Homework 3:
The ELF Format

This assignment will make you more familiar with the organization of ELF files. You can do this assignment on any operating system that supports the Unix API (Linux Openlab machines, your laptop that runs Linux or Linux VM, and even MacOS). **You don't need to set up Xv6 for this assignment.**

For MacOS users, the support of 32 bit applications is deprecated in the latest version of your system. So if you already updated your system to MacOS Catalina or have updated your XCode then we recommend you to do the homework at the Openlab machines.

Part 1: Take a Look at ELF files

Top

Download the main.c , and elf.c and look them over. At a high level this homework asks you to implement a simple ELF loader (you will extend the `main.c` file) and use it to load a simple ELF object file (the one compiled from `elf.c`). However, before starting this task, lets make ourselves familiar with ELF files.

We provide a simple Makefile that compiles `elf.o` and `main` as ELF executables. Look over the Makefile and then compile both files by running:

```
$ make
```

Now, let's take a look at the ELF files that we just compiled. We will use the `readelf` tool:

```
$ readelf -a elf
```

ELF is the file format used for `.o` object files, binaries, shared libraries and core dumps in Linux. It's actually pretty simple and well thought-out.

ELF has the same layout for all architectures, however endianness and word size can differ; relocation types, symbol types and the like may have platform-specific values, and of course the contained code is architecture specific.

ELF files are used by two tools: the **linker** and the **loader**. A linker combines multiple ELF files into an executable or a library and a loader loads the executable ELF file in the memory of the process. On real operating systems, loading may require relocation (e.g., if the file is dynamically linked it has to be linked again with all the shared libraries it depends on). In this homework we will not do any relocation (it's too complicated), we'll simply load an ELF file in memory and run it.

Linker and loader need two different views of the ELF file, i.e., **they access it differently**.

On the one hand, the linker needs to know where the DATA, TEXT, BSS, and other sections are to merge them with sections from other libraries. If relocation is required the linker needs to know where the symbol tables and relocation information is.

On the other hand, the loader does not need any of these details. It simply needs to know which parts of the ELF file are code (executable), which are data and read-only data, and where to put the BSS in the memory of a process.

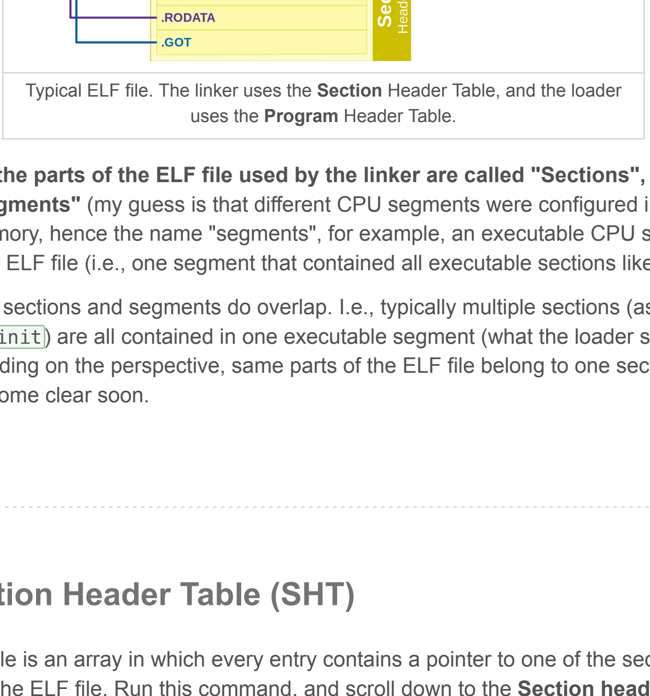
Hence, the ELF file provides two separate views on the data inside the ELF file: **a linker view** with several details, and a **loader view**, a higher level view with less details.

To provide these views each ELF file contains two tables (or arrays):

- **Section Header Table:** With pointers to **sections** to be used by the **linker**.
- **Program Header Table:** With pointers to **segments** to be used by the **loader**.

Both tables are simply arrays of entries that contain information about each part of the ELF file (e.g., where the sections/segments used by the linker/loader are inside the ELF file).

Here is a simple figure of a typical ELF file that starts with the ELF header. The header contains pointers to the locations of Section Header Table and Program Header Table within the ELF file. Then each tables have entries that point to the starting locations of individual sections and segments.



Typical ELF file. The linker uses the **Section** Header Table, and the loader uses the **Program** Header Table.

