (Refer Figure)**

EXTREF (EXTERNAL REFERENCE)

- The EXTREF statement names symbols used in this (present) control section and are defined elsewhere
  - **Ex: EXTREF RDREC, WRREC (Refer Figure) EXTREF LISTB, ENDB, LISTC, ENDC (Refer Figure)**

HOW TO IMPLEMENT THESE…

- The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of
- **- Define record**
- **- Refer record**

DEFINE RECORD

- Col. 1 D
- Col. 2-7Name of external symbol defined in this control section
- Col. 8-13 Relative address within this control section (hexadecimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols
  - **- D LISTA 000040 ENDA 000054**
  - **- D LISTB 000060 ENDB 000070**

REFER RECORD

- Col. 1 R
- Col. 2-7 Name of external symbol referred to in this control section
- Col. 8-73 Name of other external reference symbols

R LISTB ENDB LISTC ENDC

R LISTA ENDA LISTC ENDC

R LISTA ENDA LISTB ENDB

```
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
```

```
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
```

```
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
```

```
H PROGA 000000 000063
D LISTA    000040 ENDA    000054
R LISTB    ENDB   LISTC   ENDC
.
.
T 000020 0A 03201D 77100004 050014
.
.
T 000054 0F 000014 FFFF6 00003F 000014 FFFFC0
M000024 05+LISTB
M000054 06+LISTC
M000057 06+ENDC
M000057 06 -LISTC
M00005A06+ENDC
M00005A06 -LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

```
H PROGB 000000 00007F
D LISTB    000060 ENDB    000070
R LISTA    ENDA   LISTC   ENDC
.
.
T 000036 0B 03100000 772027 05100000
.
.
T 000007 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037 05+LISTA
M00003E 06+ENDA
M00003E 06 -LISTA
M000070 06 +ENDA
M000070 06 -LISTA
M000070 06 +LISTC
M000073 06 +ENDC
M000073 06 -LISTC
M000073 06 +ENDC
M000076 06 -LISTC
M000076 06+LISTA
```

