Operating System

Dionysios Kalofonos

January 22, 2017

CHAPTER

ONE

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.

CHAPTER

TWO

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

```
~ # export TARGET=i686-elf

~ # mkdir -p ${HOME}/Applications/cross-i686/bin
~ # export PREFIX="${HOME}/Applications/cross-i686"
~ # export PATH="${PREFIX}/bin:${PATH}"
```

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

```
~ # wget http://ftp.gnu.org/gnu/binutils/binutils-2.26.tar.bz2
~ # tar xvfj binutils-2.26.tar.bz2
~ # mkdir binutils-2.26-build
~ # cd binutils-2.26-build
binutils-2.26-build # ../binutils-2.26/configure --target=${TARGET}
                                                 --prefix=${PREFIX}
                                                 --with-sysroot
                                                 --disable-nls
                                                 --disable-werror
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

```
~ # gcc --version
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/

```
~ # 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

```
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:

```
--- contrib/download_prerequisites   2014-02-13 14:06:48.000000000 +0000
+++ download_prerequisites   2016-07-02 14:25:57.444427985 +0100
@@ -29,15 +29,17 @@
 GMP=gmp-4.3.2
 MPC=mpc-0.8.1

-wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$MPFR.tar.bz2 || exit 1
```

```
+URL='http://ftp.vim.org/languages/gcc/infrastructure'
+
+wget ${URL}/$MPFR.tar.bz2 || exit 1
 tar xjf $MPFR.tar.bz2 || exit 1
 ln -sf $MPFR mpfr || exit 1

-wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$GMP.tar.bz2 || exit 1
+wget ${URL}/$GMP.tar.bz2 || exit 1
 tar xjf $GMP.tar.bz2  || exit 1
 ln -sf $GMP gmp || exit 1

-wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$MPC.tar.gz || exit 1
+wget ${URL}/$MPC.tar.gz || exit 1
 tar xzf $MPC.tar.gz || exit 1
 ln -sf $MPC mpc || exit 1

@@ -46,11 +48,11 @@
   ISL=isl-0.12.2
   CLOOG=cloog-0.18.1

-  wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$ISL.tar.bz2 || exit 1
+  wget ${URL}/$ISL.tar.bz2 || exit 1
   tar xjf $ISL.tar.bz2  || exit 1
   ln -sf $ISL isl || exit 1

-  wget ftp://gcc.gnu.org/pub/gcc/infrastructure/$CLOOG.tar.gz || exit 1
+  wget ${URL}/$CLOOG.tar.gz || exit 1
   tar xzf $CLOOG.tar.gz || exit 1
   ln -sf $CLOOG cloog || exit 1
 fi
```

Finally I compiled GCC

```
~ # mkdir gcc-4.9.3-build
~ # cd gcc-4.9.3-build
gcc-4.9.3-build # ../gcc-4.9.3/configure --target=${TARGET}
                                         --prefix=${PREFIX}
                                         --disable-nls
                                         --enable-languages=c,c++
                                         --without-headers
gcc-4.9.3-build # make all-gcc
gcc-4.9.3-build # make all-target-libgcc
gcc-4.9.3-build # make install-gcc
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

```
~ # export TARGET=x86_64-elf

~ # mkdir -p ${HOME}/Applications/cross-x86_64/bin
~ # export PREFIX="$HOME/Applications/cross-x86_64"
~ # export PATH="$PREFIX/bin:$PATH"
```

2.2 Building GRUB

Instructions for downloading GRUB can be found on the GRUB 2 website. I cloned the GRUB repository:

```
# 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:

```
# ./autogen.sh
# ./configure
# 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:

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

2.3 Building Linux

Linux uses the kbuild framework. We can obtain the list of targets by running

```
# make help
```

We can specify a build directory in the following fashion

```
# make O=_build ARCH="x86" tinyconfig
```

In this example the build directory is '_build' and I made the tinyconfig target, which configures the tiniest possible kernel. An alternative method of configuring the kernel easily would be to use the default configuration