It's a bit annoying but the parts of the ELF file used by the linker are called "Sections", and the parts used by the loader are called "segments" (my guess is that different CPU segments were configured in the past for each part of the program loaded in memory, hence the name "segments", for example, an executable CPU segment was created for the executable parts of the ELF file (i.e., one segment that contained all executable sections like `.text`, and `.init`, etc.).

Also don't get confused: sections and segments do overlap. I.e., typically multiple sections (as the linker sees the ELF file, such as `.text`, and `.init`) are all contained in one executable segment (what the loader sees). Check the previous image and see how depending on the perspective, same parts of the ELF file belong to one section **and** one segment. Confusing, huh? It will become clear soon.

Linking View: Section Header Table (SHT)

Top

The Section Header Table is an array in which every entry contains a pointer to one of the sections of the ELF file. Lets take a look at what inside the ELF file. Run this command, and scroll down to the **Section headers** you will see all "sections" of the ELF file that the linker can use:

```
$ readelf --section-headers elf
There are 12 section headers, starting at offset 0x458:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000074 00000d 00 WAX  0   0  1
  [ 2] .eh_frame         PROGBITS        00000010 000084 000038 00   A  0   0  4
  [ 3] .comment          PROGBITS        00000000 0000bc 00002b 01  MS  0   0  1
  [ 4] .debug_aranges    PROGBITS        00000000 0000e7 000020 00      0   0  1
  [ 5] .debug_info       PROGBITS        00000000 000107 000066 00      0   0  1
  [ 6] .debug_abbrev     PROGBITS        00000000 00016d 000055 00      0   0  1
  [ 7] .debug_line       PROGBITS        00000000 0001c2 000035 00      0   0  1
  [ 8] .debug_str        PROGBITS        00000000 0001f7 000106 01  MS  0   0  1
  [ 9] .symtab           SYMTAB          00000000 000300 0000e0 10     10  10  4
  [10] .strtab           STRTAB          00000000 0003e0 000024 00      0   0  1
  [11] .shstrtab         STRTAB          00000000 000404 000074 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

Since `elf.c` is a very simple program, it has only `.text` section (i.e., code of the program), a bunch of sections that contain debugging information, and a `.symtab` section that contains imported and exported symbols.

Note, there is no `.data` or `.bss` sections for global variables (there are no globals in `elf.c`).

> You can experiment by adding a not initialized global variable to `elf.c`, recompile, and see the difference in the SHT. Then add an initialized global variable, recompile and check again the SHT.
>
> Make sure of removing the global variables and recompile before continuing with the assignment.)

Moreover, since we linked `elf.c` to be a static executable that is linked to run if loaded at address 0x0 (the `-Ttext 0` in the `Makefile` tells the linker to relocate the executable at linking time to work at `0x0`):

```
$ cat Makefile
...
elf: elf.o
	ld -m elf_i386 -N -e main -Ttext 0 -o elf elf.o
elf.o: elf.c
	$(CC) -c -fno-pic -static -fno-builtin -ggdb -m32 -fno-omit-frame-pointer elf.c
```

The **symbol table** contains these symbols

```
$ readelf --syms elf
Symbol table '.symtab' contains 14 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000010     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 00000000     0 SECTION LOCAL  DEFAULT    7
     8: 00000000     0 SECTION LOCAL  DEFAULT    8
     9: 00000000     0 FILE    LOCAL  DEFAULT  ABS elf.c
    10: 00000048     0 NOTYPE  GLOBAL DEFAULT    2 __bss_start
    11: 00000000    13 FUNC    GLOBAL DEFAULT    1 main
    12: 00000048     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    13: 00000048     0 NOTYPE  GLOBAL DEFAULT    2 _end
```

The `main` is our function (it's `FUNC`, and `GLOBAL`), the `__bss_start`, `_edata`, and `_end` are added by the linker to mark the start and end of the BSS, TEXT, and DATA sections.

If we take a look at the `main` executable, the ELF file is more complicated.

