ASSIGNMENT-4

Detailed Report on ELF Loader Code

This report provides a line-by-line breakdown of a custom ELF loader in C, which handles loading and executing an ELF (Executable and Linkable Format) file while managing page faults. The code performs the following main tasks:

1. Loads ELF headers and program headers from an ELF file.
2. Maps segments into memory when required.
3. Handles page faults using a segmentation fault handler.
4. Tracks and displays statistics on page faults, page allocations, and internal fragmentation.

Header Files and Constant Definitions

```
#include <elf.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
```

- These headers provide functions and data types essential for:
  - ELF file handling (`elf.h`)
  - System calls and memory management (`sys/types.h` , `unistd.h` , `fcntl.h` , `sys/mman.h`)
  - Standard input/output, error handling, and signal handling (`stdio.h` , `stdlib.h` , `signal.h` , `errno.h`).

```
#define P_SIZE 4096 // Size of a page in Linux OS
#define MAX_DATA 100
```

- `P_SIZE` is defined as 4096, representing the memory page size in bytes (common in Linux).
- `MAX_DATA` sets a maximum buffer size for storing fault addresses.

Function Prototypes and Global Variables

```
void print_data();
void execute_elf(char **argv);
Elf32_Phdr* find_segment(unsigned long addr);
```

```
void setup();
void loader_cleanup();
void segmentation_handler(int signal, siginfo_t *info, void *context);
```

- Declares function prototypes for various tasks performed by the loader.

```
Elf32_Phdr *phdr;
Elf32_Ehdr *ehdr;
int fd;
unsigned long fault_addresses[MAX_DATA];
int pointer_f = 0;
int numPageFaults;
int numPageAllocations;
long long internalFragmentation;
```

- Global variables:
    - `phdr` and `ehdr`: Pointers to ELF program and file headers.
    - `fd`: File descriptor for the ELF file.
    - `fault_addresses[]`: Stores addresses where page faults occurred.
    - `pointer_f`: Tracks the number of fault addresses recorded.
    - `numPageFaults`: Counts the total page faults.
    - `numPageAllocations`: Counts total pages allocated.
    - `internalFragmentation`: Tracks internal fragmentation.

`find_segment` Function

```
Elf32_Phdr* find_segment(unsigned long addr) {
    lseek(fd, ehdr->e_phoff, SEEK_SET);
    int len = ehdr->e_phnum;
    for (int i = 0; i < len; i++) {
        if (read(fd, phdr, sizeof(Elf32_Phdr)) != sizeof(Elf32_Phdr)) {
            free(phdr);
            return NULL;
        }
        if (phdr->p_type == PT_LOAD && addr < phdr->p_vaddr + phdr->p_memsz &
& addr >= phdr->p_vaddr)
            return phdr;
    }
    return NULL;
}
```

- **Purpose**: Locates the program segment containing the fault address `addr`.

- `lseek` positions the file descriptor to the program header offset in the ELF header.
- It reads each program header and checks if:
    - The segment is loadable (`PT_LOAD`).
    - `addr` is within the segment's address range.
- If a match is found, the function returns the segment; otherwise, it returns `NULL`.

`segmentation_handler` Function

```
void segmentation_handler(int signal, siginfo_t *info, void *context) {
    numPageFaults++;
    unsigned long fault_addr = (unsigned long)info->si_addr;
    fault_addresses[pointer_f] = fault_addr;
    pointer_f++;
    Elf32_Phdr *segment = find_segment(fault_addr);
    if (segment == NULL) {
        fprintf(stderr, "No Such Segment Requested %lx\\n", fault_addr);
        exit(1);
    }

    unsigned long seg_off = (fault_addr / P_SIZE) * P_SIZE - segment->p_vadd
r;
    unsigned long aligned_file_offset = segment->p_offset + seg_off;
    unsigned long page_start_address = (fault_addr / P_SIZE) * P_SIZE;

    void *address_mapped = mmap((void *)page_start_address, P_SIZE, PROT_READ
| PROT_WRITE | PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (address_mapped == MAP_FAILED) {
        perror("Page Allocation Failed!");
        exit(1);
    }

    if (pread(fd, address_mapped, P_SIZE, aligned_file_offset) < 0) {
        perror("Copy Operation on segment failed!");
        exit(1);
    }
    numPageAllocations++;
    internalFragmentation += P_SIZE - (segment->p_memsz - seg_off < P_SIZE ?
segment->p_memsz - seg_off : P_SIZE);
}
```

- **Purpose**: Handles segmentation faults (interpreted as page faults).
- Retrieves the fault address from `info`, then finds the segment where the fault occurred using `find_segment`.

