Q1. Explain Differences between Static Link Library and Dynamic Link Library

A **library** in software engineering is a collection of precompiled routines that a program can use. The way these libraries are linked to the main program affects program size, efficiency, and maintainability. Broadly, there are two types: **Static Link Libraries (SLL)** and **Dynamic Link Libraries (DLL)**.

Static Link Library (SLL)

A static link library is linked to a program during the **compilation stage**. Once the program is compiled, the machine code from the static library is copied directly into the executable file. This creates a standalone executable that contains everything required to run without external dependencies.

Characteristics:

1. Linking happens at **compile time**.
2. Library code is merged with the application, producing a **single, independent executable file**.
3. Every program that uses a static library gets its own copy of the routines.

Advantages:

- **Portability**: Once compiled, the program can run on any system without worrying about missing libraries.
- **Performance**: Since all code is already in the executable, there's no runtime overhead of searching for libraries.
- **Reliability**: No risk of version mismatches or missing DLL errors.

Disadvantages:

- **Large Executable Size**: Code from the library is copied into every program that uses it, leading to redundancy.
- **Recompilation Required**: If the library changes (e.g., bug fixes), every program that uses it must be recompiled.
- **Memory Wastage**: When multiple programs are running, each has its own copy of the library routines in memory.

Use Cases:

- Embedded systems where external dependencies are not practical.
- Programs that must be distributed as single standalone files.

Dynamic Link Library (DLL)

A dynamic link library, on the other hand, is linked to the program during **execution (runtime)**. The executable contains only references (addresses) to the library routines. The actual library is stored separately, and when the program runs, the operating system loads the library into memory and binds it with the executable.

Characteristics:

1. Linking happens at **runtime**.
2. Executable file contains references to the DLL rather than the actual code.
3. Multiple applications can share the same library in memory.

Advantages:

- **Reduced File Size**: Since library code is not embedded, the executable is smaller.
- **Memory Efficiency**: A DLL loaded once can be shared among multiple applications.
- **Easy Maintenance**: Updating the DLL automatically updates all applications that use it.
- **Flexibility**: Supports modular design, where optional features can be packaged separately.

Disadvantages:

- **Runtime Dependency**: The program won't run if the required DLL is missing or corrupted.
- **Compatibility Issues**: Different applications may require different versions of the same DLL (the classic "DLL Hell").
- **Slight Performance Overhead**: Dynamic binding introduces some delay during program startup.

Use Cases:

- Modern operating systems (e.g., Windows uses DLLs extensively).
- Large applications where modular updates are important (e.g., Microsoft Office shares common DLLs among all apps).

Key Differences

Aspect	**Static Link Library (SLL)**	**Dynamic Link Library (DLL)**
Linking Time	Compile time	Runtime
Executable Size	Large	Small
Memory Usage	Each program has its own copy	Shared across multiple programs
Updates	Requires recompilation	DLL can be updated independently
Performance	Faster execution	Slight runtime overhead
Dependency	Self-contained	Depends on DLL presence at runtime

Conclusion

Both static and dynamic linking have their own significance. Static linking provides reliability and portability but at the cost of size and memory redundancy. Dynamic linking, on the other hand, saves space and allows modular updates, but it introduces runtime dependencies. In real-world systems, **both**

methods are often combined—some critical parts are statically linked, while optional or frequently updated parts are dynamically linked.

Q2. What are the Different Types of Loaders? Explain Compile and Go Loader in Detail

A **loader** is a system program that loads an object program into memory and prepares it for execution. Without loaders, object code would need to be manually placed in memory by the programmer, which is impractical. Different loader schemes exist to meet different system requirements.

Types of Loaders

1. **Absolute Loader**
    - Loads the program into a fixed memory address defined by the programmer.
    - Simple, but inflexible since program must always run at the same location.
2. **Relocating Loader**
    - Modifies program addresses so it can be loaded at any memory location.
    - Useful in multiprogramming environments where memory allocation changes dynamically.
3. **Direct Linking Loader**
    - Handles multiple object modules.
    - Resolves external references and adjusts addresses automatically.
    - Used when large programs are divided into modules.
4. **Compile-and-Go Loader**
    - Compilation and loading are combined in one step.
    - No object file is produced; program is directly compiled into memory.
5. **Bootstrap Loader**
    - Special loader stored in ROM that loads the operating system into memory when the computer starts.

Compile-and-Go Loader in Detail

The Compile-and-Go loader is one of the simplest loader schemes, primarily used in early computing systems.