```
$ readelf --section-headers main
There are 35 section headers, starting at offset 0x4794:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            00000168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            00000188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        000001ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          000001cc 0001cc 000090 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0000025c 00025c 0000b8 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          00000314 000314 000012 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         00000328 000328 000040 00   A  6   1  4
  [ 9] .rel.dyn          REL             00000368 000368 000058 08   A  5   0  4
  [10] .rel.plt          REL             000003c0 0003c0 000018 08  AI  5  22  4
  [11] .init             PROGBITS        000003d8 0003d8 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        00000400 000400 000040 04  AX  0   0 16
  [13] .plt.got          PROGBITS        00000440 000440 000010 08  AX  0   0  8
  [14] .text             PROGBITS        00000450 000450 000222 00  AX  0   0 16
  [15] .fini             PROGBITS        00000674 000674 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        00000688 000688 000010 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        00000698 000698 000034 00   A  0   0  4
  [18] .eh_frame         PROGBITS        000006cc 0006cc 0000e0 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      00001ecc 000ecc 000004 04  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      00001ed0 000ed0 000004 04  WA  0   0  4
  [21] .dynamic          DYNAMIC         00001ed4 000ed4 000100 08  WA  6   0  4
  [22] .got              PROGBITS        00001fd4 000fd4 00002c 04  WA  0   0  4
  [23] .data             PROGBITS        00002000 001000 000008 00  WA  0   0  4
  [24] .bss              NOBITS          00002008 001008 000004 00  WA  0   0  1
  [25] .comment          PROGBITS        00000000 001008 00002b 01  MS  0   0  1
  [26] .debug_aranges    PROGBITS        00000000 001033 000020 00      0   0  1
  [27] .debug_info       PROGBITS        00000000 001053 000554 00      0   0  1
  [28] .debug_abbrev     PROGBITS        00000000 0015a7 000132 00      0   0  1
  [29] .debug_line       PROGBITS        00000000 0016d9 0000ee 00      0   0  1
  [30] .debug_str        PROGBITS        00000000 0017c7 0003b4 01  MS  0   0  1
  [31] .symtab           SYMTAB          00000000 001b7c 000480 10     32  48  4
  [32] .strtab           STRTAB          00000000 001ffc 00024f 00      0   0  1
  [33] .shstrtab         STRTAB          00000000 00224b 00013c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

It contains all the section we've mentioned in class: `.text` (main code of the program), `.data` (data section for global variables), `.rodata` (data section for global read-only variables), `.bss` (uninitialized global variables), `.init` (init section to call the constructors that run before `main()`), `.got`(Global Offset Table), `.plt` (Procedure Linking Table for lazy linking of imported functions), and even the `.interp` (the section for the interpreter, i.e., the linker that links dynamically linked program before it runs, typically it's `/lib/ld-linux.so.2` on Linux systems.

Execution view: Program Header Table (PHT)

Top

The Program Header Table contains information for the kernel on how to start the program. The `LOAD` directives determinate what parts of the ELF file get mapped into program memory.

Again, in our `elf` example the program header defines only two segments. And only one of them should be loaded by the operating system in memory to run.

```
$ readelf --program-headers elf

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000074 0x00000000 0x00000000 0x00048 0x00048 RWE 0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .eh_frame
   01
```

The only loadable section is linked to run at address `0x00000000`. We can inspect the `elf` binary with the `objdump` tool to see what is there:

```
$ objdump -d elf

elf:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 55 08                mov    0x8(%ebp),%edx
   6:   8b 45 0c                mov    0xc(%ebp),%eax
   9:   01 d0                   add    %edx,%eax
   b:   5d                      pop    %ebp
   c:   c3                      ret
```

Well, no surprises: it's the main function compiled into machine code.

Putting it All Together

Top

Neither the SHT nor the PHT have fixed positions, they can be located anywhere in an ELF file. To find them the ELF header is used, which is located at the very start of the file.

The first bytes contain the elf magic `"\x7fELF"`, followed by the class ID (32 or 64 bit ELF file), the data format ID (little endian/big endian), the machine type, etc.

At the end of the ELF header are then pointers to the SHT and PHT. Specifically, the Segment Header Table which is used by the linker starts at byte 1120 in the ELF file, and the Program Header Table starts at byte 52 (right after the ELF header)

```
$ readelf --file-header elf
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:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1120 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         12
  Section header string table index: 11
```

Finally, the entry point of this file is at address `0x0`. This is exactly what we told the linker to do — link the program to run at address `0x00000000`. And this is where the `main` function of the `elf.c` file is shown in the `objdump`.

Program Loading in the Kernel

Top

The execution of a program starts inside the kernel, in the `exec("/bin/wc",...)` system call takes a path to the executable file. The kernel reads the ELF header and the program header table (PHT), followed by lots of sanity checks.

For statically linked executables:

1. The kernel reads the PHT, and loads the parts specified in the `LOAD` directives into memory.
2. Once the parts specified in `LOAD` directives are loaded, control can be transferred to the entry point of the program, which is `main()`.

