3.3 Machine-Independent Loader Features

- Loading and linking are often thought of as operating system service functions.

- Machine independent loader features:
  - **3.3.1 Automatic Library Search**
  - **3.3.2 Loader Options**

Machine-Independent Features (Cont.)

- Automatic Library Search for handling external references
  - Allows programmers 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.
- Loader Options
  - Common options that can be selected at the time of loading and linking

3.3.1 Automatic Library Search

- Standard system library, subprogram library
- Automatic library search (Automatic library call)
  - The programmer does not need to take any action beyond mentioning the subroutine names as external references
- Linking loaders that support automatic library search:
  - Must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

Automatic Library Search (Cont.)

- Linking loaders that support automatic library search:
  - Enter the symbols from each ***Refer record*** into ***ESTAB***
  - When the definition is encountered ***(Define record)***, the address is assigned
  - At the end of Pass 1, the symbols in ESTAB that remain undefined represent ***unresolved external references***
  - The ***loader*** searches the libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream

Automatic Library Search (Cont.)

- Since the subroutines fetched from a library may themselves contain external references
  - The library search process may be repeated
- The programmers can override the standard subroutines in the library by supplying their own routines

Automatic Library Search (Cont.)

- Libraries ordinarily contain assembled or compiled versions of subroutines (object programs)
- Library search by scanning the Define records for all of the object programs on the library is quite inefficient
  - =>special file structure is used for the libraries and this structure contains a directory that gives the name of each routine and a pointer to its address within the file.

3.3.2 Loader Options

- Many loaders have a special command language that is used to specify options
  - a separate input file to the loader that contain control statements
  - Control statements embedded in the primary input stream between object programs
  - Control statements are included in the source program, and the assembler or compiler retains these commands as a part of the object program

Common Loader Options – **Command Language**

- Specifying alternative sources of input
  - **INCLUDE** program-name(library-name)
    - Direct the loader to read the designed object program name specified as a part of input program.
- Changing or deleting external references
  - **DELETE** csect-name
    - Delete the named control section(s) from the program loaded when not used
  - **CHANGE** name1, name2
    - Change the external symbol name 1 to name 2 appeared in the object program

Common Loader Options – **Command Language** (Cont.)

- Example for Fig. 2.15
  - INCLUDE READ(UTLIB)
  - INCLUDE WRITE(UTILB)
  - DELETE RDREC, WRREC
  - CHANGE RDREC, READ
  - CHANGE WRREC, WRITE

Common Loader Options – **Command Language** (Cont.)

- User-specified libraries are normally searched before the standard system libraries
  - **LIBRARY** MYLIB
- Specify that no external references be resolved by library search
- Specify the location at which execution is to begin
- Control whether or not the loader should attempt to execute the program if errors are detected during the load

Common Loader Options – **Command Language** (Cont.)

- Specify that some external references not be resolved
  - **NOCALL** name
  - Example
    - If it is know that the statistical analysis is not to be performed in a particular execution

      NOCALL STDDEV, PLOT, CORREL
    - External references are to remain unresolved, but force the program to execute.
    - This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

Common Loader Options – **Command Language** (Cont.)

- Output from the loader
  - Specified as the option of command language
  - Load map
    - Control section names and addresses
    - External symbol addresses
    - Cross-reference table that shows references to each external symbol

3.4 Loader Design Options

- 3.4.1 Linkage Editors
- 3.4.2 Dynamic Linking
- 3.4.3 Bootstrap Loaders

Loader Design Options (Cont.)

- *Linking loaders*
  - Perform all linking and relocation at **load time**
- *Linking editors*
  - Perform linking **before the program is loaded for execution.**
- *Dynamic linking*
  - Perform linking at **execution time**

3.4.1 Linkage Editors

- *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 to a file or library for later execution

Linkage Editors (Cont.)

