
Operating System

Dionysios Kalofonos

July 16, 2016

CONTENTS

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Toolchain | 2 |
| 2.1 | Building a cross-compiler | 2 |
| 2.2 | Building GRUB | 3 |
| 2.3 | Debugging with a VM | 4 |
| 3 | Program loading | 6 |
| 3.1 | Introduction | 6 |
| 3.2 | ELF: Executable and Linking Format | 7 |
| 3.3 | Program linking and loading | 11 |
| 4 | Boot Process | 19 |
| 4.1 | Background | 19 |
| 4.2 | GRUB - Multiboot | 19 |
| 4.3 | Bootable ISO | 22 |
| 5 | Descriptor Tables | 24 |
| 5.1 | Global Descriptor Table | 24 |

INTRODUCTION

Operating system is a fork of the Linux kernel version 3.19. The aim of this project is to study computer architecture and provide documentation that covers the areas of study. Such a study requires a kernel. At first I started writing my own kernel, but soon I found myself using snippets from my own code in the documentation. In order to bring greater value to the reader, I decided to use instead the code base of a kernel with real world application. Hence, this fork was born.

The latest version of this document is available on [GitHub Pages](#). The repository of this project is hosted on [GitHub](#).

TOOLCHAIN

2.1 Building a cross-compiler

Since I will be building my OS on Linux it is a good idea to prepare and use a cross compiler. On OSDev wiki there is a very nice [article](#) on the subject. Moreover, on OSDev wiki there is a [tutorial](#) on how to prepare a toolchain for cross compiling an OS. Here I assume you have read the OSDev wiki pages.

First I set a few environmental variables

```
1 ~ # export TARGET=i686-elf
2
3 ~ # mkdir -p ${HOME}/Applications/cross-i686/bin
4 ~ # export PREFIX="${HOME}/Applications/cross-i686"
5 ~ # export PATH="${PREFIX}/bin:${PATH}"
```

Before building GCC, I built binutils (see <https://gnu.org/software/binutils/>)

```
1 ~ # wget http://ftp.gnu.org/gnu/binutils/binutils-2.26.tar.bz2
2 ~ # tar xvfj binutils-2.26.tar.bz2
3 ~ # mkdir binutils-2.26-build
4 ~ # cd binutils-2.26-build
5 binutils-2.26-build # ../binutils-2.26/configure --target=${TARGET}
6                                     --prefix=${PREFIX}
7                                     --with-sysroot
8                                     --disable-nls
9                                     --disable-werror
10 binutils-2.26-build # make && make install
```

It is advisable that when we build a GCC cross compiler to build a version that is close to the version of the GCC compiler we use to compile. My laptop has

```
1 ~ # gcc --version
2 gcc (Debian 4.9.2-10) 4.9.2
```

so I downloaded a version based on the 4.9.x release (see <https://gcc.gnu.org/>)

```
1 ~ # wget http://www.netgull.com/gcc/releases/gcc-4.9.3/gcc-4.9.3.tar.gz
```

As advised on the OSDev wiki, I also downloaded the GNU GMP, GNU MPFR, GNU MPC and the ISL library

```
1 gcc-4.9.3 # ./contrib/download_prerequisites
```

However, the script that came with GCC did not work for me. This is the patch i applied:

```
1 --- contrib/download_prerequisites    2014-02-13 14:06:48.000000000 +0000
2 +++ download_prerequisites    2016-07-02 14:25:57.444427985 +0100
3 @@ -29,15 +29,17 @@
4     GMP=gmp-4.3.2
5     MPC=mpc-0.8.1
6
7 -wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$MPFR.tar.bz2 || exit 1
```

```

8 +URL='http://ftp.vim.org/languages/gcc/infrastructure'
9 +
10 +wget ${URL}/${MPFR.tar.bz2} || exit 1
11 tar xjf $MPFR.tar.bz2 || exit 1
12 ln -sf $MPFR mpfr || exit 1
13
14 -wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$GMP.tar.bz2 || exit 1
15 +wget ${URL}/${GMP.tar.bz2} || exit 1
16 tar xjf $GMP.tar.bz2 || exit 1
17 ln -sf $GMP gmp || exit 1
18
19 -wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$MPC.tar.gz || exit 1
20 +wget ${URL}/${MPC.tar.gz} || exit 1
21 tar xzf $MPC.tar.gz || exit 1
22 ln -sf $MPC mpc || exit 1
23
24 @@ -46,11 +48,11 @@
25     ISL=isl-0.12.2
26     CLOOG=cloog-0.18.1
27
28 - wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$ISL.tar.bz2 || exit 1
29 + wget ${URL}/${ISL.tar.bz2} || exit 1
30 tar xjf $ISL.tar.bz2 || exit 1
31 ln -sf $ISL isl || exit 1
32
33 - wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$CLOOG.tar.gz || exit 1
34 + wget ${URL}/${CLOOG.tar.gz} || exit 1
35 tar xzf $CLOOG.tar.gz || exit 1
36 ln -sf $CLOOG cloog || exit 1
37 fi

```

Finally I compiled GCC

```

1 ~ # mkdir gcc-4.9.3-build
2 ~ # cd gcc-4.9.3-build
3 gcc-4.9.3-build # ../gcc-4.9.3/configure --target=${TARGET}
4                                     --prefix=${PREFIX}
5                                     --disable-nls
6                                     --enable-languages=c,c++
7                                     --without-headers
8 gcc-4.9.3-build # make all-gcc
9 gcc-4.9.3-build # make all-target-libgcc
10 gcc-4.9.3-build # make install-gcc
11 gcc-4.9.3-build # make install-target-libgcc

```

I wanted also to prepare a cross compiler for x86_64 so I set the following environmental variables and repeated the previous steps

```

1 ~ # export TARGET=x86_64-elf
2
3 ~ # mkdir -p ${HOME}/Applications/cross-x86_64/bin
4 ~ # export PREFIX="$HOME/Applications/cross-x86_64"
5 ~ # export PATH="$PREFIX/bin:$PATH"

```

2.2 Building GRUB

Instructions for downloading GRUB can be found on the [GRUB 2 website](http://www.gnu.org/software/grub/). I cloned the GRUB repository:

```

1 # git clone git://git.savannah.gnu.org/grub.git

```

In the root directory there is an INSTALL file with instructions for compiling GRUB. I compiled GRUB through

the following steps:

```
1 # ./autogen.sh
2 # ./configure
3 # make install
```

From the installation of GRUB we are interested in grub-mkrescue which we will be using for preparing bootable ISOs, and the lib directory which contains the modules:

```
1 # find ./grub2 -iwholename '*mkrescue' -o -iwholename '*lib/grub/*' -prune
2 ./grub2/lib/grub/i386-pc
3 ./grub2/bin/grub-mkrescue
```

