UNIT – II (Loaders and Linkers)

Remaining topics of Unit-I-II

- Loaders: Loader Schemes (Unit-II)
  - Compile and Go, * Compile-assemble-link-load
  - General Loader Scheme, *
  - Absolute Loader Scheme, *-load --Compile-assemble-link(,exe)
  - Subroutine Linkages, *
  - Relocation and linking concepts,*
  - Self-relocating programs,
  - Relocating Loaders,
  - Direct Linking Loaders,
  - Overlay Structure*
- *Linkers (Unit-II)*

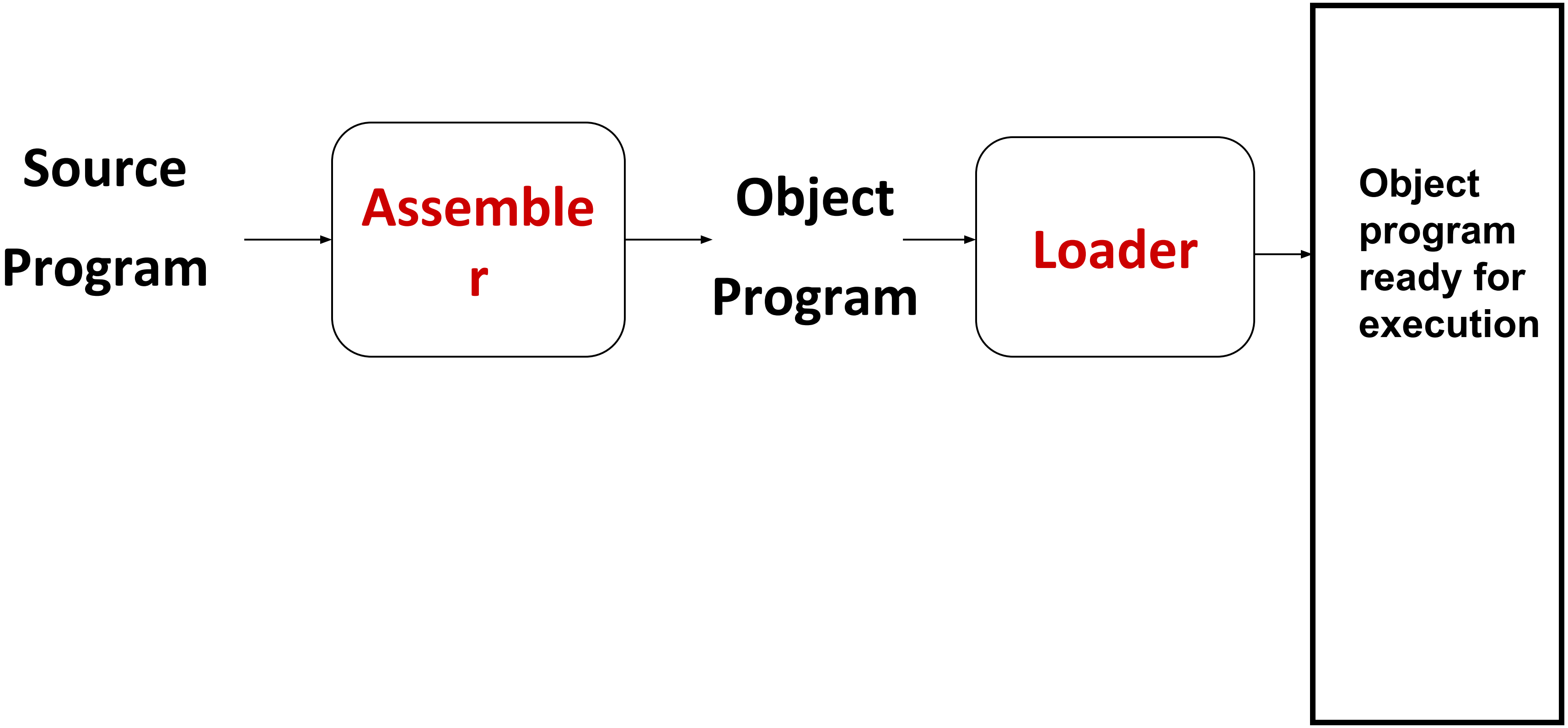
LOADERS and LINKERS

Basics

Loader VS Linker

- Program Loading   (task of LOADER)
- Relocation   (??)
- Symbol Resolution (task of LINKER)

