Operating System: ELF Format

July 29, 2023

1 ELF format header

1. Compile `hello-world.cpp` under WSL.
2. Use the command `file a.out`, where `a.out` is the compiled output executable.

   ```
   a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
   interpreter /lib64/ld-linux-x86-64.so.2,
   BuildID[sha1]=2317c609af99bb13a86e3c299af29c9a650777e3,
   for GNU/Linux 3.2.0, not stripped
   ```

3. Use the command `readelf -h a.out` to read the header of ELF format, which is defined as follows[1]

   ```
   # define EI_NIDENT 16

   typedef struct {
   unsigned char
   e_ident[EI_NIDENT];
   Elf64_Half e_type;
   Elf64_Half e_machine;
   Elf64_Word e_version;
   Elf64_Addr e_entry;
   Elf64_Off e_phoff;
   Elf64_Off e_shoff;
   Elf64_Word e_flags;
   Elf64_Half e_ehsize;
   Elf64_Half e_phentsize;
   Elf64_Half e_phnum;
   Elf64_Half e_shentsize;
   Elf64_Half e_shnum;
   Elf64_Half e_shstrndx;
   } Elf64_Ehdr;
   ```

   The ELF header information of `a.out` is :

[1] Please refer to here.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1100
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14192 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

2 ELF section

1. Use the following command `readelf -S a.out` to see all the sections that are present in the example `hello-world.cpp` :

```
There are 31 section headers, starting at offset 0x3770:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       0000000000000120  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           00000000000004f8  000004f8
       00000000000000be  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000000005b6  000005b6
       0000000000000018  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          00000000000005d0  000005d0
       0000000000000040  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000610  00000610
       00000000000000c0  0000000000000018   A       6     0     8
```

```
  [11] .rela.plt          RELA             00000000000006d0  000006d0
       0000000000000090  0000000000000018  AI       6     24     8
  [12] .init              PROGBITS         0000000000001000  00001000
       000000000000001b  0000000000000000  AX       0      0     4
  [13] .plt               PROGBITS         0000000000001020  00001020
       0000000000000070  0000000000000010  AX       0      0     16
  [14] .plt.got           PROGBITS         0000000000001090  00001090
       0000000000000010  0000000000000010  AX       0      0     16
  [15] .plt.sec           PROGBITS         00000000000010a0  000010a0
       0000000000000060  0000000000000010  AX       0      0     16
  [16] .text              PROGBITS         0000000000001100  00001100
       00000000000001bd  0000000000000000  AX       0      0     16
  [17] .fini              PROGBITS         00000000000012c0  000012c0
       000000000000000d  0000000000000000  AX       0      0     4
  [18] .rodata            PROGBITS         0000000000002000  00002000
       000000000000000a  0000000000000000   A       0      0     4
  [19] .eh_frame_hdr      PROGBITS         000000000000200c  0000200c
       0000000000000034  0000000000000000   A       0      0     4
  [20] .eh_frame          PROGBITS         0000000000002040  00002040
       00000000000000ac  0000000000000000   A       0      0     8
  [21] .init_array        INIT_ARRAY       0000000000003d90  00002d90
       0000000000000008  0000000000000008  WA       0      0     8
  [22] .fini_array        FINI_ARRAY       0000000000003d98  00002d98
       0000000000000008  0000000000000008  WA       0      0     8
  [23] .dynamic           DYNAMIC          0000000000003da0  00002da0
       00000000000001f0  0000000000000010  WA       7      0     8
  [24] .got               PROGBITS         0000000000003f90  00002f90
       0000000000000070  0000000000000008  WA       0      0     8
  [25] .data              PROGBITS         0000000000004000  00003000
       0000000000000010  0000000000000000  WA       0      0     8
  [26] .bss               NOBITS           0000000000004010  00003010
       0000000000000008  0000000000000000  WA       0      0     1
  [27] .comment           PROGBITS         0000000000000000  00003010
       000000000000002b  0000000000000001  MS       0      0     1
  [28] .symtab            SYMTAB           0000000000000000  00003040
       00000000000003d8  0000000000000018          29     18     8
  [29] .strtab            STRTAB           0000000000000000  00003418
       000000000000023d  0000000000000000          0      0     1
  [30] .shstrtab          STRTAB           0000000000000000  00003655
       000000000000011a  0000000000000000          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),
  D (mbind), l (large), p (processor specific)
```

sections.txt

From the output, it can be seen that there are `.text`, `.data`, `.bss`, `.symtab` and `.shstrtab` sections.

2. To check the code that is present in .text, use the command ọ́bjdump -d a.out.

3. How does the compiler come to know where to put each section in the final ELF. Linkers use a file called the linker descriptor file. It contains information about all the memory in the target machine with its starting address and size, as well as information about the different sections that should be present in the final ELF file and where each section should be loaded in the target machine.

4. The program header in the ELF specifies which section of the ELF file goes to a particular memory location. The program header location in the ELF file is provided by the `e_phoff` variable present in the ELF header. `e_phoff` provides the offset at which the ELF program header is present in the ELF file. `e_phnum` is the number of program header entries in the ELF file and the size of each entry is `e_phentsize`. Use the following command to see the program header information of the example: `readelf -l a.out`

```
Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1100
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000760 0x0000000000000760  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x00000000000002cd 0x00000000000002cd  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000ec 0x00000000000000ec  R      0x1000
  LOAD           0x0000000000002d90 0x0000000000003d90 0x0000000000003d90
                 0x0000000000000280 0x0000000000000288  RW     0x1000
  DYNAMIC        0x0000000000002da0 0x0000000000003da0 0x0000000000003da0
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x000000000000200c 0x000000000000200c 0x000000000000200c
                 0x0000000000000034 0x0000000000000034  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002d90 0x0000000000003d90 0x0000000000003d90
                 0x0000000000000270 0x0000000000000270  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .
       gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .
       rela.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
```

```
    10      .eh_frame_hdr
50  11
    12      .init_array .fini_array .dynamic .got
```

programsection.txt

3 Linker

This is the final stage of compilation.
We know that library functions are not a part of any C program. Thus, the compiler doesn't know the operation of any function, whether it be `std::cin>>` or `std::cout <<`. The definitions of these functions are stored in their respective library but the compiler can link them. So, when we write #include, it includes `iostream` library which gives access to Standard Input and Output. The linker links the object files to the library functions and the program becomes an executable file. Here, `a.out` will be created in an executable format.
In this stage, the final executable ELF is generated. Linker uses all object files created for each programming file in earlier stages as inputs.
It takes a linker descriptor file as input. The linker combines sections from all object files into one final ELF. It instructs the linker descriptor file as to what addresses are to be provided for each section.
It also resolves the symbols used by each object file. If any symbol is not defined, then linker gives an error. These are the stages by which a final executable is generated.

4 Program execution

If we execute any C program, then main() is always the entry point. This is because a startup code (provided by compiler) is always linked to the original program by linker. This startup code consists of a startup function, which usually is `__start` or `Reset_handler`. This is placed at the address that the CPU will execute at boot. This function calls the main() function, which is the starting point of the application.
Whenever we give the command to execute a program, the loader will load the ELF file into memory and inform the CPU with the starting point of the address where this program is loaded.

5 Code Listings

```
1 #include <iostream>
  using namespace std;
3 int main()
  {
5     std::cout << "Hello World!" << std::endl;
  }
```

hello–world.cpp