CS_344 Assignment 0A

Akshat Mittal, 200101011

Aug 12, 2022

Part 1: PC Boot Strap

Exercise 1

Modified code is shown along with output below:

```c
// Simple inline assembly example
#include <stdio.h>
int main(int argc, char **argv){
    int x = 1;
    printf("Hello x = %d\n", x);

    // Put in-line assembly here to increment
    // the value of x by 1 using in-line assembly

    asm("add $1 , %0"
        : "=r"(x)
        : "0"(x)
    );

    printf("Hello x = %d after increment\n",x);
    if(x == 2){
        printf("OK\n");
    }
    else{
        printf("ERROR\n");
    }
}
```

```
akshat@akshat:~/Desktop$ g++ ex1.c
akshat@akshat:~/Desktop$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
akshat@akshat:~/Desktop$
```

Using **extended inline assembly**, I added the value **1** to input register and stored the result in output register.

The syntax is:

asm asm-qualifiers (

AssemblerTemplate

: OutputOperands

: InputOperands

: Clobbers

: GotoLabels

);

(Here, we do not require Clobbers and GotoLabels.)

The **asm** keyword is a GNU extension.

From the output, we can see that value of **x** has been incremented by **1**.

- After this task, we are required to launch **QEMU** after cloning it.
- We will open two terminals, in first one, run ***make qemu-nox-gdb*** to start QEMU and in second one, from the same directory, run ***gdb*** and type ***source .gdbinit*** at the prompt.

Exercise 2

I executed the ***si*** command some number of times and got following results:

```
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb)
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb)
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb)
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb)
[f000:e070]    0xfe070: mov    $0x7c4,%dx
0x0000e070 in ?? ()
(gdb)
[f000:e076]    0xfe076: jmp    0x5576cf26
0x0000e076 in ?? ()
(gdb)
[f000:cf24]    0xfcf24: cli
0x0000cf24 in ?? ()
(gdb)
```

1. Compare the values at 0xffc8 with one at given CS and IP location.
2. Conditionally jump if result of previous instruction is not equal
3. Set %edx to 0 since taking xor with itself results in 0.
4. Set %ss (**stack segment register**) to %edx, i.e., 0.
5. Set %sp (**stack point register**) to 0x700.
6. Set %dx to 0x7c4.
7. Jump to the specified location.
8. Clear the **interrupt** flag.

In short, various **flags** and **registers** were initialized here.

Part 2: The Boot Loader

Exercise 3

- We have to set a breakpoint at location 0x7c00, i.e., the location where boot sector will be loaded, using b command: **b *0x7c00**.
- Then using **c** command, we will continue the execution till next breakpoint.
- Then we have to trace the output of **x/Ni** (here N=15) command in source code **bootasm.S** and disassembly in **bootblock.asm**.

Fig. GDB Output

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/15i $eip
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
```

Fig. bootblock.asm

```
10  .code16                          # Assemble for 16-bit mode
11  .globl start
    18 references
12  start:
13    cli                            # BIOS enabled interrupts;
14      7c00: fa                       cli
15
16    # Zero data segment registers DS, ES, and SS.
17    xorw    %ax,%ax                # Set %ax to zero
18      7c01: 31 c0                    xor    %eax,%eax
19    movw    %ax,%ds                # -> Data Segment
20      7c03: 8e d8                    mov    %eax,%ds
21    movw    %ax,%es                # -> Extra Segment
22      7c05: 8e c0                    mov    %eax,%es
23    movw    %ax,%ss                # -> Stack Segment
24      7c07: 8e d0                    mov    %eax,%ss
25
26  00007c09 <seta20.1>:
27
28    # Physical address line A20 is tied to zero so that the
29    # with 2 MB would run software that assumed 1 MB.  Undo
    3 references
30  seta20.1:
31    inb     $0x64,%al                  # Wait for not busy
32      7c09: e4 64                    in     $0x64,%al
33    testb   $0x2,%al
34      7c0b: a8 02                    test   $0x2,%al
35    jnz     seta20.1
36      7c0d: 75 fa                    jne    7c09 <seta20.1>
37
38    movb    $0xd1,%al                  # 0xd1 -> port 0x64
39      7c0f: b0 d1                    mov    $0xd1,%al
40    outb    %al,$0x64
41      7c11: e6 64                    out    %al,$0x64
42
43  00007c13 <seta20.2>:
44
    3 references
45  seta20.2:
46    inb     $0x64,%al                  # Wait for not busy
47      7c13: e4 64                    in     $0x64,%al
```

Fig. bootasm.S

