



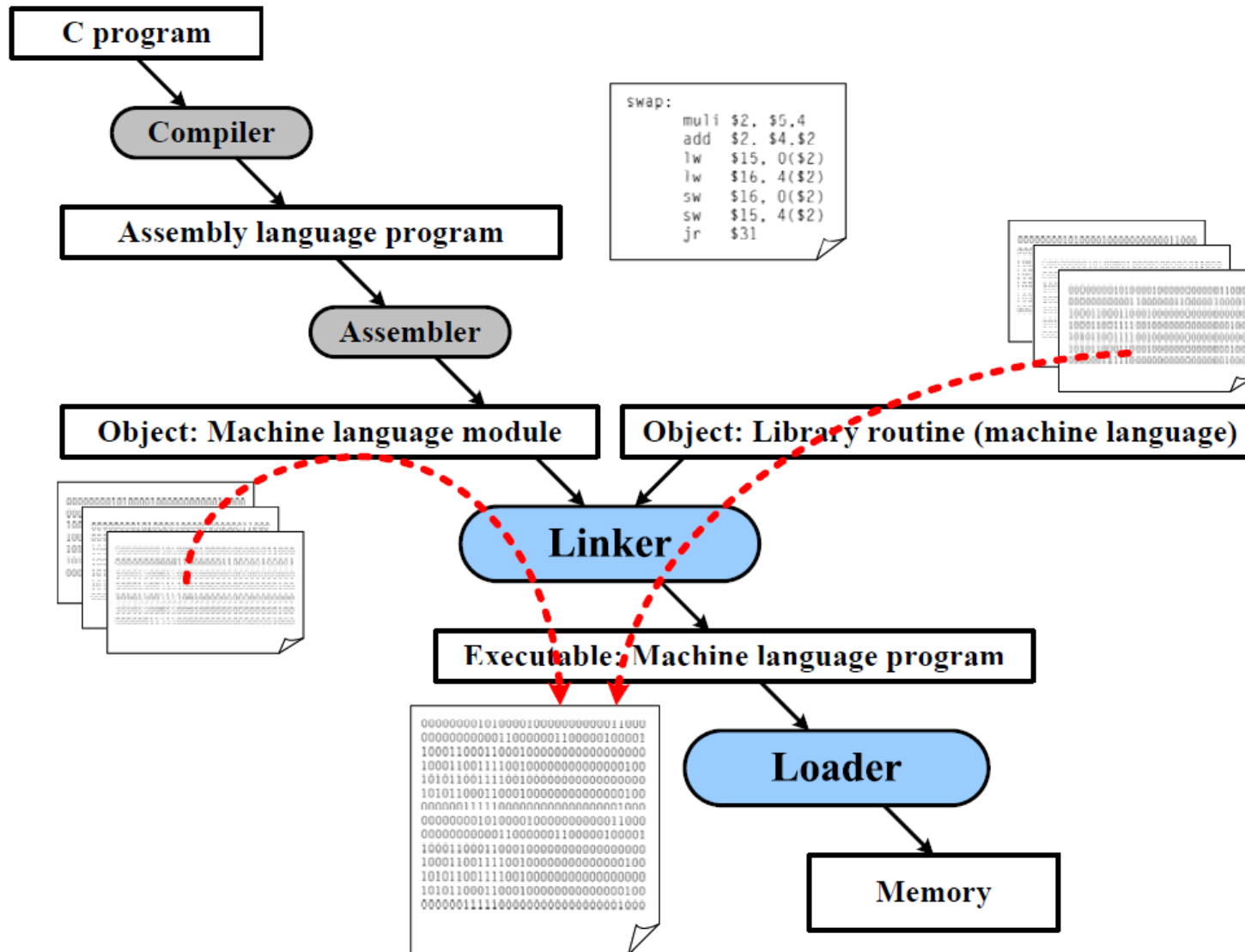
CS202 – System Software

Dr. Manish Khare

Linker and Loader



Need for Linker and Loader



MACHINE DEPENDENT LOADER FEATURES




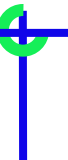
➤ The features of loader that depends on machine architecture are called machine dependent loader features. It includes:

- 1. Program Relocation
- 2. Program Linking

Program Relocation (Relocating Loader)

- The absolute loader has several disadvantages. One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory.
- On a simple computer with a small memory the actual address at which the program will be loaded can be specified easily.
- On a larger and more advanced machine, we often like to run several independent programs together, sharing memory between them. We do not know in advance where a program will be loaded. Hence we write relocatable programs instead of absolute ones.

- 
- Writing absolute programs also makes it difficult to use **subroutine libraries** efficiently. This could not be done effectively if all of the subroutines had preassigned absolute addresses.
 - The need for program relocation is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics.


