# Saket Kumar Singh - 190101081

## Exercise 1

- The modified part of the code along with the output is shown below

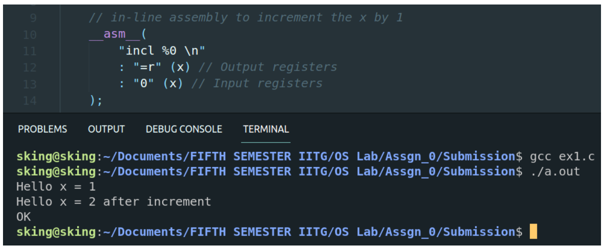


Fig 1.1: in-line assembly to increment x

- **__asm__** function takes in **three** arguments separated by ':' and are described below
  - `incl %0 \n`: **Increases** the value of register `%0` by **1** (l indicates **32 bit** register)
  - `"=r" (x)`: **Dynamically** allocate an **output** register for variable **x** (referred by **%0**)
  - `"0" (x)`: Use the register `%0` as an **input** register for variable **x**
- From the second line of output it is clear that the value of x has been incremented (refer ex1.c code file for details).

## Exercise 2 (pre – tasks)

- After cloning QEMU and installing the dependencies following output is obtained as a result of **make qemu** command followed by **ls** command on the VGA:

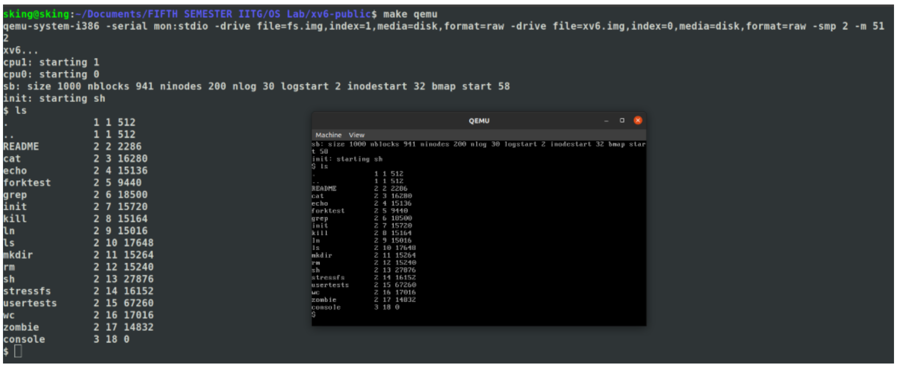


Fig 2.1: Launching QEMU

- The background window is the **ubuntu terminal** and the smaller window is the **QEMU VGA**. The output of **ls** command appears on **both** the screens which demonstrates that both the windows' **input** and **output** are **synced**.

## Exercise 2

- The screenshot of final output is attached below after which there is a detailed explanation of output and also of each step which was followed to get the output. (The labels at the left and right correspond to the serial no. used for explaining)

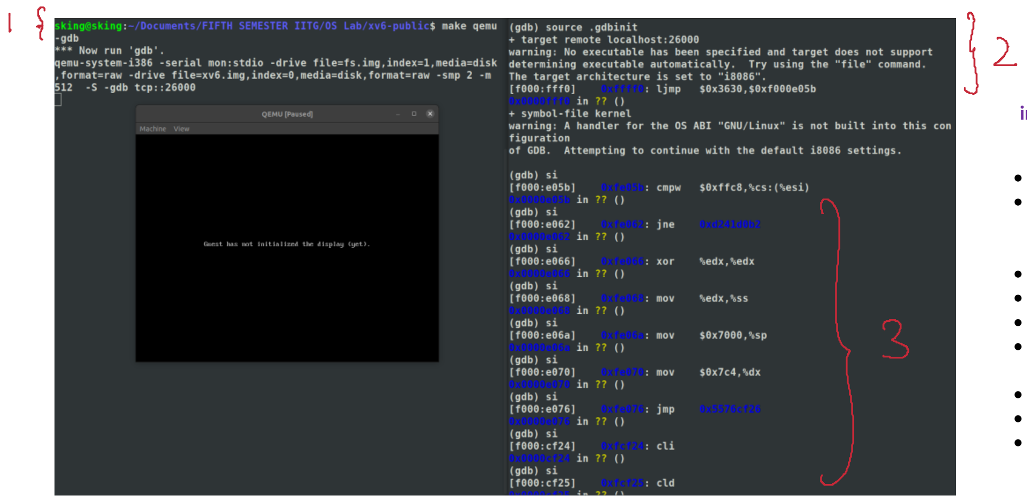


Fig 2.2: Tracing first few instructions of the ROM BIOS

- Compare the operands
- Conditionally jump if result of previous instruction is not equal

- Set %edx to 0
- Set %ss to 0
- Set %sp to 0x700
- Set %dx to 0x7c4

- Jump to the specified location.
- Clear the interrupt flag.
- Clear the direction flag.