2.3 Debugging with a VM

During development I will be using a virtual machine for debugging. One solution is [QEMU](#) which has support for [GDB](#) as well as [Valgrind](#). An alternative is [VirtualBox](#) which has a built in [debugger](#). Unfortunately on VirtualBox breakpoints do not work when hardware virtualisation is enabled, which is a requirement for a 64bit VM. The following is an example of a VirtualBox VM configuration with which debugging can be used:

```
1 # VBoxManage showvminfo "OS"
2 [...]
3 Guest OS:          Other/Unknown
4 [...]
5 Memory size:       16MB
6 Page Fusion:       off
7 VRAM size:         16MB
8 CPU exec cap:      20%
9 HPET:              off
10 Chipset:           ich9
11 Firmware:          BIOS
12 Number of CPUs:    1
13 PAE:               off
14 Long Mode:         off
15 Synthetic CPU:     off
16 CUID overrides:    None
17 Boot menu mode:    message and menu
18 Boot Device (1):   DVD
19 Boot Device (2):   HardDisk
20 Boot Device (3):   Not Assigned
21 Boot Device (4):   Not Assigned
22 ACPI:              on
23 IOAPIC:             on
24 Time offset:       0ms
25 RTC:               UTC
26 Hardw. virt.ext:   off
27 Nested Paging:     off
28 Large Pages:       off
29 VT-x VPID:         on
30 VT-x unr. exec.:   on
31 [...]
```

To enable the debug menu by default we need to set the GUI/Dbg/Enabled property of the VM:

```
1 # VBoxManage setextradata "OS" 'GUI/Dbg/Enabled' true
2 # VBoxManage getextradata "OS" 'GUI/Dbg/Enabled'
3 Value: true
```

From the debug menu we can launch the console, through which we can set breakpoints, step through instructions, display the registers and many more. For example let us suppose we have a kernel with the following main function at address 0x0010200b:

```

1 0010200b <main>:
2   10200b:      55                push    %ebp
3   10200c:      89 e5                mov     %esp,%ebp
4   10200e:      b8 ef be ad de      mov     $0xdeadbeef,%eax
5   102013:      5d                  pop     %ebp
6   102014:      c3                  ret

```

We can set a breakpoint in the VirtualBox console to break the execution at our main function:

```

1 VBoxDbg> br 0010200b 1 'echo main'
2 Set REM breakpoint 4 at 000000000010200b

```

When the breakpoint is reached the console shows:

```

1 VBoxDbg> main
2 eax=2badb002 ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
3 eip=0010200b esp=0007fefc ebp=00000000 iopl=0  nv up di pl nz na po nc
4 cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018             eflags=00000006
5 0010:0010200b 55                push    ebp

```

We can press 't' to do a single step:

```

1 VBoxDbg> t
2 VBoxDbg>
3 dbgf event: Single step! (rem)
4 eax=2badb002 ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
5 eip=0010200c esp=0007fef8 ebp=00000000 iopl=0  nv up di pl nz na po nc
6 cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018             eflags=00000006
7 0010:0010200c 89 e5                mov     ebp, esp
8 VBoxDbg> t
9 VBoxDbg>
10 dbgf event: Single step! (rem)
11 eax=2badb002 ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
12 eip=0010200e esp=0007fef8 ebp=0007fef8 iopl=0  nv up di pl nz na po nc
13 cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018             eflags=00000006
14 0010:0010200e b8 ef be ad de      mov     eax, 0deadbeefh
15 VBoxDbg> t
16 VBoxDbg>
17 dbgf event: Single step! (rem)
18 eax=deadbeef ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
19 eip=00102013 esp=0007fef8 ebp=0007fef8 iopl=0  nv up di pl nz na po nc
20 cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018             eflags=00000006
21 0010:00102013 5d                  pop     ebp

```

We can display the registers with 'r':

```

1 VBoxDbg> r
2 eax=deadbeef ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
3 eip=00102013 esp=0007fef8 ebp=0007fef8 iopl=0  nv up di pl nz na po nc
4 cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018             eflags=00000006
5 0010:00102013 5d                  pop     ebp

```

PROGRAM LOADING

3.1 Introduction

Let us start the discussion by writing a hello world program

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     printf("Hello world!\n");
6     return 0;
7 }
```

Using the compiler of our distribution we can build this program as expected

```
1 # gcc main.c
2 # ./a.out
3 Hello world!
```

However, if we try to build the same program with our cross compilers we get

```
1 # i686-elf-gcc main.c
2 main.c:1:19: fatal error: stdio.h: No such file or directory
3 #include <stdio.h>
4             ^
5 compilation terminated.
```

Many parts of the C standard library rely on having an operating system. Since we are writing an operating system, we get only a small subset of the C standard library, which can be found in the include subdirectory of our cross compiler.

If we strip away all the code that relies on the standard library we end up

```
1 int main(void)
2 {
3     return 0;
4 }
```

If we try to compile this, we get

```
1 # i686-elf-gcc main.c -nostdlib -ffreestanding
2 ld: warning: cannot find entry symbol _start; defaulting to 08048054
```

This says that we do not have a C runtime either. Since our cross compiler is built without targeting a specific OS, it does not know which loader will be used to execute our program.

3.2 ELF: Executable and Linking Format

Before we discuss in detail the C runtime, and how a program is loaded for execution, we need to have a look at the **ELF** format. **ELF** is the format we chose for our kernel executable, by building our cross-compilers with the target

```
1 ~ # export TARGET=i686-elf
```

Let us compile and study the executable of the program

```
1 int main(void)
2 {
3     return 0;
4 }

1 # gcc main.c
2 # readelf -a a.out
3 ELF Header:
4   Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
5   Class:                               ELF32
6   Data:                               2's complement, little endian
7   Version:                             1 (current)
8   OS/ABI:                              UNIX - System V
9   ABI Version:                         0
10  Type:                                 EXEC (Executable file)
11  Machine:                             Intel 80386
12  Version:                             0x1
13  Entry point address:                  0x80482d0
14  Start of program headers:             52 (bytes into file)
15  Start of section headers:            3584 (bytes into file)
16  Flags:                                0x0
17  Size of this header:                  52 (bytes)
18  Size of program headers:              32 (bytes)
19  Number of program headers:            8
20  Size of section headers:              40 (bytes)
21  Number of section headers:            30
22  Section header string table index: 27
```

Magic The first four bytes of the file hold a magic number identifying the file as an **ELF** object file, ie. 0x7f, 0x45 = E, 0x4c = L, 0x46 = F.

Entry point address The address of the `_start` function of the program. This is the first function that is being run during program execution. In the next chapter we discuss it in detail.

Flags Flags associated with the file. For 32 bit files this is always zero.