- 
- Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting address is not known until the load time.
 - In an absolute program the starting address to which the program has to be loaded is mentioned in the program itself using the START directive. So the address of every instruction and labels are known while assembling itself.
 - **Loaders that has the capability to perform relocation are called relocating loaders or relative loaders.**
 - There are two methods for specifying relocation in object program
 - 1. Modification Record (for SIC/XE)
 - 2. Relocation Bit (for SIC)

Modification Record

- A Modification record is used to describe each part of the object code that must be changed when the program is relocated.

Modification record	
Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9	Length of the address field to be modified, in half-bytes (Hex)

- The length is stored in half-bytes (4 bits)
- The starting location is the location of the byte containing the leftmost bits of the address field to be modified.
- If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.



Modification Record (revised)	
Col. 1	M
Col. 2-7	Starting location of the target address to be modified, relative to the beginning of the program (not relative to the first text record)
Col. 8-9	Length of this record in half-byte
Col. 10	Modification flag (+ or -)
Col. 11-16	External symbol whose value is to be added to or subtracted from the indicated field


- Each Modification record specifies the starting address and length of the field whose value is to be altered. It then describes the modification to be performed.

- Consider the following object program, here the records starting with M represents the modification record.
- In this example, the record M 000007 05 + COPY is the modification suggested for the statement at location 000007 and requires modification of 5-half bytes and the modification to be performed is add the value of the symbol COPY, which represents the starting address of the program.(means add the starting address of program to the statement at 000007).

```

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_001083_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

```

- 
- The Modification record is not well suited for certain cases.
 - In some programs the addresses in majority of instructions need to be modified when the program is relocated. This would require large number of Modification records, which results in an object program more than twice as large as the normal.
 - In such cases, the second method called relocation bit is used.

Relocation Bit

- To overcome the disadvantage of modification record, relocation bit is used.
- The Text records are the same as before except that there is a relocation bit associated with each word of object code.
- Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.
- The relocation bits are gathered together into a **bit mask** following the length indicator in each Text record.

col 1:	T
col 2-7:	starting address
col 8-9:	length (byte)
col 10-12:	relocation bits
col 13-72:	object code

- If the relocation bit corresponding to a word of object code is set to 1, the programs starting address is to be added to this word when the program is relocated.
- A bit value of 0 indicates that no modification is necessary.
- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
- In the following object code, the bit mask FFC (representing the bit string 11111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.

1111 1111 1100

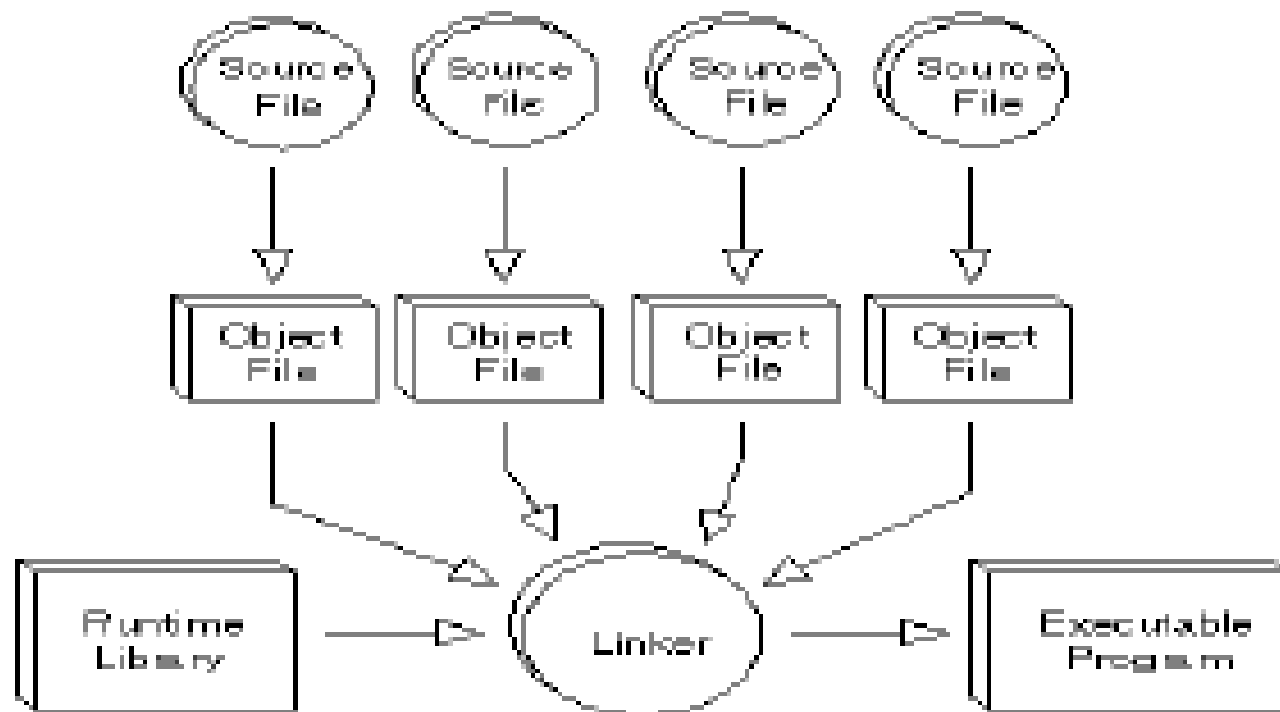
```

HCOPY 00000000107A
T0000001FFC400334810390000362800303000154810613C000300002A0C003900002D
T00001E1E00C00364810610800334C0000454F46000003000000
T0010391EFFC040030000030E0105030103FD810502800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
  