Working:

1. The source code is submitted to the compiler.

2. Compiler translates it into machine code and directly stores it in memory instead of creating an object file.
3. Once compilation is finished, control is transferred to the starting address of the program in memory.
4. Program begins execution immediately.

This eliminates the need for a separate loader program since the compiler itself handles loading.

Advantages

1. **Simplicity**: No need for separate loader or linker.
2. **Speed**: Execution begins immediately after compilation.
3. **Useful for Small Programs**: Handy in academic or development settings where quick testing is needed.

Disadvantages

1. **Recompilation Overhead**: Every time the program is run, it must be recompiled, even if there are no changes.
2. **Memory Inefficiency**: For large programs, memory management is poor since all compilation output stays in memory.
3. **No Reusability**: Object code is not saved; each run starts from scratch.
4. **Difficult Debugging**: Since object files are absent, debugging tools have less information to work with.

Example

Suppose a small program is written to add two numbers. In a compile-and-go system:

- The compiler directly places machine code instructions into memory (say from address 2000H).
- Once compiled, control jumps to 2000H.
- The addition routine executes without needing a loader.

In contrast, a modern system would generate an object file, then a loader would load it into memory.

Conclusion

The compile-and-go loader is straightforward and efficient for small-scale programs but unsuitable for large or complex systems due to its memory and recompilation drawbacks. While obsolete today, it played a significant role in early system design and laid the foundation for more advanced loader schemes.

Q3. List and Explain Different Loader Schemes in Detail

A **loader** is an essential part of system software that loads machine code into memory for execution. Over the years, different loader schemes were designed to balance performance, flexibility, and ease of use.

---

1. Absolute Loader

- The simplest loader scheme.
- It loads the program exactly at the memory location specified in the object code.
- No relocation or linking is performed.

Advantages:

- Very simple to implement.
- Minimal overhead.

Disadvantages:

- Program must be written for a specific memory address.
- Difficult to run multiple programs in memory simultaneously.

---

2. Relocating Loader

- A more flexible loader that can load a program into any memory location.
- Adjusts addresses in the program to fit the actual memory location assigned at runtime.

Advantages:

- Enables multiprogramming.
- Same object code can run in different memory partitions.

Disadvantages:

- Slightly more complex than absolute loader.

---

3. Direct Linking Loader

- Handles multiple object modules.
- Resolves **external references** between modules.
- Performs relocation as needed.

Advantages:

- Supports modular programming.
- Libraries and external routines can be easily managed.

Disadvantages:

- More complex.
- Requires data structures like External Symbol Table (EST).

---

4. Compile-and-Go Loader

- Compiler generates machine code directly in memory.
- No object file is created.

Advantages:

- Simple and fast for small programs.

Disadvantages:

- Inefficient for large programs.
- Code must be recompiled every time.

---

5. Bootstrap Loader

- Special loader permanently stored in ROM.
- Used during system startup to load the operating system into memory.

Advantages:

- Essential for system booting.

Disadvantages:

- Very limited in functionality (just loads OS).

---

6. Dynamic Loading

- Program modules are loaded **only when required**.
- Saves memory since unused modules are never loaded.

Advantages:

- Efficient memory use.
- Faster startup for large programs.

Disadvantages:

- More complicated OS support required.

Conclusion:
Loader schemes evolved from simple absolute loaders to sophisticated linking loaders and dynamic loading mechanisms. Each scheme has a role: absolute loaders for simplicity, relocating loaders for multiprogramming, linking loaders for modularity, and dynamic loaders for memory efficiency.

Q4. Explain Design of Direct Linking Loaders and Explain Required Data Structures

The **Direct Linking Loader (DLLD)** is one of the most widely used loader schemes in modern systems. It supports modular programming by resolving external references, relocating addresses, and loading modules into memory.

Functions of Direct Linking Loader

1. **Loading**: Reads object code into memory.
2. **Relocation**: Adjusts memory addresses according to allocated space.
3. **Linking**: Resolves external symbols across different object modules.
4. **Execution**: Transfers control to the start of the program.

Data Structures Required

1. **External Symbol Table (EST)**
   - Stores external symbols (function names, variables) and their addresses.
   - Used to resolve references between modules.
2. **Relocation Dictionary (RLD)**
   - Maintains information about addresses in object code that need to be modified during relocation.
3. **Text Records**
   - Contain actual machine code to be loaded into memory.
4. **Loader Map**
   - A map showing memory allocation for different modules and symbols.

Working of Direct Linking Loader

Pass 1 (Analysis Phase):

- Reads object modules.
- Builds the External Symbol Table (EST) by collecting all defined symbols.
- Allocates memory space for each module.