```
1 Section Headers:
2   [Nr] Name                Type                Addr      Off      Size    ES Flg Lk Inf Al
3   [ 0]                     NULL                00000000  000000  000000  00      0  0  0  0
4   [ 1] .interp              PROGBITS            08048134  000134  000013  00      A  0  0  1
5   [ 2] .note.ABI-tag         NOTE                08048148  000148  000020  00      A  0  0  4
6   [ 3] .note.gnu.build-id    NOTE                08048168  000168  000024  00      A  0  0  4
7   [ 4] .gnu.hash             GNU_HASH            0804818c  00018c  000020  04      A  5  0  4
8   [ 5] .dynsym               DYNSYM             080481ac  0001ac  000040  10      A  6  1  4
9   [ 6] .dynstr               STRTAB             080481ec  0001ec  000045  00      A  0  0  1
10  [ 7] .gnu.version          VERSYM             08048232  000232  000008  02      A  5  0  2
11  [ 8] .gnu.version_r         VERNEED            0804823c  00023c  000020  00      A  6  1  4
12  [ 9] .rel.dyn               REL                0804825c  00025c  000008  08      A  5  0  4
13  [10] .rel.plt               REL                08048264  000264  000010  08     AI  5 12  4
14  [11] .init                 PROGBITS            08048274  000274  000023  00     AX  0  0  4
15  [12] .plt                  PROGBITS            080482a0  0002a0  000030  04     AX  0  0 16
16  [13] .text                  PROGBITS            080482d0  0002d0  000182  00     AX  0  0 16
17  [14] .fini                 PROGBITS            08048454  000454  000014  00     AX  0  0  4
```

| | | | | | | | | | | | |
|----|------|---------------|------------|----------|--------|--------|----|----|----|----|---|
| 18 | [15] | .rodata | PROGBITS | 08048468 | 000468 | 000008 | 00 | A | 0 | 0 | 4 |
| 19 | [16] | .eh_frame_hdr | PROGBITS | 08048470 | 000470 | 00002c | 00 | A | 0 | 0 | 4 |
| 20 | [17] | .eh_frame | PROGBITS | 0804849c | 00049c | 0000b0 | 00 | A | 0 | 0 | 4 |
| 21 | [18] | .init_array | INIT_ARRAY | 0804954c | 00054c | 000004 | 00 | WA | 0 | 0 | 4 |
| 22 | [19] | .fini_array | FINI_ARRAY | 08049550 | 000550 | 000004 | 00 | WA | 0 | 0 | 4 |
| 23 | [20] | .jcr | PROGBITS | 08049554 | 000554 | 000004 | 00 | WA | 0 | 0 | 4 |
| 24 | [21] | .dynamic | DYNAMIC | 08049558 | 000558 | 0000e8 | 08 | WA | 6 | 0 | 4 |
| 25 | [22] | .got | PROGBITS | 08049640 | 000640 | 000004 | 04 | WA | 0 | 0 | 4 |
| 26 | [23] | .got.plt | PROGBITS | 08049644 | 000644 | 000014 | 04 | WA | 0 | 0 | 4 |
| 27 | [24] | .data | PROGBITS | 08049658 | 000658 | 000008 | 00 | WA | 0 | 0 | 4 |
| 28 | [25] | .bss | NOBITS | 08049660 | 000660 | 000004 | 00 | WA | 0 | 0 | 1 |
| 29 | [26] | .comment | PROGBITS | 00000000 | 000660 | 000039 | 01 | MS | 0 | 0 | 1 |
| 30 | [27] | .shstrtab | STRTAB | 00000000 | 000699 | 000106 | 00 | | 0 | 0 | 1 |
| 31 | [28] | .symtab | SYMTAB | 00000000 | 0007a0 | 000420 | 10 | | 29 | 45 | 4 |
| 32 | [29] | .strtab | STRTAB | 00000000 | 000bc0 | 00023f | 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)

.init This section holds initialisation routines, which are executed before the main program entry point (ie. main function for C programs). This section is populated by the linker, according to the target OS, and it can be customised with a linker script.

.fini This section holds termination routines, which are executed when a program terminates. This section is populated by the linker, according to the target OS, and it can be customised with a linker script.

.text This section holds the executable instructions of a program.

.data This section holds initialised variables.

.bss This section holds uninitialised variables, which are initialised to zero when the program starts executing.

.rodata This section holds initialised readonly variables.

.plt This section holds the procedure linkage table.

| | | | | | | | | | |
|----|--|--|------------|------------|---------|---------|-----|--------|--|
| 1 | Program Headers: | | | | | | | | |
| 2 | Type | Offset | VirtAddr | PhysAddr | FileSiz | MemSiz | Flg | Align | |
| 3 | PHDR | 0x000034 | 0x08048034 | 0x08048034 | 0x00100 | 0x00100 | R E | 0x4 | |
| 4 | INTERP | 0x000134 | 0x08048134 | 0x08048134 | 0x00013 | 0x00013 | R | 0x1 | |
| 5 | [Requesting program interpreter: /lib/ld-linux.so.2] | | | | | | | | |
| 6 | LOAD | 0x000000 | 0x08048000 | 0x08048000 | 0x0054c | 0x0054c | R E | 0x1000 | |
| 7 | LOAD | 0x00054c | 0x0804954c | 0x0804954c | 0x00114 | 0x00118 | RW | 0x1000 | |
| 8 | DYNAMIC | 0x000558 | 0x08049558 | 0x08049558 | 0x000e8 | 0x000e8 | RW | 0x4 | |
| 9 | NOTE | 0x000148 | 0x08048148 | 0x08048148 | 0x00044 | 0x00044 | R | 0x4 | |
| 10 | GNU_EH_FRAME | 0x000470 | 0x08048470 | 0x08048470 | 0x0002c | 0x0002c | R | 0x4 | |
| 11 | GNU_STACK | 0x000000 | 0x00000000 | 0x00000000 | 0x00000 | 0x00000 | RW | 0x10 | |
| 12 | | | | | | | | | |
| 13 | Section to Segment mapping: | | | | | | | | |
| 14 | Segment Sections... | | | | | | | | |
| 15 | 00 | | | | | | | | |
| 16 | 01 | .interp | | | | | | | |
| 17 | 02 | .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr | | | | | | | |
| 18 | | .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text | | | | | | | |
| 19 | | .fini .rodata .eh_frame_hdr .eh_frame | | | | | | | |
| 20 | 03 | .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss | | | | | | | |
| 21 | 04 | .dynamic | | | | | | | |
| 22 | 05 | .note.ABI-tag .note.gnu.build-id | | | | | | | |
| 23 | 06 | .eh_frame_hdr | | | | | | | |
| 24 | 07 | | | | | | | | |

To quote the [ELF](#) standard:

“An executable or shared object file’s program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file segment contains one or more sections.”

```

1 Dynamic section at offset 0x558 contains 24 entries:
2   Tag          Type          Name/Value
3   0x00000001 (NEEDED)          Shared library: [libc.so.6]
4   0x0000000c (INIT)          0x8048274
5   0x0000000d (FINI)          0x8048454
6   0x00000019 (INIT_ARRAY)      0x804954c
7   0x0000001b (INIT_ARRAYSZ)    4 (bytes)
8   0x0000001a (FINI_ARRAY)      0x8049550
9   0x0000001c (FINI_ARRAYSZ)    4 (bytes)
10  0x6ffffef5 (GNU_HASH)        0x804818c
11  0x00000005 (STRTAB)          0x80481ec
12  0x00000006 (SYMTAB)          0x80481ac
13  0x0000000a (STRSZ)           69 (bytes)
14  0x0000000b (SYMENT)          16 (bytes)
15  0x00000015 (DEBUG)           0x0
16  0x00000003 (PLTGOT)          0x8049644
17  0x00000002 (PLTRELSZ)        16 (bytes)
18  0x00000014 (PLTREL)          REL
19  0x00000017 (JMPREL)          0x8048264
20  0x00000011 (REL)             0x804825c
21  0x00000012 (RELSZ)           8 (bytes)
22  0x00000013 (RELENT)          8 (bytes)
23  0x6ffffffe (VERNEED)         0x804823c
24  0x6fffffff (VERNEEDNUM)      1
25  0x6ffffff0 (VERSYM)          0x8048232
26  0x00000000 (NULL)           0x0

```

.dynamic This section holds dynamic linking information. Dynamic linking (see the [ELF](#) standard, part 2), takes place during program execution. During the `exec()` system call, control is passed to an interpreter who is responsible for reading the executable’s segments into memory.