```
H PROGC 000000 000051
D LISTC    000030 ENDC    000042
R LISTA    ENDA   LISTB   ENDB
.
T 000018 0C 03100000 77100004 05100000
.
T 000042 0F 000030 000008 000011 000000 000000
M000019 05+LISTA
M00001D 06+LISTB
M000021 06+ENDA
M000021 06 -LISTA
M000042 06+ENDA
M000042 06 -LISTA
M000042 06+PROGC
M000048 06+LISTA
M00004B 06+ENDA
M00004B 006-LISTA
M00004B 06-ENDB
M00004B 06+LISTB
M00004E 06+LISTB
M00004E 06-LISTA
E
```

Program Linking Example

- Refer Figure ***3.5.2***
- Load address for control sections
  - PROGA 004000 63
  - PROGB 004063 7F
  - PROGC 0040E2 51

- Load address for symbols
  - LISTA: PROGA+0040=4040
  - LISTB: PROGB+0060=40C3
  - LISTC: PROGC+0030=4112

- REF4 in PROGA

ENDA-LISTA+LISTC=14+4112=4126

(ENDA-LISTA = 14 (4054-4040)

T0000540F000014FFFFF600003F000014FFFFC0

M00005406+LISTC

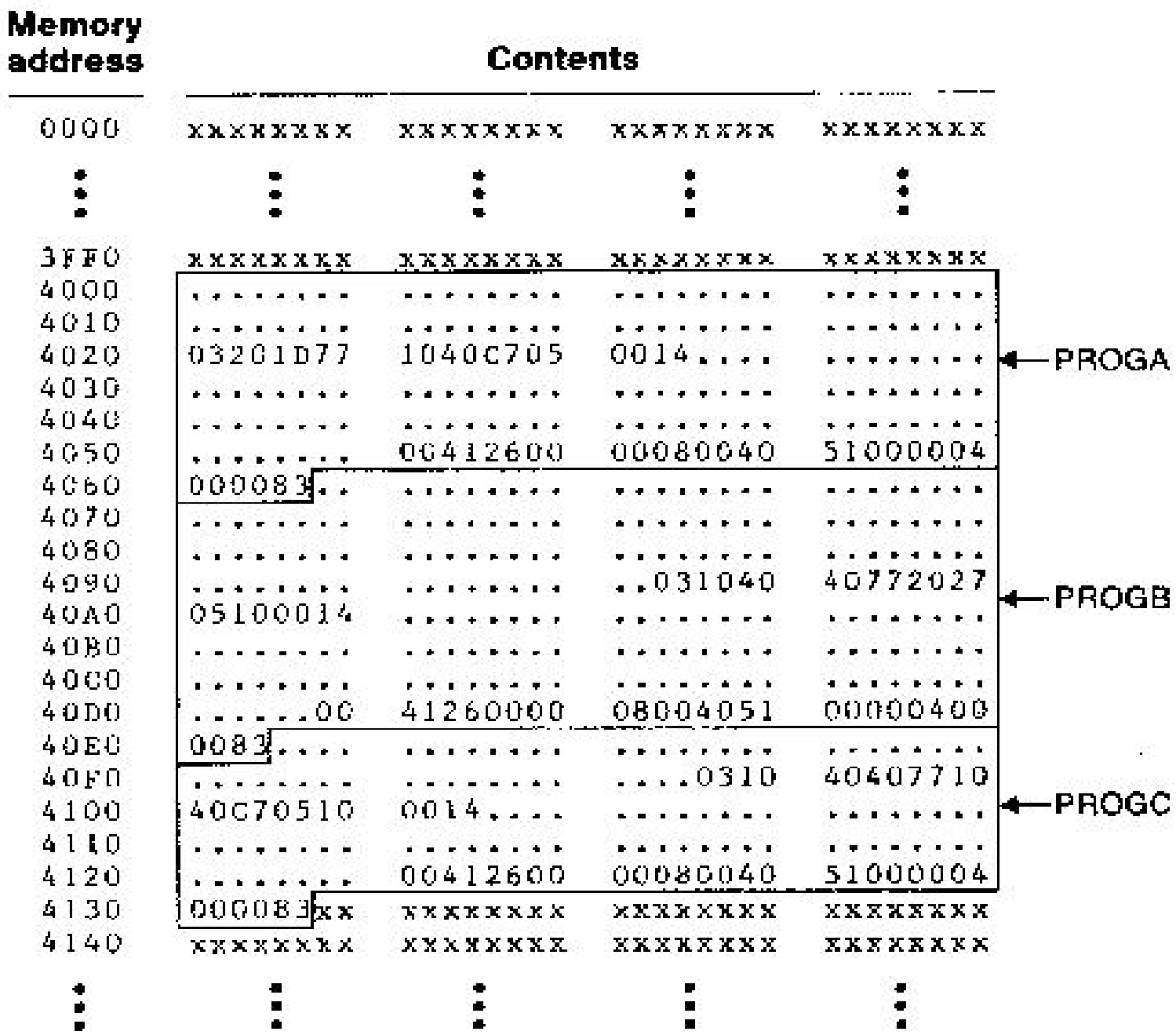


Figure 3.5.1: Programs form the above figure after linking and loading

