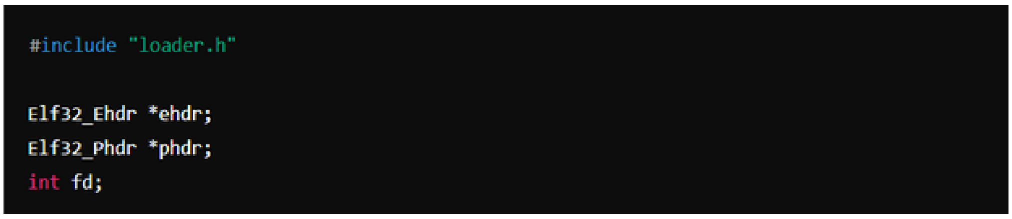
ELF LOADER

This project involves developing a program in C that loads and executes ELF (Executable and Linkable Format) files. ELF is a common file format for executables, object code, shared libraries, and core dumps in Unix-like operating systems. The purpose of this program is to load an ELF file, locate the entry point of the executable code, and execute it.

CODE EXPLANATION

1. Necessary functions and global variables



```c
#include "loader.h"

Elf32_Ehdr *ehdr;
Elf32_Phdr *phdr;
int fd;
```

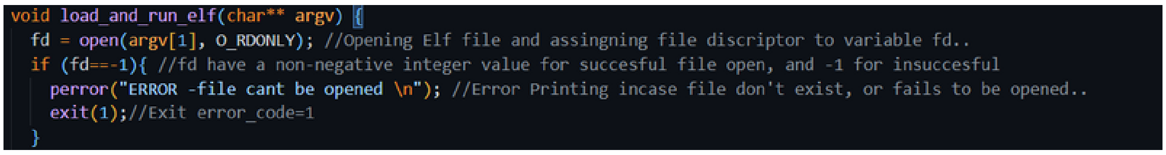
- Header Inclusion: The code begins by including "loader. h", which defines necessary libraries that will be needed in this project.

Global Variables:

- ehdr: A pointer to the ELF header structure (Elf32_Ehdr). The ELF header contains important metadata about the ELF file, such as its type, entry point, and program header table.
- phdr: A pointer to the Program Header (Elf32_Phdr). The program header table contains information needed to create the memory image of the process during execution.
- fd: A file descriptor for the ELF file, which is used for file operations.

2. ELF loader and runner

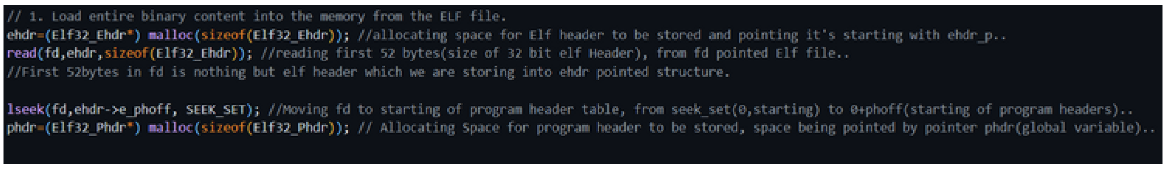
The load_and_run_elf function is the core of this program. It handles the process of loading an ELF file into memory and executing it.



```c
void load_and_run_elf(char** argv) {
  fd = open(argv[1], O_RDONLY); //Opening Elf file and assingning file discriptor to variable fd..
  if (fd==-1){ //fd have a non-negative integer value for succesful file open, and -1 for insuccesful
    perror("ERROR -file cant be opened \n"); //Error Printing incase file don't exist, or fails to be opened..
    exit(1);//Exit error_code=1
  }
```

Opening the ELF File (open(filepath, O_RDONLY)):