- A linkage editor
  - Perform relocation of all *control sections* relative to the start of the linked program,
  - Resolve all *external reference*
  - 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 required

Linking Loader vs. Linkage Editor

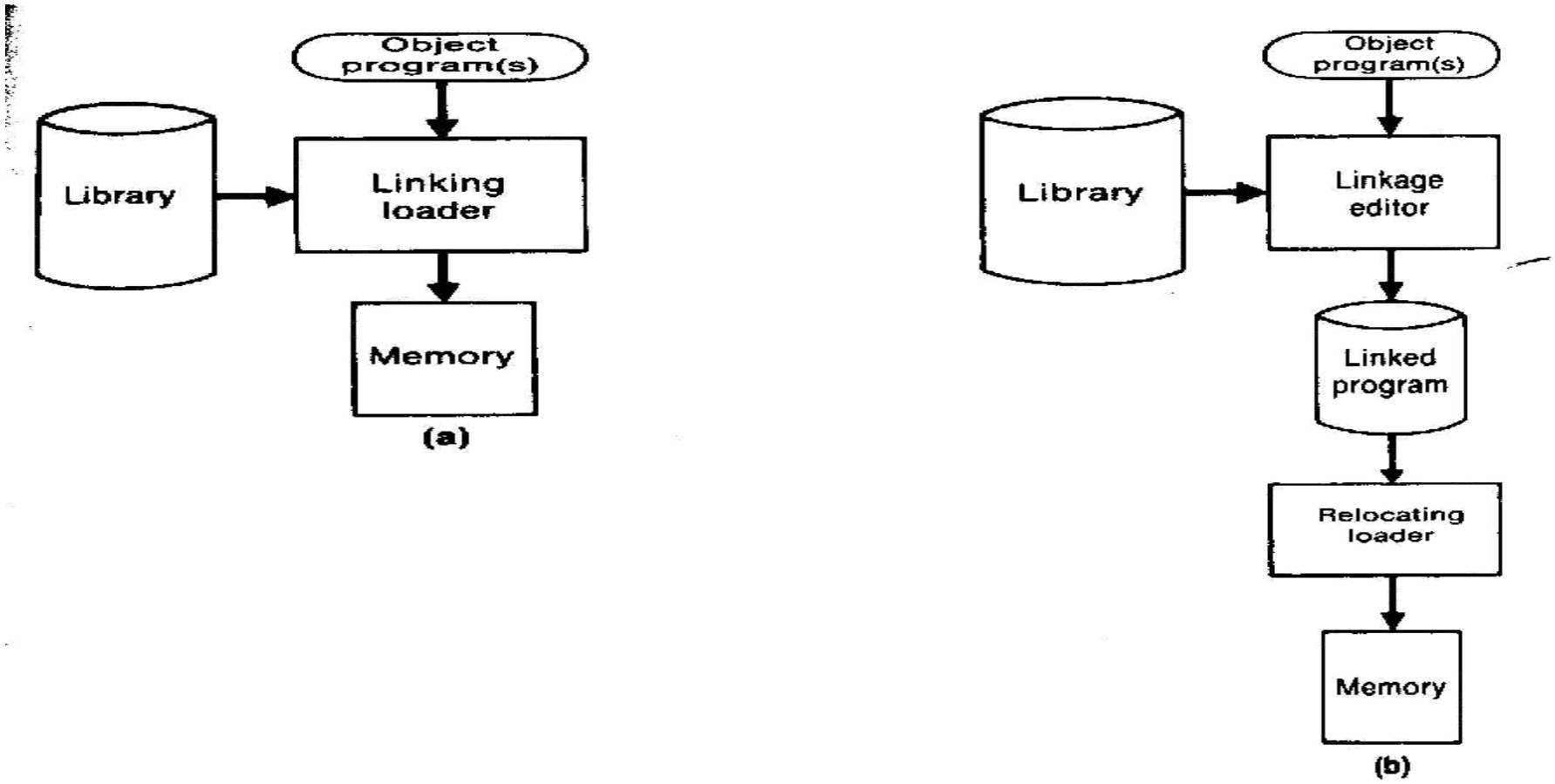


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

Linking Loader vs. Linkage Editor

- *Linking loader*
  - *Searches libraries* and *resolves external references* **every time** the program is executed.
  - Avoid the writing and reading the linked program.

- *Linkage editor*
  - *Resolution of external reference* and *library searching* are only performed **once**

Linking Loader vs. Linkage Editor (Cont.)

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

Additional Functions of Linkage Editors

- Replacement of subroutines in the linked program
- Construction of a package for subroutines generally used together
- Specification of external references not to be resolved by automatic library search

Additional Functions of Linkage Editors (Cont.)