```

Object program with relocation by bit mask.


Program Linking (Linking Loader)

- Many programming languages allow us to write different pieces of code called *modules*, separately. This simplifies the programming task because we can break a large program into small, more manageable pieces. Eventually, though, we need to put all the modules together. Apart from this, a user code often makes references to code and data defined in some "libraries".
- Linking is the process in which references to "externally" defined symbols are processed so as to make them operational.
- A linker or link editor is a program that combines object modules to form an executable program.
- A Linking Loader is a program that has the capability to perform relocation, linking and loading. Linking and relocation is performed at load time.



Algorithm and Data Structures for a Linking Loader

- The algorithm for a *linking loader* is considerably more complicated than the *absolute loader* algorithm.
- A linking loader usually makes *two passes* over its input, just as an assembler does.
- In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler:
 - Pass 1 assigns addresses to all external symbols.
 - Pass 2 performs the actual loading, relocation, and linking.
- The main data structure needed for our linking loader is an *external symbol table* **ESTAB**. This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the *name* and *address* of each external symbol in the set of control sections being loaded.

- 
- Two other important variables are **PROGADDR** (program load address) and **CSADDR** (control section address).
 - (1) PROGADDR is the *beginning address in memory* where the linked program is to be loaded. Its value is supplied to the loader by the OS.
 - (2) CSADDR contains the *starting address* assigned to the control section currently being scanned by the loader. This value is added to all *relative addresses within* the control section to convert them to actual addresses.

Linking loader PASS 1

- During Pass 1, the loader is concerned only with Header and Define records.
- Variables and Data structures used in PASS1
 - PROGADDR (Program Load Address) from OS
 - CSADDR (Control Section Address)
 - CSLTH (Control Section Length)
 - ESTAB (External Symbol Table)

How Pass 1 Algorithm work

- The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.
- The control section name from Header record is entered into ESTAB, with value given by CSADDR.
- All external symbols appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR.
- When the End record is read, the control section length CSLTH (which was saved from the End record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.
- At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.

```

HPROGA 00000000000063
DLISTA 000040^END^A 000054
RLISTB ENDB ^LISTC ^ENDC
:

```

```

HPROGB 0000000000007F
DLISTB 000060^ENDB ^000070
RLISTA ENDA ^LISTC ^ENDC
:

```

```

HPROGC 00000000000051
DLISTC 000030^ENDC ^000042
RLISTA ENDA ^LISTB ^ENDB
:

```

**Linking Loader
Pass 1**


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:

```
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}
```

Linking loader PASS 2

- 
- Pass 2 of linking loader performs the actual loading, relocation, and linking of the program.


How Pass 2 Algorithm work

- 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.
- The last step performed by the loader is usually the transferring of control to the loaded program to begin execution. The End record for each control section may contain the address of the first instruction in that control section to be executed. Loader takes this as the transfer point to begin execution.

Pass 2:

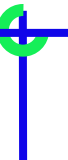
```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
  begin
    read next input record {Header record}
    set CSLTH to control section length
    while record type ≠ 'E' do
      begin
        read next input record
        if record type = 'T' then
          begin
            {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}
```

MACHINE INDEPENDENT LOADER FEATURES

- 
- The features of loader that doesn't depend on the architecture of machine are called machine independent loader features. It includes:
 - Automatic Library search
 - Loader Options that can be selected at the time of loading and linking