Pass 2 (Loading and Linking Phase):

- Reads each object module again.
- Uses EST to resolve external references.
- Relocates addresses based on memory allocation.
- Loads machine code into memory.
- Transfers control to the starting address.

Example

Suppose two modules are given:

- **Module 1** defines symbol A and references B.
- **Module 2** defines symbol B and references A.

During Pass 1:

- EST is built: A → 2000, B → 2500.

During Pass 2:

- Module 1 references to B are replaced with 2500.
- Module 2 references to A are replaced with 2000.

Thus, both modules are correctly linked and loaded.

Advantages

- Supports modularity.
- Allows independent compilation of modules.
- Facilitates use of standard libraries.

Disadvantages

- More complex implementation.
- Requires additional processing time during loading.

Conclusion:
The Direct Linking Loader is powerful because it combines linking, relocation, and loading into one process. With EST, RLD, and text records, it supports modular and large-scale software development efficiently.

Q5. What is the Need of DLL? Explain with Example

A **Dynamic Link Library (DLL)** provides a way for programs to use shared code that can be loaded at runtime. The need for DLLs arises from requirements of modularity, memory efficiency, and ease of updates.

Need for DLL

1. **Memory Sharing**
   - Multiple programs can share the same DLL loaded into memory, reducing overall memory consumption.
2. **Code Reusability**
   - Common routines can be stored in a DLL and reused across applications (e.g., GUI functions, math libraries).
3. **Modularity**
   - Programs can be divided into separate modules. Each DLL can handle a specific function.
4. **Easy Updates and Maintenance**
   - Updating a DLL automatically benefits all programs that depend on it. No need to recompile everything.
5. **Reduced File Size**
   - Since the code is not embedded into executables, the final program files are smaller.

Example

Consider Microsoft Windows:

- DLLs like user32.dll, kernel32.dll, and gdi32.dll contain essential APIs.
- When multiple programs (Word, Excel, Browser) run, they all use these same DLLs without duplicating the code in each executable.

Another example:

- A company might build a dbconnect.dll library for database operations. Any application in the company can reuse the same DLL.

Advantages

- Efficient use of memory.
- Easy to distribute updates.
- Programs become modular and maintainable.

Disadvantages

- Dependency on DLL files at runtime.
- Version conflicts may occur (DLL Hell).

Conclusion:
DLLs solve the problems of redundancy and large executable sizes by allowing modular and shareable code. They are a backbone of modern software systems where code reuse and memory efficiency are crucial.

Q6. What is Absolute Loader? Explain Design of Absolute Loader with Example and Flowchart

The **Absolute Loader** is the simplest form of loader. It loads the object program into the memory at **fixed locations** specified in the program itself. It neither performs relocation nor linking.

Design of Absolute Loader

1. **Input**: Object program containing text records with absolute addresses.
2. **Operation**: Loader reads these instructions and places them directly at specified memory locations.
3. **Transfer**: Control is transferred to the execution start address given in the End record.

Components

- **Header Record**: Specifies program name, length, and starting address.
- **Text Record**: Contains actual object code with memory addresses.
- **End Record**: Specifies program termination and starting execution address.

Flowchart (described in words)

1. Start loader.
2. Read Header Record → Initialize memory.
3. Read Text Records → For each instruction, copy code into specified memory location.
4. Read End Record → Transfer control to start address.
5. Execution begins.

Example

Suppose the object code contains:

- H COPY 001000 00107A → program starts at address 1000.

- T 001000 14 141033 281030 ... → machine code instructions at addresses starting from 1000.
- E 001000 → execution begins at 1000.

The absolute loader will directly copy these instructions into memory starting from address 1000 and then start execution.

Advantages

- Very simple and easy to implement.
- Minimal overhead.

Disadvantages

- Inflexible; program must run at one specific location.
- Not suitable for multiprogramming.
- No support for modular programming.

Q7. Explain Formats of ESD, RLD, TXT and END Cards with Example

In **direct linking loaders**, control cards are used to describe different parts of an object program. These cards are essential for communication between the compiler, assembler, and loader. The main cards are **ESD (External Symbol Dictionary), RLD (Relocation Dictionary), TXT (Text Card), and END Card**.

---

1. External Symbol Dictionary (ESD) Card

- **Purpose**: Defines external symbols and their addresses.
- Contains names of variables and functions that may be referenced in other modules.
- Loader uses ESD to resolve external references during linking.

