Linker & Loader

Presented By: Foram Thakor

Overview

- Introduction to Linkers
- Relocation of Linking Concept
- Design of Linker
- Self-Relocating Programs
- Linking in Linux
- Linking Overlay Structured Programs
- Dynamic Linking
- Loaders
- Different Loading Schemes
- Sequential and Direct Loader
- Compile-and-Go Loaders.

Linker

- A linker is a system software that combines multiple object files into a single executable file.
- When a program is compiled, the compiler translates the source code into an intermediate object code, but this object code may reference external functions or libraries.
- The linker resolves these references by locating the corresponding object code (or library code) and linking them together.
- Linkers are also called as **link editors**.
- Linking is a process of collecting and maintaining piece of code and data into a single file.

Linker (Contd.)

- Linker also links a particular module into system library.
- It takes **object modules** from assembler as input and forms an executable file as output for the loader.
- Linking is performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory by the loader.
- Linking is performed at the last step in compiling a program.

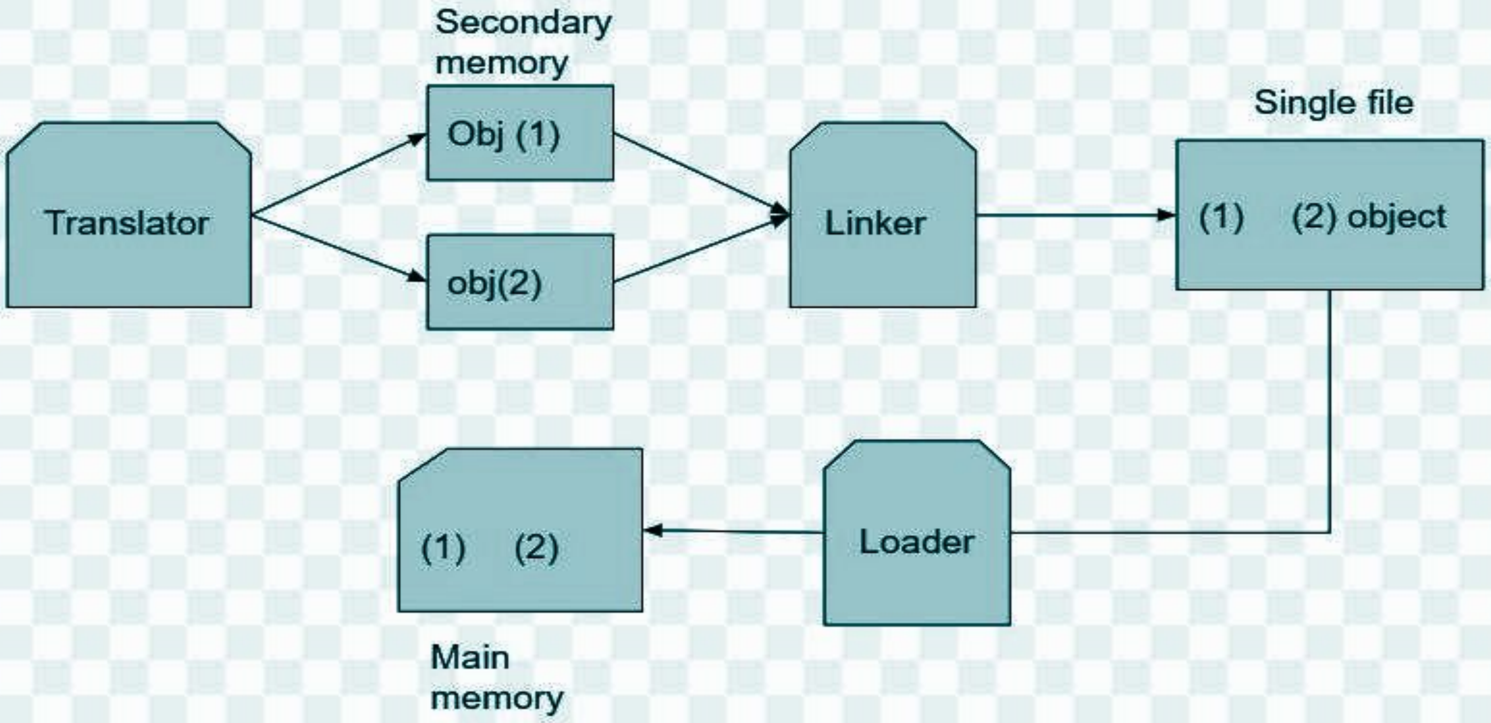
Secondary memory
Obj (1)
obj(2)
Translator
Linker
Single file
(1) (2) object
Loader
(1) (2)
Main memory

Fig : LINKER

Linker (Contd.)