Automatic Library Search

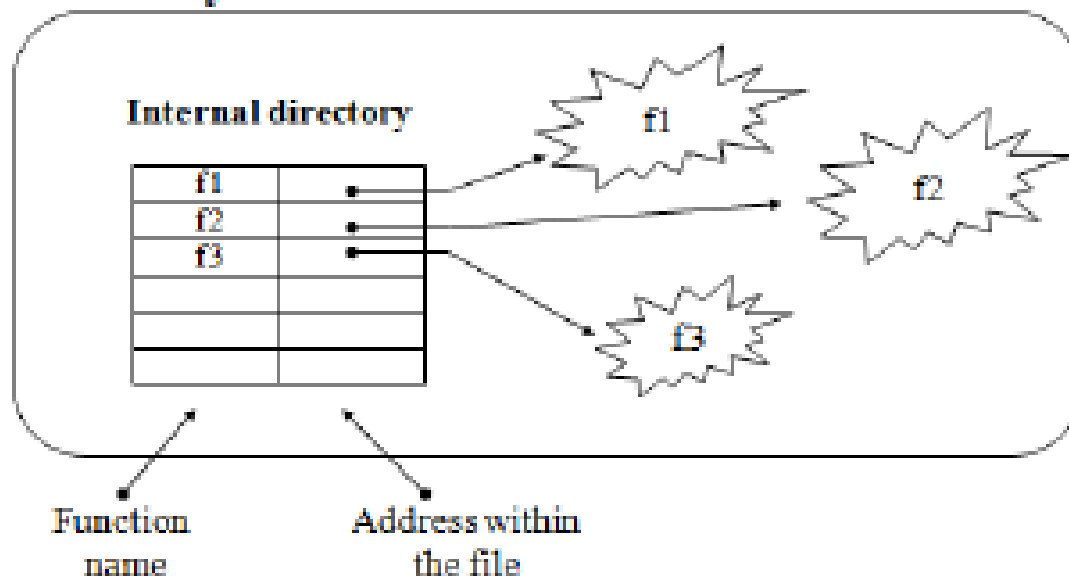
- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- In most cases there is a standard system library that is used in this way. Other libraries may be specified by control statements or by parameters to the loader.
- The subroutines called by the program being loaded are automatically fetched from the library linked with the main library, program, and loaded.
- 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.
- At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.

- 
- The loader searches the **library** or **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.
 - The process just described allows the programmer to **override** the standard subroutines in the library by supplying his or her own routines.
 - In most cases a **special file structure** is used for the libraries
 - This structure contains a **directory** that gives the name of each routine and a pointer to its address within the file.
 - Some operating systems can keep the **directory for commonly used libraries permanently in memory**. This can expedite the search process if a large number of external references are to be resolved.

Linking



Library




Loader Options

- Many loaders allow the user to specify options that modify the standard processing.
- Many loaders have a special **command language** that is used to specify options.
 - Sometimes there is a **separate input file** to the loader that contains such control statements.
 - Sometimes these same statements can also be embedded in the **primary input stream between object programs**.
 - On a few systems the programmer can even include loader control statements in the **source program**.
 - On some systems options are specified as **a part of the job control language** that is processed by the O/S.

Command Language


- **INCLUDE** program-name (library-name)
 - Include the designated object program from a library as a part of input
- **DELETE** csect-name
 - Delete the named control section from the set of programs being loaded
- **CHANGE** name1, name2
 - Replace the external symbol name1 with name2
- **LIBRARY** library-name
 - Specify alternative libraries to be searched
- **NOCALL** name1, name2 ...
 - Instruct the loader that these external references are to remain unresolved

- 
-
- If we would like to use the utility routines READ and WRITE instead of RDREC and WRREC in our programs, for a temporary measure, we use the following loader commands

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

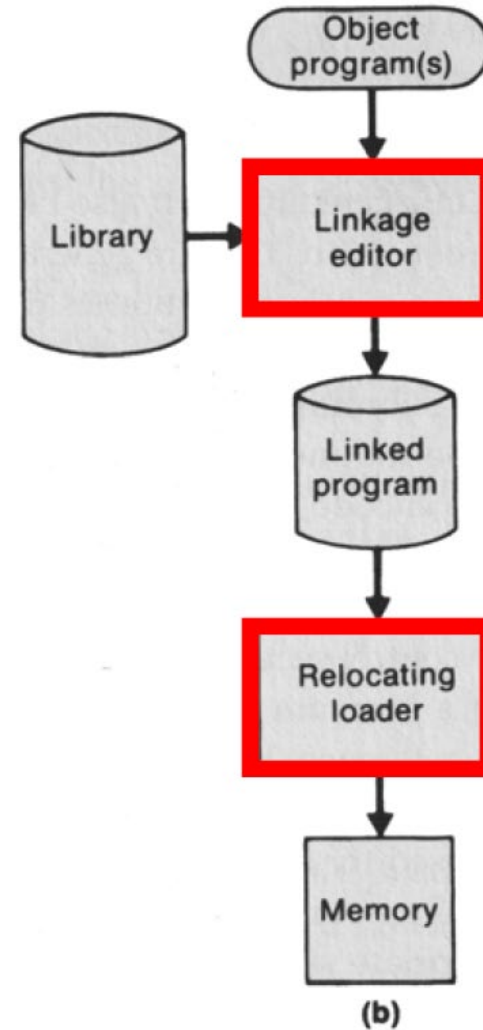
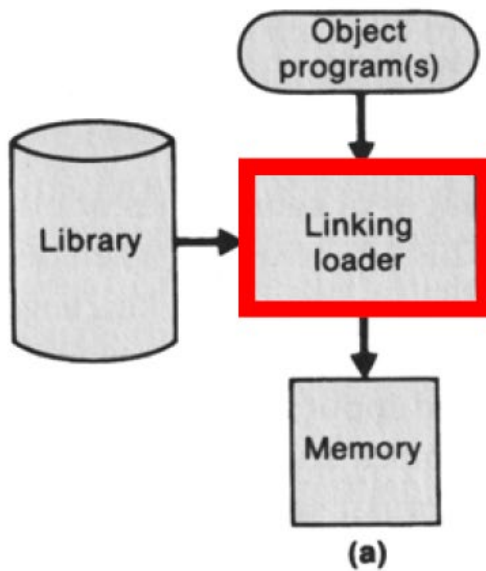
- These commands would ask the loader to include control sections READ and WRITE from the library UTLIB and to delete the control sections WRREC and RDREC. The first CHANGE command would change all the external references to the symbol RDREC to be changed to refer to READ and second CHANGE will cause references to WRREC to be changed to WRITE.