```
# make O=_build ARCH="x86" defconfig
```

Having a minimal or default configuration we can run menuconfig to tweak it

```
# make O=_build ARCH="x86" menuconfig
```

Once we are happy with the configuration, we can build the kernel. For the x86 architecture I usually make the isoimage target which creates a bootable iso

```
# make O=_build -j2 isoimage
```

The original isoimage target uses the isolinux boot loader. I replaced isolinux with GRUB, as GRUB is the bootloader that most probably the reader is familiar with. GRUB provides the command grub-mkrescue which builds an ISO image:

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

This command is given the directory where the GRUB modules live. We can set this by running make menuconfig as shown above, and navigating into Operating System > "GRUB modules path". When I ran grub-mkrescue for the first time, I got an error message that it could not locate xorriso

```
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

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

so I had to install the mtools package.

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

```
# find isoimage
isoimage/
isoimage/boot
isoimage/boot/grub
isoimage/boot/grub/grub.cfg
isoimage/boot/bzImage
```

Our kernel is the file "bzImage". The GRUB configuration file contains

```
# cat isoimage/boot/grub/grub.cfg
menuentry "Linux" {
   linux /boot/bzImage
   boot
}
```

Make sure the open brace is at the same line as the menuentry definition.

As part of this project we are customising the Linux build system so that it can build our additions. Previously we mentioned the menu Operating System > "GRUB modules path" where we can declare the path where the GRUB modules live. The menu Device Drivers > Operating System lists the modules that we have added and that we can compile as part of Linux.

2.4 Building Linux modules

An example module is drivers/os/modapi.c, whose contents are not particularly interesting at this point. However, it can serve as an example of how a module can be compiled.

We can run make menuconfig, go to Device Drivers > Operating System and set the "Poke the Module API" to "M". "M" means that the module will be built for dynamic loading, while "*" means that the module will be statically linked against our kernel. Having done so, we can compile all activated modules, against our source tree with the command:

```
make O=_build ARCH="x86" -j4 modules
```

The following command does all the above, plus it compiles our module against the sources of an installed kernel:

```
make -C /lib/modules/3.16.0-4-586/build M=${PWD}/drivers/os CONFIG_OS_MODAPI_C=m modules
```

Having compiled our module we can copy it to a VM running the corresponding kernel and try it out:

```
# mkdir -p /lib/modules/3.16.0-4-586/extra
# mv /tmp/modapi.ko /lib/modules/3.16.0-4-586/extra/
# depmod /lib/modules/3.16.0-4-586/extra/modapi.ko
# modprobe modapi
# lsmod |grep modapi
modapi                 12386  0
# modprobe -r modapi
# tail -n 2 /var/log/messages
Nov 19 17:42:26 vbdeb kernel: [ 2198.076194] modapi.c: 28: mod_init
Nov 19 17:42:40 vbdeb kernel: [ 2211.619753] modapi.c: 36: mod_exit
```

2.5 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:

```
# VBoxManage showvminfo "OS"
[...]
Guest OS:        Other/Unknown
[...]
Memory size:     16MB
Page Fusion:     off
VRAM size:       16MB
CPU exec cap:    20%
HPET:            off
```

```
Chipset:         ich9
Firmware:        BIOS
Number of CPUs:  1
PAE:             off
Long Mode:       off
Synthetic CPU:   off
CPUID overrides: None
Boot menu mode:  message and menu
Boot Device (1): DVD
Boot Device (2): HardDisk
Boot Device (3): Not Assigned
Boot Device (4): Not Assigned
ACPI:            on
IOAPIC:          on
Time offset:     0ms
RTC:             UTC
Hardw. virt.ext: off
Nested Paging:   off
Large Pages:     off
VT-x VPID:       on
VT-x unr. exec.: on
[...]
```

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

```
# VBoxManage setextradata "OS" 'GUI/Dbg/Enabled' true
# VBoxManage getextradata "OS" 'GUI/Dbg/Enabled'
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:

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

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

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

When the breakpoint is reached the console shows:

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

We can press ‘t’ to do a single step:

```
VBoxDbg> t
VBoxDbg>
dbgf event: Single step! (rem)
eax=2badb002 ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=0010200c esp=0007fef8 ebp=00000000 iopl=0 nv up di pl nz na po nc
cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018               eflags=00000006
0010:0010200c 89 e5                   mov ebp, esp
VBoxDbg> t
VBoxDbg>
```

```
dbgf event: Single step! (rem)
eax=2badb002 ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=0010200e esp=0007fef8 ebp=0007fef8 iopl=0 nv up di pl nz na po nc
cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018               eflags=00000006
0010:0010200e b8 ef be ad de          mov eax, 0deadbeefh
VBoxDbg> t
VBoxDbg>
dbgf event: Single step! (rem)
eax=deadbeef ebx=00010000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=00102013 esp=0007fef8 ebp=0007fef8 iopl=0 nv up di pl nz na po nc
cs=0010 ds=0018 es=0018 fs=0018 gs=0018 ss=0018               eflags=00000006
0010:00102013 5d                      pop ebp
```

We can display the registers with ‘r’:

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

CHAPTER THREE

PROGRAM LOADING

3.1 Introduction

Let us start the discussion by writing a hello world program

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello world!\n");
    return 0;
}
```

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

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

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

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

```c
int main(void)
{
    return 0;
}
```

If we try to compile this, we get

```
# i686-elf-gcc main.c -nostdlib -ffreestanding
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

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

Let us compile and study the executable of the program

```
int main(void)
{
    return 0;
}
```

```
# gcc main.c
# readelf -a a.out
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:               0x80482d0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          3584 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  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.

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

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

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0054c 0x0054c R E 0x1000
  LOAD           0x00054c 0x0804954c 0x0804954c 0x00114 0x00118 RW  0x1000
  DYNAMIC        0x000558 0x08049558 0x08049558 0x000e8 0x000e8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x000470 0x08048470 0x08048470 0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10

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

```
Dynamic section at offset 0x558 contains 24 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x8048274
 0x0000000d (FINI)                       0x8048454
 0x00000019 (INIT_ARRAY)                 0x804954c
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x8049550
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x804818c
 0x00000005 (STRTAB)                     0x80481ec
 0x00000006 (SYMTAB)                     0x80481ac
 0x0000000a (STRSZ)                      69 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x8049644
 0x00000002 (PLTRELSZ)                   16 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048264
 0x00000011 (REL)                        0x804825c
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x804823c
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x8048232
 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.

```
Relocation section '.rel.dyn' at offset 0x25c contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049640  00000106 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x264 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049650  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
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."

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

.dynsym This section holds the dynamic linking symbol table.

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

```
    61: 080483cb    10 FUNC    GLOBAL DEFAULT   13 main
    62: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    63: 08049660     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
    64: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    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

```c
int main(void)
{
    return 0;
}
```

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

```
# i686-elf-gcc main.c
ld: cannot find crt0.o: No such file or directory
ld: cannot find -lc
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?

```
# i686-elf-gcc -nostdlib -ffreestanding main.c
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’

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

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

```
# objdump -S a.out
a.out:     file format elf32-i386


Disassembly of section .plt:

[...]

080482c0 <__libc_start_main@plt>:
 80482c0:   ff 25 54 96 04 08       jmp    *0x8049654
 80482c6:   68 08 00 00 00          push   $0x8
 80482cb:   e9 d0 ff ff ff          jmp    80482a0 <_init+0x2c>

Disassembly of section .text:

080482d0 <_start>:
 80482d0:   31 ed                   xor    %ebp,%ebp
 80482d2:   5e                      pop    %esi
```