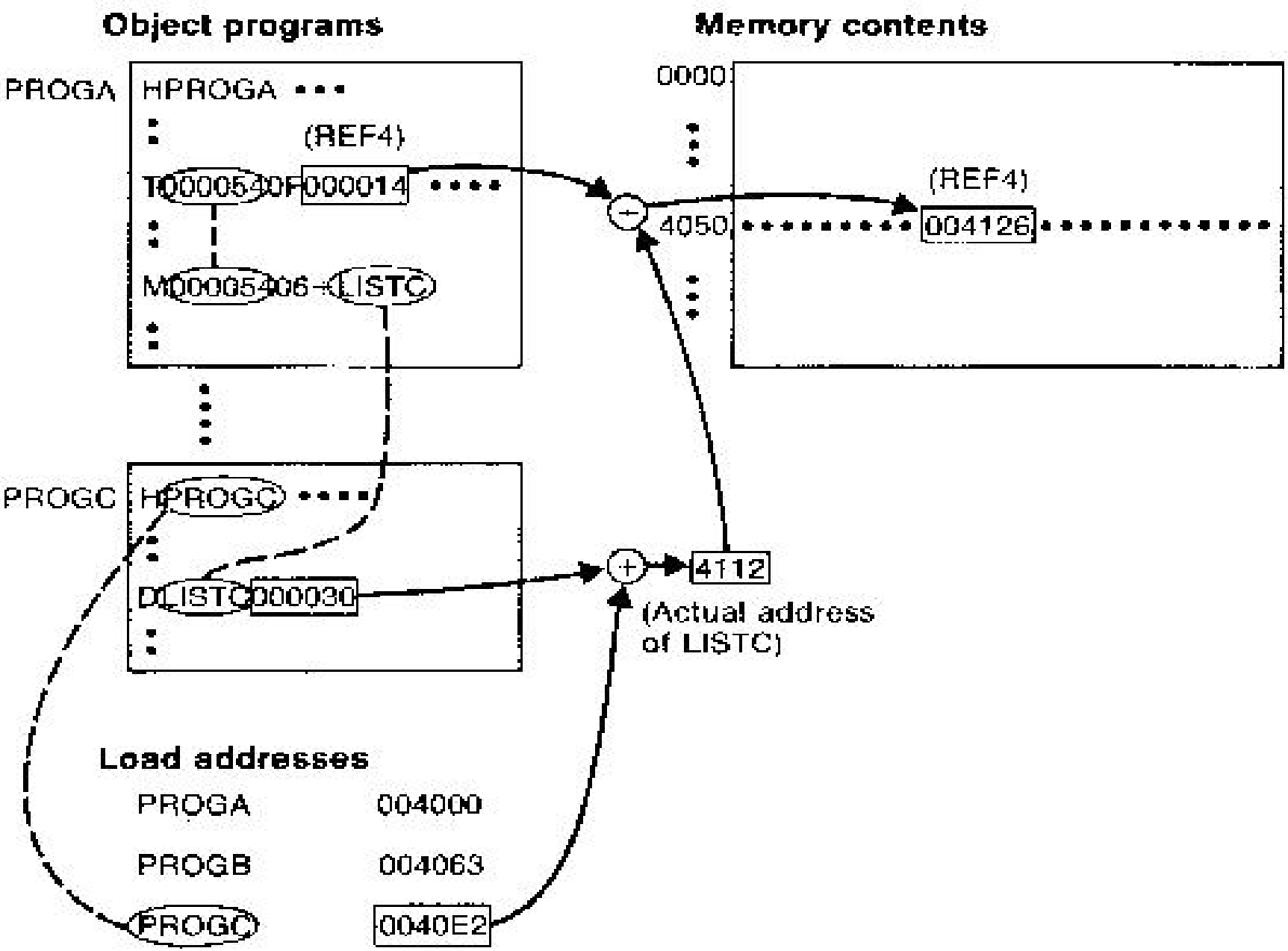


Figure 3.5.2: *relocation and linking operations performed on REF4 from PROGA*

ALGORITHM AND DATA STRUCTURES FOR A LINKING LOADER

- Linking Loader uses two-passes logic
- - Pass 1: assign addresses to all external symbols
- - Pass 2: perform the actual loading, relocation, and linking
- ESTAB (external symbol table) – main data structure for a linking loader

ESTAB FOR THE EXAMPLE GIVEN

Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

PROGRAM LOGIC FOR PASS 1

- Pass 1:
  - - Assign addresses to all external symbols
- Variables & Data structures
- - PROGADDR (program load address) from OS
- - CSADDR (control section address)
- - CSLTH (control section length
- - ESTAB
- Refer Algorithm for Pass 1 of LL in Fig. 3.11(a)
  - - Process Define Record

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}
```