Loader Design Options

- 
- 1. **Linkage Editors** – which perform linking prior to load time
 - 2. **Dynamic Linking** – which perform the linking function at execution time.
 - 3. **Bootstrap Loaders** – used to load operating system or the loader into the memory

Linkage Editors

- Linking loaders perform all linking and relocation at load time. There are two alternatives: Linkage editors, which perform linking prior to load time, and dynamic linking, in which the linking function is performed at execution time.
- Difference between linkage editor and linking loader is explained below:
- 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.
- 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.
- This means that the loading can be accomplished in one pass with no external symbol table required.
- This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.





➤ 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 Linkage Editor(LE) performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program. This means that the loading can be accomplished in one pass with no external symbol table required.

Dynamic Linking

➤ Linkage editor


- Perform linking operations before the program is loaded for execution

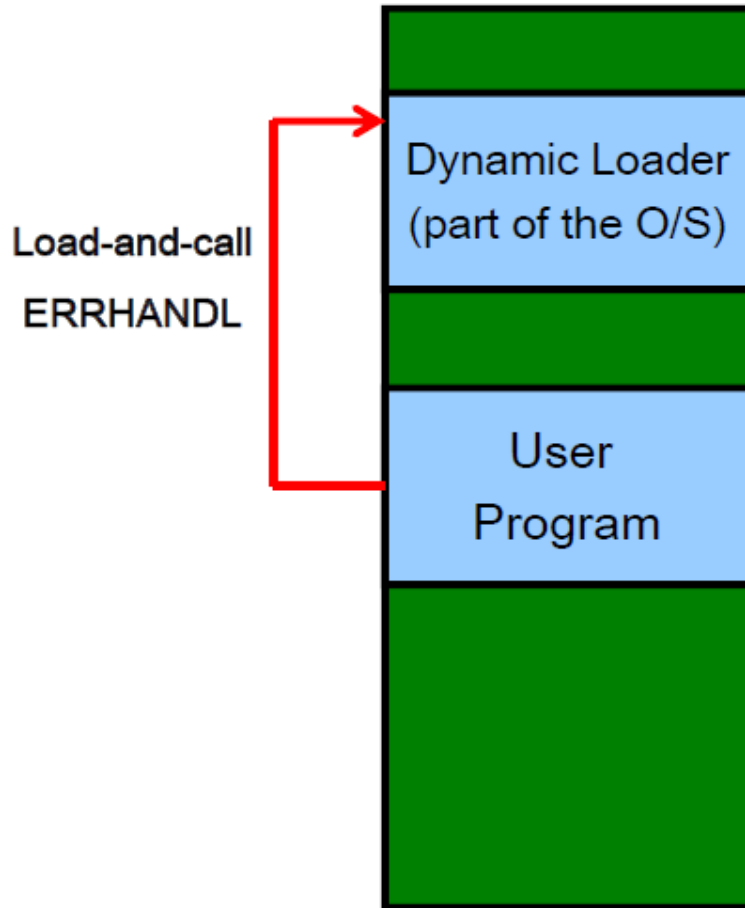
➤ Linking loader

- Perform linking operations at **load time**

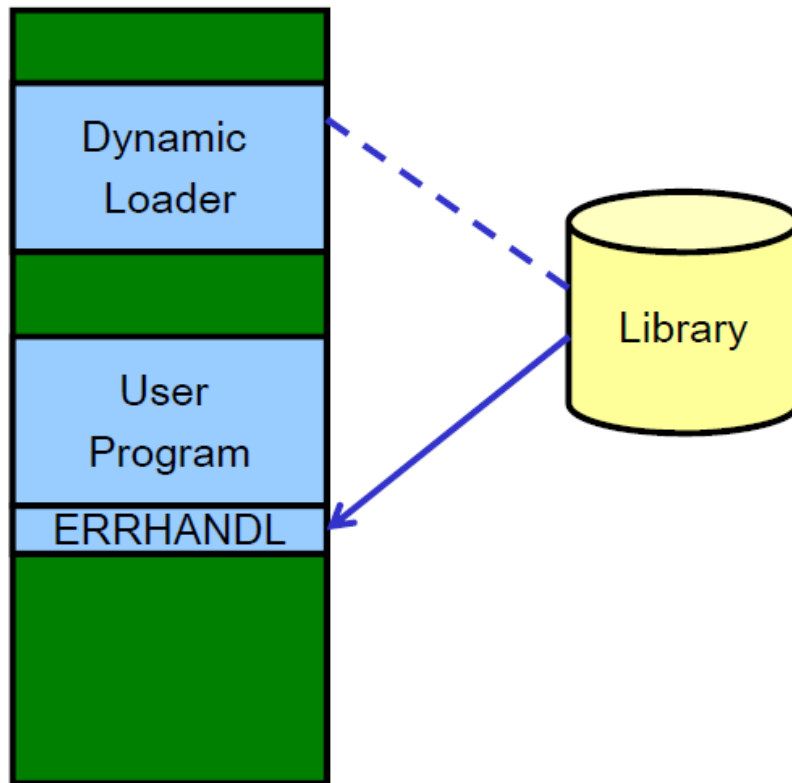
➤ Dynamic linking dynamic loading, load on call)

- Postpone the linking function until **execution time**
- A subroutine is loaded and linked to the rest of the program when it is first called

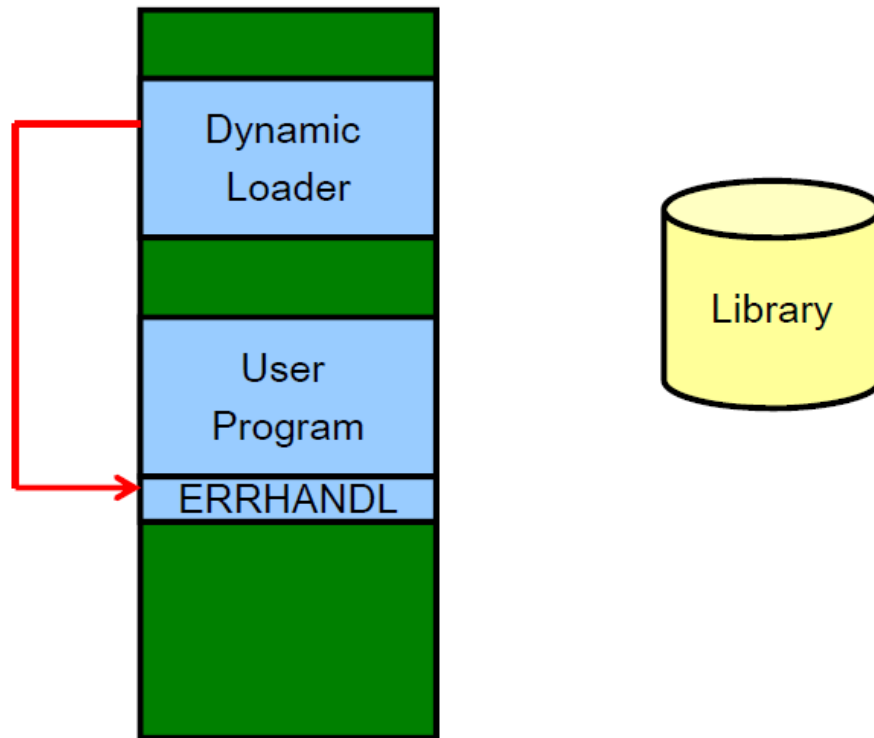
- 
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library (eg. run-time support routines for a high-level language like C.)
 - With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution. Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.