1. Command executed (in the left terminal window): `make qemu-gdb`
   Output: **Blank** QEMU VGA with a message - "*Guest has not initialized the display yet*"
   Explanation: This command is used to **debug** and tweak into **step by step** process for the **OS** to **boot** up. As a consequence, even the VGA has not been initialized yet.
2. Command executed (in the right gdb terminal window): `source .gdbinit`
   Output: **GDB session** has started and **first** instruction is present at the location **[f000: fff0]** and is a **jump** instruction.
   Explanation: The first command to be executed is at the **end of the BIOS** section of memory (precisely speaking **16 bits** from the end). This command causes a **jump to the first** instruction of **boot-up section** of BIOS (i.e, at the location **[f000:e05b]**) from where BIOS tries to boot-up the **OS** into the system. This mechanism allows the BIOS to re-execute in case of any failure (as it towards the end of the BIOS section of memory).
3. Command executed (in the right gdb terminal window): `si`
   Output: **Assembly instructions** of boot-up process in BIOS.

Explanation: BIOS is trying to **set** up the **Interrupt Descriptor Table** in the first few commands which is used to initialize devices like the VGA display. The instructions in the screenshot **set** the stack segment register (**%ss**) and the stack pointer register (**%sp**) with the appropriate values. The in detail meaning of steps are written at the **right** side of the image attached.

## Exercise 3

- The first part describes the setting up of the breakpoint at 0x7c00 address location and tracing the source code in bootasm.S (left part of the image), disassembly in bootblock.asm (middle part of the image) and in the GDB terminal (right part of the image). Below the screenshot is a description of the tasks performed.

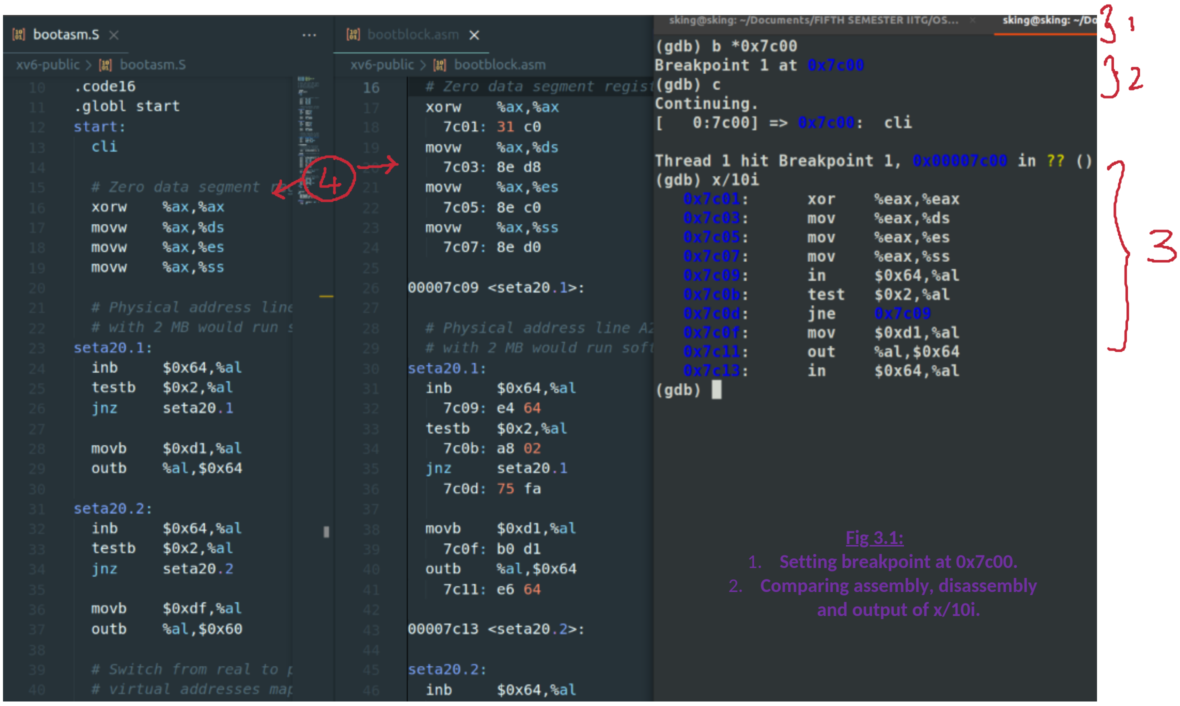


Fig 3.1:
1. Setting breakpoint at 0x7c00.
2. Comparing assembly, disassembly and output of x/10i.

1. Command: `b *0x700`
   Explanation and Output: This is used to set **break-point** at the instruction to be executed at memory location with address **0x7c00**. The output tells that the breakpoint is **successfully** set.

2. Command: c
   Explanation and Output: It **executes** all the instructions up to the next break-point (here at address 0x7c00). After the execution is done, it prints the next instruction to be executed onto the terminal (here ***cli*** instruction).

3. Command: x/10i
   Output: **Ten lines** are printed and each line constitutes of a **disassembly** instruction.
   Explanation: This command **prints** the **next 10 disassembly instructions** to be executed. In general 10 can be replaced with any positive integer.

4. The **disassembly** code and **assembly** source code are **compared** with the disassembly **output** of Step 3. in GDB window. It is observed that all the instructions starting from ***cli*** all the way up to ***inb $0x64 %al*** are **exactly the same** except a few changes in the syntax (**majorly** the suffix of **b/w/l** standing for byte/word/long word is **not** present in **GDB** output).

