Loaders and Linkers

Jian-hua Yeh (葉建華)
真理大學資訊科學系助理教授
au4290@email.au.edu.tw
Introduction

• To execute an object program, we needs
  – **Relocation**, which modifies the object program so that it can be loaded at an address different from the location originally specified
  – **Linking**, which combines two or more separate object programs and supplies the information needed to allow references between them
  – **Loading and Allocation**, which allocates memory location and brings the object program into memory for execution
Overview of Chapter 3

• Type of loaders
  – assemble-and-go loader
  – absolute loader (bootstrap loader)
  – relocating loader (relative loader)
  – direct linking loader

• Design options
  – linkage editors
  – dynamic linking
  – bootstrap loaders
Assemble-and-go Loader

• Characteristic
  – the object code is stored in memory after assembly
  – single JUMP instruction

• Advantage
  – simple, developing environment

• Disadvantage
  – whenever the assembly program is to be executed, it has to be assembled again
  – programs have to be coded in the same language
Design of an Absolute Loader

• Absolute Program
  – Advantage
    • Simple and efficient
  – Disadvantage
    • the need for programmer to specify the actual address
    • difficult to use subroutine libraries

• Program Logic
Fig. 3.2 Algorithm for an absolute loader

Begin
read Header record
verify program name and length
read first Text record
while record type is not ‘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
(a) Object program
<table>
<thead>
<tr>
<th>Memory address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>0010</td>
<td>xxxxxxxx</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0FF0</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>1000</td>
<td>14103348</td>
</tr>
<tr>
<td>1010</td>
<td>20613C10</td>
</tr>
<tr>
<td>1020</td>
<td>36482061</td>
</tr>
<tr>
<td>1030</td>
<td>000000xx</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2030</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>2040</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>2050</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>2060</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>2070</td>
<td>xxxxxxxxx</td>
</tr>
<tr>
<td>2080</td>
<td>xxxxxxxxx</td>
</tr>
</tbody>
</table>

(b) Program loaded in memory
Object Code Representation

• Figure 3.1 (a)
  – each byte of assembled code is given using its hexadecimal representation in character form
  – easy to read by human beings

• In general
  – 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

- **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.
Fig. 3.3 SIC Bootstrap Loader Logic

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

0~9 : 30~39
A~F : 41~46

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
Relocating Loaders

• Motivation
  – 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

• Two methods for specifying relocation
  – modification record (Fig. 3.4, 3.5)
  – relocation bit (Fig. 3.6, 3.7)
    • each instruction is associated with one relocation bit
    • these relocation bits in a Text record is gathered into bit masks
Modification Record

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

<table>
<thead>
<tr>
<th>Modification record</th>
</tr>
</thead>
<tbody>
<tr>
<td>col 1: M</td>
</tr>
<tr>
<td>col 2-7: relocation address</td>
</tr>
<tr>
<td>col 8-9: length (halfbyte)</td>
</tr>
<tr>
<td>col 10: flag (+/-)</td>
</tr>
<tr>
<td>col 11-17: segment name</td>
</tr>
</tbody>
</table>
Fig. 3.5

H COPY 000000 001077
T 000000 1D 17202D 69202D 48101036 032026 ... 3F2FEC 032010
T 00001D 13 0F2016 010003 0F200D 4B10105D 3E2003 454F46
T 001035 1D B410 B400 B440 75101000 E32019 ... 57C003 B850
T 001053 1D 3B2FEA 134000 4F0000 F1 B410 ... DF2008 B850
T 00070 07 3B2FEF 4F0000 05
M 000007 05+COPY
M 000014 05+COPY
M 000027 05+COPY
E 000000
Relocation Bit

• For simple machines

• Relocation bit
  – 0: no modification is necessary
  – 1: modification is needed

<table>
<thead>
<tr>
<th>Text record</th>
</tr>
</thead>
<tbody>
<tr>
<td>col 1: T</td>
</tr>
<tr>
<td>col 2-7: starting address</td>
</tr>
<tr>
<td>col 8-9: length (byte)</td>
</tr>
<tr>
<td>col 10-12: relocation bits</td>
</tr>
<tr>
<td>col 13-72: object code</td>
</tr>
</tbody>
</table>

• Twelve-bit mask is used in each Text record
  – 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
Fig. 3-7

H COPY 000000 00107A
T 000000 1E FFC 140033 481039 000036 280030 300015 481061 ...
T 00001E 15 E00 0C0036 481061 080033 4C0000 454F46 000003 000000
T 001039 1E FFC 040030 000030 E0105D 30103F D8105D 280030 ...
T 001057 0A 800 100036 4C0000 F1 001000
T 001061 19 FE0 040030 E01079 301064 508039 DC1079 2C0036 ...
E 000000
Program Linking