Role of a Loader



Source Program

Assembler

Object Program

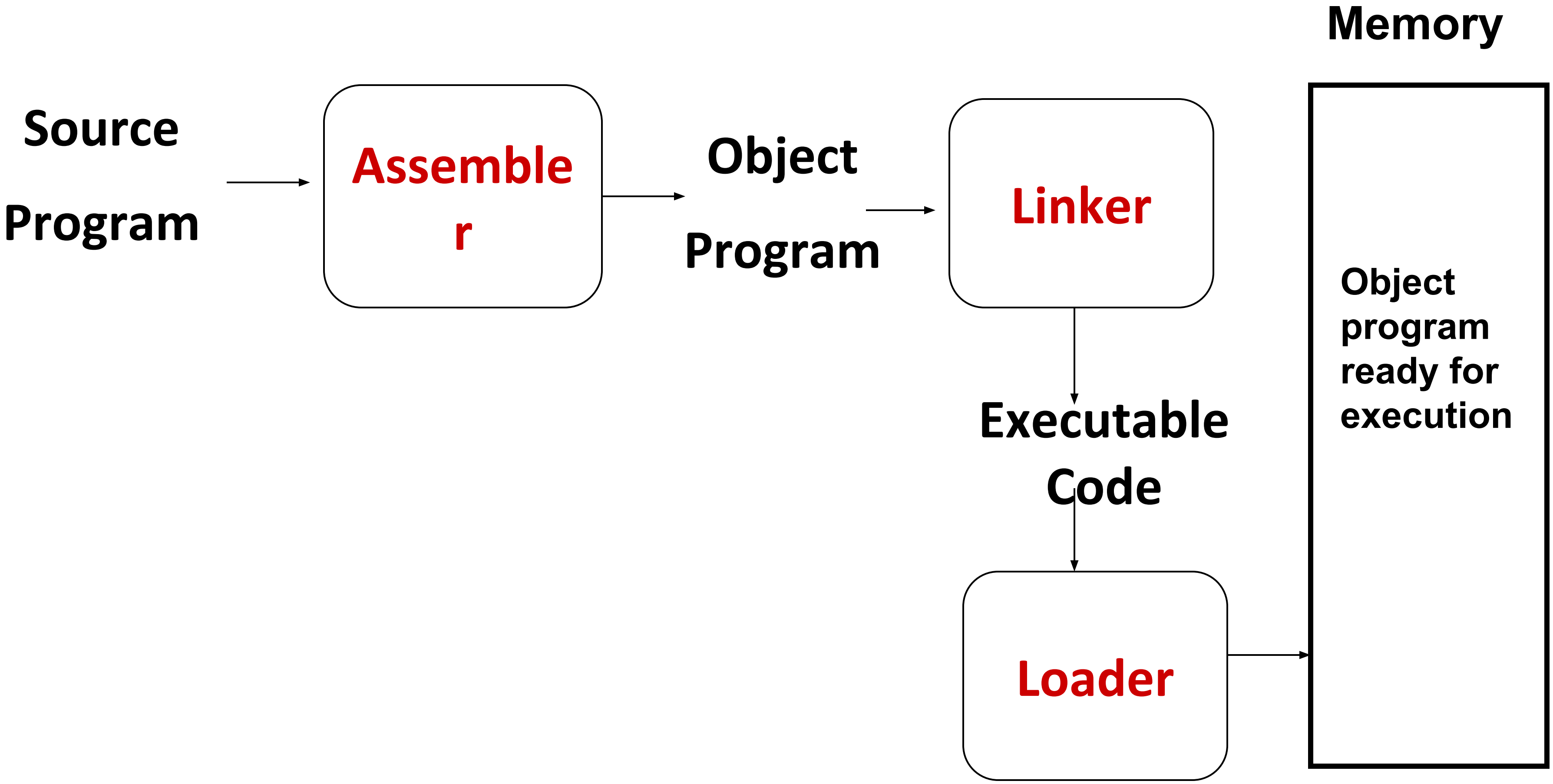
Loader

Object program ready for execution

Linker

- A Program, that takes **one** or **more object files** generated by a compiler and **combines** them into an executable file

Role of a Loader and Linker



What is a Loader?

- A Program, which **loads a program**, from a secondary memory to main memory, when it is to be executed
- A Program, which accepts the object program, **prepares it for execution** by the computer, and initiates the execution

Sample operations of Language Processors (GCC)

Sample operations of Language Processors (GCC)

- >gcc a1.c b1.c

*(GNU preprocessor- **cpp**, compiler proper-**cc1**, assembler–**as;** are part of standard GCC distribution)*

```
cpp some options a1.c      /tmp/a1.i
cc1 some options /tmp/a1.i   /tmp/a1.s
as  some options /tmp/a1.s   /tmp/a1.o
```

```
cpp some options b1.c   /tmp/b1.i
cc1 some options /tmp/b1.i  /tmp/b1.s
as  some options /tmp/b1.s  /tmp/b1.o
```

2 object files – a1.o and b1.o are ready

- **LINKER** takes these input object files (a1.o and b1.o) and generates the final executable

```
ld some options /tmp/a1.o /tmp/b1.o -o a.out
```

- Final executable (a.out) then is ready to be LOADED
- >./a.out

…contd….**Sample** operations of Language Processors (GCC)

- >./a.out
  - The shell invokes the **loader** function
    - Loader copies the code and data in the **executable file a.out** into memory, and then transfers control to the beginning of the program
    - The **loader** (program called **execve)** loads the code and data of the executable object file into memory and then runs the program by jumping to the first instruction

Concept of Loading, Relocating and Linking

Program loading

- Copy a program from secondary storage into main memory so it’s ready to run

- Loading involves copying the data from disk to memory, or also involves **allocating storage**, **setting protection bits**, or arranging for **virtual memory** to map virtual addresses to disk pages

