Simple Smart Loader

Contributors:
Vishal Kumar Maurya (2022580)
Subham Maurya (2022510)

GitHub Link for the Repository (Private) :-
https://github.com/vmaurya6622/OS-Assignments/tree/792b268c3348858046e324feddc8bd0a36b9c15c/Assignment-04

Files Contained :- loader.c , README , Documentation, fib.c and Makefile,loader.h,sum.c,prime.c we have used deberian based (KALI linux) to complete our assignment and we have completed Bonus part also .

Contribution by Vishal Kumar Maurya (2022580) :- implemented the loader.c cleanup function that cleans up the unused segments and header functions and helped in debugging the code. Improved the code by adding additional comments and made the documentation and readme files to display our approach. I made most of the effort at the debugging part of the code and improved it for NO ERROR. I also searched for the resources from the OSTEP book and given an idea about the program handling.

Contribution by Subham Maurya (2022580) :- implemented the loader. C and added the segmentation fault handler that handles the segmentation fault by providing a new space of PAGESIZE (Take to be 4KB initially), I also completed the load and run elf function to give the segmentation fault and then dealt with it in segmentation fault handler function and improved it to handle the segmentation fault effectively. I added the calculation of PAGESIZE, total internal segmentation etc.

Click Here for the Source Code (Seen locally) :- CLICK HERE

```c
#include "loader.h"

Elf32_Ehdr *ehdr;
Elf32_Phdr *phdr;
// default allocation size taken here is 4096 B or 4KB
int File_Descriptor;
size_t Entry_Offset_Size;
void *Previous_Address;
size_t Entry_Point_Address_Size;
int (*_address)();
int PHDR_Table[1000][2];
int Page_Fault_Counter = 0;
size_t Total_Internal_Fragmentation = 0;
void *Virtual_Memory_Addr;
size_t Total_MEM_sz = 0;
int Page_Allocation_Counter = 0;

void loader_cleanup();
void Call_entrypoint(Elf32_Ehdr *ehdr, Elf32_Phdr *phdr);
void SegmentationFaultHandler(int signal, siginfo_t *info, void *context);
int Ceil(int i, int k);

void load_and_run_elf(char **argv)
{
    File_Descriptor = open(argv[1], O_RDONLY);
    if (File_Descriptor == -1)
    {
        perror("Failed to open file");
        loader_cleanup();
        exit(1);
    }

    ehdr = (Elf32_Ehdr *)malloc(sizeof(Elf32_Ehdr));
    if (read(File_Descriptor, ehdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr))
    {
        perror("Error reading ELF header");
        loader_cleanup();
        exit(1);
    }
```

```
    phdr = (Elf32_Phdr *)malloc(ehdr->e_phentsize * ehdr->e_phnum);
    if (lseek(File_Descriptor, ehdr->e_phoff, SEEK_SET) == -1)
    {
        perror("lseek");
        exit(0);
    }

    if (read(File_Descriptor, phdr, ehdr->e_phentsize * ehdr->e_phnum) !=
(ssize_t)(ehdr->e_phentsize * ehdr->e_phnum))
    {
        perror("Error reading program header table");
        loader_cleanup();
        exit(1);
    }

    Call_entrypoint(ehdr, phdr);
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("Usage: %s <ELF Executable> \n", argv[0]);
        exit(1);
    }

    struct sigaction sa;
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = SegmentationFaultHandler;

    if (sigaction(SIGSEGV, &sa, NULL) == -1)
    {
        perror("sigaction");
        return 1;
    }

    load_and_run_elf(argv);
    loader_cleanup();

    if (munmap(Virtual_Memory_Addr, Total_MEM_sz) == -1)
    {
        perror("Error unmapping memory");
        loader_cleanup();
```

```
        exit(1);
    }

    printf("Total Page Faults            : %d\n", Page_Fault_Counter);
    printf("Total Page Allocations       : %d\n", Page_Allocation_Counter);
    printf("Total Memory Allocated       : %d Bytes or %f KB\n",
(int)Total_MEM_sz, Total_MEM_sz / 1000.0);