- **Functions of a Linker:**
- **Symbol Resolution:** The linker resolves symbols, or identifiers, used in one object file but defined in another.
- **Relocation:** It adjusts memory addresses for code and data so that they can be loaded correctly into memory.
- **Library Handling:** The linker also links in necessary library code (such as standard libraries) that the program depends on.

Linker (Contd.)

- **Types of Linkers:**
- **Static Linker:** Links object files and libraries at compile time to create a single executable. Once linked, it cannot be changed unless recompiled.
- **Dynamic Linker:** Links libraries and other object files at runtime, allowing more flexibility but requiring that libraries be available on the system when the program is executed.

Linker (Contd.)

- A program execution consists of the following steps:
- **Translation**: A program is translated into a target program
- **Linking**: The code of a program is combined with codes of those programs and library routine that it calls.
- **Relocation**: It is the action of changing memory addresses used in the code of the program so that it can execute correctly in the allocated memory area.
- **Loading**: The program is loaded in a specific memory area for execution

Linker (Contd.)

- The following terms are used related to the addresses:
- **Translated address:** Address assigned by the translator
- **Linked address**: Address assigned by the linker
- **Load address:** Address assigned by the loader
- **Translated origin**: Address of the origin used by the translator. It is either the address specified by the programmer in ORIGIN or START statement or a default value
- **Linked origin**: Address of the origin assigned by the linker while producing a binary program.
- **Load origin:** Address of the origin assigned by the loader while loading the program in memory for execution.

Relocation of Linking Concept

- Relocation is the process of adjusting memory addresses in a program to ensure that code and data references are resolved correctly when a program is loaded into memory for execution.
- **Why is Relocation Needed?**
- **Flexible Memory Allocation:** Programs can't always predict where they'll be loaded in memory.
- **Modular Development:** Programs are written and compiled in parts (modules), so addresses need adjustment when combined.
- **Dynamic Libraries:** Shared libraries need to work regardless of where they're loaded.

Relocation of Linking Concept (Contd.)

- **Key Concepts & Terms**
- **Relocatable Code:**
- Code that doesn’t rely on fixed memory addresses.
- Example: Relative or symbolic addresses instead of hardcoded absolute addresses.
- **Absolute Address:**
- A fixed memory location, e.g., 0x4000. Needs adjustment during relocation if the program isn’t loaded at the expected address.
- **Relative Address:**
- An offset from a base address, e.g., Base + Offset. Easier to adjust during relocation.
- **Symbol Table:**
- A table containing information about symbols (variables, functions) and their locations.
- **Relocation Table:**
- A table that records parts of the program needing address adjustments (e.g., where symbols are referenced).

Relocation of Linking Concept (Contd.)

- Let **AA** be the set of Absolute Addresses (Address of instruction or data) of a program P.
- An address sensitive program P, contains An address sensitive instruction: instruction address a, included in the set AA An address constant: data word address a, included in the set AA.
- If **linked origin != translated origin** => relocation must be performed by the linker.
- If **load origin != linked origin** => relocation must be performed by the loader

Algorithm for Relocation in Linkers

- **Purpose**: Adjust memory addresses at link-time to produce an executable with fixed addresses.
- **Algorithm**:
- **Input**:
  - Object files with relocatable code, symbol table, and relocation table.
- **Steps:**
  - **Read Object Files:**
    - Parse object files to extract sections, symbols, and relocation table entries.
  - **Resolve Symbols:**
    - Assign addresses to all global symbols based on the memory layout.
    - Example: Place the .text section first, followed by .data and .bss.

Algorithm for Relocation in Linkers (Contd.)

- **Adjust Relocatable Addresses:**
  - For each entry in the relocation table:
  - Identify the reference type (code/data) and location.
  - Add the symbol's resolved address to the base address of the program.
  - Update the instruction or data reference in the code.
- **Generate Output:**
  - Create an executable file with all addresses fixed.
- **Output**:
  - A fully linked executable with absolute addresses.