• Goal
  – Resolve the problems with EXTREF and EXTDEF from different control sections (sec 2.3.5)

• 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
<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PROGA</td>
<td>START</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTDEF LISTA, ENDA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTREF LISTB, ENDB, LISTC, ENDC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td>REF1</td>
<td>LDA LISTA</td>
<td>003201D</td>
</tr>
<tr>
<td>0023</td>
<td>REF2</td>
<td>+LDT LISTB+4</td>
<td>77100004</td>
</tr>
<tr>
<td>0027</td>
<td>REF3</td>
<td>LDX #ENDA-LISTA</td>
<td>050014</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040</td>
<td>LISTA</td>
<td>EQU *</td>
<td></td>
</tr>
<tr>
<td>0054</td>
<td>ENDA</td>
<td>EQU *</td>
<td></td>
</tr>
<tr>
<td>0054</td>
<td>REF4</td>
<td>WORD ENDA-LISTA+LISTC</td>
<td>000014</td>
</tr>
<tr>
<td>0057</td>
<td>REF5</td>
<td>WORD ENDC-LISTC-10</td>
<td>FFFFFF6</td>
</tr>
<tr>
<td>005A</td>
<td>REF6</td>
<td>WORD ENDC-LISTC+LISTA-1</td>
<td>00003F</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Field 1</th>
<th>Field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PROGC</td>
<td>START</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTDEF LISTC, ENDC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTREF LISTA, ENDA, LISTB, ENDB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018</td>
<td>REF1</td>
<td>+LDA LISTA</td>
<td>03100000</td>
</tr>
<tr>
<td>001C</td>
<td>REF2</td>
<td>+LDT LISTB+4</td>
<td>77100004</td>
</tr>
<tr>
<td>0020</td>
<td>REF3</td>
<td>+LDX #ENDA-LISTA</td>
<td>05100000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0030</td>
<td>LISTC</td>
<td>EQU</td>
<td>*</td>
</tr>
<tr>
<td>0042</td>
<td>ENDC</td>
<td>EQU</td>
<td>*</td>
</tr>
<tr>
<td>0042</td>
<td>REF4</td>
<td>WORD ENDA-LISTA+LISTC</td>
<td>000030</td>
</tr>
<tr>
<td>0045</td>
<td>REF5</td>
<td>WORD ENDC-LISTC-10</td>
<td>000008</td>
</tr>
<tr>
<td>0045</td>
<td>REF6</td>
<td>WORD ENDC-LISTC+LISTA-1</td>
<td>000011</td>
</tr>
<tr>
<td>004B</td>
<td>REF7</td>
<td>WORD ENDA-LISTA-(ENDB-LISTB)</td>
<td>000000</td>
</tr>
<tr>
<td>004E</td>
<td>REF8</td>
<td>WORD LISTB-LISTA</td>
<td>000000</td>
</tr>
</tbody>
</table>
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
Program Linking Example

• Fig. 3.10

• 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
  – T0000540F000014FFFFFF600003F000014FFFFFC0
  – M00005406+LISTC
<table>
<thead>
<tr>
<th>Memory address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x x x x x x</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3FF0</td>
<td>x x x x x x</td>
</tr>
<tr>
<td>4000</td>
<td></td>
</tr>
<tr>
<td>4010</td>
<td></td>
</tr>
<tr>
<td>4020</td>
<td>03201D77</td>
</tr>
<tr>
<td>4030</td>
<td></td>
</tr>
<tr>
<td>4040</td>
<td></td>
</tr>
<tr>
<td>4050</td>
<td></td>
</tr>
<tr>
<td>4060</td>
<td>000083</td>
</tr>
<tr>
<td>4070</td>
<td></td>
</tr>
<tr>
<td>4080</td>
<td></td>
</tr>
<tr>
<td>4090</td>
<td></td>
</tr>
<tr>
<td>40A0</td>
<td>05100014</td>
</tr>
<tr>
<td>40B0</td>
<td></td>
</tr>
<tr>
<td>40C0</td>
<td></td>
</tr>
<tr>
<td>40D0</td>
<td></td>
</tr>
<tr>
<td>40E0</td>
<td>0083</td>
</tr>
<tr>
<td>40F0</td>
<td></td>
</tr>
<tr>
<td>4100</td>
<td>40C70510</td>
</tr>
<tr>
<td>4110</td>
<td></td>
</tr>
<tr>
<td>4120</td>
<td></td>
</tr>
<tr>
<td>4130</td>
<td>000083xx</td>
</tr>
<tr>
<td>4140</td>
<td>x x x x x</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Program Logic and Data Structure