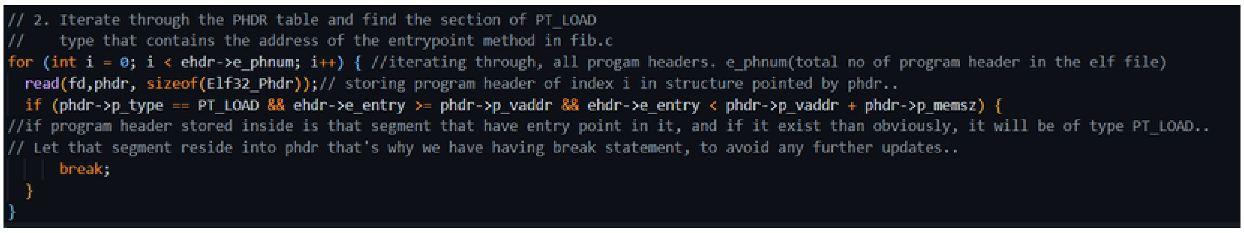
- The open function opens the specified file in read-only mode (O_RDONLY). The file descriptor fd is used for subsequent file operations.
- If the file cannot be opened (e.g., if the file doesn't exist or the process doesn't have permission to read it), open returns -1. The perror function then prints a descriptive error message, and the program exits .



```c
// 1. Load entire binary content into the memory from the ELF file.
ehdr=(Elf32_Ehdr*) malloc(sizeof(Elf32_Ehdr)); //allocating space for Elf header to be stored and pointing it's starting with ehdr_p..
read(fd,ehdr,sizeof(Elf32_Ehdr)); //reading first 52 bytes(size of 32 bit elf Header), from fd pointed Elf file..
//First 52bytes in fd is nothing but elf header which we are storing into ehdr pointed structure.

lseek(fd,ehdr->e_phoff, SEEK_SET); //Moving fd to starting of program header table, from seek_set(0,starting) to 0+phoff(starting of program headers)..
phdr=(Elf32_Phdr*) malloc(sizeof(Elf32_Phdr)); // Allocating Space for program header to be stored, space being pointed by pointer phdr(global variable)..
```

Allocating space to ehdr and phdr:

- malloc : Allocates memory for the ELF header and program header.
- read: Reads the ELF header from the file into the allocated memory.
- lseek: Moves the file pointer to the location of the program header table (e_phoff).

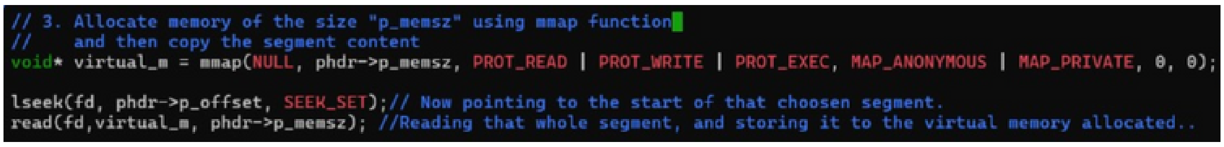


```c
// 2. Iterate through the PHDR table and find the section of PT_LOAD
//    type that contains the address of the entrypoint method in fib.c
for (int i = 0; i < ehdr->e_phnum; i++) { //iterating through, all progam headers. e_phnum(total no of program header in the elf file)
  read(fd,phdr, sizeof(Elf32_Phdr));// storing program header of index i in structure pointed by phdr..
  if (phdr->p_type == PT_LOAD && ehdr->e_entry >= phdr->p_vaddr && ehdr->e_entry < phdr->p_vaddr + phdr->p_memsz) {
//if program header stored inside is that segment that have entry point in it, and if it exist than obviously, it will be of type PT_LOAD..
// Let that segment reside into phdr that's why we have having break statement, to avoid any further updates..
      break;
  }
}
```

This section responsible for determining the correct entry point within the ELF (Executable and Linkable Format) file by iterating over the program headers and checking if the entry point lies within the virtual address range of any loadable segment. This process involves the following steps:

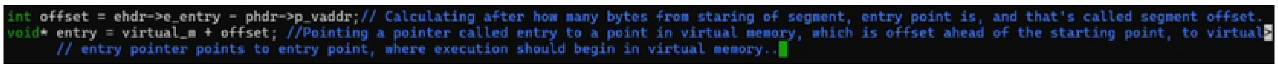
1. **Iteration and Header Reading:**
    - The loop iterates over all program headers (e_phnum), reading each one into phdr using fread.
    - This iteration ensures that every program header is examined to identify the segment that contains the entry point.