- If `segment` is `NULL`, it logs an error and exits.
- Calculates the offset within the segment and maps a new memory page at the fault address (`mmap`).
- If mapping fails, it logs an error and exits.
- Uses `pread` to copy data from the ELF file into memory, updating page allocation statistics and internal fragmentation.

`execute_elf` Function

```
void execute_elf(char **exe) {
    fd = open(exe[1], O_RDONLY);
    if (fd == -1) {
        perror("Failed to open ELF file");
        exit(1);
    }
    if (read(fd, ehdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr)) {
        perror("Elf Header Can't be readed");
        close(fd);
        exit(1);
    }

    int (*_start)() = (int (*)())(ehdr->e_entry);
    int res = _start();
    print_data(res);
    close(fd);
}
```

- **Purpose**: Opens, reads, and executes an ELF file.
- Opens the ELF file in read mode and reads the ELF header (`ehdr`).
- Uses the entry point (`e_entry`) to start execution.
- Calls `print_data` to display statistics after execution, then closes the file.

`loader_cleanup` Function

```
void loader_cleanup() {
    if (phdr != NULL) free(phdr);
    if (ehdr != NULL) free(ehdr);
}
```

- **Purpose**: Frees memory allocated to `phdr` and `ehdr` to avoid memory leaks.

`setup` Function

```
void setup() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(struct sigaction));
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = segmentation_handler;
    if (sigaction(SIGSEGV, &sa, NULL) < 0) {
        perror("Failed to set up signal handler");
        exit(1);
    }
}
```

- **Purpose**: Sets up a signal handler for `SIGSEGV` (segmentation fault) signals, linking it to `segmentation_handler`.

`print_data` Function

```
void print_data(int res) {
    printf("_start address Value = %d\\n", res);
    printf("Total Page Faults = %d\\n", numPageFaults);
    printf("Total Page Allocated %d\\n", numPageAllocations);
    printf("Internally Space Fragmented: %lld Bytes \\n", internalFragmentati
on);
    printf("Fault Occured at: \\n");
    for (int i = 0; i < numPageFaults; i++) {
        printf("%d %lu\\n", i + 1, fault_addresses[i]);
    }
}
```

- **Purpose**: Displays recorded data on program execution:
    - ELF entry address.
    - Total page faults, allocations, internal fragmentation.
    - List of fault addresses.