Design of Linker

- A linker is designed to combine multiple object files into a single executable file. Its architecture is modular, involving several components that perform specific tasks.

1. **Input to the Linker**

- The linker takes the following inputs:
- **Object Files**: Contain machine code generated by the compiler for each module.
- **Symbol Table**: Lists symbols (variables, functions, etc.) defined or referenced in the object files.
- **Relocation Information**: Details memory addresses requiring adjustments.

Design of Linker (Contd.)

2. Components of a Linker

- **A. Symbol Resolution Unit**
  - Resolves external references by matching symbols between object files.
  - Tasks:
    - Identify symbols defined in one object file but used in others.
    - Replace undefined references with actual memory addresses or offsets.
- **B. Relocation Handler**
  - Adjusts memory addresses in code and data sections based on the program’s placement in memory.
  - Uses the relocation table to identify where changes are needed.
- **C. Section Merger**
  - Combines sections of the same type from multiple object files into a single section.
  - .text → Code Section
  - .data → Initialized Data
  - .bss → Uninitialized Data
  - Ensures consistent ordering to prevent address conflicts.

Design of Linker (Contd.)

- **D. Library Manager**
  - Handles linking to external libraries:
  - Static Libraries: Combines required code into the executable.
  - Dynamic Libraries: Creates placeholders for functions, resolved at runtime.
- **E. Error Checker**
  - Validates the correctness of:
  - Undefined symbols that cannot be resolved.
  - Addressing errors due to overlapping memory allocation.
  - Conflicts in section merging.
- **F. Output Generator**
  - Produces the final executable file:
  - Contains absolute addresses for symbols.
  - Includes relocation and symbol tables if dynamic linking is used.

Self-Relocating Programs

- Self-relocating programs are designed to modify their own code and data addresses during execution to adapt to different memory locations. Unlike conventional programs that rely on external software (like a loader or linker) for relocation, self-relocating programs handle relocation internally.
- **WORKING:**
- The program is loaded into memory at an arbitrary location.
- A small section of the program, typically located at the start, adjusts memory addresses of the program's code and data.
- The relocation code identifies all addresses that need adjustment.
- The program updates all absolute addresses (e.g., function calls, variable references) based on the actual starting location in memory.
- After relocation is complete, the program begins normal execution with all addresses adjusted.

Linking in Linux

- Linking in Linux is the process of combining multiple object files and libraries into an executable program. This process can happen in two stages: **static linking and dynamic linking**. Linux uses the GNU toolchain, primarily the GNU linker (ld) and the dynamic linker (ld-linux.so), to handle linking.
- **Static Linking:**
- Combines all required object files and libraries into a single executable at link time.
- The resulting binary is self-contained and doesn’t depend on external libraries during execution.
- Used in scenarios where portability and independence are essential (e.g., embedded systems).
- **Dynamic Linking:**
- Links to shared libraries (dynamic libraries) at runtime.
- Shared libraries are loaded into memory when the program starts.
- Reduces the size of the executable and allows sharing of common libraries among multiple programs.

Linking Overlay Structured Programs

- Overlay-structured programs are designed to run in environments with limited memory by dynamically loading only the necessary portions of the program into memory.
- Overlays help reduce the memory footprint of large programs by dividing them into smaller modules, or "overlays," which are loaded and executed as needed.
- Linking overlay-structured programs involves special handling to ensure that the overlays are correctly managed in memory, their dependencies are resolved, and they can access shared resources effectively.

Linking Overlay Structured Programs (Contd.)

- **Structure:**
- The program is divided into overlays based on functionality or dependencies.
- Each overlay represents a logical module that can be loaded into a predefined memory area when required.
- Common sections (e.g., global variables) are kept in a resident area to allow shared access.
- **Memory Layout:**
- **Resident Area**: Contains code and data always available in memory, like shared routines and global data.
- **Overlay Area:** A reserved region where different overlays are loaded and replaced dynamically.

Linking Overlay Structured Programs (Contd.)1

- **ADVANTAGES:**
- **Memory Efficiency:**
- Allows large programs to run in systems with limited memory by keeping only the required overlay in memory.
- **Modularity**:
- Easier to develop and debug smaller overlays than a monolithic program.
- **Dynamic Loading:**
- Reduces the overall memory footprint by loading overlays on demand.