- Replacement of subroutines in the linked program
  - For example: 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)
=> New version of PLANNER
```

Additional Functions of Linkage Editors (Cont.)

- Build *packages of subroutines* or *other control sections* that are generally used **together**
  - For example: build a new *linked module* FINIO instead of search all subroutines in FINLIB

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

Additional Functions of Linkage Editors (Cont.)

- Specify that external references are not to be resolved by automatic library search
  - Thus, only the external references between *user-written routines* would be resolved
    - i.e., does not include library
  - Can avoid multiple storage of common libraries in programs.
    - If 100 programs using the routines on the same library
      - A total copy of 100 libraries would to stored, waste space
  - Need a linking loader to combine the common libraries at execution time.

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 rest of the program *when it is first called*
  - Other names: *dynamic loading*, *load on call*

3.4.2 Dynamic Linking

- Advantages
  - Load the routines when they are needed, the time and memory space will be saved.
  - 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**)

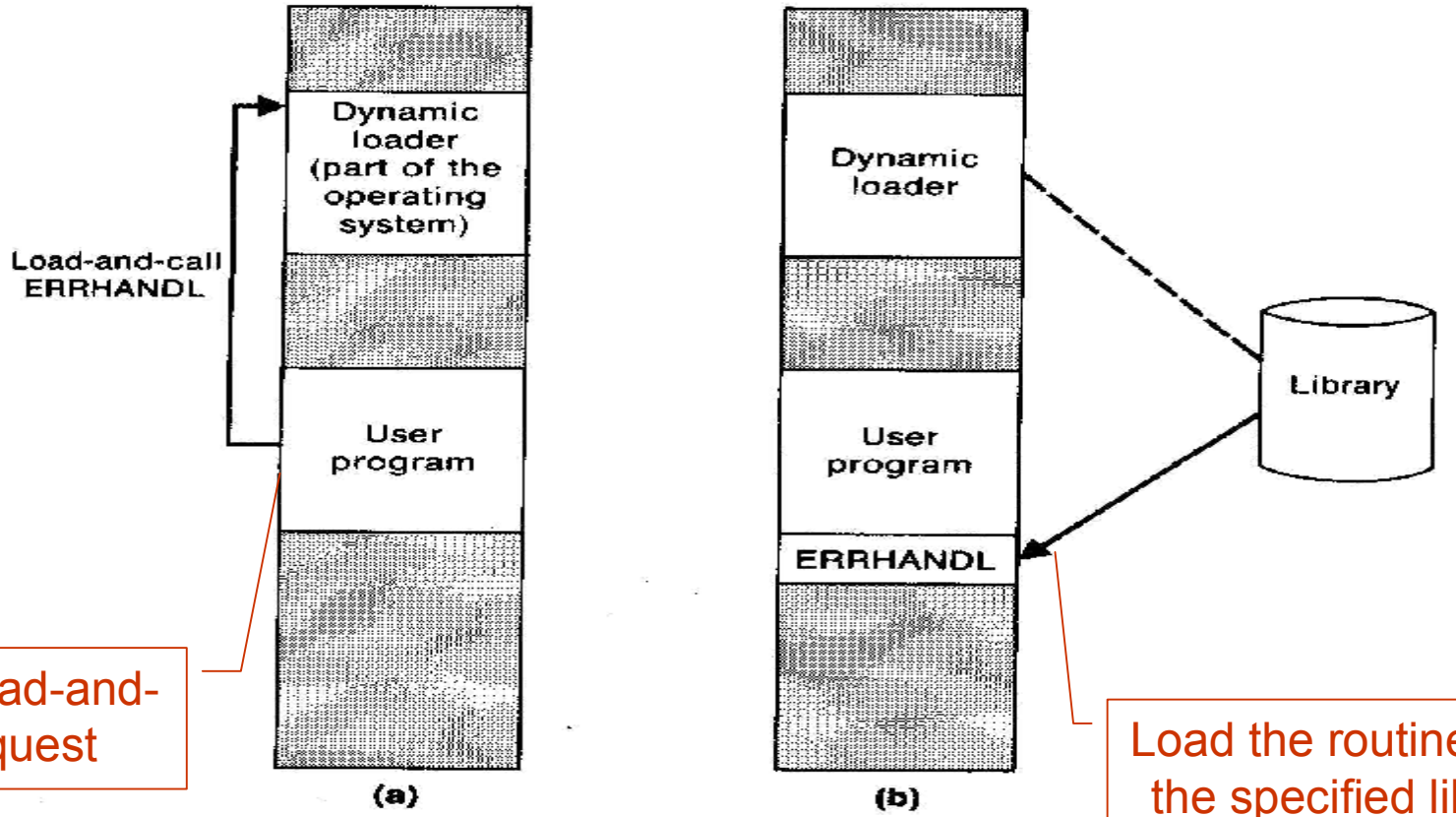
Implementation of Dynamic Linking

- Need the help of OS
  - Dynamic loader is one part of the OS
  - OS should provide load-and-call system call
- Instead of executing a JSUB instruction, the program makes a load-and-call service request to the OS
  - The parameter of this request is the *symbolic name of the routine to be called*
- Processing procedures of load-and-call:
  - Pass control to OS's dynamic loader
  - OS checks the routine in memory or not. If in memory, pass control to the routine. If not, load the routine and pass control to the routine.

Implementation of Dynamic Linking (Cont.)