```

1 Relocation section '.rel.dyn' at offset 0x25c contains 1 entries:
2   Offset      Info      Type           Sym.Value  Sym. Name
3   08049640    00000106 R_386_GLOB_DAT  00000000   __gmon_start__
4
5 Relocation section '.rel.plt' at offset 0x264 contains 2 entries:
6   Offset      Info      Type           Sym.Value  Sym. Name
7   08049650    00000107 R_386_JUMP_SLOT 00000000   __gmon_start__
8   08049654    00000207 R_386_JUMP_SLOT 00000000   __libc_start_main

```

.rel.dyn This section holds relocation information for the `.dynamic` section.

.rel.plt This section holds relocation information for the `.plt` section.

From the [ELF](#) standard:

“Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process’s program image.”

```

1 Symbol table '.dynsym' contains 4 entries:
2   Num:      Value      Size Type      Bind      Vis      Ndx Name
3   0: 00000000      0 NOTYPE   LOCAL   DEFAULT   UND
4   1: 00000000      0 NOTYPE   WEAK    DEFAULT   UND __gmon_start__
5   2: 00000000      0 FUNC     GLOBAL  DEFAULT   UND __libc_start_main@GLIBC_2.0 (2)
6   3: 0804846c      4 OBJECT   GLOBAL  DEFAULT   15 _IO_stdin_used

```

.dynsym This section holds the dynamic linking symbol table.

```

1 Symbol table '.symtab' contains 66 entries:
2   Num:      Value      Size Type      Bind      Vis      Ndx Name
3       0: 00000000      0 NOTYPE   LOCAL   DEFAULT   UND
4       1: 08048134      0 SECTION LOCAL   DEFAULT    1
5       2: 08048148      0 SECTION LOCAL   DEFAULT    2
6       3: 08048168      0 SECTION LOCAL   DEFAULT    3
7       4: 0804818c      0 SECTION LOCAL   DEFAULT    4
8       5: 080481ac      0 SECTION LOCAL   DEFAULT    5
9       6: 080481ec      0 SECTION LOCAL   DEFAULT    6
10      7: 08048232      0 SECTION LOCAL   DEFAULT    7
11      8: 0804823c      0 SECTION LOCAL   DEFAULT    8
12      9: 0804825c      0 SECTION LOCAL   DEFAULT    9
13     10: 08048264      0 SECTION LOCAL   DEFAULT   10
14     11: 08048274      0 SECTION LOCAL   DEFAULT   11
15     12: 080482a0      0 SECTION LOCAL   DEFAULT   12
16     13: 080482d0      0 SECTION LOCAL   DEFAULT   13
17     14: 08048454      0 SECTION LOCAL   DEFAULT   14
18     15: 08048468      0 SECTION LOCAL   DEFAULT   15
19     16: 08048470      0 SECTION LOCAL   DEFAULT   16
20     17: 0804849c      0 SECTION LOCAL   DEFAULT   17
21     18: 0804954c      0 SECTION LOCAL   DEFAULT   18
22     19: 08049550      0 SECTION LOCAL   DEFAULT   19
23     20: 08049554      0 SECTION LOCAL   DEFAULT   20
24     21: 08049558      0 SECTION LOCAL   DEFAULT   21
25     22: 08049640      0 SECTION LOCAL   DEFAULT   22
26     23: 08049644      0 SECTION LOCAL   DEFAULT   23
27     24: 08049658      0 SECTION LOCAL   DEFAULT   24
28     25: 08049660      0 SECTION LOCAL   DEFAULT   25
29     26: 00000000      0 SECTION LOCAL   DEFAULT   26
30     27: 00000000      0 FILE     LOCAL   DEFAULT  ABS crtstuff.c
31     28: 08049554      0 OBJECT   LOCAL   DEFAULT   20 __JCR_LIST__
32     29: 08048310      0 FUNC     LOCAL   DEFAULT   13 deregister_tm_clones
33     30: 08048340      0 FUNC     LOCAL   DEFAULT   13 register_tm_clones
34     31: 08048380      0 FUNC     LOCAL   DEFAULT   13 __do_global_ctors_aux
35     32: 08049660      1 OBJECT   LOCAL   DEFAULT   25 completed.6279
36     33: 08049550      0 OBJECT   LOCAL   DEFAULT   19 __do_global_ctors_aux_fini
37     34: 080483a0      0 FUNC     LOCAL   DEFAULT   13 frame_dummy
38     35: 0804954c      0 OBJECT   LOCAL   DEFAULT   18 __frame_dummy_init_array_
39     36: 00000000      0 FILE     LOCAL   DEFAULT  ABS main.c
40     37: 00000000      0 FILE     LOCAL   DEFAULT  ABS crtstuff.c
41     38: 08048548      0 OBJECT   LOCAL   DEFAULT   17 __FRAME_END__
42     39: 08049554      0 OBJECT   LOCAL   DEFAULT   20 __JCR_END__
43     40: 00000000      0 FILE     LOCAL   DEFAULT  ABS
44     41: 08049550      0 NOTYPE   LOCAL   DEFAULT   18 __init_array_end
45     42: 08049558      0 OBJECT   LOCAL   DEFAULT   21 _DYNAMIC
46     43: 0804954c      0 NOTYPE   LOCAL   DEFAULT   18 __init_array_start
47     44: 08049644      0 OBJECT   LOCAL   DEFAULT   23 _GLOBAL_OFFSET_TABLE_
48     45: 08048450      2 FUNC     GLOBAL   DEFAULT   13 __libc_csu_fini
49     46: 00000000      0 NOTYPE   WEAK     DEFAULT  UND _ITM_deregisterTMCleanup
50     47: 08048300      4 FUNC     GLOBAL   HIDDEN   13 __x86.get_pc_thunk.bx
51     48: 08049658      0 NOTYPE   WEAK     DEFAULT   24 data_start
52     49: 08049660      0 NOTYPE   GLOBAL   DEFAULT   24 _edata
53     50: 08048454      0 FUNC     GLOBAL   DEFAULT   14 _fini
54     51: 08049658      0 NOTYPE   GLOBAL   DEFAULT   24 __data_start
55     52: 00000000      0 NOTYPE   WEAK     DEFAULT  UND __gmon_start__
56     53: 0804965c      0 OBJECT   GLOBAL   HIDDEN   24 __dso_handle
57     54: 0804846c      4 OBJECT   GLOBAL   DEFAULT   15 _IO_stdin_used
58     55: 00000000      0 FUNC     GLOBAL   DEFAULT  UND __libc_start_main@@GLIBC_
59     56: 080483e0      97 FUNC     GLOBAL   DEFAULT   13 __libc_csu_init
60     57: 08049664      0 NOTYPE   GLOBAL   DEFAULT   25 _end
61     58: 080482d0      0 FUNC     GLOBAL   DEFAULT   13 _start
62     59: 08048468      4 OBJECT   GLOBAL   DEFAULT   15 _fp_hw
63     60: 08049660      0 NOTYPE   GLOBAL   DEFAULT   25 __bss_start

```