PROGRAM LOGIC FOR PASS 2

- Pass 2:
  - - Perform the actual loading, relocation, and linking
- Modification record
  - - Lookup the symbol in ESTAB
- End record for a main program
  - - Transfer address
- Refer Algorithm for Pass 2 of LL in Fig. 3.11(b)
  - - Process Text record and Modification record

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}
```

IMPROVE EFFICIENCY, HOW?

- Use of local searching instead of multiple searches of ESTAB for the same symbol
  - **- assign a reference number to each external symbol**
  - **- the reference number is used in Modification records**
- Implementation
  - **- 01: control section name**
  - **- other: external reference symbols**
- Example - **Refer Figure**

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
.
.
T000029000320ED7710000405001 4
.
.
T000054000C014FFFF60000300001 4FFFC0
M000024 05+02
M000054 06+04
M000057 06+05
M000057 06-04
```

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
.
.
T000036000B031000007720205100000
.
.
T000070000000000FFFFF6FFFFFFFFFF0000060
M000037005+02
M00003E005+03
M00003E005-02
M000070006+03
M000070006-02
M000070006+04
M000073006+05
M000073006-04
M000076006+05
M000076006-04
M000076006+02
M000079006+03
M000079006-02
M00007C006+01
M00007C006-02
E
```

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
*
*
T000018C03100000771000040510000
*
*
T000042F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E
```

SYMBOL AND ADDRESSES IN PROGA

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

PROGA

SYMBOL AND ADDRESSES IN PROGB

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

PROGB

SYMBOL AND ADDRESSES IN PROGC

Ref No.	Symbol	Address
1	PROGC	4063
2	LISTA	4040
3	ENDA	4054
4	LISTB	40C3
5	ENDB	40D3

PROGC

ADVANTAGE OF REFERENCE-NUMBER

The main advantage of reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section.

MACHINE-INDEPENDENT LOADER FEATURES

- Features that are not directly related to machine architecture and design
- - Automatic Library Search
- - Loader Options

AUTOMATIC LIBRARY SEARCH

- This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded
- The routines are automatically retrieved from a library as they are needed during linking

IMPLEMENTATION

- Allows programmer to use subroutines from one or more libraries
- The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded
- The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program

LOADER OPTIONS

- Allow the user to specify options that modify the standard processing
- Specified using a command language
- Specified as a part of job control language that is processed by the operating system
- Specified using loader control statements in the source program.

EXAMPLE OPTIONS

- INCLUDE program-name (library-name) - read the designated object program from a library
- DELETE csect-name – delete the named control section from the set pf programs being loaded
- CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs
- LIBRARY MYLIB – search MYLIB library before standard libraries
- NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines

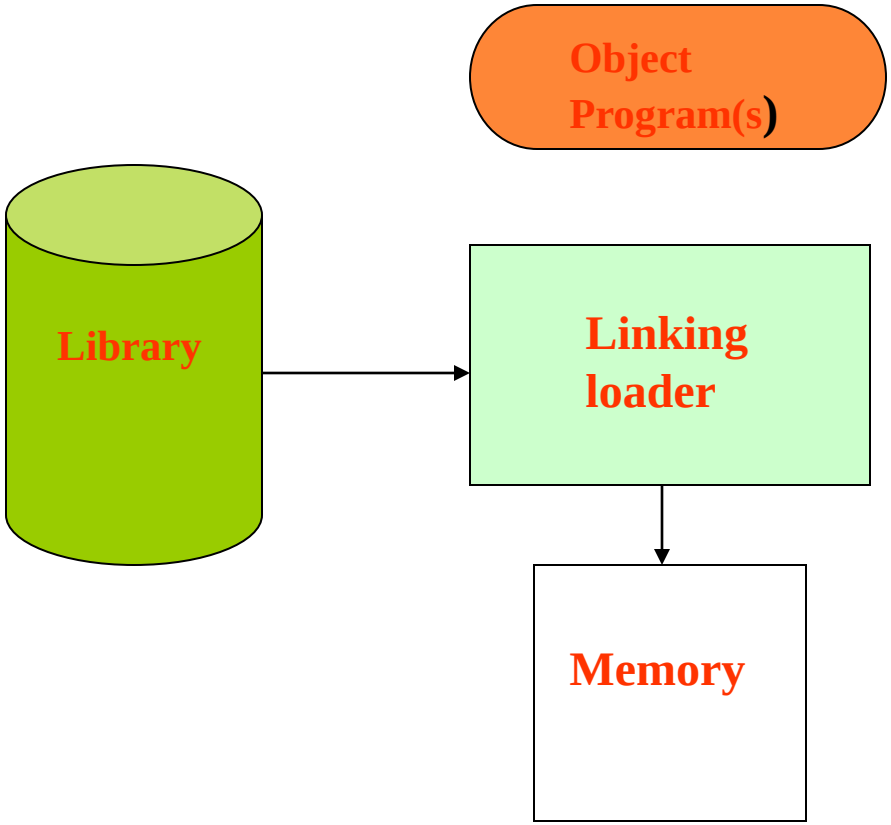
EXAMPLE

- LIBRARY UTLIB
- INCLUDE READ (UTLIB)
- INCLUDE WRITE (UTLIB)
- DELETE RDREC, WRREC
- CHANGE RDREC, READ
- CHANGE WRREC, WRITE
- NOCALL SQRT, PLOT

LOADER DESIGN OPTIONS

- Common alternatives for organizing the loading functions, including relocation and linking
- Linking Loaders – Perform all linking and relocation at load time
- Other Alternatives
- Linkage editors – Perform linking prior to load time
- Dynamic linking – Linking function is performed at execution time