```
10  .code16                       # As
11  .globl start
12  start:
13    cli                         # BIO
14
15    # Zero data segment registers DS,
16    xorw    %ax,%ax             # Se
17    movw    %ax,%ds             # ->
18    movw    %ax,%es             # ->
19    movw    %ax,%ss             # ->
20
21    # Physical address line A20 is ti
22    # with 2 MB would run software th
23  seta20.1:
24    inb     $0x64,%al
25    testb   $0x2,%al
26    jnz     seta20.1
27
28    movb    $0xd1,%al
29    outb    %al,$0x64
30
31  seta20.2:
32    inb     $0x64,%al
33    testb   $0x2,%al
34    jnz     seta20.2
35
36    movb    $0xdf,%al
37    outb    %al,$0x60
38
```

- It is observed that all the instructions are same except few changes in syntax (the suffix **b/w/l** stands for **block/ word/ long word** respectively and is not present in output).

- Here, the underlying commands are same, the only difference being the style of writing.

- The **line 2 of gdb output** is same as **line 17 of bootblock.asm** and **line 16 of bootasm.S** (marked with red arrow) and similarly those marked with blue arrows are same.

- Now, we have to trace various parts of source code and corresponding assembly instructions in **bootmain.c** and **bootblock.asm.**

1. <u>readsect():</u> Read a single sector at offset from disk.

```c
// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

Fig. Source code in bootmain.c

```
// Read a single sector at offset into dst.
381 references
void
readsect(void *dst, uint offset)
{
    7c90: f3 0f 1e fb           endbr32
    7c94: 55                    push   %ebp
    7c95: 89 e5                 mov    %esp,%ebp
    7c97: 57                    push   %edi
    7c98: 53                    push   %ebx
    7c99: 8b 5d 0c              mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c9c: e8 dd ff ff ff        call   7c7e <waitdisk>
}
```

Fig. Corresponding assembly code in bootblock.asm

2. For loop that reads remaining sector of kernel from disk.

```c
  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Fig. bootmain.c

For loop runs from line 327 – 348 in bootblock.asm. We can see on line 329 loop condition is checked. If the condition fails, on line 330, conditional jump will move to address **0x7d91** on line 319.
Now the control will be given to the OS.

```
315    for(; ph < eph; ph++){
316      7d8d: 39 f3                  cmp    %esi,%ebx
317      7d8f: 72 15                  jb     7da6 <bootmain+0x5d>
318    entry();
319      7d91: ff 15 18 00 01 00      call   *0x10018
320  }
321      7d97: 8d 65 f4               lea    -0xc(%ebp),%esp
322      7d9a: 5b                     pop    %ebx
323      7d9b: 5e                     pop    %esi
324      7d9c: 5f                     pop    %edi
325      7d9d: 5d                     pop    %ebp
326      7d9e: c3                     ret
327    for(; ph < eph; ph++){
328      7d9f: 83 c3 20               add    $0x20,%ebx
329      7da2: 39 de                  cmp    %ebx,%esi
330      7da4: 76 eb                  jbe    7d91 <bootmain+0x48>
331      pa = (uchar*)ph->paddr;
332      7da6: 8b 7b 0c               mov    0xc(%ebx),%edi
333      readseg(pa, ph->filesz, ph->off);
334      7da9: 83 ec 04               sub    $0x4,%esp
335      7dac: ff 73 04               pushl  0x4(%ebx)
336      7daf: ff 73 10               pushl  0x10(%ebx)
337      7db2: 57                     push   %edi
338      7db3: e8 44 ff ff ff         call   7cfc <readseg>
339      if(ph->memsz > ph->filesz)
340      7db8: 8b 4b 14               mov    0x14(%ebx),%ecx
341      7dbb: 8b 43 10               mov    0x10(%ebx),%eax
342      7dbe: 83 c4 10               add    $0x10,%esp
343      7dc1: 39 c1                  cmp    %eax,%ecx
344      7dc3: 76 da                  jbe    7d9f <bootmain+0x56>
345        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
346      7dc5: 01 c7                  add    %eax,%edi
347      7dc7: 29 c1                  sub    %eax,%ecx
348  }
```

Fig. bootblock.asm

```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
```

Setting breakpoint at **0x7d91**, i.e., the end of for loop and stepping through boot loader.

- Now let us answer the questions asked:

```
# Switch from real to protected mode.  Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0

//PAGEBREAK!
# Complete the transition to 32-bit protected mode by using a long jmp
# to reload %cs and %eip.  The segment descriptors are set up with no
# translation, so that the mapping is still the identity mapping.
ljmp    $(SEG_KCODE<<3), $start32
```

1. The **ljmp** command makes instruction pointer move from 16-bit to 32-bit mode. It is caused by the 4 lines above this command which ask OS to switch. All instructions after this are in 32-bit mode.

