BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]
Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300126
Name	Pranay Singhvi
Class and Batch	TE Computer Engineering - Batch C
Date	01-05-2024
Lab #	10
Aim	Design linker/loader
Objective	Implement file parsing, symbol table creation, and pretty table representation to analyze C code structure efficiently.
Theory	# 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:

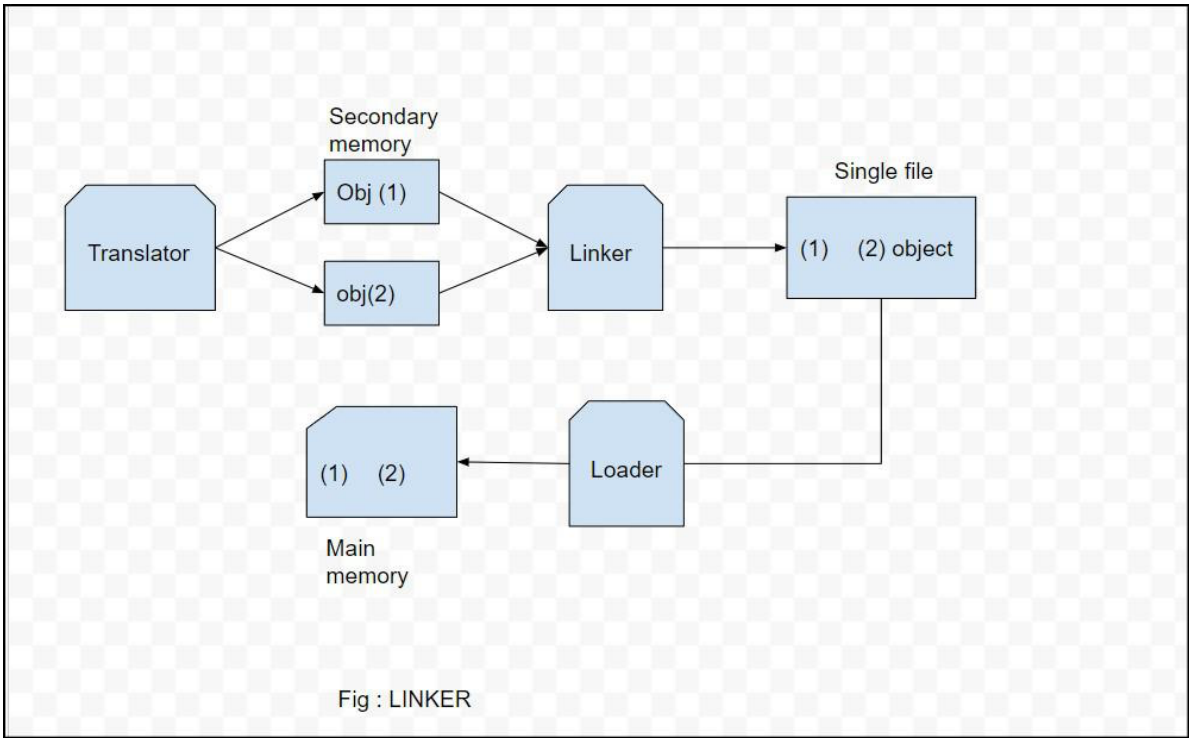


Fig : LINKER

- **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.

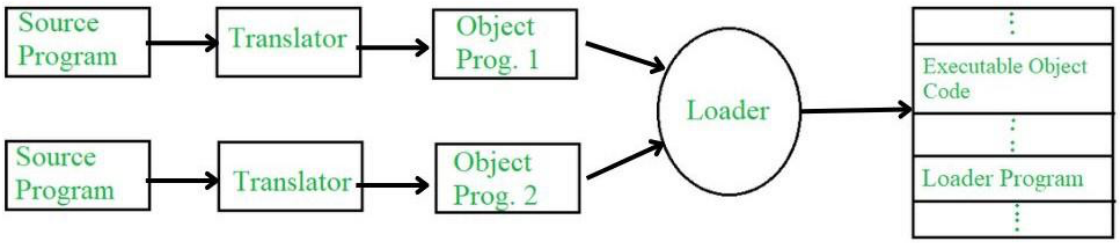
Loader

In compiler design, a loader is a program that is responsible for loading executable programs into memory for execution. The loader reads the object code of a program, which is usually in binary form, and copies it into memory. It also performs other tasks such as allocating memory for the program's data and resolving any external references to other programs or libraries. The loader is typically part of the operating system and is invoked by the system's bootstrap program or by a command from a user. Loaders can be of two types:

- **Absolute Loader:** It loads a program at a specific memory location, specified in the program's object code. This location is usually absolute and does not change when the program is loaded into memory.
- **Relocating Loader:** It loads a program at any memory location, and then adjusts all memory references in the program to reflect the new location. This allows the same program to be loaded into different memory locations without having to modify the program's object code.