2. **Condition Check and Entry Point Adjustment:**
    - The code checks if the segment is of type PT_LOAD, meaning it's a loadable segment.
    - It then verifies if the entry point (e_entry) falls within the segment's virtual address range (p_vaddr to p_vaddr + p_memsz).
    - If this condition is met, the entry point is adjusted to the end of the segment, ensuring that when execution begins, it starts at the correct location.



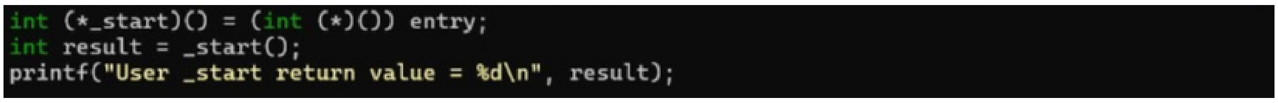
```c
// 3. Allocate memory of the size "p_memsz" using mmap function
//    and then copy the segment content
void* virtual_m = mmap(NULL, phdr->p_memsz, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);

lseek(fd, phdr->p_offset, SEEK_SET);// Now pointing to the start of that choosen segment.
read(fd,virtual_m, phdr->p_memsz); //Reading that whole segment, and storing it to the virtual memory allocated..
```

- **Memory Allocation:**
    - mmap allocates a block of memory in the process's virtual address space. This memory is where the segment will be loaded, allowing the program to access and execute the code, or use the data within this segment.
- **Setting Permissions:**
    - The PROT_READ, PROT_WRITE, and PROT_EXEC flags are crucial because they ensure that the allocated memory has the correct permissions. Without the PROT_EXEC flag, for example, the code in this segment could not be executed.
- **Correct Data Loading:**
    - By moving the file pointer to phdr->p_offset, the program ensures that the next read operation (fread) starts at the correct location in the file. This is critical because the segment's data needs to be loaded exactly as it is stored in the ELF file.
- **Loading Code/Data into Memory:**
    - This step transfers the segment's data from the ELF file into the process's virtual memory.
    - If the segment contains initialized data (e.g., global variables), those values are now set in the appropriate memory locations.



```
int offset = ehdr->e_entry - phdr->p_vaddr;// Calculating after how many bytes from staring of segment, entry point is, and that's called segment offset.
void* entry = virtual_m + offset; //Pointing a pointer called entry to a point in virtual memory, which is offset ahead of the starting point, to virtual
    // entry pointer points to entry point, where execution should begin in virtual memory..
```

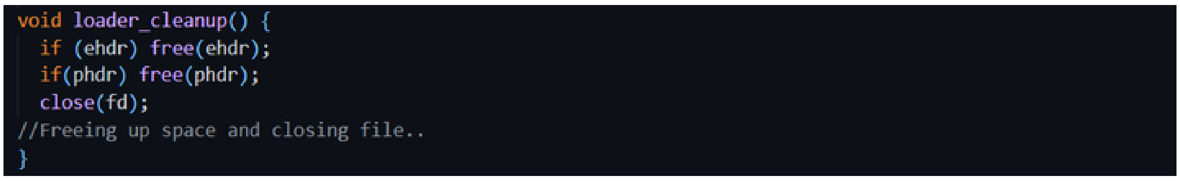
- **Locating the Entry Point in Memory:**
  - The entry point is given as an absolute address in the ELF header, but the code running the loader needs to know the offset from the beginning of the loaded segment to this entry point.
  - By calculating offset, the code identifies the exact location within the mapped memory (virtual_m) where execution should begin. This is crucial for jumping to the correct starting address of the program.
- **Preparing for Execution:**
  - By adding offset to virtual_m, the code calculates the exact memory address where the program's execution should start.
  - This is where the initial setup or main function of the ELF executable resides. Setting entry to this location prepares the loader to transfer control to the executable.



```
int (*_start)() = (int (*)()) entry;
int result = _start();
printf("User _start return value = %d\n", result);
```