Dynamic Linking

- Dynamic linking is the process of deferring the resolution of program dependencies on libraries until runtime.
- Unlike static linking, where all required code is embedded into the executable, dynamic linking creates a smaller executable that relies on shared libraries, which are loaded into memory as needed.

Dynamic Linking (Contd.)

How Dynamic Linking Works

1. **Compile-Time**:
   - The program is compiled into an object file with unresolved references to shared libraries.
   - A list of required shared libraries is embedded in the executable.
2. **Link-Time (Linking Phase)**:
   - During linking, the linker (`ld`) includes references to dynamic libraries but doesn't include their code in the executable.
   - The resulting executable contains placeholders (stubs) for the library functions and symbols.
3. **Load-Time**:
   - When the program is executed, the dynamic linker (`ld-linux.so` on Linux) resolves the unresolved symbols by loading the required shared libraries into memory.
   - Memory addresses for functions and variables are patched in the executable at runtime.
4. **Run-Time**:
   - If new shared libraries are needed during execution, they can be loaded dynamically using system calls like `dlopen()`.

Loader

- A loader is the system software responsible for loading the executable file produced by the linker into the system memory so it can be run by the CPU. The loader performs the final step of preparing a program for execution.
- **Functions of a Loader:**
- **Loading:** Loads the executable code into the appropriate memory locations for execution.
- **Relocation:** Similar to the linker, the loader adjusts addresses for memory references in the executable, resolving them to match the actual memory addresses where the program will reside.
- **Program Execution:** After loading and relocating the program, the loader passes control to the program, initiating its execution.

Loader (Contd.)

- **Types of Loaders:**
- **Absolute Loader:** A simple loader that places the program at a specific location in memory without any further adjustments.
- **Relocating Loader:** Adjusts address-dependent locations, allowing the program to be loaded into different memory locations.
- **Dynamic Loader:** Loads parts of the program as needed during execution, supporting features like dynamic linking and demand paging.

Different Loading Schemes

- A loading scheme is a strategy used to load an executable program into memory for execution.
- Different schemes vary based on system architecture, memory management, and program requirements.

Absolute Loading

- **Description:**
  - The **absolute loader** places a program into memory at a specific, predetermined address.
  - Requires that the program is already assembled or compiled for a specific memory location.
- **Process:**
  - The assembler or compiler generates an **absolute object file** with fixed addresses for all instructions and data.
  - The loader reads the object file and places the program directly into memory at the specified address.
- **Advantages:**
  - Simple and fast.
  - No relocation or address adjustment is required.
- **Disadvantages:**
  - Lack of flexibility; the program must always run at the same memory address.
  - Inefficient memory usage in multiprogramming systems.
- **Use Case:**
  - Embedded systems and single-program environments.

Dynamic Loading

- **Description:**
  - In dynamic loading, parts of a program are loaded into memory only when they are needed.
  - Reduces memory usage by avoiding loading unused modules.
- **Process:**
  - Initially, only the main program is loaded.
  - As execution proceeds, required modules or routines are loaded dynamically.
- **Advantages:**
  - Efficient memory utilization.
  - Reduces program load time.
- **Disadvantages:**
  - Runtime overhead due to on-demand loading.
  - Additional complexity for managing dynamic memory.
- **Use Case:**
  - Large programs with infrequently used modules.
  - Operating systems with limited memory, such as mobile or IoT devices.

Relocatable Loading

- **Description:**
  - A **relocatable loader** allows a program to be loaded at any memory location.
  - The program uses **relocation information** (like a relocation table) to adjust addresses during loading.
- **Process:**
  - The compiler or assembler generates a **relocatable object file**.
  - The loader reads relocation information and adjusts addresses to fit the allocated memory segment.
- **Advantages:**
  - Flexibility in program placement.
  - Efficient memory usage in multitasking environments.
- **Disadvantages:**
  - Slightly more complex than absolute loading.
  - Requires relocation overhead during loading.
- **Use Case:**
  - General-purpose operating systems with multiprogramming.

