Lab 5 (C) - Loader

Introduction

In this lab you will implement a static loader. Your loader will be able to load (run) static executable files. These are executable files that do not use dynamic library code. In particular, your loader will be able to load your code which uses the system_call interface, and no standard libraries. **Note**: Remember to compile your code with the -m32 flag.

Task 0

Write a program, which gets a single command line argument. The argument will be the file name of a 32bit ELF formatted executable.

Your task is to write an iterator over program headers in the file. Implement a function with the following signature:

```
int foreach_phdr(void *map_start, void (*func)(Elf32_Phdr *,int), int arg);
```

The function arguments are:

1. map_start: The address in virtual memory the executable is mapped to.
2. func: the function which will be applied to each Phdr.
3. arg: an additional argument to be passed to func, for later use (not used in this task).

This function will apply func to each program header.

Verify that your iterator works by applying it to an 32bit ELF file, with a function that prints out a message: "Program header number i at address x" for each program header i it visits.

Task 1

Task 1a:

In this task you will use the iterator you created in Task 0, and implement the *readelf -l* functionality.

Using the functions from task 0 (the iterator), your task is to go over the program headers in a file and for each header, print all the information which resides in the corresponding Elf32_Phdr structure.

The output should look similar to *readelf -l*:

```
Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
PHDR 0x000034 0x04048034 0x04048034 0x00100 0x00100 R E 0x4
INTERP 0x000134 0x04048134 0x04048134 0x00013 0x00013 R 0x1
LOAD 0x000000 0x04048000 0x04048000 0x008a4 0x008a4 R E 0x1000
LOAD 0x0008a4 0x040498a4 0x040498a4 0x0011c 0x00120 RW 0x1000
DYNAMIC 0x0008b0 0x040498b0 0x040498b0 0x000c8 0x000c8 RW 0x4
NOTE 0x000148 0x04048148 0x04048148 0x00020 0x00020 R 0x4
```

Task 1b:

Note that tasks 0 and 1 should work on any Linux 32 bit ELF file whatsoever, as at this point we are merely looking at the program headers (empty if the ELF is non-executable, of course). But starting from task 2, we actually attempt to load and execute, so there you only need to support static linked files, i.e. without program headers of type DYNAMIC or INTERP.

In the next task we will map the needed segments to memory, so we need to prepare the appropriate data for mapping. The loader uses the Program Headers to find the information needed for loading the program to memory and uses mmap system call for actually mapping the needed segments to process's memory. Specifically, it is VERY important to pass mmap the appropriate protection and mapping flags.

For each program header that should be mapped to memory, in addition to the information in task 1a, print the appropriate protection flags and mapping flags that should be passed to the mmap function for this header. **Note**: The protection flags used by mmap, though similar to the flags in the program header, differ in bit position!

Task 2

Now you will write the actual loader, using the iterator from task 1. The loader is mainly a single function, which maps each relevant chunk of the executable into memory. In the end, you should produce an executable program, which receives one command-line argument (an ELF executable), loads it in to memory and passes control to the loaded code.

Task 2a

Recall that in order to actually load the files using mmap as required, you need to obey the instructions in the reading material so as to avoid memory space clashes. Download the linking_script file, and compile your code as shown in the reading material. Note that we link the program without using any standard libraries, but the loaded program still needs to get a specific stack before it starts executing. Also, we still want the ability to use the system_call interface (what the "glue" code file start.s from lab 3 was about). So you have to link your code together with start.o (you do not need startup.o yet). Verify linking worked properly using *readelf -h loader*

You should be able to explain to the lab instructor why the linking script is needed and how you verified that it worked.

Task 2b

Implement the following function:
void load_phdr(Elf32_Phdr *phdr, int fd);

This function takes two arguments, a pointer to the Phdr struct and the file descriptor of the executable file. It should map each Phdr that has the PT_LOAD flag set, into memory, starting from the specified offset, and place it at the virtual address stated in the Phdr. Each map should be according to the flags set in the Phdr struct. In addition, this function should print to the screen the information about each program header it maps (you can use the function from Task 1 to print the information).

Recommended operating procedure: make sure system calls succeed before proceeding, most especially mmap.

Task 2c

After successfully completing the previous function, you should now pass control to the loaded program. To achieve this, we provide the code in assembly language (startup.s), you should examine the code we provide. You may download its object file startup.o. You need to execute the loaded program using our function startup(), with the following signature:

```
int startup(int argc, char **argv, void (*start)());
```

and start is the entry point of your executable.

Your loader should be able to load and run all code from previous labs which uses the system_call interface, provided that they are compiled with the -m32 flag and according to the compilation instructions in the system calls lab.

However, first try it for a program that does not expect command-line arguments, such as this file: loadme. In case of a bug note the following:

- The "arg" of foreach_phdr is used correctly - it should be the file descriptor of an open file.
- The file/s that the segments are mapped from must remain open and mapped to memory.
- The correct segments are mapped to memory and flags are set appropriately.
- startup is used correctly as following:
  startup(argc-1, argv+1, (void *)(elf_head->e_entry))

Task 2d

Now, we assemble the command line arguments and pass them to the loaded program. A command line looks like:

```
my_loader my_test_program arg1 arg2 ...
```

Where *my_loader* is the executable program you implemented in this lab, *my_test_program* is the program you are trying to load and run, and arg1, arg2, etc. are the command-line arguments that *my_test_program* should see. These are to be passed to it using the startup function.

Deliverables

This lab is not too much work ASSUMING you read the reading material and did task 0 before the lab. So you should complete everything from task 1 to task 2d during lab hours. having to do completion labs next week.