RELOCATION: Due to Loading from any designated area of Memory

- Process of **modifying the addresses** used in the "**address sensitive instructions**" so that the program executes correctly from **any designated area** of the memory
  - Sample of (memory) address sensitive instruction:-

    MOVER AREG**, X**

    BC ANY, **L1**

Relocation of Multiple Subprograms

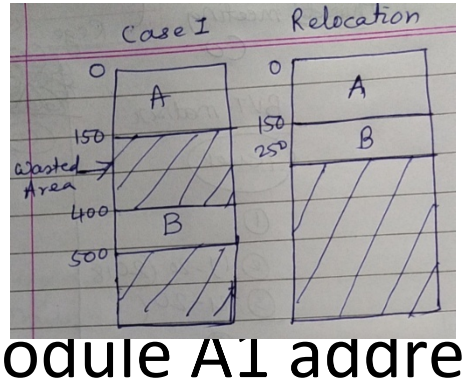
- Compilers/assemblers generate the object code for each input module **with a starting address of zero** (generally)
  - If a program is created from **multiple subprograms**, all the subprograms have to be loaded at non-overlapping addresses
  - Commonly, **linker combines the multiple subprograms**, and creates **one linked output program that starts at zero**, with the various subprograms **relocated to locations within the big program**

- **Relocation** is the process of assigning **load addresses** to different parts of the program when **combining pieces of code and data** by merging all sections of the same type (Code/data) into one section
  - The code and data section also are **adjusted** so they **point to the correct runtime addresses**
  - Then when the program is loaded, the system picks the **actual load address** and the linked program is **relocated** as a whole to the load address

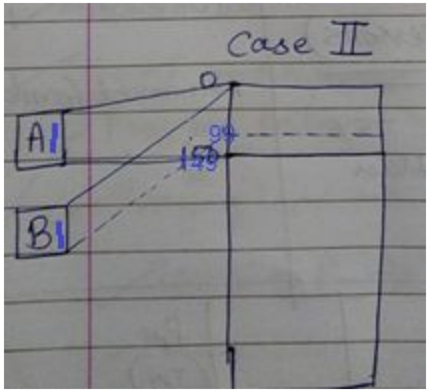
….contd.. RELOCATION

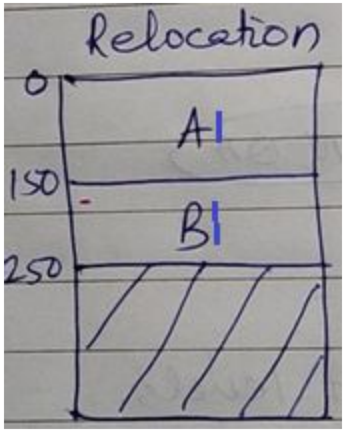
Sample Case for Need of Relocation

- Module A needs Module B, and both have assigned addresses
  - Case I: Module A address: 0-149 and Module B 400-499