Sequential and Direct Loaders

1. **Sequential Loader**

- **Definition:**
- A sequential loader loads the program into memory sequentially, starting from the beginning and continuing linearly until the entire program is in memory.
- Working:
- The program is read from storage in the order it is stored.
- Instructions and data are loaded into consecutive memory locations.
- There is no random access; the loader processes the program as a single linear entity.

Sequential and Direct Loaders (Contd.)

- **Process:**
- The loader reads the program header for size and location information.
- The loader sequentially transfers the program's code and data into memory, starting at the specified base address.
- Once the transfer is complete, control is transferred to the program's starting instruction.
- **Advantages:**
- Simplicity:
- Easy to implement and requires minimal logic.
- Efficiency for Simple Programs:
- Works well when the program is small and contiguous in memory.
- Predictable Memory Usage:
- All parts of the program are loaded in memory, ensuring availability at runtime.

Sequential and Direct Loaders (Contd.)

- **Disadvantages**:
- Inefficient Memory Usage:
- Entire program must fit into memory, even if some parts are unused.
- Lack of Flexibility:
- No ability to selectively load specific sections of the program.
- Slow for Large Programs:
- Sequential access is time-consuming for large programs.
- **Use Case:**
- Suitable for simple, standalone systems with limited memory (e.g., early operating systems, embedded systems).

Sequential and Direct Loaders (Contd.)

2. Direct Loader

- **Definition:**
- A direct loader loads specific parts of the program into memory directly, often based on pre-determined addresses or based on instructions in a relocation table. It can use random access to load program segments more efficiently.
- **Working:**
- The loader reads only the necessary portions of the program.
- Code and data segments are directly loaded into their respective memory locations.
- Symbol resolution and relocation may occur during loading.

Sequential and Direct Loaders (Contd.)

- **Process:**
- The program file includes metadata like a symbol table and relocation information.
- The loader identifies the memory requirements and allocation for the program's segments.
- Program sections are loaded directly into the allocated memory locations, skipping unused parts.
- **Advantages:**
- Efficient Memory Utilization:
- Only the required parts of the program are loaded, reducing memory usage.
- Random Access:
- Can load program segments independently and in parallel.
- Supports Dynamic Loading:
- Works well with large programs and dynamic libraries.

Sequential and Direct Loaders (Contd.)

- **Disadvantages:**
- Complexity:
- Requires more logic to manage relocation and segment dependencies.
- Higher Overhead:
- Loading may involve extra computations for symbol resolution and relocation.
- **Use Case:**
- Suitable for modern multitasking operating systems and systems supporting dynamic loading (e.g., Linux, Windows).

Compile-and-Go Loader

- A Compile-and-Go Loader, also known as a load-and-go loader, is a simple type of loader that combines the processes of compilation, assembly, and loading into one step.
- It translates the source code into machine code and immediately places it in memory for execution without generating an intermediate object file.

Compile-and-Go Loader (Contd.)

- **<u>How It Works</u>**
- **Source Code Input:**
  - The programmer provides the source code written in a high-level or assembly language.
- **Compilation/Assembly:**
  - The Compile-and-Go loader translates the source code directly into machine code. This step involves:
  - Lexical analysis
  - Syntax analysis
  - Code generation
- **Loading:**
  - The generated machine code is directly loaded into memory at a specific location.
- **Execution:**
  - The program starts executing immediately after it is loaded.

Compile-and-Go Loader (Contd.)

- **Advantages**
- **Simplicity:**
  - The entire process is performed in one step without requiring external tools for linking or loading.
- **No Intermediate Files:**
  - Saves disk space by not creating object files.
- **Fast for Small Programs:**
  - Suitable for small programs where the entire code is assembled and loaded quickly.
- **Ideal for Development:**
  - Useful for debugging and educational purposes, where quick compilation and execution are desired.

Compile-and-Go Loader (Contd.)

- **Disadvantages**
- **Inefficiency in Recompilation:** Any change to the program requires recompiling the entire code, even if only a small part is modified.
- **Poor Memory Utilization:** The machine code and the compiler/assembler occupy memory simultaneously, limiting the space available for larger programs.
- **Lack of Modularity:** Cannot handle modular or large programs effectively since there is no support for separate compilation and linking.
- **Execution Overhead:** The compiler or assembler must remain in memory during execution, which can be inefficient.
- **Error Handling:** Errors in the source code may not be detected until runtime, making debugging harder for complex programs.

thank you