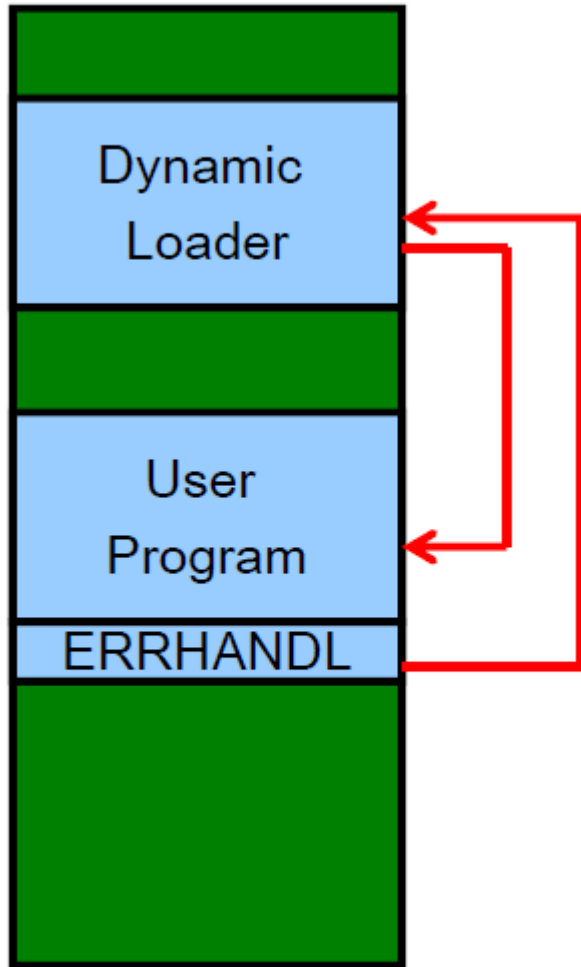
Whenever the user program needs a subroutine for its execution, the program makes a load-and-call service request to OS (instead of executing a JSUB instruction referring to an external symbol). The parameter of this request is the symbolic name (ERRHANDL) of the routine to be called.



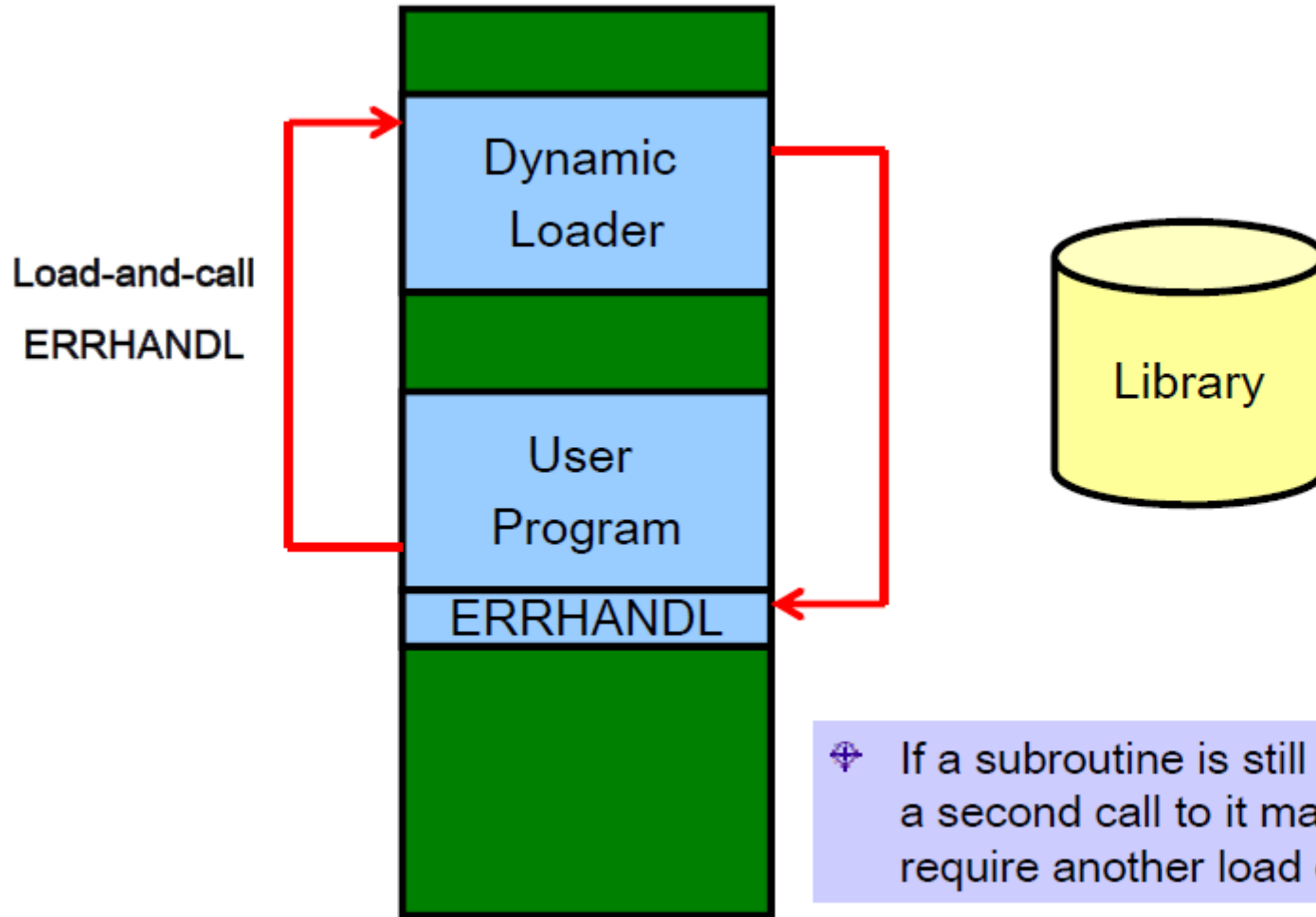
OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries.