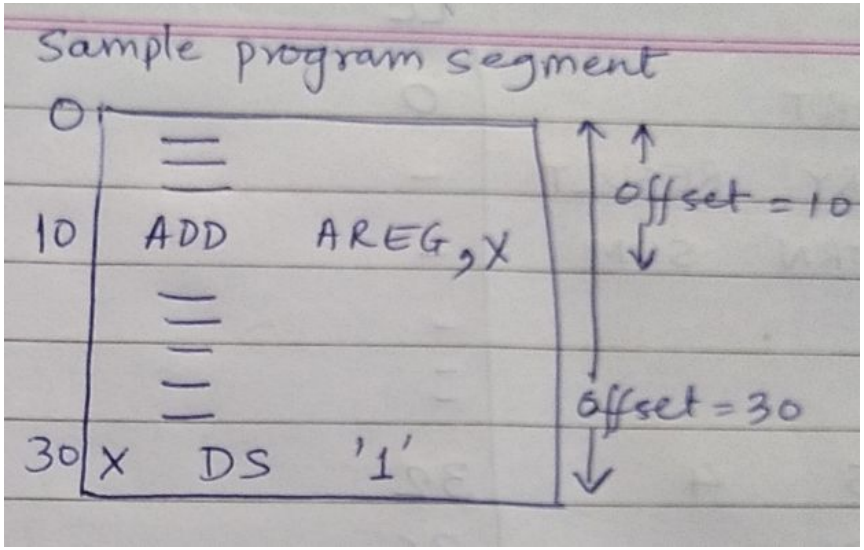


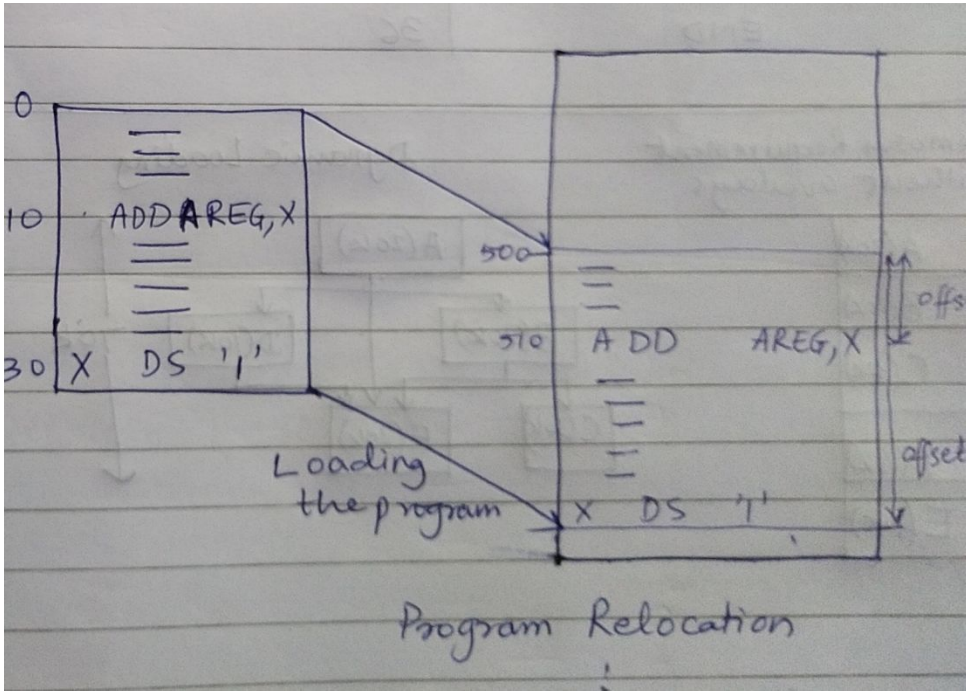
  - Case II: Module A1 address: 0-149 and Module B1 0-99





Relocation Concept with sample example





…contd..Relocation concepts

- Program **Relocatability** : Ability to load and execute program in into arbitrary location in memory
- 2 basic type of relocation: (Based on when mapping of **virtual address** to **physical address** is done)
  - Static Relocation
  - Dynamic Relocation

Basic LINKING Process

- **Combining pieces of dependent codes** and **data** together to form a **single executable** that can be loaded in memory by **loader**
  - **By Resolving External References** (Subroutine Linkage and Data Linkage)
- Commonly, Linking **combines** all the necessary code and data into a **single executable file**
  - This file contains all the information needed to run the program
- **LINKING = Replace :** ***Symbolic*** **addresses** with **real addresses**

Symbol Resolution

- When a program is built from multiple subprograms, the **references** from one subprogram to another are made using **symbols**
  - Ex: a main program might **use** a square root routine called **sqrt**, and the **math library** **defines** **sqrt**
- A linker **resolves the symbol** by **noting the location** assigned to sqrt in the library, and **patching** the caller's object code to so the **call instruction** refers to **that location**

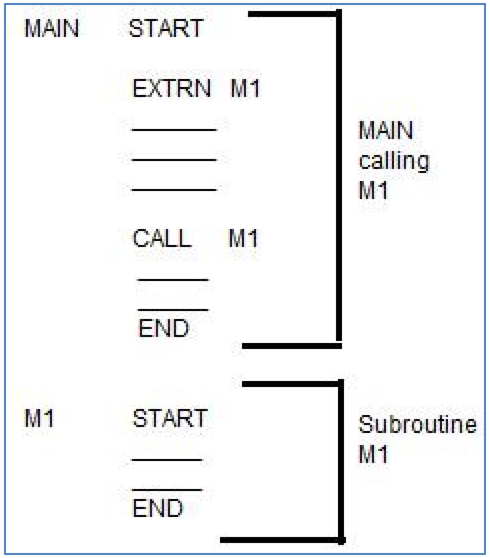
Subroutine Linkages

- Subroutines (functions) may lie in different files, which may be assembled separately
  - Prog A calls function B, residing in **separate file**
    - Assembler, **without help**, will declare B as **undefined**

- How to allow this interaction?
  - Use of -> Public Definitions, External References

…contd..**Subroutine Linkages**

- **EXTRN Statements**: they list the SYMBOLS to which external references are made in current program unit
  - These SYMBOLS are defined in other program unit
  - Diagram shows MAIN and M1 in separate program units
  - Assembler cannot provide address of external symbols



```
MAIN    START
        EXTRN  M1
        ______
        ______
        ______
        CALL   M1
        ______
        ______
        END
M1      START
        ______
        ______
        END
```

- **ENTRY Statements**: they list the public definitions of a program unit
  - Lists SYMBOLS defined the program unit which may be used in other program units

- External symbols are **UNRESOLVED** until **LINKING**

Tasks of the linker

- **Combine** parts / modules of a program
  - (Large programs written in parts/modules, more simplified/manageable pieces)
  - (Pieces of code (Modules) written separately, eventually, need to **put together**)

- **Linking = Replace : Symbolic addresses** with **real addresses**

Linking Time - Variations

- Linking can be done at **compile time**, at **load time (by loaders)** and **also at run time** (by application programs)

Compile Time Linking: **Static Linking**

- All the necessary code and libraries are combined into the final executable before the program is run
  - Resolves references and addresses during this compile/assemble time process
- Static linking produces a standalone executable, and the **linking is performed before the program is run**