```
 80482d3:	89 e1                	mov    %esp,%ecx
 80482d5:	83 e4 f0             	and    $0xfffffff0,%esp
 80482d8:	50                   	push   %eax
 80482d9:	54                   	push   %esp
 80482da:	52                   	push   %edx
 80482db:	68 50 84 04 08       	push   $0x8048450
 80482e0:	68 e0 83 04 08       	push   $0x80483e0
 80482e5:	51                   	push   %ecx
 80482e6:	56                   	push   %esi
 80482e7:	68 cb 83 04 08       	push   $0x80483cb
 80482ec:	e8 cf ff ff ff       	call   80482c0 <__libc_start_main@plt>
 80482f1:	f4                   	hlt
 80482f2:	66 90                	xchg   %ax,%ax
 80482f4:	66 90                	xchg   %ax,%ax
 80482f6:	66 90                	xchg   %ax,%ax
 80482f8:	66 90                	xchg   %ax,%ax
 80482fa:	66 90                	xchg   %ax,%ax
 80482fc:	66 90                	xchg   %ax,%ax
 80482fe:	66 90                	xchg   %ax,%ax

08048300 <__x86.get_pc_thunk.bx>:
 8048300:	8b 1c 24             	mov    (%esp),%ebx
 8048303:	c3                   	ret
 8048304:	66 90                	xchg   %ax,%ax
 8048306:	66 90                	xchg   %ax,%ax
 8048308:	66 90                	xchg   %ax,%ax
 804830a:	66 90                	xchg   %ax,%ax
 804830c:	66 90                	xchg   %ax,%ax
 804830e:	66 90                	xchg   %ax,%ax

08048310 <deregister_tm_clones>:
 8048310:	b8 63 96 04 08       	mov    $0x8049663,%eax
 8048315:	2d 60 96 04 08       	sub    $0x8049660,%eax
 804831a:	83 f8 06             	cmp    $0x6,%eax
 804831d:	76 1a                	jbe    8048339 <deregister_tm_clones+0x29>
 804831f:	b8 00 00 00 00       	mov    $0x0,%eax
 8048324:	85 c0                	test   %eax,%eax
 8048326:	74 11                	je     8048339 <deregister_tm_clones+0x29>
 8048328:	55                   	push   %ebp
 8048329:	89 e5                	mov    %esp,%ebp
 804832b:	83 ec 14             	sub    $0x14,%esp
 804832e:	68 60 96 04 08       	push   $0x8049660
 8048333:	ff d0                	call   *%eax
 8048335:	83 c4 10             	add    $0x10,%esp
 8048338:	c9                   	leave
 8048339:	f3 c3                	repz ret
 804833b:	90                   	nop
 804833c:	8d 74 26 00          	lea    0x0(%esi,%eiz,1),%esi

08048340 <register_tm_clones>:
 8048340:	b8 60 96 04 08       	mov    $0x8049660,%eax
 8048345:	2d 60 96 04 08       	sub    $0x8049660,%eax
 804834a:	c1 f8 02             	sar    $0x2,%eax
 804834d:	89 c2                	mov    %eax,%edx
 804834f:	c1 ea 1f             	shr    $0x1f,%edx
 8048352:	01 d0                	add    %edx,%eax
 8048354:	d1 f8                	sar    %eax
 8048356:	74 1b                	je     8048373 <register_tm_clones+0x33>
 8048358:	ba 00 00 00 00       	mov    $0x0,%edx
 804835d:	85 d2                	test   %edx,%edx
 804835f:	74 12                	je     8048373 <register_tm_clones+0x33>
 8048361:	55                   	push   %ebp
 8048362:	89 e5                	mov    %esp,%ebp
```

```
 8048364:	83 ec 10             	sub    $0x10,%esp
 8048367:	50                   	push   %eax
 8048368:	68 60 96 04 08       	push   $0x8049660
 804836d:	ff d2                	call   *%edx
 804836f:	83 c4 10             	add    $0x10,%esp
 8048372:	c9                   	leave
 8048373:	f3 c3                	repz ret
 8048375:	8d 74 26 00          	lea    0x0(%esi,%eiz,1),%esi
 8048379:	8d bc 27 00 00 00 00 	lea    0x0(%edi,%eiz,1),%edi