```

64      61: 080483cb      10 FUNC      GLOBAL DEFAULT   13 main
65      62: 00000000         0 NOTYPE     WEAK      DEFAULT   UND __Jv_RegisterClasses
66      63: 08049660         0 OBJECT     GLOBAL HIDDEN    24 __TMC_END__
67      64: 00000000         0 NOTYPE     WEAK      DEFAULT   UND __ITM_registerTMCloneTable
68      65: 08048274         0 FUNC      GLOBAL DEFAULT   11 __init

```

.symtab This section holds a symbol table. We can see in this table our main.c file, and our main function at address 080483cb. Try compiling the same program with -static, and see how the symbol table changes.

3.3 Program linking and loading

Let us now study the linking and loading processes. The program we will study is the following

```

1  int main(void)
2  {
3      return 0;
4  }

```

If we try to compile this with the compiler that comes with our distribution, the program compiles cleanly. However, if we use our cross-compilers to compile this program we get errors

```

1  # i686-elf-gcc main.c
2  ld: cannot find crt0.o: No such file or directory
3  ld: cannot find -lc
4  collect2: error: ld returned 1 exit status

```

In this example we asked our cross-compiler to link against the C standard library, but the linker could not find the crt0.o file. What is the crt0.o file?

```

1  # i686-elf-gcc -nostdlib -ffreestanding main.c
2  ld: warning: cannot find entry symbol _start; defaulting to 08048054

```

In this example we asked our cross-compiler to not use the C standard library, but the linker could not find the _start symbol. What is the _start symbol?

If we have a look at the ELF again, we see in the header the field 'Entry point address'

```

1  # readelf -h a.out | grep 'Entry'
2  Entry point address:                0x80482d0

```

Let us disassemble our program and focus on the .text section, which is the section that holds the executable instructions of a program

```

1  # objdump -S a.out
2  a.out:          file format elf32-i386
3
4
5  Disassembly of section .plt:
6
7  [...]
8
9  080482c0 <__libc_start_main@plt>:
10  80482c0:  ff 25 54 96 04 08      jmp     *0x8049654
11  80482c6:  68 08 00 00 00        push   $0x8
12  80482cb:  e9 d0 ff ff ff        jmp     80482a0 <__init+0x2c>
13
14  Disassembly of section .text:
15
16  080482d0 <_start>:
17  80482d0:  31 ed                xor     %ebp,%ebp
18  80482d2:  5e                  pop     %esi

```

```

19 80482d3: 89 e1          mov     %esp,%ecx
20 80482d5: 83 e4 f0       and     $0xfffffffff0,%esp
21 80482d8: 50            push    %eax
22 80482d9: 54            push    %esp
23 80482da: 52            push    %edx
24 80482db: 68 50 84 04 08 push    $0x8048450
25 80482e0: 68 e0 83 04 08 push    $0x80483e0
26 80482e5: 51            push    %ecx
27 80482e6: 56            push    %esi
28 80482e7: 68 cb 83 04 08 push    $0x80483cb
29 80482ec: e8 cf ff ff ff call    80482c0 <__libc_start_main@plt>
30 80482f1: f4            hlt
31 80482f2: 66 90         xchg    %ax,%ax
32 80482f4: 66 90         xchg    %ax,%ax
33 80482f6: 66 90         xchg    %ax,%ax
34 80482f8: 66 90         xchg    %ax,%ax
35 80482fa: 66 90         xchg    %ax,%ax
36 80482fc: 66 90         xchg    %ax,%ax
37 80482fe: 66 90         xchg    %ax,%ax
38
39 08048300 <_x86.get_pc_thunk.bx>:
40 8048300: 8b 1c 24       mov     (%esp),%ebx
41 8048303: c3            ret
42 8048304: 66 90         xchg    %ax,%ax
43 8048306: 66 90         xchg    %ax,%ax
44 8048308: 66 90         xchg    %ax,%ax
45 804830a: 66 90         xchg    %ax,%ax
46 804830c: 66 90         xchg    %ax,%ax
47 804830e: 66 90         xchg    %ax,%ax
48
49 08048310 <deregister_tm_clones>:
50 8048310: b8 63 96 04 08 mov     $0x8049663,%eax
51 8048315: 2d 60 96 04 08 sub     $0x8049660,%eax
52 804831a: 83 f8 06       cmp     $0x6,%eax
53 804831d: 76 1a         jbe     8048339 <deregister_tm_clones+0x29>
54 804831f: b8 00 00 00 00 mov     $0x0,%eax
55 8048324: 85 c0         test    %eax,%eax
56 8048326: 74 11         je      8048339 <deregister_tm_clones+0x29>
57 8048328: 55            push    %ebp
58 8048329: 89 e5         mov     %esp,%ebp
59 804832b: 83 ec 14       sub     $0x14,%esp
60 804832e: 68 60 96 04 08 push    $0x8049660
61 8048333: ff d0         call    *%eax
62 8048335: 83 c4 10       add     $0x10,%esp
63 8048338: c9            leave
64 8048339: f3 c3         repz ret
65 804833b: 90            nop
66 804833c: 8d 74 26 00    lea     0x0(%esi,%eiz,1),%esi
67
68 08048340 <register_tm_clones>:
69 8048340: b8 60 96 04 08 mov     $0x8049660,%eax
70 8048345: 2d 60 96 04 08 sub     $0x8049660,%eax
71 804834a: c1 f8 02       sar     $0x2,%eax
72 804834d: 89 c2         mov     %eax,%edx
73 804834f: c1 ea 1f       shr     $0x1f,%edx
74 8048352: 01 d0         add     %edx,%eax
75 8048354: d1 f8         sar     %eax
76 8048356: 74 1b         je      8048373 <register_tm_clones+0x33>
77 8048358: ba 00 00 00 00 mov     $0x0,%edx
78 804835d: 85 d2         test    %edx,%edx
79 804835f: 74 12         je      8048373 <register_tm_clones+0x33>
80 8048361: 55            push    %ebp
81 8048362: 89 e5         mov     %esp,%ebp

```