Load Time Linking : Dynamic Linking

- Instead of including all the code and data in the executable file, the executable file **contains references** to **external modules**
- These references are resolved by the loader at the time the program is loaded into memory
- This reduces task of reassembly of the whole program, with small change in any 1 module
  - Only that changed 1module is reassembled

Run Time: Dynamic linking: Dynamic Link Libraries (DLLs) and Shared Objects

- Linking can also occur at run time, when program accesses a shared library
- Referenced shared libraries are loaded into memory when the **program starts** or when the **specific module is first accessed**
  - This allows for more efficient use of memory and facilitates **sharing of code** among multiple processes

- Dynamic Link Libraries (DLLs) in Windows or Shared Objects in Unix/Linux are examples of dynamically linked libraries that are linked to a program during its execution
- The program loads these libraries into memory when needed

Static vs Dynamic vs Runtime Linking

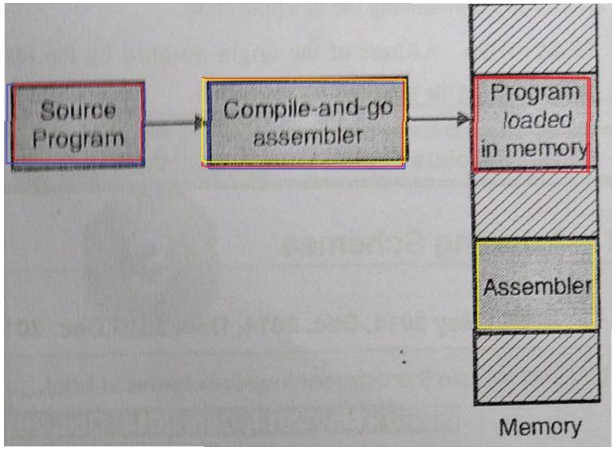
Each approach has its advantages and disadvantages

- Static linking produces a **self-contained executable** but may result in **larger file sizes** and less **flexibility**
- Dynamic linking
  - allows for **more efficient use of resources** and **easier updates** to shared libraries
  - introduces dependencies on external libraries
- Run-time linking
  - Provides the flexibility to load and link modules on demand during program execution

Choice of linking method depends on factors such as **performance requirements**, **resource constraints**, and **the desired level of flexibility** and **modularity** in the software system

Types of Loaders

Compile and Go Loaders



Advantage: Simple, so, easy to implement

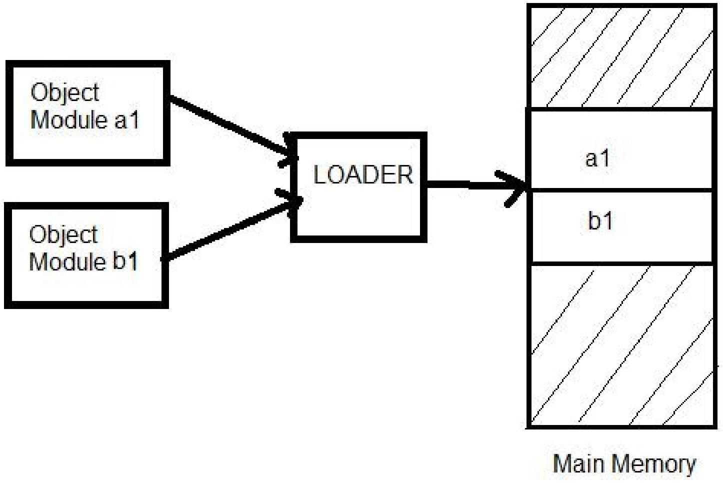
Disdv:

Assembler uses memory

Retranslate every time

Multiple segments handling is difficult

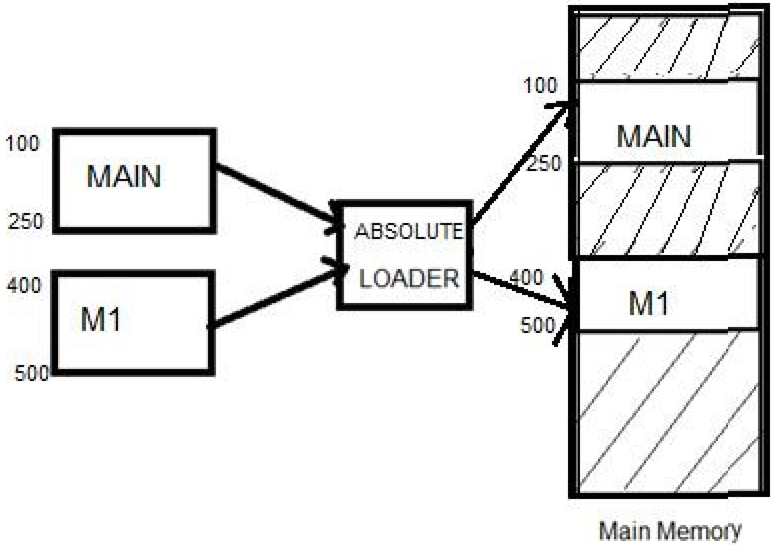
General Loading Scheme :
Loader accepts object files, and places machine instructions and data in the memory for execution



Advantage:

- No need for **retraslation** of Program, everytime it is run
- A small loader(instead of big Compile and Go loader) need reside in memory
- Modular program writing is possible

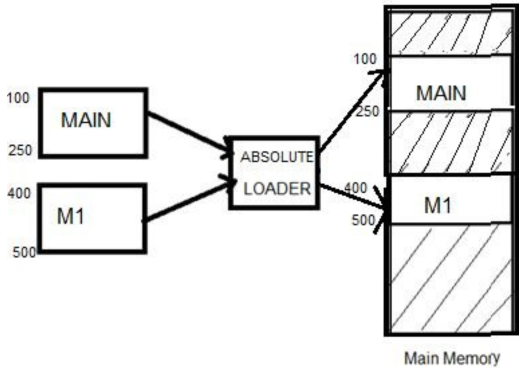
Absolute Loader Scheme



- Loader simply accepts the machine language code produced by assembler and places it at location specified by Assembler
- Accomplishment of Four Loader Functions :

Allocation	By Programmer
Linking	By Programmer
Relocation	By Assembler
Loading	Loader (Absolute Loader)

Absolute Loader Scheme: Advantages and Disadvantages



Advantages:

- **No** **Relocation information** is required, so size of object module is small
- **Simple** to implement
- Small loader is only in memory
- No modification of address sensitive entities is required at time of loading
- Multiple object modules can reside in memory

Disadvantages:

- Programmer has to perform linking, by remembering addresses of each module, and using it explicitly for linking
- Programmer should not assign overlapping locations to modules to be linked
- Memory wastage between modules
- Changes to one module, if leading to overlap with other module, may need tedious manual shifting

Relocating Loaders

BSS Loader (Multiple code segments-but 1 data segment)

Direct Linking Loader (Multiple code segments- multiple data segment)

Relocating Loader

- **Benefits** of general class of Relocating Loader
  - Avoid **re-assembling** all subroutines when a single subroutine is **changed**
  - Provide task of **allocatio**n and **linking** for programmer

- **Output required from assembler :** Program + **Information** about all other programs it references + (Relocation information)
  - (Relocation information = locations in program that need to be changed if it is to be loaded in any arbitrary location in memory)

Relocation – **Hardware support**: **Segment Registers**

- **Segment Registers** makes a program Address **Insensitive**
- All memory addressing is performed using displacement (Offset)
  - Starting Memory Address is stored in segment register and actual address is given by:
    - Content of memory register + Address of operand in instruction
      - Thus, if address of X is 30, and content of memory register is 500, then actual address of X is -> 500+30=530

Examples of Relocating Loaders

Relocating Loader Example :
Binary Symbolic Loader (BSS)

- Allows multiple code segments, but only 1 data segment
- Output of assembler using BSS loader
  - 1.Object Program
  - 2.Reference about other programs to be accessed
  - 3.Information about address sensitive entities

…contd.. **BSS Loader**

- **Assembler** assembles each code segment separately and passes on the information to the loader:

  1. **Object program** prefixed by a **TRANSFER VECTOR**.

     Transfer vector contains information about subroutines used by program

  2. Relocation information = locations in program that need to be changed if it is to be loaded in any arbitrary location in memory

  3. Length of source code, length of transfer vector

- **Loader loads** transfer vector + object code in memory
- Loader then loads each subroutine identified in Transfer Vector
- Transfer vector is used to solve linking problem
- Length of program is used for solving relocation

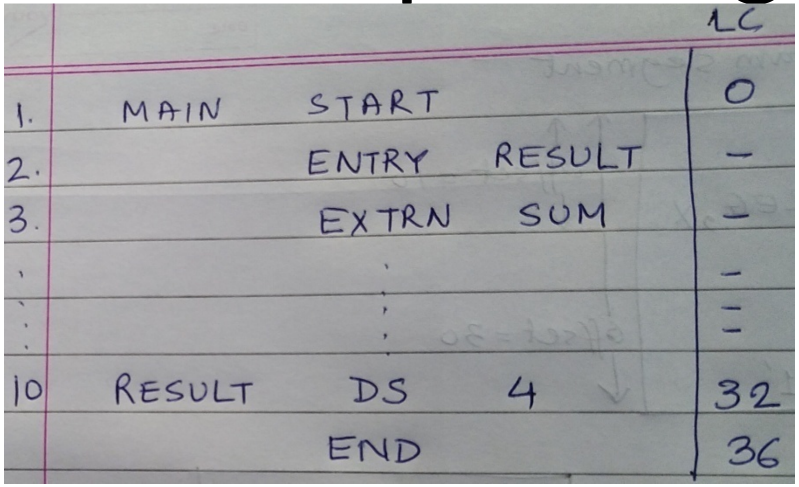
Relocating Loader Example: **Direct Linking Loader**

- Allows multiple code **and** data segments
- Assembler must provide the following with each segment (code/data):
  - Length of segment
  - A list of symbols defined in the current segment that may be referenced by other segments – public declarations
  - A list of symbols not defined in the current segment but used in the segments – external variables
  - Information about address constants
  - Machine code translation and relative addresses assigned