08048380 <__do_global_dtors_aux>:
 8048380:	80 3d 60 96 04 08 00 	cmpb   $0x0,0x8049660
 8048387:	75 13                	jne    804839c <__do_global_dtors_aux+0x1c>
 8048389:	55                   	push   %ebp
 804838a:	89 e5                	mov    %esp,%ebp
 804838c:	83 ec 08             	sub    $0x8,%esp
 804838f:	e8 7c ff ff ff       	call   8048310 <deregister_tm_clones>
 8048394:	c6 05 60 96 04 08 01 	movb   $0x1,0x8049660
 804839b:	c9                   	leave
 804839c:	f3 c3                	repz ret
 804839e:	66 90                	xchg   %ax,%ax

080483a0 <frame_dummy>:
 80483a0:	b8 54 95 04 08       	mov    $0x8049554,%eax
 80483a5:	8b 10                	mov    (%eax),%edx
 80483a7:	85 d2                	test   %edx,%edx
 80483a9:	75 05                	jne    80483b0 <frame_dummy+0x10>
 80483ab:	eb 93                	jmp    8048340 <register_tm_clones>
 80483ad:	8d 76 00             	lea    0x0(%esi),%esi
 80483b0:	ba 00 00 00 00       	mov    $0x0,%edx
 80483b5:	85 d2                	test   %edx,%edx
 80483b7:	74 f2                	je     80483ab <frame_dummy+0xb>
 80483b9:	55                   	push   %ebp
 80483ba:	89 e5                	mov    %esp,%ebp
 80483bc:	83 ec 14             	sub    $0x14,%esp
 80483bf:	50                   	push   %eax
 80483c0:	ff d2                	call   *%edx
 80483c2:	83 c4 10             	add    $0x10,%esp
 80483c5:	c9                   	leave
 80483c6:	e9 75 ff ff ff       	jmp    8048340 <register_tm_clones>

080483cb <main>:
 80483cb:	55                   	push   %ebp
 80483cc:	89 e5                	mov    %esp,%ebp
 80483ce:	b8 00 00 00 00       	mov    $0x0,%eax
 80483d3:	5d                   	pop    %ebp
 80483d4:	c3                   	ret
 80483d5:	66 90                	xchg   %ax,%ax
 80483d7:	66 90                	xchg   %ax,%ax
 80483d9:	66 90                	xchg   %ax,%ax
 80483db:	66 90                	xchg   %ax,%ax
 80483dd:	66 90                	xchg   %ax,%ax
 80483df:	90                   	nop

080483e0 <__libc_csu_init>:
 80483e0:	55                   	push   %ebp
 80483e1:	57                   	push   %edi
 80483e2:	31 ff                	xor    %edi,%edi
 80483e4:	56                   	push   %esi
 80483e5:	53                   	push   %ebx
 80483e6:	e8 15 ff ff ff       	call   8048300 <__x86.get_pc_thunk.bx>
 80483eb:	81 c3 59 12 00 00    	add    $0x1259,%ebx
 80483f1:	83 ec 1c             	sub    $0x1c,%esp
```

```
 80483f4:   8b 6c 24 30             mov    0x30(%esp),%ebp
 80483f8:   8d b3 0c ff ff ff       lea    -0xf4(%ebx),%esi
 80483fe:   e8 71 fe ff ff          call   8048274 <_init>
 8048403:   8d 83 08 ff ff ff       lea    -0xf8(%ebx),%eax
 8048409:   29 c6                   sub    %eax,%esi
 804840b:   c1 fe 02                sar    $0x2,%esi
 804840e:   85 f6                   test   %esi,%esi
 8048410:   74 27                   je     8048439 <__libc_csu_init+0x59>
 8048412:   8d b6 00 00 00 00       lea    0x0(%esi),%esi
 8048418:   8b 44 24 38             mov    0x38(%esp),%eax
 804841c:   89 2c 24                mov    %ebp,(%esp)
 804841f:   89 44 24 08             mov    %eax,0x8(%esp)
 8048423:   8b 44 24 34             mov    0x34(%esp),%eax
 8048427:   89 44 24 04             mov    %eax,0x4(%esp)
 804842b:   ff 94 bb 08 ff ff ff    call   *-0xf8(%ebx,%edi,4)
 8048432:   83 c7 01                add    $0x1,%edi
 8048435:   39 f7                   cmp    %esi,%edi
 8048437:   75 df                   jne    8048418 <__libc_csu_init+0x38>
 8048439:   83 c4 1c                add    $0x1c,%esp
 804843c:   5b                      pop    %ebx
 804843d:   5e                      pop    %esi
 804843e:   5f                      pop    %edi
 804843f:   5d                      pop    %ebp
 8048440:   c3                      ret
 8048441:   eb 0d                   jmp    8048450 <__libc_csu_fini>
 8048443:   90                      nop
 8048444:   90                      nop
 8048445:   90                      nop
 8048446:   90                      nop
 8048447:   90                      nop
 8048448:   90                      nop
 8048449:   90                      nop
 804844a:   90                      nop
 804844b:   90                      nop
 804844c:   90                      nop
 804844d:   90                      nop
 804844e:   90                      nop
 804844f:   90                      nop