Architecture of Loader

Below is the Architecture of the Loader:



The architecture of a loader in a compiler design typically consists of several components:

1. **Source program:** This is a program written in a high-level programming language that needs to be executed.
2. **Translator:** This component, such as a compiler or interpreter, converts the source program into an object program.
3. **Object program:** This is the program in a machine-readable form, usually in binary, that contains both the instructions and data of the program.
4. **Executable object code:** This is the object program that has been processed by the loader and is ready to be executed.

Overall, the Loader is responsible for loading the program into memory, preparing it for execution, and transferring control to the program's entry point. It acts as a bridge between the Operating System and the program being loaded.

Role of Loader in Compilation

In the compilation process, the Loader is responsible for bringing the machine code into memory for execution. It performs the following key functions:

- Loading the executable program into memory from the secondary storage device.
- Allocating memory space to the program and its data.
- Resolving external references between different parts of the program.
- Setting up the initial values of the program counter and stack pointer.
- Preparing the program for execution by the CPU.

The Loader plays a critical role in the compilation process as it ensures that the program is properly loaded into memory, and the necessary memory space is allocated for the program and its data. It also resolves external references between different parts of the program and prepares it for execution by the CPU.

Features of Loaders

- **Relocation:** Loaders can relocate the program to different memory locations to avoid memory conflicts with other programs.
- **Linking:** Loaders can link different parts of the program to resolve external references and create a single executable program.
- **Error Detection:** Loaders can detect and report errors that occur during the loading process.
- **Memory Allocation:** Loaders can allocate memory space to the program and its data, ensuring that the program has enough memory to execute efficiently.
- **Execution Preparation:** Loaders can prepare the program for execution by setting up the initial values of the program counter and stack pointer.
- **Dynamic Loading:** Loaders can load program segments dynamically, allowing the program to only load the necessary segments into memory as they are needed.

Advantages of Loader

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

1. **Memory management:** The loader is responsible for allocating memory for the program's instructions and data. This allows the program to execute in a separate, protected area of memory, which can help prevent errors in the program from affecting the rest of the system.
2. **Dynamic linking:** The loader can resolve external references to other programs or libraries at runtime, which allows for more flexibility in the design of the program. This means that if a library is updated, the program will automatically use the new version without requiring any changes to the program's object code.
3. **Relocation:** The loader can relocate a program to any memory location, which allows for efficient use of memory and prevents conflicts with other programs.

4. **Error handling:** The loader can check the compatibility of the program with the system and handle any errors that occur during loading, such as missing libraries or incompatible instruction sets.
5. **Modularity:** The loader makes it possible to develop and use separate modules or components, which can be linked together to form a complete program. This allows for a more modular design, which can make programs easier to maintain and update.
6. **Reusability:** As the program is separated into different modules, it can be reused in other programs. The loader also allows the use of shared libraries, which can be used by multiple programs.
7. **Ease of use:** The loader provides a simple and consistent interface for loading and executing programs, which makes it easier for users to run and manage programs on their system.

Disadvantages of Loader

There are several disadvantages of using a loader in compiler design:

1. **Complexity:** Loaders can be complex to implement and maintain, as they need to perform a variety of tasks such as memory management, symbol resolution, and relocation.
2. **Overhead:** Loaders introduce additional overhead in terms of memory usage and execution time, as they need to read the object code from storage and perform various operations before the program can be executed.
3. **Size limitations:** Loaders have limitations on the size of the program that can be loaded and might not be able to handle large programs or programs with a lot of external references.
4. **Limited Flexibility:** Loaders are typically specific to a particular operating system or architecture, and may not be easily portable to other systems.
5. **Security:** A poorly designed or implemented loader can introduce security vulnerabilities, such as buffer overflows or other types of memory corruption.
6. **Error handling:** Loaders need to handle various types of errors, such as missing libraries, incompatible object code, and insufficient memory.
7. **Overlapping Memory:** Due to the use of dynamic loading and relocation, the same memory location may be used by multiple programs leading to overlapping memory.

8. **Dependency issues:** Programs might have dependencies on external libraries or other programs, and the loader needs to resolve these dependencies before the program can be executed. This can lead to issues if the required dependencies are not present in the system.

Input File

File a.c:

```c
#include<stdio.h>

void main()
{
    extern int no2;
    extern int no2;
    float f1;
    float f2;
    int d1;
    int d2;
    char e;
}
```

File b.c:

```c
void main()
{
    extern int d1;
    extern int d2;
    extern float f1;
    extern float f2;
    extern char e;
    int no1;
    int no2;
}
```