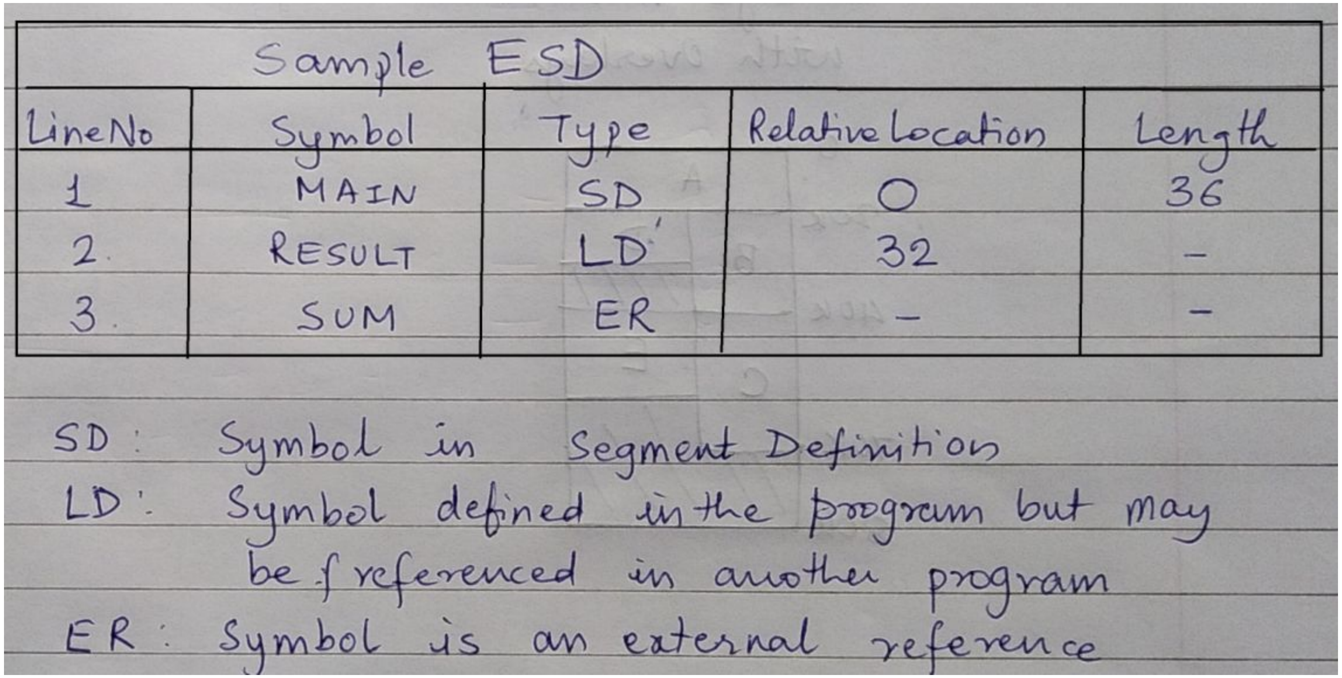
…contd.. **Direct Linking Loaders**

- Object module produced by Assembler has 4 sections
  - External Symbol Directory (ESD)
  - Actual Assembled Program (TXT)
    - Relocatable instructions, data produced during translation
  - Relocation Directory (RLD)
    - One entry for each address that must be changed when the module is loaded into the main memory
  - End of the object module(END)

Sample Program, its ESD



```
                                         LC
1.    MAIN      START                  |  0
2.              ENTRY    RESULT        |  -
3.              EXTRN    SUM           |  -
,                 .                    |  -
:                 :                    |  =
:                 :                    |  -
10    RESULT      DS      4            |  32
                  END                  |  36
```



Sample ESD				
Line No	Symbol	Type	Relative Location	Length
1	MAIN	SD	0	36
2.	RESULT	LD	32	-
3.	SUM	ER	-	-

SD : Symbol in Segment Definition

LD : Symbol defined in the program but may be f referenced in another program

ER : Symbol is an external reference

…contd.. **Direct Linking Loaders**

- Relocation directory (RLD) Information
  - Address of each operand that needs to be changed due to relocation
  - By what it has to be changed
  - Operation to be performed