For dynamically linked executables:

1. The kernel reads the PHT, and loads the parts specified in the `INTERP` and `LOAD` directives into memory. Dynamically linked programs always need `/lib/ld-linux.so` as interpreter because it includes some startup code, loads shared libraries needed by the binary, and performs relocations.
2. Once the parts specified in `INTERP` and `LOAD` directives are loaded, control can be transferred to the interpreter.
3. The dynamic linker (contained within the interpreter):
   1. Looks at the `.dynamic` section, whose address is stored in the PHT; and finds:
   2. The `NEEDED` entries determining which libraries have to be loaded before the program can be run.
   3. The `*REL*` entries containing the address of the relocation tables.
   4. The `VER*` entries which contain symbol versioning information.
   5. Etc...
4. The dynamic linker loads the needed libraries and performs relocations (either directly at program startup or later, as soon as the relocated symbol is needed, depending on the relocation type).
5. Finally, control is transferred to the address given by the symbol `_start` in the binary. Normally some gcc/glibc startup code lives there, which in the end calls `main()`.

What do you need to do in this homework: load an ELF file

Top

While ELF might look a bit intimidating, in practice the loading algorithm is trivial:

1. Read the ELF header.
2. One of the ELF header fields tells you the offset of the program header table inside the file.
3. Read each entry of the program header table (i.e., read each program header)
4. Each program header has an offset and size of a specific segment inside the ELF file (e.g., a executable code). You have to read it from the file and load it in memory.
5. When done with all segments, jump to the entry point of the program. (Note since we don't control layout of the address space at the moment, we load the sections at some random place in memory (the place that is allocated for us by the `mmap()` function). Obviously the address of the entry point should be an offset within that random area. Such loading will not work for a real ELF file, but ours is simple: it's statically linked, and contains the code that can run at any location in memory. So even though it's linked to run at `0x0` it will run anywhere where you load it.

Looks manageable. We make a couple of simplifications. First we create a very simple ELF file out of `elf.c`: it contains only one function, it has no data, and the code can be placed anywhere in memory and will run ok (it simply does not refer to any global addresses, all variables are on the stack).

The `main.c` file provides definitions for the structs that match the ELF header and entries of the Program Header Table. So you can simply read the header out of the ELF file with the `open`, `lseek`, and `read` functions, read the offset of the program header table, get the number of the entries in the program header table from the ELF header, and read all entries one by one. If the entry has `ELF_PROG_LOAD` type, you will load it in memory.

To load the segment in memory, you can allocate executable memory with the following function

```
code_va = mmap(NULL, ph.memsz, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS | MAP_PRIVATE,
0, 0);
```

where `ph.memsz` is the size of the segment that we currently are loading.

You should figure out where the entry point of your program is (it is in one of the segments, and since you map the segments at the addresses returned by the `mmap()` function, you need to find where it ends up in your address space.

If you found the entry point then type cast it to the function pointer that matches the function signature of the `sum` function and call it.

```
if(entry != NULL) {
    sum = entry;
    ret = sum(1, 2);
    printf("sum:%d\n", ret);
};
```

Your program should take the name of the executable elf file as its only argument and return the result produced by the function `main` present in that elf file. In particular, if the source code of the elf is:

```
unsigned int main(int a, int b) {
    return a + b;
}
```

Then your loader should print the following result:

```
$ ./main elf
sum:3
```

Please note that different elf files (one at the time) with different definitions of the function `main` will be passed to your loader in order to test the correctness of it.

Extra credit: (10% bonus)

Top

Try loading elf-data.c file. Can you explain why it crashes?

Fix the crash by either ensuring that the file is linked to work at the address that is available inside your process (it's a bit tricky to ensure that any specific address is available, but we'll accept any meaningful solution), or by performing a simple relocation step for the ELF file when it is loaded.

These ELF tutorial and Executable and Linkable Format 101 Part 3: Relocations resources can be helpful.

Submit your work

Top

Submit your solution (the zip archive) through Gradescope in the assignment called HW3 - The ELF Format . Please zip all of your files (main.c, Makefile) and submit them. If you have done extra credit then place that main.c and Makefile in an additional folder coled "extra". **Please note that the non-optional part must be in the root of the zip archive, not inside yet another folder**.

You can resubmit as many times as you wish. If you have any problems with the structure the autograder will tell you. The structure of the zip file should be the following:

```
homework3.zip
├── main.c
├── Makefile
└── extra/       # optional
    ├── main.c   # optional
    └── Makefile # optional
```