```

82 8048364: 83 ec 10          sub    $0x10,%esp
83 8048367: 50               push   %eax
84 8048368: 68 60 96 04 08   push   $0x8049660
85 804836d: ff d2           call   *%edx
86 804836f: 83 c4 10          add    $0x10,%esp
87 8048372: c9              leave
88 8048373: f3 c3           repz  ret
89 8048375: 8d 74 26 00      lea    0x0(%esi,%eiz,1),%esi
90 8048379: 8d bc 27 00 00 00 lea    0x0(%edi,%eiz,1),%edi
91
92 08048380 <__do_global_dtors_aux>:
93 8048380: 80 3d 60 96 04 08 00 cmpb   $0x0,0x8049660
94 8048387: 75 13           jne    804839c <__do_global_dtors_aux+0x1c>
95 8048389: 55             push   %ebp
96 804838a: 89 e5           mov     %esp,%ebp
97 804838c: 83 ec 08        sub    $0x8,%esp
98 804838f: e8 7c ff ff ff   call   8048310 <deregister_tm_clones>
99 8048394: c6 05 60 96 04 08 01 movb   $0x1,0x8049660
100 804839b: c9             leave
101 804839c: f3 c3           repz  ret
102 804839e: 66 90          xchg   %ax,%ax
103
104 080483a0 <frame_dummy>:
105 80483a0: b8 54 95 04 08   mov     $0x8049554,%eax
106 80483a5: 8b 10           mov     (%eax),%edx
107 80483a7: 85 d2           test    %edx,%edx
108 80483a9: 75 05           jne     80483b0 <frame_dummy+0x10>
109 80483ab: eb 93           jmp     8048340 <register_tm_clones>
110 80483ad: 8d 76 00        lea     0x0(%esi),%esi
111 80483b0: ba 00 00 00 00   mov     $0x0,%edx
112 80483b5: 85 d2           test    %edx,%edx
113 80483b7: 74 f2           je      80483ab <frame_dummy+0xb>
114 80483b9: 55             push   %ebp
115 80483ba: 89 e5           mov     %esp,%ebp
116 80483bc: 83 ec 14        sub    $0x14,%esp
117 80483bf: 50             push   %eax
118 80483c0: ff d2           call   *%edx
119 80483c2: 83 c4 10          add    $0x10,%esp
120 80483c5: c9             leave
121 80483c6: e9 75 ff ff ff   jmp     8048340 <register_tm_clones>
122
123 080483cb <main>:
124 80483cb: 55             push   %ebp
125 80483cc: 89 e5           mov     %esp,%ebp
126 80483ce: b8 00 00 00 00   mov     $0x0,%eax
127 80483d3: 5d             pop     %ebp
128 80483d4: c3             ret
129 80483d5: 66 90          xchg   %ax,%ax
130 80483d7: 66 90          xchg   %ax,%ax
131 80483d9: 66 90          xchg   %ax,%ax
132 80483db: 66 90          xchg   %ax,%ax
133 80483dd: 66 90          xchg   %ax,%ax
134 80483df: 90             nop
135
136 080483e0 <__libc_csu_init>:
137 80483e0: 55             push   %ebp
138 80483e1: 57             push   %edi
139 80483e2: 31 ff          xor     %edi,%edi
140 80483e4: 56             push   %esi
141 80483e5: 53             push   %ebx
142 80483e6: e8 15 ff ff ff   call   8048300 <__x86.get_pc_thunk.bx>
143 80483eb: 81 c3 59 12 00 00 add     $0x1259,%ebx
144 80483f1: 83 ec 1c        sub    $0x1c,%esp

```

```

145 80483f4: 8b 6c 24 30      mov     0x30(%esp),%ebp
146 80483f8: 8d b3 0c ff ff ff lea     -0xf4(%ebx),%esi
147 80483fe: e8 71 fe ff ff    call   8048274 <__init>
148 8048403: 8d 83 08 ff ff ff lea     -0xf8(%ebx),%eax
149 8048409: 29 c6            sub     %eax,%esi
150 804840b: c1 fe 02          sar     $0x2,%esi
151 804840e: 85 f6            test    %esi,%esi
152 8048410: 74 27            je      8048439 <__libc_csu_init+0x59>
153 8048412: 8d b6 00 00 00 00 lea     0x0(%esi),%esi
154 8048418: 8b 44 24 38      mov     0x38(%esp),%eax
155 804841c: 89 2c 24          mov     %ebp,(%esp)
156 804841f: 89 44 24 08      mov     %eax,0x8(%esp)
157 8048423: 8b 44 24 34      mov     0x34(%esp),%eax
158 8048427: 89 44 24 04      mov     %eax,0x4(%esp)
159 804842b: ff 94 bb 08 ff ff ff call   *-0xf8(%ebx,%edi,4)
160 8048432: 83 c7 01          add     $0x1,%edi
161 8048435: 39 f7            cmp     %esi,%edi
162 8048437: 75 df            jne     8048418 <__libc_csu_init+0x38>
163 8048439: 83 c4 1c          add     $0x1c,%esp
164 804843c: 5b              pop     %ebx
165 804843d: 5e              pop     %esi
166 804843e: 5f              pop     %edi
167 804843f: 5d              pop     %ebp
168 8048440: c3              ret
169 8048441: eb 0d            jmp     8048450 <__libc_csu_fini>
170 8048443: 90              nop
171 8048444: 90              nop
172 8048445: 90              nop
173 8048446: 90              nop
174 8048447: 90              nop
175 8048448: 90              nop
176 8048449: 90              nop
177 804844a: 90              nop
178 804844b: 90              nop
179 804844c: 90              nop
180 804844d: 90              nop
181 804844e: 90              nop
182 804844f: 90              nop
183
184 08048450 <__libc_csu_fini>:
185 8048450: f3 c3            repz ret

```

Surprisingly, our main function is only a tiny part of the program's .text section, and is not even the first function being run when the program is loaded. The entry point address we got by reading the ELF, points to the `_start` function. So where does this function come from?

The `_start` function is part of the C library, and is contained in the `crt0.o` file. To quote the [Gentoo documentation](#):

"On uClibc/glibc systems, this object initializes very early ABI requirements (like the stack or frame pointer), setting up the `argc/argv/env` values, and then passing pointers to the `init/fini/main` funcs to the internal `libc` main which in turn does more general bootstrapping before finally calling the real main function.

glibc ports call this file 'start.S' while uClibc ports call this `crt0.S` or `crt1.S` (depending on what their gcc expects)."

But before we go through the `_start` section we need to discuss what happens when a program is run. A program is run through the `execve()` system call (see the man page for `execve`). To quote the [Linux x86 Program Start Up](#):

"To summarize, it will set up a stack for you, and push onto it `argc`, `argv`, and `envp`. The file descriptions 0, 1, and 2, (`stdin`, `stdout`, `stderr`), are left to whatever the shell set them to. The loader does much work for you setting up your relocations, and as we'll see much later, calling your preinitializers. When everything is ready, control is handed to your program by calling `_start()`"

So, in detail, the `_start` section does the following (from the `start.S` [source code](#)):