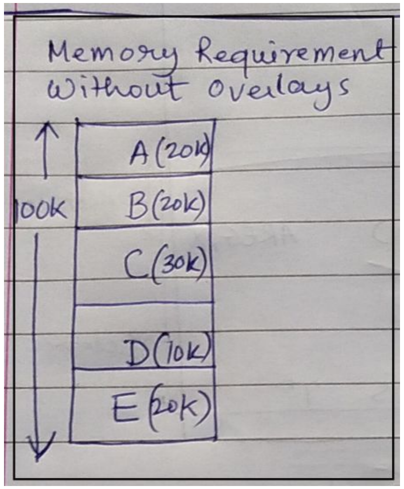
Overlay Structures

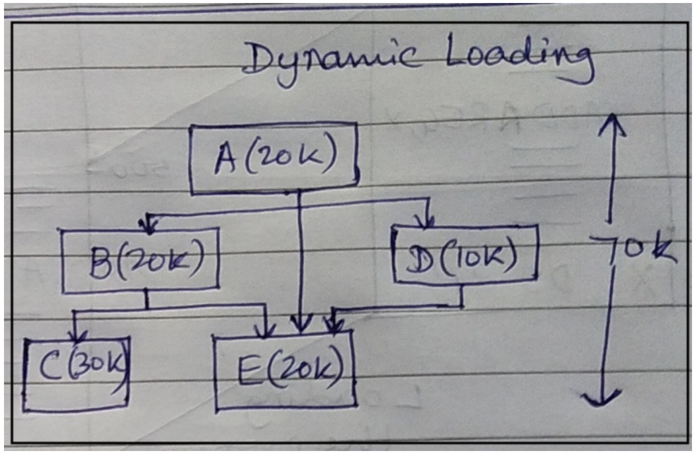
- What is Overlay?
  - Part of a program which has **same load origin**, as some other part of program
- When is it used?
  - When system **does not support virtual memory**

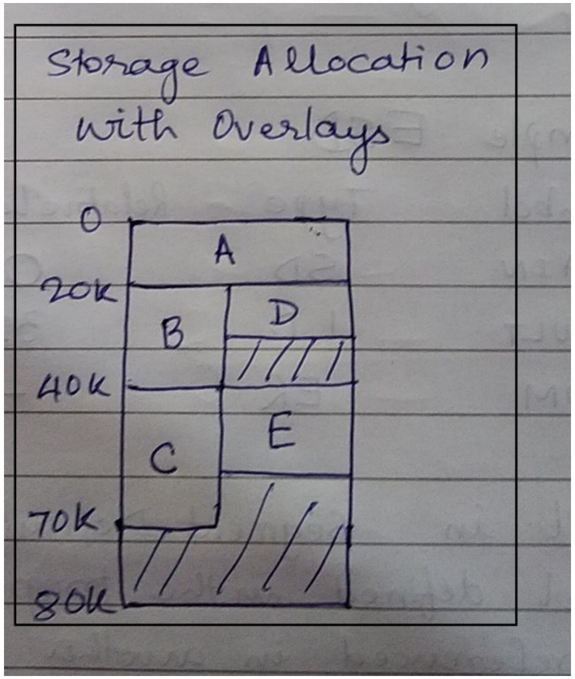
  And **static linking/loading requires all subroutines to be in main memory at the same time**

  And **total memory required exceeds the amount available**

…contd.. Overlay Structures







Linkers

Linker

- A Program, that takes one or more object files generated by a compiler and combines them into an executable file

Linker VS Loader

- Although there’s considerable overlap between linking and loading, it’s reasonable to define a program that does program loading as a loader, and one that does symbol resolution as a linker
- Either can do relocation, and there have been all-in-one linking loaders that do all three functions

Symbol Resolution

- **Reference** of one subprogram to another is made through **symbols**
- A linker's job is to resolve the reference by noting the symbol's location and patching the caller's object code

Tasks of the linker

- **Combine** parts / modules of a program
  - (Large programs are written in parts, which are more simplified/manageable pieces of a large program), Pieces of code (Modules) are written separately, which eventually, though, need to put together)
- **Replace:** Symbolic addresses with real addresses

Linker

HANDLES - Resolving of addresses of **symbolic references**

- Linking Process makes address of modules known, so that transfer of control takes place during execution
  - (During execution of the Main program, control must go correctly to Library functions/ user-defined functions, when they are called)
- Parameter passing (by value/reference) and returning a value by functions, is handled by linker
- Same Public variable, in all modules, must have same address
- EXTERN variable (defined in one module, and used in another) should have same address

Static Vs Dynamic Linking

- Static linking takes object files produced by compiler including library functions, and produces an executable file.
  - Thus an executable file contains contains a copy of every subroutine (user defined / library function)
- Disadvantage??

….contd.. Static Vs Dynamic Linking

- Dynamic linking defers much of the linking process until a program starts running
- Steps:
  - Reference to external module during the run time causes loader to find the target module and load it
  - Perform relocation during runtime
- Advantages:??

(Read Windows DLL)

Self Relocating Programs

- Contain Relocating Logic
- Contains:
  - Table of information about **address sensitive instructions** in program
  - Relocating logic for performing relocation of address sensitive instructions

Set of Programming and Compilation Techniques
Combinations of which aid in position independent (PIC) code

- Relative Addressing : Addresses are expressed as offsets (not absolute addresses)
- GOT (Global Offset Table) and PLT (Procedure Linkage Table)
- Architectures provide a dedicated register
  - Ex: Global Offset Table Pointer or GOTP register, points to the Global Offset Table, where GOT is used to access global variables or functions in a position-independent manner
- Compilers (such as GCC) provide options to generate PIC code, like -fpic or -fPIC for position-independent code. Linker flags, such as -shared, are used when creating shared libraries
- Assembler directives specific to creating position-independent code
  - x86, NASM assembler supports the section .text global _start directive for generating position-independent code
- Deferring certain relocations until load time - The dynamic linker then performs these relocations during the program's loading phase
- HLLs have features or constructs that facilitate the creation of PICs
  - Ex: In C, the use of pointers, and avoiding absolute addresses contributes to PIC

Thank You!!