Implementation/ Code

```python
import os
from prettytable import PrettyTable

class Variable:
    def __init__(self, name, type, size, address):
        self.name = name
        self.type = type
        self.size = size
        self.address = address

class SymbolTable:
    def __init__(self):
        self.variables = []
```

```
def read_file_size_and_content(filename):
    with open(filename, 'rb') as file:
        content = file.read()
        size = os.path.getsize(filename)
    return size, content.decode('utf-8')

def parse_variables(content, symbol_table, start_address):
    lines = content.split('\n')
    for line in lines:
        parts = line.split()
        if len(parts) >= 2:
            type, name = parts[:2]
            size = None
            if type == "int":
                size = 4
            elif type == "char":
                size = 1
            elif type == "float":
                size = 4
            elif type == "double":
                size = 8
            if size is not None:
                symbol_table.variables.append(Variable(name, type, size,
start_address))
                start_address += size

def parse_ext_variables(content, symbol_table):
    lines = content.split('\n')
    for line in lines:
        parts = line.split()
        if len(parts) >= 3 and parts[0] == "extern":
            symbol_table.variables.append(Variable(parts[2], parts[1], 0, -1))

def print_symbol_table(symbol_table):
    table = PrettyTable(["Variable", "Type", "Size", "Address"])
    for variable in symbol_table.variables:
        table.add_row([variable.name, variable.type, variable.size,
variable.address])
    print(table)

def print_symbol_tablet(symbol_table):
    table = PrettyTable(["Variable", "Type"])
    for variable in symbol_table.variables:
        table.add_row([variable.name, variable.type])
    print(table)

def print_symbol_tables(symbol_table1, symbol_table2):
```

```
    table = PrettyTable(["Variable", "Type", "Size", "Address"])
    for variable in symbol_table1.variables:
        table.add_row([variable.name, variable.type, variable.size,
variable.address])
    for variable in symbol_table2.variables:
        table.add_row([variable.name, variable.type, variable.size,
variable.address])
    print(table)

def main():
    memory_size = int(input("Enter the size of memory: "))

    size_a, content_a = read_file_size_and_content("a.c")
    size_b, content_b = read_file_size_and_content("b.c")

    total_size = size_a + size_b

    if total_size > memory_size:
        print("Insufficient memory.")
        return

    symbol_table_a = SymbolTable()
    symbol_table_b = SymbolTable()
    symbol_table_c = SymbolTable()
    symbol_table_d = SymbolTable()

    parse_variables(content_a, symbol_table_a, 1000)
    parse_variables(content_b, symbol_table_b, 5000)

    parse_ext_variables(content_a, symbol_table_c)
    parse_ext_variables(content_b, symbol_table_d)

    print("Symbol Table for a.c")
    print_symbol_table(symbol_table_a)

    print("Symbol Table for extern a.c")
    print_symbol_tablet(symbol_table_c)

    print("Symbol Table for b.c")
    print_symbol_table(symbol_table_b)

    print("Symbol Table for extern b.c")
    print_symbol_tablet(symbol_table_d)

    print("Global variable Table")
    print_symbol_tables(symbol_table_a, symbol_table_b)
```

```
if __name__ == "__main__":
    main()
```

Output

```
pranaysinghvi@Pranays-MacBook-Air 10_ % python3 linker.py
Enter the size of memory: 500
Symbol Table for a.c
+----------+-------+------+---------+
| Variable |  Type | Size | Address |
+----------+-------+------+---------+
|   f1;    | float |  4   |   1000  |
|   f2;    | float |  4   |   1004  |
|   d1;    |  int  |  4   |   1008  |
|   d2;    |  int  |  4   |   1012  |
|    e;    |  char |  1   |   1016  |
+----------+-------+------+---------+
Symbol Table for extern a.c
+----------+------+
| Variable | Type |
+----------+------+
|   no2;   | int  |
|   no2;   | int  |
+----------+------+
Symbol Table for b.c
+----------+------+------+---------+
| Variable | Type | Size | Address |
+----------+------+------+---------+
|   no1;   | int  |  4   |   5000  |
|   no2;   | int  |  4   |   5004  |
+----------+------+------+---------+
Symbol Table for extern b.c
+----------+-------+
| Variable |  Type |
+----------+-------+
|   d1;    |  int  |
|   d2;    |  int  |
|   f1;    | float |
|   f2;    | float |
|    e;    |  char |
+----------+-------+
Global variable Table
+----------+-------+------+---------+
| Variable |  Type | Size | Address |
+----------+-------+------+---------+
|   f1;    | float |  4   |   1000  |
|   f2;    | float |  4   |   1004  |
|   d1;    |  int  |  4   |   1008  |
|   d2;    |  int  |  4   |   1012  |
|    e;    |  char |  1   |   1016  |
|   no1;   |  int  |  4   |   5000  |
|   no2;   |  int  |  4   |   5004  |
+----------+-------+------+---------+
```

Conclusion	I successfully implemented file parsing, symbol table creation, and utilized pretty tables for efficient C code structure analysis.
References	[1] GeeksforGeeks (17 Feb, 2023) Loader in Compiler Design https://www.geeksforgeeks.org/loader-in-compiler-design/ [2] GeeksforGeeks (09 May, 2023) Linker https://www.geeksforgeeks.org/linker/ [3] Github System-Programming https://github.com/Kartik-Katkar/System-Programming/tree/main