08048450 <__libc_csu_fini>:
 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):

```
080482d0 <_start>:
 /* Clear the frame pointer, to mark the outermost frame. */
 80482d0:    31 ed                      xor    %ebp,%ebp
 /* Put argc into %esi. */
 80482d2:    5e                         pop    %esi
 /* Put argv into %ecx. */
 80482d3:    89 e1                      mov    %esp,%ecx
 /* 16-byte alignment. */
 80482d5:    83 e4 f0                   and    $0xfffffff0,%esp
 80482d8:    50                         push   %eax
 80482d9:    54                         push   %esp
 80482da:    52                         push   %edx
 /* Push the address of .fini. */
 80482db:    68 50 84 04 08             push   $0x8048450
 /* Push the address of .init. */
 80482e0:    68 e0 83 04 08             push   $0x80483e0
 /* Push argv. */
 80482e5:    51                         push   %ecx
 /* Push argc. */
 80482e6:    56                         push   %esi
 /* Push the address of the main function. */
 80482e7:    68 cb 83 04 08             push   $0x80483cb
 /* Call the main function through __libc_start_main. */
 80482ec:    e8 cf ff ff ff             call   80482c0 <__libc_start_main@plt>
 80482f1:    f4                         hlt
 80482f2:    66 90                      xchg   %ax,%ax
 80482f4:    66 90                      xchg   %ax,%ax
 80482f6:    66 90                      xchg   %ax,%ax
 80482f8:    66 90                      xchg   %ax,%ax
 80482fa:    66 90                      xchg   %ax,%ax
 80482fc:    66 90                      xchg   %ax,%ax
 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.

```
# i686-elf-gcc main.c
ld: cannot find crt0.o: No such file or directory
ld: cannot find -lc
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 uknown to glibc. As such there is no crt0.o file.

```
# i686-elf-gcc -nostdlib -ffreestanding main.c
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

```
.section .text
.global _start
.type _start, @function
_start:
   andl $0xfffffff0, %esp  # align the stack to a 16-byte boundary
   call main               # call the main function
loop:
```

```
    jmp _start                # go back to _start
.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

```
#if defined(__cplusplus)
extern "C"
#endif
int main(void)
{
   return 0;
}
```

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

```
ENTRY(_start)

SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

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

```
# i686-elf-as start.S -o start.o
# i686-elf-g++ -c -o main.o main.cpp
# i686-elf-g++ -T i686.ld -o a.out -ffreestanding -nostdlib start.o main.o
# readelf -a a.out
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:               0x10000
Start of program headers:          52 (bytes into file)
Start of section headers:          4424 (bytes into file)
Flags:                             0x0
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         1
Size of section headers:           40 (bytes)
Number of section headers:         7
Section header string table index: 4

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        00010000 001000 000014 00  AX  0   0  1
[ 2] .eh_frame         PROGBITS        00010014 001014 000038 00   A  0   0  4
[ 3] .comment          PROGBITS        00000000 00104c 000011 01  MS  0   0  1
[ 4] .shstrtab         STRTAB          00000000 001113 000034 00      0   0  1
[ 5] .symtab           SYMTAB          00000000 001060 000090 10      6   7  4
[ 6] .strtab           STRTAB          00000000 0010f0 000023 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)

There are no section groups in this file.

Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x001000 0x00010000 0x00010000 0x0004c 0x0004c R E 0x1000

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

There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type Intel 80386 is not currently supported.

Symbol table '.symtab' contains 9 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
1: 00010000     0 SECTION LOCAL  DEFAULT    1
2: 00010014     0 SECTION LOCAL  DEFAULT    2
3: 00000000     0 SECTION LOCAL  DEFAULT    3
4: 00000000     0 FILE    LOCAL  DEFAULT  ABS start.o
5: 00010008     0 NOTYPE  LOCAL  DEFAULT    1 loop
6: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.cpp
7: 00010000    10 FUNC    GLOBAL DEFAULT    1 _start
8: 0001000a    10 FUNC    GLOBAL DEFAULT    1 main

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 dont see the .data and .bss sections. Well, we do not have any variables, so these sections have been dropped. So lets recompile with debug information and lets go through GDB

```
# i686-elf-as -g start.S -o start.o
# i686-elf-g++ -g -c -o main.o main.cpp
# i686-elf-g++ -g -T i686.ld -o a.out -ffreestanding -nostdlib start.o main.o
# gdb ./a.out
Reading symbols from ./a.out...done.
   (gdb) b _start
Breakpoint 1 at 0x10000: file start.S, line 5.
   (gdb) run
Starting program: a.out

   Breakpoint 1, _start () at start.S:5
5               andl $0xfffffff0, %esp
   (gdb) s
6               call main
   (gdb) s
```

```
main () at main.cpp:6
   6              return 0;
(gdb) s
   7       }
(gdb) s
   _start () at start.S:12
12              jmp _start
   (gdb) s

Breakpoint 1, _start () at start.S:5
   5              andl $0xfffffff0, %esp
   (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.

CHAPTER

FOUR

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 conrol 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 syntaxt). For more information please see the source of the lecacy 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 any subsequent arguments are passed as arguments to the kernel. The command adheres to the version 2.10 of the Linux boot protocol. The Linux boot protocol describes the memory layout for BIOS, loader and the kernel. It also describes the header that is used to communicate information between the kernel and the loader.

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:

```
.section .text
.global _start
.type _start, @function
_start:
```

```
    andl $0xfffffff0, %esp
    call main
loop:
    hlt
    jmp loop
.size _start, . - _start
```

The main program just returns 0xDEADBEEF:

```
#if defined(__cplusplus)
extern "C"
#endif
int main(void)
{
    return 0xDEADBEEF;
}
```

The linker script puts the pieces together:

```
ENTRY(_start)

SECTIONS
{
    . = 1M;
    .text :
    {
        *(.text)
        . = ALIGN(8K);
    }
    .data :
    {
        *(.data)
        . = ALIGN(8K);
    }
    .bss :
    {
        *(.bss)
        . = ALIGN(8K);
    }
    .comment :
    {
        *(.comment)
    }
}
```

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

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:

```
.section .multiboot
.align 8
mbAs:
    .long 0x1BADB002            # MAGIC
    .long 0x1                   # FLAGS
    .long 0 - 0x1BADB002 - 0x1  # CHECKSUM
mbAe:
.align 8
mbBs:
    .long 0xE85250D6                           # MAGIC
    .long 0                                    # ARCHITECTURE
    .long mbBe - mbBs                          # HEADER LENGTH
    .long 0 - 0xE85250D6 - 0 - (mbBe - mbBs)  # CHECKSUM
    .short 0                                   # END TAG
    .short 0                                   # TAG FLAG
    .long 8                                    # TAG SIZE
mbBe:
.section .text
.global _start
.type _start, @function
_start:
    andl $0xfffffff0, %esp
    call main
loop:
    hlt
    jmp loop
.size _start, . - _start
```

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

```
ENTRY(_start)