```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call    *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov     %cr4,%eax
0x0010000c in ?? ()
(gdb)
```

2. To do so, we set breakpoint at **0x791** (as found above) and using **c** and **si** command found the required instructions.

<u>Last instruction of boot loader:</u>

*call *0x10018*

<u>First instruction of kernel:</u>

mov %cr4, %eax

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

3. The for loop runs starting from **ph** whose value can be found out using **ELF header** file and number of iterations are given using **elf->phnum**. Thus, end of for loop, i.e., **eph** becomes: **eph = ph + elf->phnum**.

<u>Exercise 4</u>

- The first part says to understand the output of ***pointer.c*** file line by line. Here is the output upon execution of the program:

```
akshat@akshat:~/Desktop$ gcc pointer.c
akshat@akshat:~/Desktop$ ./a.out
1: a = 0x7ffe0788ec40, b = 0x55c02c4602a0, c = 0x7ffe0788ec67
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7ffe0788ec40, b = 0x7ffe0788ec44, c = 0x7ffe0788ec41
```

```
int a[4];
int *b = (int*)malloc(16);
int *c;
int i;

printf("1: a = %p, b = %p, c = %p\n", a, b, c);
```

<u>Line 1:</u> The variables **a** and **c** belong to **stack** memory address while **malloc** is used to dynamically allocate memory in the **heap** space. Hence, the address of **b** is at totally different location.

```
c = a;
for (i = 0; i < 4; i++)
a[i] = 100 + i;
c[0] = 200;
printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
    a[0], a[1], a[2], a[3]);
```

Line 2: The pointer **c** is made to point the location same as **a**. Thus, after **a[0]** is changed to **100** inside **for loop**, the **c[0]** changes it to **200** while another values **a[1…3**] remain as after the **for loop**.

```
c[1] = 300;
*(c + 2) = 301;
3[c] = 302;
printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
    a[0], a[1], a[2], a[3]);
```

Line 3: Now **c[1]** is changed to **300** which changes **a[1]** as well since **c** is still storing address of **a**. The next lines are other ways of dereferencing and which change values of **a[2]** and **a[3]** respectively.

```
c = c + 1;
*c = 400;
printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
    a[0], a[1], a[2], a[3]);
```

Line 4: **c = c+1** increases the address inside **c** by **4 bytes** due to which **c** now stores the address of **a[1]** rather than **a[0]** and hence value of **a[1]** is changed to **400.**

```
c = (int *) ((char *) c + 1);
*c = 500;
printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
    a[0], a[1], a[2], a[3]);
```

Line 5: This time, address in **c** is increased by only **1 byte** since **char is 1 byte** and thus **c** does not point to any specific value in a but an **address between a[1] and a[2]** and updating it affect both.

Note: *Numbers are stored in binary representation from left to right inside memory, i.e., left-most bit is LSB. For eg. '40' in 16-bit system will be stored as 00010100 00000000 rather than 00000000 00101000.*

a[1] = 400 and a[2] = 301

00001001 10000000 00000000 00000000 10110100 10000000 00000000 00000000

^a[1] ^ location c is pointing ^a[2]

After updating *c = 500: 4 bytes from the position of pointer c will be affected.

00001001 00101111 10000000 00000000 00000000 10000000 00000000 00000000

^a[1] ^ location c is pointing ^a[2]

Hence, new value of a[1] = 128144 and a[2] = 256

```
b = (int *) a + 1;
c = (int *) ((char *) a + 1);
printf("6: a = %p, b = %p, c = %p\n", a, b, c);
```

Line 6: It stores the (**address of a) + 4 bytes in b** and (**address of a) + 1 byte in c** (like above one) and thus the address inside **a** is same as initial while in **b** is **a+4 byte** and in **c** is **a+1 byte**.

- Now we have to solve **K-splice** problem:

What does this program print?