```

1  080482d0 <_start>:
2  /* Clear the frame pointer, to mark the outermost frame. */
3  80482d0:  31 ed                xor    %ebp,%ebp
4  /* Put argc into %esi. */
5  80482d2:  5e                  pop    %esi
6  /* Put argv into %ecx. */
7  80482d3:  89 e1                mov    %esp,%ecx
8  /* 16-byte alignment. */
9  80482d5:  83 e4 f0             and    $0xfffffffff0,%esp
10 80482d8:  50                  push   %eax
11 80482d9:  54                  push   %esp
12 80482da:  52                  push   %edx
13 /* Push the address of .fini. */
14 80482db:  68 50 84 04 08       push   $0x8048450
15 /* Push the address of .init. */
16 80482e0:  68 e0 83 04 08       push   $0x80483e0
17 /* Push argv. */
18 80482e5:  51                  push   %ecx
19 /* Push argc. */
20 80482e6:  56                  push   %esi
21 /* Push the address of the main function. */
22 80482e7:  68 cb 83 04 08       push   $0x80483cb
23 /* Call the main function through __libc_start_main. */
24 80482ec:  e8 cf ff ff ff       call   80482c0 <__libc_start_main@plt>
25 80482f1:  f4                  hlt
26 80482f2:  66 90                xchg   %ax,%ax
27 80482f4:  66 90                xchg   %ax,%ax
28 80482f6:  66 90                xchg   %ax,%ax
29 80482f8:  66 90                xchg   %ax,%ax
30 80482fa:  66 90                xchg   %ax,%ax
31 80482fc:  66 90                xchg   %ax,%ax
32 80482fe:  66 90                xchg   %ax,%ax

```

The function `__libc_start_main` lives in glibc source tree in `csu/libc-start.c`. The function `__libc_csu_init` is a constructor, while the function `__libc_csu_fini` is a destructor. These functions live in glibc source tree in `csu/elf-init.c`. I will not discuss the inner workings of these function, but if you want to know more please consult the [Linux x86 Program Start Up](#).

So now that we know what `crt0.o` and `_start` are, lets revisit the two error messages that we received when were building with our cross compiler.

```

1  # i686-elf-gcc main.c
2  ld: cannot find crt0.o: No such file or directory
3  ld: cannot find -lc
4  collect2: error: ld returned 1 exit status

```

The standard library that comes with the cross compiler is minimal, as the cross compiler targets an OS unknown to glibc. As such there is no `crt0.o` file.

```

1  # i686-elf-gcc -nostdlib -ffreestanding main.c
2  ld: warning: cannot find entry symbol _start; defaulting to 08048054

```

In this example we asked the cross compiler to build the program without using the standard library. Even though it does not attempt to locate the `crt0.o` file, it needs the entry point of the program. Hence, we need to provide our own `_start` function. A minimal `start.S` file is the following

```

1  .section .text
2  .global _start
3  .type _start, @function
4  _start:
5      andl $0xfffffffff0, %esp    # align the stack to a 16-byte boundary
6      call main                  # call the main function
7  loop:

```

```

8     jmp _start          # go back to _start
9 .size _start, . - _start

```

The most confusing part of this program is the call to “`jmp _start`”. The `_start` function cannot return, as there is no frame before `_start` to continue execution at. Returning from `_start` would result in a segmentation fault. If we were building a program for linux, at this point we would be making a system call to exit the program. However, this is a standalone program and it cannot make system calls. So i decided to let it loop forever, even though that might not be the best approach.

The main program is a minimal CPP program

```

1  #if defined(__cplusplus)
2  extern "C"
3  #endif
4  int main(void)
5  {
6      return 0;
7  }

```

Since we are using our own `start.S` script, we need to provide our own linker script. A simple linker script is the following

```

1  ENTRY(_start)
2
3  SECTIONS
4  {
5      . = 0x10000;
6      .text : { *(.text) }
7      . = 0x8000000;
8      .data : { *(.data) }
9      .bss : { *(.bss) }
10 }

```

This script tells the linker that `_start` is the entry point of the program, and that the program has the sections `.text` starting at address `0x10000`, `.data` starting at address `0x8000000`, and `.bss`. Lets compile and have a look at the ELF

```

1  # i686-elf-as start.S -o start.o
2  # i686-elf-g++ -c -o main.o main.cpp
3  # i686-elf-g++ -T i686.ld -o a.out -ffreestanding -nostdlib start.o main.o
4  # readelf -a a.out
5  ELF Header:
6  Magic:      7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
7  Class:                                ELF32
8  Data:                                2's complement, little endian
9  Version:                                1 (current)
10 OS/ABI:                                UNIX - System V
11 ABI Version:                            0
12 Type:                                    EXEC (Executable file)
13 Machine:                                Intel 80386
14 Version:                                0x1
15 Entry point address:                    0x10000
16 Start of program headers:                52 (bytes into file)
17 Start of section headers:                4424 (bytes into file)
18 Flags:                                    0x0
19 Size of this header:                      52 (bytes)
20 Size of program headers:                  32 (bytes)
21 Number of program headers:                1
22 Size of section headers:                  40 (bytes)
23 Number of section headers:                7
24 Section header string table index:        4
25
26 Section Headers:

```

```

27 [Nr] Name                Type                Addr      Off      Size   ES Flg Lk Inf Al
28 [ 0]                   NULL                00000000 000000 000000 00      0 0 0
29 [ 1] .text                PROGBITS                00010000 001000 000014 00  AX  0 0 1
30 [ 2] .eh_frame            PROGBITS                00010014 001014 000038 00  A   0 0 4
31 [ 3] .comment              PROGBITS                00000000 00104c 000011 01  MS  0 0 1
32 [ 4] .shstrtab             STRTAB                 00000000 001113 000034 00      0 0 1
33 [ 5] .symtab               SYMTAB                 00000000 001060 000090 10      6 7 4
34 [ 6] .strtab              STRTAB                 00000000 0010f0 000023 00      0 0 1
35 Key to Flags:
36 W (write), A (alloc), X (execute), M (merge), S (strings)
37 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
38 O (extra OS processing required) o (OS specific), p (processor specific)
39
40 There are no section groups in this file.
41
42 Program Headers:
43 Type                Offset      VirtAddr    PhysAddr    FileSiz MemSiz   Flg Align
44 LOAD                0x001000 0x00010000 0x00010000 0x0004c 0x0004c R E 0x1000
45
46 Section to Segment mapping:
47 Segment Sections...
48 00      .text .eh_frame
49
50 There is no dynamic section in this file.
51
52 There are no relocations in this file.
53
54 The decoding of unwind sections for machine type Intel 80386 is not currently supported.
55
56 Symbol table '.symtab' contains 9 entries:
57 Num:      Value      Size Type      Bind      Vis      Ndx Name
58 0: 00000000      0 NOTYPE  LOCAL  DEFAULT  UND
59 1: 00010000      0 SECTION LOCAL  DEFAULT    1
60 2: 00010014      0 SECTION LOCAL  DEFAULT    2
61 3: 00000000      0 SECTION LOCAL  DEFAULT    3
62 4: 00000000      0 FILE    LOCAL  DEFAULT  ABS start.o
63 5: 00010008      0 NOTYPE  LOCAL  DEFAULT    1 loop
64 6: 00000000      0 FILE    LOCAL  DEFAULT  ABS main.cpp
65 7: 00010000     10 FUNC    GLOBAL DEFAULT    1 _start
66 8: 0001000a     10 FUNC    GLOBAL DEFAULT    1 main
67
68 No version information found in this file.

```

In the output of `readelf` we see that the entry point is at `0x10000`, we see that the `.text` section is located at the same address, and in the symbol table we see all the symbols of our program. However, we don't see the `.data` and `.bss` sections. Well, we do not have any variables, so these sections have been dropped. So let's recompile with debug information and let's go through GDB