• Two Passes Logic
  – Pass 1: assign addresses to all external symbols
  – Pass 2: perform the actual loading, relocation, and linking

• ESTAB (external symbol table)

<table>
<thead>
<tr>
<th>Control section</th>
<th>Symbol</th>
<th>Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program A</td>
<td>LISTA</td>
<td>4040</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDA</td>
<td>4054</td>
<td></td>
</tr>
<tr>
<td>Program B</td>
<td>LISTB</td>
<td>40C3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDB</td>
<td>40D3</td>
<td></td>
</tr>
<tr>
<td>Program C</td>
<td>LISTC</td>
<td>4112</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDC</td>
<td>4124</td>
<td></td>
</tr>
</tbody>
</table>
Pass 1 Program Logic

• 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

• Fig. 3.11(a)
  – Process Define Record
Pass 1:

begin
get PROCADDR from operating system
set CSADDR to PROCADDR {for first control section}
while not end of input do
  begin
    read next input record {Header record for control section}
    set CS1TH 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 CS1TH to CSADDR {starting address for next control section}
      end {while not EOF}
  end {Pass 1}
Pass 2 Program Logic

- Pass 2:
  - perform the actual loading, relocation, and linking

- Modification record
  - lookup the symbol in ESTAB

- End record for a main program
  - transfer address

- 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}
      end {while record type () 'E'}
    end {while not end of input}
  end {begin}
end {begin}
Improve Efficiency

• Use **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
  – Fig. 3.12
### Figure 3.12

<table>
<thead>
<tr>
<th>Ref No.</th>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROGA</td>
<td>4000</td>
</tr>
<tr>
<td>2</td>
<td>LISTB</td>
<td>40C3</td>
</tr>
<tr>
<td>3</td>
<td>ENDB</td>
<td>40D3</td>
</tr>
<tr>
<td>4</td>
<td>LISTC</td>
<td>4112</td>
</tr>
<tr>
<td>5</td>
<td>ENDC</td>
<td>4124</td>
</tr>
</tbody>
</table>

**PROGA**

<table>
<thead>
<tr>
<th>Ref No.</th>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROGB</td>
<td>4063</td>
</tr>
<tr>
<td>2</td>
<td>LISTA</td>
<td>4040</td>
</tr>
<tr>
<td>3</td>
<td>ENDA</td>
<td>4054</td>
</tr>
<tr>
<td>4</td>
<td>LISTC</td>
<td>4112</td>
</tr>
<tr>
<td>5</td>
<td>ENDC</td>
<td>4124</td>
</tr>
</tbody>
</table>

**PROGB**

<table>
<thead>
<tr>
<th>Ref No.</th>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROGC</td>
<td>4063</td>
</tr>
<tr>
<td>2</td>
<td>LISTA</td>
<td>4040</td>
</tr>
<tr>
<td>3</td>
<td>ENDA</td>
<td>4054</td>
</tr>
<tr>
<td>4</td>
<td>LISTB</td>
<td>40C3</td>
</tr>
<tr>
<td>5</td>
<td>ENDB</td>
<td>40D3</td>
</tr>
</tbody>
</table>
Fig. 3-12 (a)
Machine-Independent Features

Automatic Library Search
automatically incorporate routines from a
subprogram library

Loading Options
Automatic Library Search

- Automatic library call
  - The programmer does not need to take any action beyond mentioning the subroutine names as external references

- Solution
  1. Enter the symbols from each Refer record into ESTAB
  2. When the definition is encountered (Define record), the address is assigned
  3. At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references
  4. The loader searches the libraries specified (or standard) for undefined symbols or subroutines
Automatic Library Search (Cont.)

• The library search process may be repeated
  – Since the subroutines fetched from a library may themselves contain external references

• Programmer defined subroutines have higher priority
  – The programmer can override the standard subroutines in the library by supplying their own routines

• Library structures
  – Assembled or compiled versions of the subroutines in a library can be structured using a directory that gives the name of each routine and a pointer to its address within the library
Loader Options

• Many loaders have a special command language that is used to specify options
  – a separate input file
  – source program
  – embedded in the primary input stream between programs

• Command Language
  – specifying alternative sources of input
    • INCLUDE program-name(library-name)
  – changing or deleting external reference
    • DELETE name
    • CHANGE symbol1, symbol2
Loader Options (cont.)