Control is then passed from OS to the routine being called.



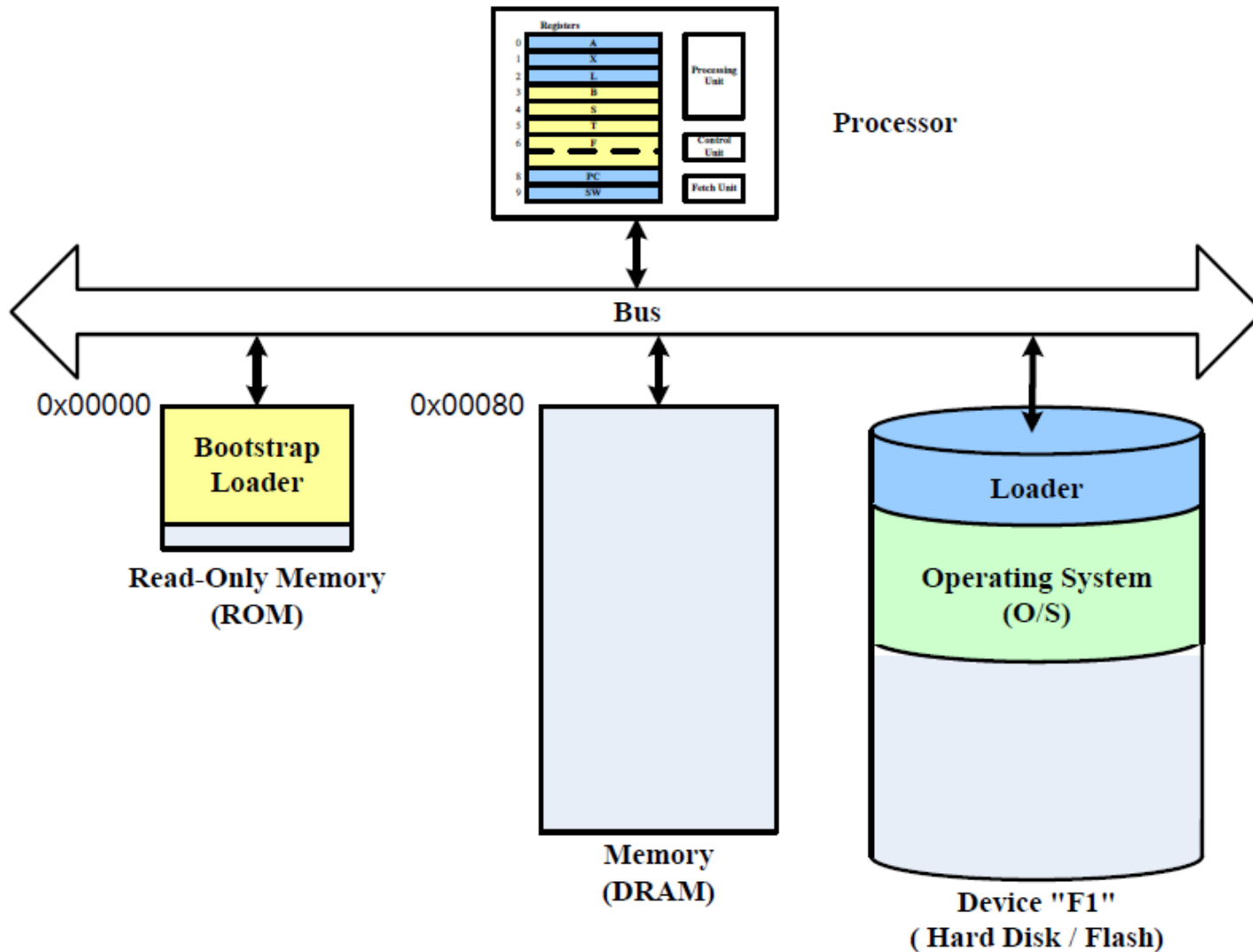
When the called subroutine completes its processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.

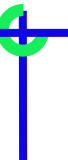


✦ If a subroutine is still in memory, a second call to it may not require another load operation.

Control may simply be passed from the dynamic loader to the called routine.

Bootstrap Loader



- 
- Given an idle computer with no program in memory, how do we get things started? Two solutions are there.
 - On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs, the machine begins to execute this ROM program. This is referred to as a bootstrap loader.
 - On 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. 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
 - When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader**, is executed. This bootstrap loader loads the first program to be run by the computer – usually an operating system