- **Transferring Control to the Program:**
  - Function pointers in C allow code to call functions indirectly, via a pointer to the function's location in memory. By setting _start to the entry point, the loader prepares to execute the ELF file by calling this function pointer.
  - This setup is necessary because the actual execution point of the ELF file is determined dynamically, based on where the segment is loaded into memory..

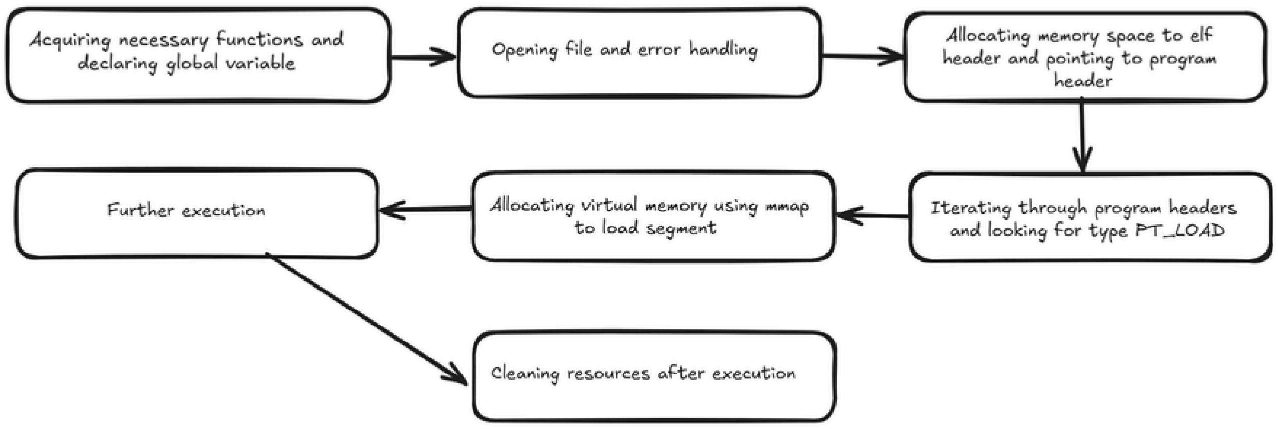
3. Memory cleanup



```
void loader_cleanup() {
  if (ehdr) free(ehdr);
  if(phdr) free(phdr);
  close(fd);
//Freeing up space and closing file..
}
```

Transferring Control to the Program: Function pointers in C allow code to call functions indirectly, via a pointer to the function's location in memory. By setting _start to the entry point, the loader prepares to execute the ELF file by calling this function pointer. This setup is necessary because the actual execution point of the ELF file is determined dynamically, based on where the segment is loaded into memory..

PROGRAM FLOW


Acquiring necessary functions and declaring global variable
Opening file and error handling
Allocating memory space to elf header and pointing to program header
Iterating through program headers and looking for type PT_LOAD
Allocating virtual memory using mmap to load segment
Further execution
Cleaning resources after execution

MEMBERS AND CONTRIBUTION

1. **Priyanshu sharma | 2023408 | priyanshu23408@iiitd.ac.in**
    - Code logic and coding portion
    - Git repository-committing and maintenance
    - Explaining code via comments

2. **Vansh tyagi | 2023582 | vansh23582@iiitd.ac.in**
    - Code logic and coding portion
    - Documentation of code
    - Explaining code via comments

GITHUB REPO

https://github.com/Priyanshu0phc/OS_66