CS 5600 – Computer Systems | Prof. Peter Desnoyers
Homework 1 – Program Environment and Loading | Fall 2018

For this assignment you will write a program loader that can load simple statically-linked programs in Linux – it will read an executable file and load it into virtual memory at the correct locations, and then transfer control to that program

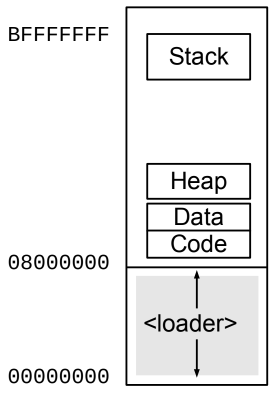
Source code and repository

The files you will need for this program are in the CCIS github, in the repository CS5600-TR-F18/team-nn-hw1 where nn is your team number.

Deliverables

You will create a program called “loader.c” which is able to load and run a simple executable compiled with Diet Libc (see below) and a script “compile.sh” which compiles “loader.c” into “loader”.

Problem Description



Linux virtual memory map

A statically-linked program is a “classic” executable (as described in class and in Chapter 2 in the text) with code, data, heap, and stack segments, as shown in the figure to the right. In 32-bit Linux these segments begin at 08000000 (128MB), so if we write a program that runs in the memory space below 08000000, it can read a standard executable into higher memory locations and then transfer control to that executable.

Linux uses the ELF executable format, a self-describing format with a series of headers as well as the executable data itself - for more information you can consule the Linux man page for the ELF format (man 5 elf), search on the web, or use the ‘readelf’ application to examine various executables. There is a standard elf.h file; however you are provided with an alternate but compatible implementation which is somewhat easier to use, especially with the debugger[1].

You are going to write code which will “live” in the bottom of the address space, and which reads an ELF executable file, iterates through the segments in the file, loads the appropriate ones into the right place in memory, and then jumps to the starting address of the program. (which is identified by a field in the ELF header) Your code can assume that programs are statically linked at the normal address; you won't be tested on other cases. (i.e. it's OK if your code crashes in those cases)

Before you get started

Update the software on your VM, either via the update manager or:
```
   sudo apt upgrade
```
Then install diet-libc:
```
   sudo apt install dietlibc-dev
```

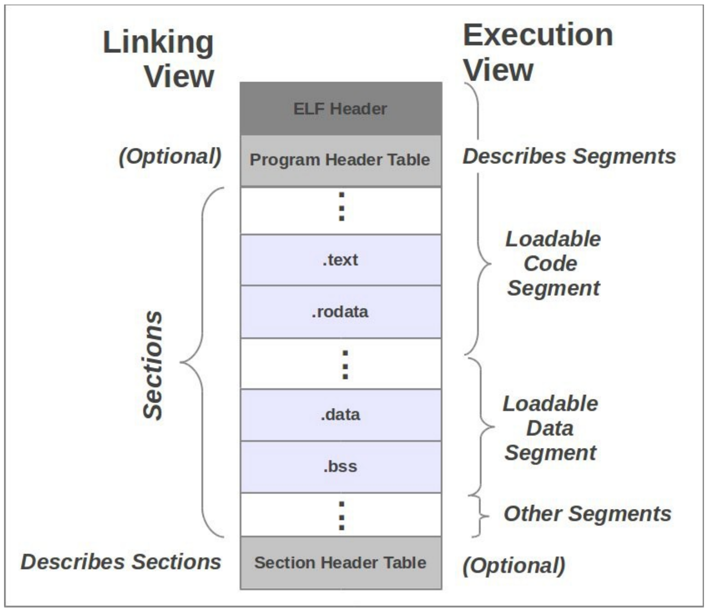
1 In other words, the standard elf.h file is horrible mess, designed by a standards committee and with no respect for the fact that it might be used by someone who didn't write it.

ELF format

You'll need to understand this in order to do the assignment. An ELF file starts with a header (struct elf32_ehdr) which contains information such as:

- "magic number" to identify the file
- type of object file - e.g. executable, shared library, object file
- CPU
- "entry point" - i.e. the address the program should start at
- pointers to the section header table and program header table (see below)

The rest of the file is made up of *sections*, described in the section header table; these sections are grouped into *segments*, described in the program header table; this structure is shown to the right[2]. Our code will iterate through the segments and load the appropriate ones into memory.



To understand it better, let's compile a very simple program:

```
$ cat hello.c
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("hello, world\n");
}
```

We're using something called "Diet LibC", which tells gcc to link against a small, simple library instead of the standard GNU library, which is huge; the result is a much simpler executable. To compare the two:

```
$ diet gcc -static hello.c -o hello
$ size hello
   text    data     bss     dec     hex filename
    466       4      12     482     1e2 hello
$ gcc -static hello.c -o hello
$ size hello
   text    data     bss     dec     hex filename
 677278    3316    5892  686486   a7996 hello
```

The extra 677KB of GNU library code includes a lot of complicated executable file segments which we're not going to bother with here.