Format:
[ESD | Symbol Name | Address | Length | Flags]

Example:
If module A defines a symbol X at address 2000 of length 4 bytes, the ESD card may look like:
ESD X 2000 04 D
(D stands for Defined symbol).

---

2. Relocation Dictionary (RLD) Card

- **Purpose**: Identifies addresses in object code that must be modified during relocation.
- Since programs may not always load at the same memory address, relocation ensures addresses point to the right location.

Format:
[RLD | Address Field | Type | Symbol Reference]

Example:
If an instruction refers to symbol Y which is external, an RLD entry would record the memory location of the instruction and the fact that it must be adjusted once Y's address is known.

---

3. Text (TXT) Card

- **Purpose**: Contains the actual object code to be placed into memory.
- Each TXT card corresponds to a block of consecutive instructions or data.

Format:
[TXT | Starting Address | Length | Object Code]

Example:
TXT 2000 10 141033 281030
→ Load instructions 141033 and 281030 starting from address 2000.

---

4. End (END) Card

- **Purpose**: Indicates end of object program.
- Also specifies the starting execution address.

Format:
[END | Start Address]

Example:
END 2000
→ Loader knows to transfer control to address 2000 after loading finishes.

Summary of Functions

Card	Function	Example
ESD	Defines symbols	ESD X 2000 04 D
RLD	Specifies relocation	RLD 2020 ADDR Y
TXT	Contains object code	TXT 2000 141033
END	Marks end of program	END 2000

Conclusion:
These cards make direct linking loaders possible by conveying symbol definitions, relocation needs, and actual object code in a standardized format.

Q8. Explain General Loading Scheme with Advantages and Disadvantages

The **general loading scheme** is a standard approach followed by loaders to bring a program into memory and prepare it for execution. It typically includes steps for **compilation, linking, relocation, and loading**.

Steps in General Loading Scheme

1. **Compilation Phase**
    - Source program is translated into object code.
    - Symbol tables and control cards (ESD, RLD, TXT, END) are generated.
2. **Pass 1 (Analysis Phase of Loader)**
    - Loader scans object modules.
    - Builds External Symbol Table (EST).
    - Allocates memory to each module.
3. **Pass 2 (Loading and Linking)**
    - Loader reads object code.
    - Relocates addresses based on memory allocation.
    - Resolves external references using EST.
    - Loads actual machine instructions into memory.
4. **Transfer Control**
    - Loader jumps to the starting address given in END card.

Advantages

1. **Supports Modular Programming**
    - Programs can be divided into multiple modules.
    - Easier debugging and development.
2. **Memory Flexibility**
    - Loader can place modules anywhere in memory.
3. **Code Reusability**
    - Libraries can be linked dynamically.
4. **Efficient Program Execution**
    - Ensures that all references and addresses are properly resolved.

Disadvantages

1. **Complexity**
    - Loader design is complicated compared to simple loaders.
2. **Execution Overhead**
    - Linking and relocation take extra time during loading.
3. **Data Structure Overhead**
    - Requires EST, RLD, Loader Map, etc.

Conclusion:
The general loading scheme provides a systematic and flexible way to prepare programs for execution, balancing modularity and efficiency at the cost of complexity.

Q9. Give Complete Design of Direct Linking Loader

A **Direct Linking Loader (DLLD)** is designed to handle external references, relocation, and loading in a single step. Its design includes multiple passes and specialized data structures.

Design Steps

Pass 1:

- Read object modules.
- Construct External Symbol Table (EST).
- Allocate memory space to modules.

Pass 2:

- Re-read object modules.
- Use EST to resolve external references.
- Modify instructions as per Relocation Dictionary (RLD).
- Load machine code into memory.
- Transfer control to program start address.

Data Structures

- **EST**: For external symbol resolution.
- **RLD**: For relocation details.
- **Loader Map**: For memory allocation visualization.
- **Text Records**: For actual machine code.

Example

If module A calls subroutine B (defined in module B):

- Pass 1 records that B is defined at address 2500.
- Pass 2 modifies call in module A to point to 2500 before loading.

Advantages

- Supports modularity.
- Allows use of common subroutine libraries.
- Flexible memory allocation.

Disadvantages

- Requires complex data structures.
- Takes more time compared to simple loaders.

Conclusion:
The direct linking loader is powerful for large-scale software systems where modularity and reusability are critical. Its design ensures that programs are correctly linked and relocated before execution.

Q10. Give Complete Design of Absolute Loader with Example

The **Absolute Loader** design is straightforward since it assumes the program is always loaded at a fixed memory location.