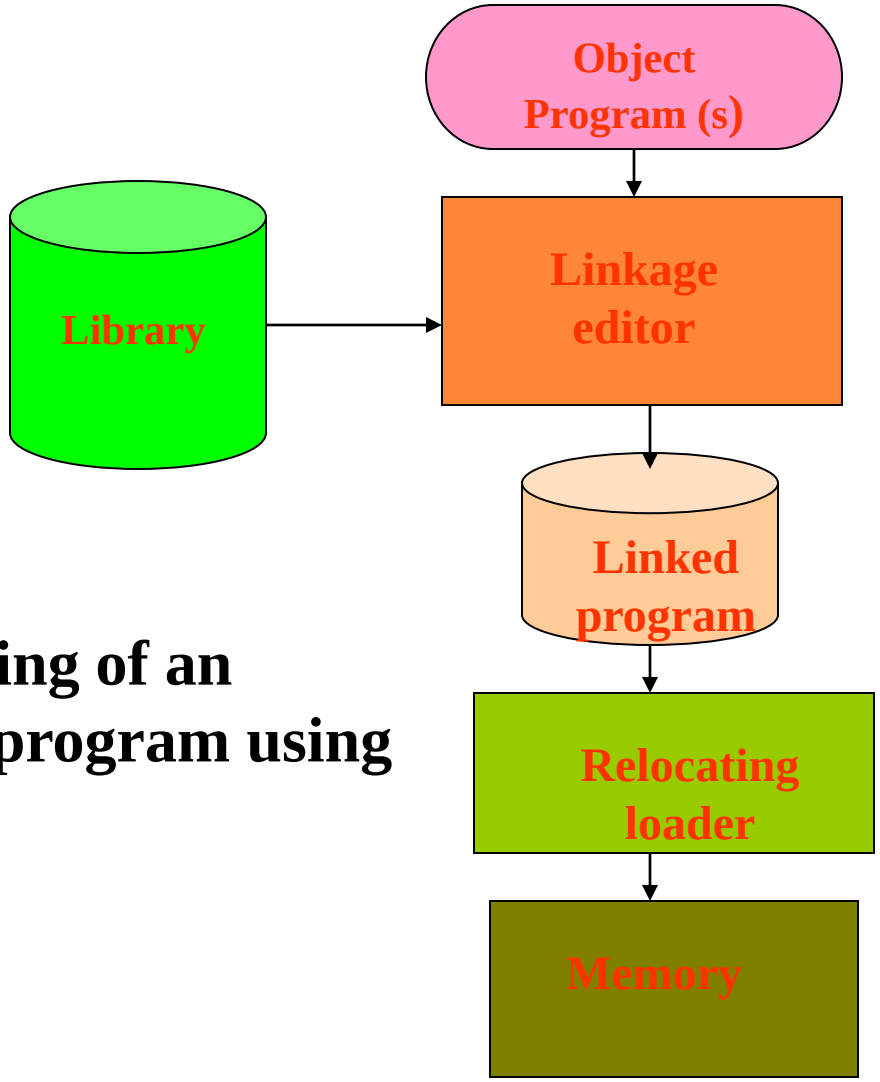
LINKING LOADERS



The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and loading operations, and loads the program into memory for execution

Processing of an Object program using LL

LINKAGE EDITORS



Processing of an Object program using LE

LINKAGE EDITORS

- A linkage editor, produces a linked version of the program – often called a load module or an executable image – which is written to a file or library for later execution
- The linked program produced is generally in a form that is suitable for processing by a relocating loader.

SOME USEFUL FUNCTIONS…

- An absolute object program can be created, if starting address is already known
- New versions of the library can be included without changing the source program
- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together
- Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space

DYNAMIC LINKING

- The scheme that postpones the linking function until execution
- A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call

ADVANTAGES…

- Allow several executing programs to share one copy of a subroutine or library
- In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs
- Dynamic linking provides the ability to load the routines only when (and if) they are needed
- The actual loading and linking can be accomplished using operating system service request – Refer Figure

BOOTSTRAP LOADERS

- How is the loader itself loaded into the memory ?
- When computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or A Bootstrap loader, which in turn loads OS and prepares it for execution.
- The first record (or records) is generally referred to as a bootstrap loader – makes the OS to be loaded
- Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system

IMPLEMENTATION EXAMPLES…

- Brief description of loaders and linkers for actual computers
- They are
- MS-DOS Linker - Pentium architecture
- SunOS Linkers - SPARC architecture
- Cray MPP Linkers – T3E architecture

MS-DOS LINKER

- Microsoft MS-DOS linker for Pentium and other x86 systems
- Most MS-DOS compilers and assemblers (MASM) produce object modules - .OBJ files
- MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program - .EXE file

MS-DOS OBJECT MODULE

Record Types	Description
THEADR	Translator Header
TYPDEF,PUBDEF, EXTDEF	External symbols and references
LNAMES, SEGDEF, GRPDEF	Segment definition and grouping
LEDATA, LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information
MODEND	End of object module

SUNOS LINKERS

- SunOS provides two different linkers – link-editor and run-time linker
- Link-editor is invoked in the process of assembling or compiling a program – produces a single output module – one of the following types (next slide)
- An object module contains one or more sections – representing instructions and data area from the source program, relocation and linking information, external symbol table

TYPES OF OBJECT MODULE

- A relocatable object module – suitable for further link-editing
- A static executable – with all symbolic references bound and ready to run
- A dynamic executable – in which some symbolic references may need to be bound at run time
- A shared object – which provides services that can be bound at run time to one ore more dynamic executables

RUN-TIME LINKER

- Uses dynamic linking approach
- Run-time linker binds dynamic executables and shared objects at execution time
- Performs relocation and linking operations to prepare the program for execution