    printf("Total Internal Fragmentation : %d Bytes or %f KB\n\n",
Total_Internal_Fragmentation, Total_Internal_Fragmentation / 1000.0);

    return 0;
}

int Ceil(int i, int k)
{ // to calculate the multiple for the number of pages to be allocated....
    if (i % k == 0)
        return i / k;
    return i / k + 1;
}

void Call_entrypoint(Elf32_Ehdr *ehdr, Elf32_Phdr *phdr)
{

    for (int i = 0; i < ehdr->e_phnum; i++)
    {
        PHDR_Table[i][0] = 0;
        PHDR_Table[i][1] = phdr[i].p_memsz;
    }

    printf("\nEntry Point Address : %p\n\n", (void *)ehdr->e_entry);
    Entry_Offset_Size = ehdr->e_entry;
    Entry_Point_Address_Size = (Entry_Offset_Size);
    _address = (int (*)())Entry_Point_Address_Size;
    int (*_start)() = (int (*)())_address;
    int result = _start();
    printf("\nUser _start return value = %d\n", result);
    printf("END OF PROGRAM!\n\n");
}

void loader_cleanup() // Function to Clean any allocated resources and memory
{
    if (ehdr != NULL) // Clearing the space allocated to ELF header
    {
        free(ehdr);
```

```
        ehdr = NULL;
    }
    if (phdr != NULL) // Clearing the space allocated to Program header
    {
        free(phdr);
        phdr = NULL;
    }
    if (File_Descriptor != -1) // Closing the file descriptor
    {
        close(File_Descriptor);
        File_Descriptor = -1;
    }
}

void SegmentationFaultHandler(int signal, siginfo_t *info, void *context)
{
    void *Fault_Address = (void *)info->si_addr;
    printf("Segmentation Fault At Address: %p\n", Fault_Address);
    ucontext_t *Old_Context = (ucontext_t *)context;
    for (int i = 0; i < ehdr->e_phnum; i++)
    {
        if ((int)Fault_Address <= phdr[i].p_vaddr + phdr[i].p_memsz &&
(int)Fault_Address >= phdr[i].p_vaddr)
        {

            int MEM_sz = phdr[i].p_memsz;
            int Page_counter = Ceil(MEM_sz, 4096);
            size_t allocationSize = 4096;

            if (PHDR_Table[i][0] == 0)
            {
                Previous_Address = (void *)phdr[i].p_vaddr;
                if (lseek(File_Descriptor, phdr[i].p_offset, SEEK_SET) == -1)
                {
                    perror("lseek");
                    exit(0);
                }
                PHDR_Table[i][0] = 1;
            }

            Virtual_Memory_Addr = mmap(Previous_Address, allocationSize,
PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
            if (Virtual_Memory_Addr == MAP_FAILED)
            {
```

```
                perror("mmap");
                exit(0);
            }

            int sizeUsing = PHDR_Table[i][1] > 4096 ? 4096 : PHDR_Table[i][1];
            Previous_Address += (sizeUsing + 1);

            printf("SEGMENT MEMORY SIZE : %d Bytes\n", phdr[i].p_memsz);
            printf("LOADING SIZE        : %d Bytes or %f KB\n", sizeUsing,
sizeUsing / 1024.0);
            printf("------------------------------------------------------------
---------\n");

            if (PHDR_Table[i][1] - 4096 < 0 && PHDR_Table[i][1] > 0)
            {
                if (read(File_Descriptor, Virtual_Memory_Addr, PHDR_Table[i][1])
== -1)
                {
                    perror("read");
                    exit(0);
                }
                Total_Internal_Fragmentation += (allocationSize -
(size_t)PHDR_Table[i][1]);
                PHDR_Table[i][1] = 0;
            }

            else if (PHDR_Table[i][1] - 4096 > 0)
            {
                PHDR_Table[i][1] -= 4096;
                if (read(File_Descriptor, Virtual_Memory_Addr, 4096) == -1)
                {
                    perror("read");
                    exit(0);
                }
            }
            Page_Fault_Counter++;
            Page_Allocation_Counter++;
            Total_MEM_sz += allocationSize;
        }
    }
}
```