**Linker**

Linker is a program in a system which helps to link object modules of a program into a single object file. It performs the process of linking. 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 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.

*Source code -> compiler -> Assembler -> Object code -> Linker -> Executable file -> Loader*

Linking is of two types:

**1. Static Linking –** It is performed during the compilation of source program. Linking is performed before execution in static linking. It takes collection of relocatable object file and command-line arguments and generates a fully linked object file that can be loaded and run. Static linker performs two major tasks:

LINKER DIAGRAM:

A picture containing text, screenshot, diagram, rectangle

Description automatically generated

* **Symbol resolution –** It associates each symbol reference with exactly one symbol definition. Every symbol has a  predefined task.
* **Relocation –** It relocates code and data section and modifies the symbol references to the relocated memory locations.

The linker copies all library routines used in the program into executable image. As a result, it requires more memory space. As it does not require the presence of library on the system when it is run, so it is faster and more portable. No failure chance and less error chance.

**2. Dynamic linking –** Dynamic linking is performed during the run time. This linking is accomplished by placing the name of a shareable library in the executable image. There are more chances of errors and failures. It require less memory space as multiple programs can share a single copy of the library. Here we can perform code sharing. It means if we are using the same object a number of times in the program, instead of linking the same object again and again into the library, each module shares information of the object with other modules having the same object. The shared library needed in the linking is stored in virtual memory to save RAM. In this linking we can also relocate the code for the smooth running of code but all the code is not relocatable. It fixes the address at run time.

**Features:**

**Symbol resolution:**The linker resolves symbols, such as function and variable names, across different object files and libraries.

**Relocation:**The linker performs relocation, adjusting the addresses of symbols within object files and libraries to match the final address space of the executable program.

**Optimization:**The linker can perform optimization, such as dead code elimination and function inlining, to improve the performance and size of the executable program.

**Library management:** The linker can manage libraries, linking in only the required functions and removing unused code to minimize the size of the executable.

**Debugging information:** The linker can include debugging information in the executable program, making it easier to debug and analyze during development.

**Cross-platform support:**The linker can generate executable programs for different platforms, including different architectures and operating systems.

**Incremental linking:**The linker can perform incremental linking, allowing changes to be made to individual object files without needing to rebuild the entire executable program.

**Versioning:**The linker can support versioning of shared libraries, allowing multiple versions of a library to coexist and preventing compatibility issues.

**Link-time code generation:**The linker can perform link-time code generation, allowing code to be generated during the linking process rather than at compile time.

**Linker scripts:** The linker can use linker scripts, which are configuration files that specify how object files and libraries should be linked together. Linker scripts can also be used to specify the layout of the executable program’s memory.

**Advantages of Linker**

There are several advantages of using a linker in compiler design:

1. **Code Reuse**: A linker allows code to be reused across multiple programs by linking in shared libraries, reducing the amount of code that needs to be written and maintained.
2. **Smaller Executable Files:**Dynamic linkers reduce the size of the executable file by linking libraries at runtime, rather than including them in the executable.
3. **Reduced Memory Footprint:**Dynamic linkers allow multiple programs to share the same library in memory, reducing the overall memory usage of the system.
4. **Reduced Disk Space:** With dynamic linking, the libraries only need to be stored on disk once, instead of being copied into the executable of each program that uses them.
5. **Improved Security:** Dynamic linkers enable the use of protected libraries, which can help prevent unauthorized access or modification of the library code.
6. **Easier to Update Libraries:**Dynamic linkers allow libraries to be updated or replaced without the need to relink the program, making it easier to fix bugs, add new features, or improve performance.
7. **Portability:**Linkers allow multiple object files generated by different compilers or written in different languages to be combined into a single executable file, allowing for more flexibility and portability in program

**Disadvantages of Linker**

1. **Complexity:** Linkers can be quite complex, especially when dealing with large and complex projects. This can make it difficult for developers to understand and troubleshoot issues that may arise.
2. **Symbol resolution:**Linkers must resolve symbols, which are names used in the source code that refer to memory locations. This can be a difficult and time-consuming process, especially when dealing with multiple object files and libraries.
3. **Compatibility issues:** Linkers must be able to work with a variety of object file formats and libraries, which can be challenging. Incompatible object files or libraries can cause errors or crashes during the linking process.
4. **Performance:**Linkers can be resource-intensive and may take a long time to process large projects. This can be a problem for developers working on large-scale projects or for embedded systems with limited resources.
5. **Security:** Linkers can also be a potential security risk if not properly implemented. Insecure linking can lead to vulnerabilities that can be exploited by malicious actors.
6. **Dependency management:**Linkers also require proper dependency management. If the dependencies are not managed properly, the final binary may not run properly or may not run at all.