SECTIONS
{
    . = 1M;
```

```
    .multiboot :
    {
       *(.multiboot)
       . = ALIGN(8K);
    }
    .text :
    {
       *(.text)
       . = ALIGN(8K);
    }
    .data :
    {
       *(.data)
       . = ALIGN(8K);
    }
    .bss :
    {
       *(.bss)
       . = ALIGN(8K);
    }
    .comment :
    {
       *(.comment)
    }
}
```

CHAPTER

FIVE

KERNEL MODULES

5.1 Introduction

A good starting point for module development is Linux Device Drivers, Third Edition. As an example we will add the following module to the kernel source tree:

```
# find ./drivers/os/
./drivers/os/
./drivers/os/Kconfig
./drivers/os/modapi.c
./drivers/os/Makefile
```

5.2 modapi.c

The file modapi.c holds the source code of the module:

```
/*
 * Copyright (C) 2016 Dionysios Kalofonos
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */

#include <linux/init.h>          // __init, __exit, module_init, and module_exit
#include <linux/kernel.h>        // several utilities, includes printk
#include <linux/module.h>        // MODULE_LICENSE, MODULE_AUTHOR, MODULE_DESCRIPTION
#include <linux/export.h>        // THIS_MODULE

// module information
#define MODULE_NAME "modapi"
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dionysios Kalofonos");
MODULE_DESCRIPTION("Poke the Module API");

/*
 * Callback for module loading. __init is an attribute meaning that the
 * memory used by this function can be thrown away after initialisation.
 */
static int __init mod_init(void)
{
    printk(KERN_INFO MODULE_NAME ": %d: %s\n", __LINE__, __func__);
    return 0;
}

/*
 * Callback for module unloading. __exit is an attribute meaning that the
```

```
 * function can be ignored when compiling for statically linking.
 */
static void __exit mod_exit(void)
{
    printk(KERN_INFO MODULE_NAME ": %d: %s\n", __LINE__, __func__);
}

// export module functions
module_init(mod_init);
module_exit(mod_exit);
```

5.3 Kconfig and Makefile

With this module we are adding to the drivers directory the sub-directory named “os”, and the module modapi.c. In order for the new directory to be considered during building, we need to append an obj target to the Makefile of the parent directory:

```
# tail drivers/Makefile
obj-$(CONFIG_IPACK_BUS)         += ipack/
obj-$(CONFIG_NTB)               += ntb/
obj-$(CONFIG_FMC)               += fmc/
obj-$(CONFIG_POWERCAP)          += powercap/
obj-$(CONFIG_MCB)               += mcb/
obj-$(CONFIG_RAS)               += ras/
obj-$(CONFIG_THUNDERBOLT)       += thunderbolt/
obj-$(CONFIG_CORESIGHT)         += coresight/
obj-$(CONFIG_ANDROID)           += android/
obj-$(CONFIG_OS_MODAPI_C)       += os/
```

Within the directory “os” we need a makefile, which lists the files that we need to build for the new obj target:

```
# cat drivers/os/Makefile
# MODAPI module

obj-$(CONFIG_OS_MODAPI_C)               += modapi.o
```

Equivalently, we need to setup the config files. In the Kconfig of the parent directory we need to source the new Kconfig:

```
# tail drivers/Kconfig

source "drivers/ras/Kconfig"

source "drivers/thunderbolt/Kconfig"

source "drivers/android/Kconfig"

source "drivers/os/Kconfig"

endmenu
```

Within the “os” directory we introduce a new config file, which allows us to set the value of the variable CONFIG_OS_MODAPI_C. Since we are introducing a new directory, in our Kconfig we also add a menu entry:

```
# cat drivers/os/Kconfig
menu "Operating System"

config OS_MODAPI_C
   tristate "Poke the Module API"
   default n
   help
```

```
      A minimal module showcasing the module API.
endmenu # "Operating System"
```

We set the type of the variable to tristate as this variable can be set to one of ‘n’ for not building the module, ‘M’ for building the module for dynamic loading, and ‘*’ for statically linking the module.

CHAPTER

SIX

DESCRIPTOR TABLES

6.1 Global Descriptor Table