- Pass the control: (Fig. 3.14)
  - Fig. 3.14 (a) User program -> OS (dynamic loader)
  - Fig. 3.14 (b) OS: load the subroutine
  - Fig. 3.14 (c) OS -> Subroutine
  - Fig. 3.14 (d) Subroutine -> OS -> User program
  - Fig. 3.14 (e) A second call to the same routine may not require load operation
- After the subroutine is completed, the memory that was allocated to load it may be released
  - A second call to the same routine may not require load operation

Loading and Calling a Subroutine Using Dynamic Linking



Issue a load-and-call request

Load the routine from the specified library

Loading and Calling a Subroutine Using Dynamic Linking (Cont.)

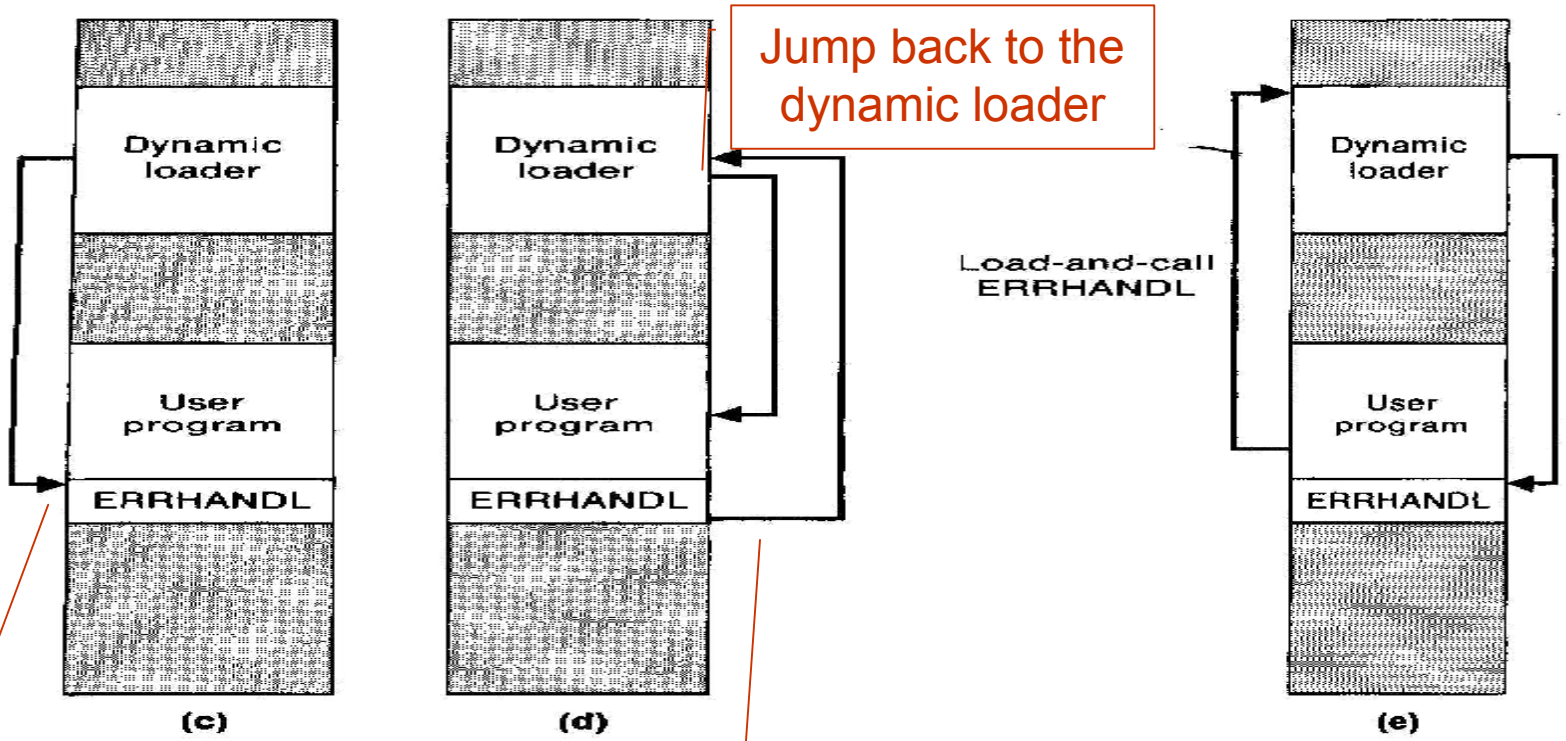


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

Jump to the loaded routine

Jump back to the user program

Second call to this subroutine may not require load operation.

Dynamic Linking (Cont.)

- The association of an *actual address* with the *symbolic name* of the called routine is not made until the call statement is **executed**.
  - ***Binding*** of the name to an actual address is delayed from load time until execution time.
    - ***Delayed binding***
  - Requires more overhead since OS must intervene in the calling process.

3.4.3 Bootstrap Loaders

- How is loader itself loaded into memory?
  - By OS
- How is OS itself loaded into memory?
  - ***Bootstrap loader***

3.4.3 Bootstrap Loaders (Cont.)

- On some computers, an absolute loader program is permanently resident in ROM
  - When power-on, the machine begins to execute this ROM program
  - Inconvenient to change a ROM program if modifications in the absolute loader are required.

3.4.3 Bootstrap Loaders (Cont.)

- An intermediate solution
  - Have a built-in hardware function that reads a fixed-length record from some device into memory at a fixed location, then jump to execute it.
    - This record contains machine instructions that load the absolute program that follows.