2 This figure is taken from the very good description at http://www.360doc.com/content/17/1204/11/7377734_709762695.shtml, although you can ignore the parts describing dynamic linking, in particular the discussion of the PLT sections.

You can see how the 'diet' command works by running it with the -v option:

```
$ diet -v gcc -static hello.c -o hello
gcc -nostdlib -static -L/usr/lib/diet/lib-i386 /usr/lib/diet/lib-i386/start.o -static
hello.c -o hello -isystem /usr/lib/diet/include -D__dietlibc__ /usr/lib/diet/lib-
i386/libc.a -lgcc -fno-stack-protector
```

It compiles your programs with a different set of standard include files (found in /usr/lib/diet/include) and links against a different libc.a, rather than the standard system version found in /usr/lib/i386-linux-gnu/libc.a[3].

We can use “readelf” to look at the resulting executable:

```
$ readelf --file-header --segments --sections hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80480f8
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4220 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```

The main pieces of information we'll be getting from the file header are the entry point address (i.e. the first instruction to execute when starting the program) and the location of the program and section headers.

The section headers describe all the content of the file in full detail. Sections that go into the executable file are PROGBITS (code, initialized data), NOBITS (data initialized to zero, so it doesn't have to be stored in the file), and NOTE (random stuff like the build ID which we don't care about). Other sections include things like symbol tables, string tables (all strings in the executable are represented by indexes into this table), and debug information. Each section has a byte offset in the executable file and a length, and sections which get loaded into memory also have an address at which they should be loaded:

3 “What's a .a file?” It's an “archive”, basically a collection of .o files, i.e. compiled (usually) C files, in this case making up the standard C library. You can list the contents with `ar -t /usr/lib/i386-linux-gnu/libc.a`, and if you want to see all the functions defined you can use the command `nm /usr/lib/i386-linux-gnu/libc.a`
In typical Unix fashion, “ar” stands for “archive” and “nm” stands for “name map”.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            080480d4 0000d4 000024 00   A  0   0  4
  [ 2] .text             PROGBITS        080480f8 0000f8 000113 00  AX  0   0  4
  [ 3] .rodata           PROGBITS        0804820b 00020b 00000f 00   A  0   0  1
  [ 4] .eh_frame         PROGBITS        0804821c 00021c 00008c 00   A  0   0  4
  [ 5] .data             PROGBITS        08049ff4 000ff4 000004 00  WA  0   0  4
  [ 6] .bss              NOBITS          08049ff8 000ff8 00000c 00  WA  0   0  8
  [ 7] .comment          PROGBITS        00000000 000ff8 00002a 01  MS  0   0  1
  [ 8] .shstrtab         STRTAB          00000000 001022 00005a 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 001234 000270 10     10  11  4
  [10] .strtab           STRTAB          00000000 0014a4 000185 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

In an executable file the actual program sections - the ones which need to be loaded into memory - are arranged into *segments* of the same type (e.g. read-only, executable, read/write), and the program header table describes these segments:

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x002a8 0x002a8 R E 0x1000
  LOAD           0x000ff4 0x08049ff4 0x08049ff4 0x00004 0x00010 RW  0x1000
  NOTE           0x0000d4 0x080480d4 0x080480d4 0x00024 0x00024 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0
 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .text .rodata .eh_frame
   01     .data .bss
   02     .note.gnu.build-id
   03
   04
```

We're only going to care about the segments with type LOAD, as these are the ones we load into memory. Note that the second segment has a larger size in memory than in the file - this is because it includes the BSS section, which is data initialized to zero. If we add a large global variable to our program, like this:

```
$ cat hello.c
#include <stdio.h>
char buffer[1000];
int main(int argc, char **argv)
{
    printf("hello, world\n");
}
```

then the BSS section will get bigger, and the memory size of this segment will grow without any increase in its file size:

```
$ diet -v gcc -static hello.c -o hello
$ readelf  --segments hello
Elf file type is EXEC (Executable file)
Entry point 0x80480b4
```

In this simple case there are 4 program headers, starting at offset 52; two of them get loaded:

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x00264 0x00264 R E 0x1000
  LOAD           0x000ff4 0x08049ff4 0x08049ff4 0x00004 0x00414 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0
```

The first thing we're going to do is to just read the ELF headers, so that our program can get the same information that readelf is displaying:

```
#include "elf32.h"
main(int argc, char **argv)
{
    int fd;
    if ((fd = open(argv[1])) < 0) {
        perror("open");
        exit(1);
    }
    struct elf32_ehdr ehdr;
    read(fd, &ehdr, sizeof(ehdr));
    assert(ehdr.e_phentsize == sizeof(struct elf32_phdr));
    int n = ehdr.e_phnum;
    struct elf32_phdr phdrs[n];
    lseek(fd, ehdr.e_phoff, SEEK_SET);
    read(fd, phdrs, sizeof(phdrs));
    /* ... */
    return 0;
}
```

Now you can iterate through the phdrs[] array, printing out information about each program section.

To actually load the program into memory we'll need a way to allocate memory at a specific address with the appropriate permissions - we use `mmap` for this ('`man 2 mmap`' for more details), e.g.:

```
char *buf = mmap((void*)base, len, PROT_READ | PROT_WRITE |
                 PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

In this case we're allocating 'len' bytes at address 'base', with read/write/execute permission, private to this process, and "anonymous" - i.e. not associated with a file. (the other way of using mmap is to map a portion of a file into memory - in that case the last two arguments would specify the file descriptor and the offset into the file)

Note that 'base' and 'len' should be multiples of the system page size, which is 4096 (0x1000) bytes. An easy way to round a number down to the nearest multiple of 4096 is to mask off the bottom 12 bits - i.e.

```
int rounded = value & 0xFFFFF000;
```

Read-only sections are a complication, as although mmap will happily allocate read-only memory, our program will crash if we try to write into it. The simple solution is to make the read-only sections writable, and hope that the program won't notice. (it won't if it's not buggy) A better solution is to use the mprotect system call to protect these segments after they've been filled.

Compiling

In order to fit in the space underneath a normal executable we'll have to link at a low starting address:

```
diet gcc -g -static loader.c -o loader -Wl,--section-start -Wl,.text=0x100000
```

"-Wl," tells gcc to pass arguments through to the linker, and "--section-start .text=0x100000" tells the linker to start the executable at address 0x100000, or 1MB. Put your compile command in a script file called `build.sh` – the grading software will expect it to be there.

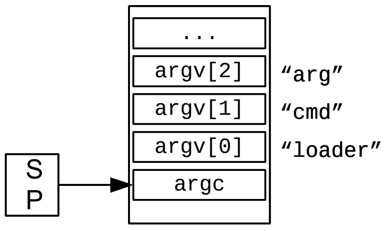
Program arguments and environment

When a C program starts, the main() function is passed 3 arguments[4] – the command line argument count (argc), the array of command line arguments (argv), and the array of environment variables (envp). Since there's no count, the end of the envp array is marked by a null pointer. (actually, the argv array has a null pointer at the end, too.)

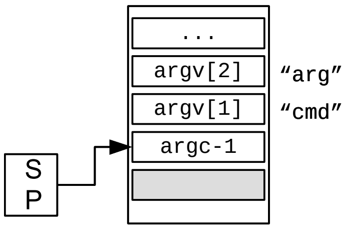
When the first instruction in an executable begins running (i.e. the library code that *calls* main) it's a bit different, since nothing actually *called* the entry point; instead, execution just began there. From the top down, the stack contains the following[5]:

- Values of all the environment variables. (these are null-terminated strings of the form "<variable>=<value>", e.g. "PATH=/usr/bin:/bin". The values in envp[] will point to these.
- Values of all the command-line arguments. Values in argv[] will point to these.
- A set of *auxiliary vectors* that pass weird system information from the kernel to the startup code. (e.g. what sort of floating point hardware the CPU has, etc.)
- The envp array, with a NULL pointer as its last entry
- The argv array, with a NULL pointer as its last entry
- The value for argc. (SP points here – no return address on stack)

When the process begins executing, the stack pointer is pointing directly at argc. However we need to adjust the argument list before jumping to our loaded executable – e.g. we might have argv = "loader", "cat", "/dev/null", but when 'cat' begins to execute, it should be argv = "cat", "/dev/null". To do this we modify it in place, like this:



Stack at startup



Modified stack for running loaded program

4 You probably thought it was 2, right? Try compiling and running the following program:

```
#include <stdio.h>
int main(int argc, char **argv, char **envp) {
    for (; *envp != NULL; envp++)
        printf("%s\n", *envp);
}
```

5 See https://lwn.net/Articles/631631/ for full details.

We can find the right location by observing that the value of envp, which is passed to our main() function, is in fact the address of envp[0]. Since integers and pointers are the same size on 32-bit Intel, you can create an 'int *' pointer and set it to the value of envp, using the appropriate cast.

Now we just have to set SP to point to that location, and jump to the entry point indicated in the ELF header. JMP instructions on Intel are complicated; instead we can do this easily by pushing the entry point on the stack and executing a RET. Combining the argc/argv adjustment and the jump, we have:

```
int *p = (int*)argv;
p[0] = argc-1;
p[-1] = ehdr.eh_entry;
__asm__("mov.l %0,%%esp\n" : p-1: );[6]
__asm__("ret");
```

And now you should be done.

6 If you *really* want to know how the GCC extensions for inline assembler work, google it yourself. That's what I do every time I work with it.