- specify that some references not be resolved
  - NOCALL name

- specify the location at which execution is to begin

**Example**

- If we would like to evaluate the use of READ and WRITE instead of RDREC and WRREC, for a temporary measure, we use the following loader commands
  - INCLUDE READ(UTLIB)
  - INCLUDE WRITE(UTILB)
  - DELETE RDREC, WRREC
  - CHANGE RDREC, READ
  - CHANGE WRREC, WRITE

- If it is know that the statistical analysis is not to be performed in an execution
  - NOCALL STDDEV, PLOT, CORREL
Loader Design Options

• Linkage Editors
• Dynamic Linking
• Bootstrap Loaders
Linkage Editors

• Definition
  – 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 a library for later execution

• Procedure
  – A linkage editor performs relocation of all control sections relative to the start of the linked program, resolves all external reference, and output a relocatable module for later execution

• A simple relocating loader can be used to load the program into memory (one-pass without external symbol table)
FIGURE 3.17 Processing of an object program using (a) linking loader and (b) linkage editor.

Processing of an object program using (a) linking loader and (b) linkage editor
Linking loader vs. linkage editors

- **Comparison**
  - Linking Loader: *performs all linking and relocation operations, including library search if specified, and loads the linked program directly into memory for execution*
  - Linkage Editors: *produces a linked version of the program (often called a load module or an executable image), which is written onto a file or library for later execution*
  - Resolution of external reference and library searching are only performed once for linkage editor
    - If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.
    - If a program is under development or is used infrequently, the use of a linking loader outperforms a linkage editor
Linkage Editors (Cont.)

- Other functions
  - produce core image if actual address is known in advance
  - improve a subroutine (PROJECT) of a program (PLANNER) without going back to the original versions of all of the other subroutines

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

- external references are retained in the linked program
Linkage Editors (Cont.)

- build packages of subroutines or other control sections THAT ARE GENERALLY USED TOGETHER

  • example

    INCLUDE       READR(FTNLIB)
    INCLUDE       WRITER(FTNLIB)
    INCLUDE       BLOCK(FTNLIB)
    INCLUDE       DEBLOCK(FTNLIB)
    INCLUDE       ENCODE(FTNLIB)
    INCLUDE       DECODE(FTNLIB)

    .

    SAVE           FTNIO(SUBLIB)

- specify that external references are not to be resolved by automatic library search
Dynamic Linking

• Comparison
  – Linkage editors perform linking operations before the program is loaded for execution
  – Linking loaders perform linking operations at load time
  – Dynamic linking (dynamic loading, load on call) perform linking at execution time

• Delayed Binding
  – Avoid the necessity of loading the entire library for each execution, i.e. load the routines only when they are needed
  – Allow several executing programs to share one copy of a subroutine or library (Dynamic Link Library, DLL)
Dynamic Linking

• Via an OS
  – dynamic loader is one part of the OS
  – Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the OS
  – pass of control
    • User program -> OS
    • OS: load the subroutine
    • OS -> Subroutine
    • Subroutine -> OS
    • OS -> User program
Pass of control
Pass of Control

FIGURE 3.18 Loading and calling of a subroutine using dynamic linking.
Bootstrap Loaders

- **Absolute loader**
  - One some computers, an absolute loader program is permanently resident in a read-only memory ROM
  - One some computers, there’s a built-in hardware which read a fixed-length record from some device into memory at a fixed location. After the read operation, control is automatically transferred to the address in memory
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
Repeated Instruction

Suppose that the SIC assembler language is changed to include a new form of the RESB statement, such as

```
RESB  n‘c’
```

which reserves n bytes of memory and initializes all of these bytes to the character ‘c’. For example

```
BUFFER   RESB 4096‘ ’
```

This feature could be implemented by simply generating the required number of bytes in Text records. However, this could lead to a large increase in the size of the object program.
LINK

• Pass 1
  – compute a starting address for each segment in the program
    • segment from different object modules that have the same
      segment name and class are combined
    • segments with the same class, but different names are
      concatenated
    • a segment’s starting address is updated as these combinations
      and concatenations are performed

• Pass 2
  – extract the translated instructions from the object modules
  – build an image of the executable program in memory
  – write it to the executable (.EXE) file
SunOS Linkers

• Link-editor
  – *relocatable object module*, for further link-editing
  – *static executable*, ready to run
  – *dynamic executable*, some symbolic references need to be bound at run time
  – *shared object*, services that can be bound at run time to dynamic executables

• Run-time linker
Run-Time Linker

- Locate and include necessary shared objects
- Lazy binding
  - Binding of procedure calls is normally deferred until the program is in execution
- Reduce the amount of overhead required for starting a program