CRAY MPP LINKER

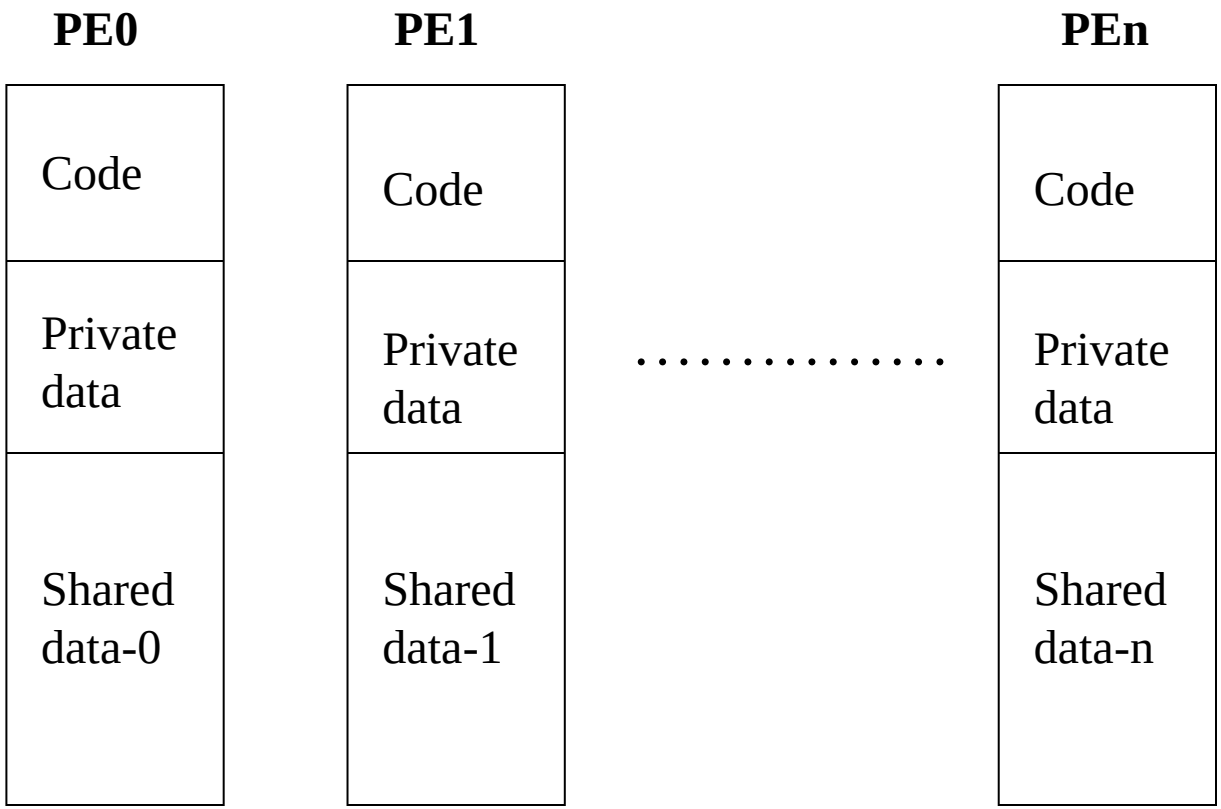
- Cray MPP (massively parallel processing) Linker
- T3E system contains large number of parallel processing elements (PEs) – Each PE has local memory and has access to remote memory (memory of other PEs)
- The processing is divided among PEs - contains shared data and private data
- The loaded program gets copy of the executable code, its private data and its portion of the shared data

Cray MPP (massively parallel processing) Linker

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)

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



T3E program loaded on multiple PEs

THIS CHAPTER GAVE YOU…

- Basic Loader Functions
- Machine-Dependent Loader Features
- Machine-Independent Loader Features
- Loader Design Options
- Implementation Examples

Thank you all