Steps

1. Read Header Record → Get program length and starting address.
2. Read Text Records → Place object code directly into specified addresses.
3. Read End Record → Transfer control to specified start address.

Example

- Header: H TEST 001000 00107A → Start at 1000.
- Text: T 001000 14 141033 281030 → Instructions stored at 1000H and onwards.
- End: E 001000 → Execution begins at 1000H.

Loader directly copies instructions to memory without relocation.

Advantages

- Very simple and fast.
- Minimal overhead.

Disadvantages

- Inflexible; program cannot run elsewhere.
- No support for modularity.
- No relocation or linking.

Conclusion:
While outdated today, the absolute loader was a cornerstone of early computing systems. Its simplicity made it useful in small systems, but modern multiprogramming environments require relocating and linking loaders.

Q11. Explain Types of Schedulers (Short, Long, Medium Term)

In operating systems, a **scheduler** determines which process runs at what time. Schedulers manage CPU utilization, throughput, and response time.

---

1. Short-Term Scheduler (CPU Scheduler)

- Runs most frequently (milliseconds).
- Selects a process from the ready queue and assigns CPU.
- Uses scheduling algorithms like FCFS, SJF, Round Robin, Priority.

Objective:

- Maximize CPU utilization.
- Minimize response time.

Example:
Choosing between two processes waiting in ready queue for CPU.

---

2. Long-Term Scheduler (Job Scheduler)

- Runs less frequently.
- Controls degree of multiprogramming by deciding which jobs enter system from job pool.

Objective:

- Balance between CPU-bound and I/O-bound processes.

- Ensure efficient use of system resources.

Example:
Batch processing systems decide which jobs to admit.

---

3. Medium-Term Scheduler

- Temporarily suspends/resumes processes to improve CPU utilization.
- Moves processes from memory to disk (swapping) and back.

Objective:

- Improve overall system performance by balancing I/O and CPU.

Example:
Suspending a low-priority process to make room for higher-priority tasks.

Conclusion:
Schedulers at different levels ensure fair and efficient process execution. Short-term focuses on CPU, long-term controls job admission, and medium-term manages suspension/resumption.

Q12. Explain Compile and Go Loading Scheme with Advantages and Disadvantages

The **Compile-and-Go loader** combines compilation and execution in one step.

Working

1. Source code → Compiler → Directly converted into machine code in memory.
2. No object file created.
3. Control is transferred to start address immediately.

Advantages

- Simple implementation.
- Fast execution for small programs.
- No separate loader needed.

Disadvantages

- Must recompile every time.
- Memory management inefficient.

- No reusability of object code.
- Debugging is difficult.

Conclusion:
Useful for small programs and academic settings, but unsuitable for large applications due to inefficiency.

Q13. Describe Concept of DLL? How Dynamic Linking Can be Done with or without Import

Concept of DLL

A **Dynamic Link Library (DLL)** is a collection of routines stored separately from the main program, loaded at runtime. It reduces redundancy and allows modular updates.

Dynamic Linking with Import

- Executable declares needed DLL functions in its Import Address Table (IAT).
- OS automatically loads DLL and resolves addresses at runtime.

Example:
A program imports MessageBoxA from user32.dll via IAT.

Dynamic Linking without Import

- Program loads DLL manually at runtime using system calls (e.g., LoadLibrary() and GetProcAddress() in Windows).
- Offers flexibility as functions can be loaded conditionally.

Example:
A media player might load mp4codec.dll only when playing MP4 files.

Conclusion:
DLLs provide modularity and efficiency. Dynamic linking can be automated with import tables or done manually for maximum flexibility.

Q14. Write Short Notes on (i) Subroutine Linkage (ii) Overlays

Subroutine Linkage

- Refers to mechanism of transferring control from main program to subroutine and back.
- Includes passing parameters and saving return addresses.

Methods:

1. Call and Return instructions.
2. Parameter passing by value, reference, or stack.
3. Loader/linker ensures subroutines are properly addressed in memory.

Importance:

- Encourages modularity and code reuse.
- Simplifies debugging and maintenance.

Overlays

- Technique to run programs larger than available memory.
- Program divided into modules (overlays).
- Only required overlay is loaded into memory at a time.

Example:
An old compiler might keep the lexical analysis in one overlay, parsing in another, and code generation in a third.

Advantages:

- Enables execution of very large programs in limited memory.

Disadvantages:

- Requires careful design.
- Slower due to frequent overlay loading.

Conclusion:
Subroutine linkage and overlays were fundamental concepts enabling modular programming and execution of large programs on limited-memory machines. They laid the foundation for modern modular and memory management techniques.