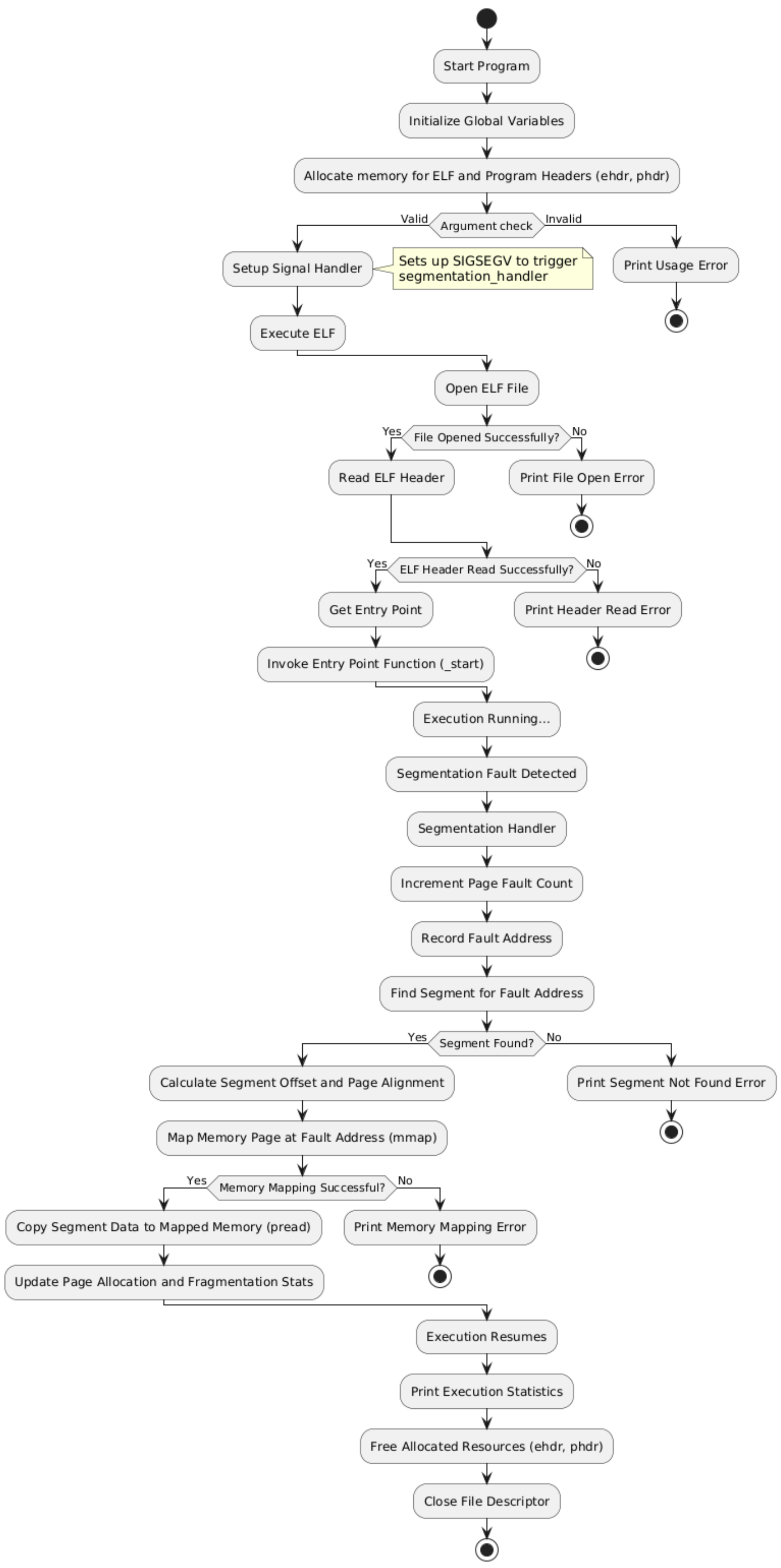
`main` Function

```
int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Wrong Usage, enter exe name after ./loader\\n");
        exit(1);
    }
    numPageAllocations = 0;
    internalFragmentation = 0;
    numPageFaults = 0;
    phdr = (Elf32_Phdr *)malloc(sizeof(Elf32_Phdr));
    if (phdr == NULL) {
        perror("Malloc Operation failed for Program Header");
        exit(1);
    }
    ehdr = (Elf32_Ehdr *)malloc(sizeof(Elf32_Ehdr));
    if (ehdr == NULL) {
        perror("Malloc Operation failed for ELF Header");
        exit(1);
    }

    setup();
    execute_elf(argv);
    loader_cleanup();
    return 0;
}
```

- **Purpose**: Initializes global variables, sets up signal handling, and executes the ELF file.
- Manages memory allocation for ELF and program headers (`ehdr`, `phdr`), ensuring space is freed with `loader_cleanup`.
- Exits with error if command-line arguments are incorrect or memory allocation fails.

PROGRAM FLOW


Start Program
Initialize Global Variables
Allocate memory for ELF and Program Headers (ehdr, phdr)
Argument check
Valid
Invalid
Setup Signal Handler
Sets up SIGSEGV to trigger segmentation_handler
Print Usage Error
Execute ELF
Open ELF File
File Opened Successfully?
Yes
No
Read ELF Header
Print File Open Error
ELF Header Read Successfully?
Yes
No
Get Entry Point
Print Header Read Error
Invoke Entry Point Function (_start)
Execution Running...
Segmentation Fault Detected
Segmentation Handler
Increment Page Fault Count
Record Fault Address
Find Segment for Fault Address
Segment Found?
Yes
No
Calculate Segment Offset and Page Alignment
Print Segment Not Found Error
Map Memory Page at Fault Address (mmap)
Memory Mapping Successful?
Yes
No
Copy Segment Data to Mapped Memory (pread)
Print Memory Mapping Error
Update Page Allocation and Fragmentation Stats
Execution Resumes
Print Execution Statistics
Free Allocated Resources (ehdr, phdr)
Close File Descriptor

CONTRIBUTIONS

1. Priyanshu Sharma / 2023408

- Code logic
- Commenting
- GitHub contribution

1. Vansh Tyagi / 2023582

- Code logic
- Documentation
- GitHub contribution