  - If the loading process requires more instructions, the first record causes the reading of the others, and these in turn can cause the reading of still more records
    - Hence the term **bootstrap**.
  - The first record (or records) is generally referred to as a **bootstrap loader**.

3.5 Implementation Example (Skip)
3.5.1 MS-DOS Linker

- 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 (Fig. 3.15)

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

Figure 3.15 MS-DOS object module.

LINK

- Pass 1
  - Constructs a symbol table that associates an address with each segment (using the LNAMES, SEGDEF, GRPDEF records) and each external symbol (using EXTDEF and PUBDEF)
- Pass 2
  - extract the translated instructions from the object modules
  - build an image of the executable program in memory (perform relocation)
  - write it to the executable (.EXE) file

Appendix

- Example of Programs Using Libraries

Example of Programs Using Libraries

main.c

```
#include <stdio.h>

extern int a();
extern int b();

main()
{
        int ret1, ret2;

        ret1=a();
        ret2=b();

        printf("\n ret from a()= %d", ret1);
        printf("\n ret from b()= %d", ret2);
}
```

a.c

```
int a()
{
        return 5;
}
int a1()
{
        return 8;
}
```

b.c

```
int b()
{
        return 5;
}
```

Example of Programs Using Libraries (Cont.)

- Compile source programs to object programs
  - gcc –c main.c
  - gcc –c a.c
  - gcc –c b.c
- Create/add/replace object programs into library under SunOS
  - ar –r libtmp.a a.o b.o
- List contents of library under SunOS
  - ar –t libtmp.a
- Delete object programs from library under SunOS
  - ar –d libtmp.a b.o

Example of Programs Using Libraries (Cont.)

- Using library in programs
  - Ex. gcc main.o libtmp.a –o prog
  - Ex. gcc main.o –L. –ltmp –o prog
- Linking editor
  - **ld** under SunOS
- References
  - man **gcc** under SunOS
  - man **ar** under SunOS
  - man **ld** under SunOS