```c
#include <stdio.h>
int main() {
  int x[5];
  printf("%p\n", x);
  printf("%p\n", x+1);
  printf("%p\n", &x);
  printf("%p\n", &x+1);
  return 0;
}
```

Let us assume that **x** is stored at address **0x7fffdfbf7f00** (mentioned in problem itself).

Output:

0x7fffdfbf7f00
0x7fffdfbf7f04
0x7fffdfbf7f00
0x7fffdfbf7f14

We can draw following conclusions:

- Pointers and arrays are different things.
- If **x** is an integer array, then **x + 1** increments the address by **4 bytes**, i.e., **x[1]**.
- **&x** is a pointer to the array and **&x + 1** increments the address by **4*n bytes** (where **n** is size of the array).

Part 3: Loading the Kernel

The commands ***objdump –h kernel*** and ***objdump –h bootblock.o*** were executed to inspect various code sections in the kernel file and text sections of the boot loader.

```
akshat@akshat:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000070da  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801070e0  001070e0  000080e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006cb5  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   000121ce  00000000  00000000  000121cb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00003fd7  00000000  00000000  00024399  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003a8  00000000  00000000  00028370  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000eab  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c3  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002b  00000000  00000000  00030ae9  2**0
                  CONTENTS, READONLY
```

Fig. objdump output for kernel

```
akshat@akshat:~/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001d3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dd4  00007dd4  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002b  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000328  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000368  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000022c  00000000  00000000  0000093a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   0000029a  00000000  00000000  00000b66  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000220  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001020  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012db  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Fig. objdump output for bootblock.o

- **objdump -h** gives the list of all sections in the executable (ELF) file of the specified file, along with some details. It gives us the following information:
  1. Size of the section.
  2. VMA, i.e. Link Address, of the section, where code execution starts.
  3. LMA, i.e. Load Address, of the section, where section is loaded into memory.

Exercise 5

- On tracing through the first few instructions of boot loader, I concluded that the **ljmp** instruction would be the first one to break on changing the link address, i.e., where we transition to 32-bit system.

```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32
```

Fig. bootasm.S

- Now, I changed the **link** address in **Makefile** from 0x7c00 to 0x7e00 and then run the ***make clean*** and ***make qemu-nox-gdb*** commands.

```
bootblock: bootasm.S bootmain.c
	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
	$(LD) $(LDFLAGS) -N -e start -Ttext 0x7E00 -o bootblock.o bootasm.o bootmain.o
	$(OBJDUMP) -S bootblock.o > bootblock.asm
	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
	./sign.pl bootblock
```

- Now, I will compare the output of **gdb terminal** in correct link address with that of incorrect link address. Doing so, following conclusions can be drawn:
  1. All instructions before **ljmp** are same for both the cases.
  2. There is no message saying "***The target architecture is assumed to be i386***" with incorrect link address which means it does not switch to 32-bit system.
  3. Instructions after this are different. Hence, **ljmp is first instruction to break**.

```
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb)
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb)
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
```

Fig. Output with **correct** link address

```
(gdb)
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb)
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87e31
0x00007c2c in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb)
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb)
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
```

Fig. Output with **incorrect** link address

- From the command ***objdump -f kernel***, we can see the entry point, i.e., the start address of kernel is at ***0x0010000c***.

```
akshat@akshat:~/xv6-public$ objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

Fig. objdump -f kernel

Exercise 6

Following conclusions can be drawn from this experiment:

1. All the **8 words** at the point where BIOS enters the boot-loader (address **0x7c00**) are **0x00000000**.
   This is because the address location **0x00100000**, i.e., **1 MB** in the main memory, i.e., the point where the **boot-loader has to load the kernel**. Since the boot-loader's first instruction is yet to be executed after the address **0x7c00**, the **loading of kernel into the RAM has not started** and hence, all the words are having the value **0**.

2. At the second breakpoint i.e., the point where **boot-loader gives the control to kernel**, i.e., **0x7d91** the 8 words now have a **non-zero** value.
   This output is expected since by this time the **boot-loader has loaded the kernel** into the memory location starting at address **1 MB**. So the words there contain meaningful information.

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
```

- Answer to the **second question** is that second breakpoint was set at the instruction corresponding to the address location **0x7d91**. This is the address of the **entry function** which is responsible for **giving the control to kernel** and is the last instruction to execute of the boot-loader (as already discussed in Exercise 3).