```

1  # i686-elf-as -g start.S -o start.o
2  # i686-elf-g++ -g -c -o main.o main.cpp
3  # i686-elf-g++ -g -T i686.ld -o a.out -ffreestanding -nostdlib start.o main.o
4  # gdb ./a.out
5  Reading symbols from ./a.out...done.
6  (gdb) b _start
7  Breakpoint 1 at 0x10000: file start.S, line 5.
8  (gdb) run
9  Starting program: a.out
10
11  Breakpoint 1, _start () at start.S:5
12  5      andl $0xfffffffff0, %esp
13  (gdb) s
14  6      call main
15  (gdb) s

```

```
16 main () at main.cpp:6
17     6         return 0;
18 (gdb) s
19     7     }
20 (gdb) s
21 _start () at start.S:12
22 12         jmp _start
23 (gdb) s
24
25 Breakpoint 1, _start () at start.S:5
26     5         andl $0xffffffff0, %esp
27 (gdb) q
```

We set a breakpoint at the `_start` function, and we ran the program. The program calls the main function, returns from it, and resumes execution at the `_start` again.

BOOT PROCESS

4.1 Background

According to the [Intel Software Developer's Manual](#) the first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. At this point control is passed to the BIOS which runs diagnostic tests and configures the devices of the system. At the very end BIOS passes control to the boot loader. When the BIOS supports [EFI](#), it loads EFI applications from the EFI System Partition. If the BIOS is legacy, it loads the boot loader from the Master Boot Record (MBR).

4.2 GRUB - Multiboot

The boot loader that we will be using is [GRUB 2](#). GRUB is modular, it is shipped with a large set of modules and it can be easily extended further. It supports both legacy and EFI capable BIOS. GRUB 2 has three interfaces for loading a kernel or a second boot loader, namely chainloader, linux and multiboot.

4.2.1 Chainloading

Chainloading is the method of loading a second boot loader. When GRUB is loaded by a legacy BIOS from the MBR, it expects that the second boot loader will be loaded in the same fashion. As such it expects that the chainloader command will be passed a series of blocks that correspond to the second boot loader (see the [block list syntax](#)). For more information please see the source of the [legacy loader](#).

When GRUB is loaded from an EFI capable BIOS, it expects that the second boot loader will also be an EFI application. As such it expects that the chainloader command will be given a file name of an EFI application to load. For more information please see the source of the [EFI loader](#).

4.2.2 Linux

This is the method of loading a Linux kernel. The first argument to the linux command is the kernel to load, and and subsequent arguments are passed as arguments to the kernel.

4.2.3 Multiboot

Multiboot is the method of loading a kernel that adheres to the Multiboot [version 1](#) or [version 2](#) specifications. For example lets take a minimal kernel program. The `_start` function calls main:

```
1 .section .text
2 .global _start
3 .type _start, @function
4 _start:
5     andl $0xffffffff, %esp
6     call main
```

```

7  loop:
8      hlt
9      jmp loop
10 .size _start, . - _start

```

The main program just returns 0xDEADBEEF:

```

1  #if defined(__cplusplus)
2  extern "C"
3  #endif
4  int main(void)
5  {
6      return 0xDEADBEEF;
7  }

```

The linker script puts the pieces together:

```

1  ENTRY(_start)
2
3  SECTIONS
4  {
5      . = 1M;
6      .text :
7      {
8          *(.text)
9          . = ALIGN(8K);
10     }
11     .data :
12     {
13         *(.data)
14         . = ALIGN(8K);
15     }
16     .bss :
17     {
18         *(.bss)
19         . = ALIGN(8K);
20     }
21     .comment :
22     {
23         *(.comment)
24     }
25 }

```

When we boot this kernel we see that GRUB complains about not being able to find the Multiboot header.

```
error: no multiboot header found.
Press any key to continue..._
```

Multiboot compliant kernels contain a Multiboot header which should appear within the first 32768 bytes of the executable. The following example introduces a `.multiboot` section, which is split in two subsections. The first holds the header for the Multiboot 1 specification, and the second holds the header for the Multiboot 2 specification:

```
1  .section .multiboot
2  .align 8
3  mbAs:
4      .long 0x1BADB002          # MAGIC
5      .long 0x1                # FLAGS
6      .long 0 - 0x1BADB002 - 0x1 # CHECKSUM
7  mbAe:
8      .align 8
9  mbBs:
10     .long 0xE85250D6           # MAGIC
11     .long 0                   # ARCHITECTURE
12     .long mbBe - mbBs         # HEADER LENGTH
13     .long 0 - 0xE85250D6 - 0 - (mbBe - mbBs) # CHECKSUM
14     .short 0                  # END TAG
15     .short 0                  # TAG FLAG
16     .long 8                   # TAG SIZE
17  mbBe:
18  .section .text
19  .global _start
20  .type _start, @function
21  _start:
22      andl $0xffffffff0, %esp
23      call main
24  loop:
25      hlt
26      jmp loop
27  .size _start, . - _start
```

We need to modify our linker script so that the `.multiboot` section appears at the beginning of the executable:

```
1  ENTRY(_start)
2
3  SECTIONS
4  {
5      . = 1M;
```

```

6     .multiboot :
7     {
8         *(.multiboot)
9         . = ALIGN(8K);
10    }
11    .text :
12    {
13        *(.text)
14        . = ALIGN(8K);
15    }
16    .data :
17    {
18        *(.data)
19        . = ALIGN(8K);
20    }
21    .bss :
22    {
23        *(.bss)
24        . = ALIGN(8K);
25    }
26    .comment :
27    {
28        *(.comment)
29    }
30 }

```

4.3 Bootable ISO

In order to boot our VM, we need a bootable device. Making a bootable ISO is the easiest method and it does not require root privileges. The tool we will be using to make the ISO images is grub-mkrescue

```
1 # ./grub2/bin/grub-mkrescue -d ./grub2/lib/grub/i386-pc -o live.iso iso
```

When I used this tool, I got an error message that it could not locate xorriso

```
1 grub-mkrescue: 323: xorriso: not found
```

so i had to install the xorriso library. Then xorriso complained that it could not find the efi.img file

```
1 xorriso : FAILURE : Cannot find path '/efi.img' in loaded ISO image
```

so I had to install the mtools package.

The last argument of this command is a directory that will be included in the ISO. GRUB expects a specific structure

```

1 # find iso
2 iso
3 iso/boot
4 iso/boot/grub
5 iso/boot/grub/grub.cfg
6 iso/boot/a.out

```

Our kernel is the file a.out. The GRUB configuration file contains

```

1 menuentry "OS" {
2     multiboot /boot/a.out
3 }

```

Make sure the open brace is at the same line as the menuentry definition. If you want to boot with the Multiboot 2 header instead, use the multiboot2 command:


```
1 menuentry "OS" {  
2     multiboot2 /boot/a.out  
3 }
```

DESCRIPTOR TABLES

5.1 Global Descriptor Table