**Course - System Programming and Compiler Construction (SPCC)**

| **UID** | 2021300101 |
| --- | --- |
| **Name** | Adwait Purao |
| **Class and Batch** | TE Computer Engineering - Batch B |
| **Date** | 01/05/24 |
| **Lab #** | 10 |
| **Aim** | Design linker/loader |
| **Objective** | Demonstrate a user-driven linker/loader simulation to resolve symbols and load an  executable based on input object file data. |
| **Theory** | **Linkers and Loaders**  **Introduction**  Linkers and loaders are essential components of a computer system that facilitate the execution of programs. They play a crucial role in the process of transforming separate object files into an executable program and loading it into memory for execution. [1, p. 641]  **Linkers**  A linker is a program that combines multiple object files generated by the compiler or assembler, along with necessary library routines, into a single executable file. [2, p. 233] The main tasks performed by a linker are:   1. Symbol Resolution: The linker resolves external references by associating each symbol reference with its corresponding definition in one of the object files or libraries. [1, p. 642] 2. Relocation: The linker assigns final memory addresses to the instructions and data in the object files, adjusting any references accordingly. [1, p. 643] 3. Library Integration: The linker includes only the required library routines in the final executable file, ensuring efficient memory usage. [2, p. 234]   The output of the linker is an executable file that contains the combined and relocated code and data from the input object files and libraries. [1, p. 645]    **Loaders**  A loader is a program responsible for loading the executable file into memory and preparing it for execution. [3, p. 147] The main tasks performed by a loader are:   1. Memory Allocation: The loader allocates memory segments for the code, data, and stack sections of the program. [1, p. 646] 2. Relocation: For relocatable executables, the loader adjusts the memory addresses of instructions and data according to the assigned memory locations. [2, p. 236] 3. Symbol Resolution: In some cases, the loader resolves external references that could not be resolved by the linker, such as dynamic linking of shared libraries. [3, p. 149] 4. Execution Preparation: The loader sets up the necessary data structures and registers for program execution, transferring control to the program's entry point. [2, p. 237]   Loaders can be classified into different types based on when and how they load the program into memory:   1. Bootstrap Loader: This loader is responsible for loading the operating system kernel into memory during system startup. [3, p. 150] 2. Relocating Loader: This loader handles the relocation of instructions and data references in the executable file. [1, p. 647] 3. Dynamic Loader: This loader loads shared libraries or dynamically linked code into memory as needed during program execution. [2, p. 238]   **Differences between Linker and Loader are as follows: [4]**   | LINKER | LOADER | | --- | --- | | The main function of Linker is to generate executable files. | Whereas main objective of Loader is to load executable files to main memory. | | The linker takes input of object code generated by compiler/assembler. | And the loader takes input of executable files generated by linker. | | Linking can be defined as process of combining various pieces of codes and source code to obtain executable code. | Loading can be defined as process of loading executable codes to main memory for further execution. | | Linkers are of 2 types: Linkage Editor and Dynamic Linker. | Loaders are of 4 types: Absolute, Relocating, Direct Linking, Bootstrap. | | Another use of linker is to combine all object modules. | It helps in allocating the address to executable codes/files. | | Linker is also responsible for arranging objects in program’s address space. | Loader is also responsible for adjusting references which are used within the program. | |
| **Implementation / Code** | import struct  class SymbolTable:  def \_\_init\_\_(self):  self.symbols = {}  def add\_symbol(self, name, address):  self.symbols[name] = address  def resolve\_symbol(self, name):  return self.symbols.get(name, None)  class Loader:  def \_\_init\_\_(self):  self.symbol\_table = SymbolTable()  def load\_object\_files(self):  object\_files\_data = []  num\_object\_files = int(input("Enter the number of object files: "))  for i in range(num\_object\_files):  object\_file\_data = input(f"Enter data for object file {i + 1} (format: symbol address symbol address ...): ").split()  object\_files\_data.extend(object\_file\_data)  for i in range(0, len(object\_files\_data), 2):  name = object\_files\_data[i]  address = int(object\_files\_data[i + 1], 16) # Parse address as hexadecimal  self.symbol\_table.add\_symbol(name, address)  def resolve\_symbols(self):  print("Resolving symbols...")  for symbol, address in self.symbol\_table.symbols.items():  print(f"Resolved symbol '{symbol}' to address {address}")  def load\_executable(self):  print("Loading executable...")  print("Executable loaded successfully.")  if \_\_name\_\_ == "\_\_main\_\_":  loader = Loader()  loader.load\_object\_files()  loader.resolve\_symbols()  loader.load\_executable() |
| **Output** |  |
| **Conclusion** | The program prompts the user to input data for object files, then proceeds to resolve symbols and simulate loading an executable into memory. This illustrates a simplified workflow akin to that of a linker/loader. |
| **References** | [1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.  [2] Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.  [3] Tanenbaum, A. S., & Bos, H. (2015). Modern Operating Systems (4th ed.). Prentice Hall.  [4] *Difference between Linker and Loader*. (2020, August 11). GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-linker-and-loader/ |