- The next section of the exercise asks to locate various **parts of source code** and their corresponding **assembly** code. Following are the **screenshots** with labels for each

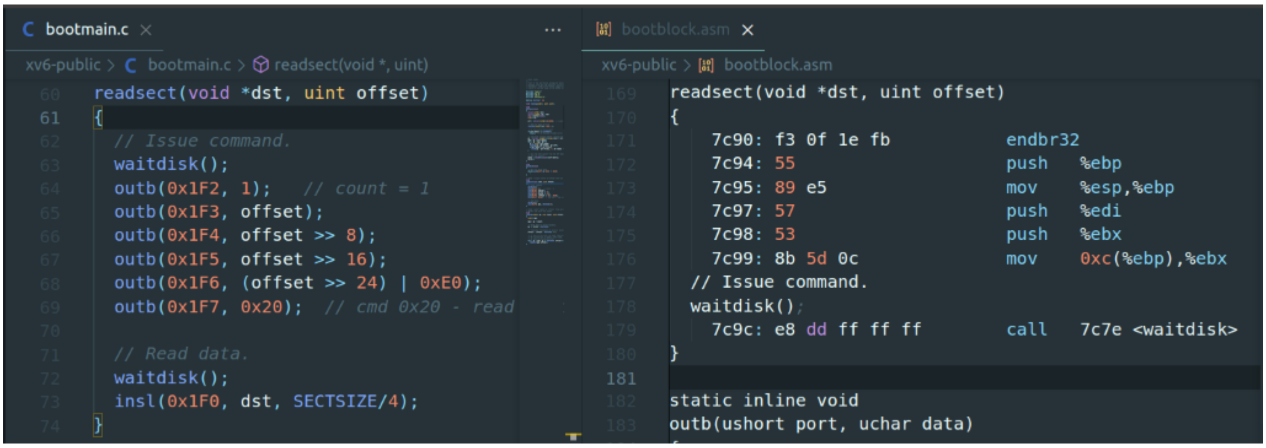


Fig 3.2:
1. Left: Source code for readsect in bootmain.c
2. Right: Corresponding assembly code in bootblock.asm

```
24
25    elf = (struct elfhdr*)0x10000;  // scra
26
27    // Read 1st page off disk
28    readseg((uchar*)elf, 4096, 0);
29
30    // Is this an ELF executable?
31    if(elf->magic != ELF_MAGIC)
32      return;  // let bootasm.S handle err
33
34    // Load each program segment (ignores
35    ph = (struct proghdr*)((uchar*)elf + el
36    eph = ph + elf->phnum;
37    for(; ph < eph; ph++){
38      pa = (uchar*)ph->paddr;
39      readseg(pa, ph->filesz, ph->off);
40      if(ph->memsz > ph->filesz)
41        stosb(pa + ph->filesz, 0, ph->mems
42    }
43
44    // Call the entry point from the ELF he
45    // Does not return!
46    entry = (void(*)(void))(elf->entry);
47    entry();
48  }
49
50  void
51  waitdisk(void)
52  {
53    // Wait for disk ready.
54    while((inb(0x1F7) & 0xC0) != 0x40)
```

```
315    for(; ph < eph; ph++){
316      7d8d: 39 f3                cmp    %esi,%ebx
317      7d8f: 72 15                jb     7da6 <bootmain+0x
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+0x
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+0x
345        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
```

**Fig 3.3: For loop reading remaining sectors**

1. **Left: Source code of for loop**
2. **Right: Assembly code of for loop**

The code on line **46-47** (in **left** side source file of **Fig 2.5**) whose assembly is on line **318-319** (**right** side assembly file of **Fig 2.5**) of above image would be executed **once the for loop is over**. This line is the **final** line in execution of boot loader and after this the **control** is given to the **OS**. The **breakpoint** is set at **0x7d91** (line **319**) as shown in **Fig 2.6**.

```
(gdb) b *7d91
Invalid number "7d91".
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) n
Cannot find bounds of current function
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x109000,%eax
0x00100015 in ?? ()
(gdb) si
```

**Fig 3.4: Breakpoint at 0x7d91 (end of for loop) and stepping through boot loader**

- Finally the 3 questions are answered below:
  1. Following image shows the last lines executed before switching to the protected mode. The **last line** to be executed is **line 51**. Before that the boot loader **deactivates the line A20** (to access **2MB** of RAM instead of default 1MB) and also initializes the **descriptor table** which would be helpful in accessing the **virtual memory** in the **protected** mode. After it's done then the line 51 causes the control to **jump** to the protected mode.

```
  # 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 3.5: Assembly code to switch from 16 bit Real mode to 32 bit Protected mode.**

  2. From the discussion in Fig 2.6 and from the source code at Fig 2.5, it is clear that the **last instruction** of boot loader would be to call the entry function (on **line 47 in Fig 2.5** bootmain.c). Thus a **breakpoint** is set for the corresponding assembly instruction and the command **si** is used to **trace** the next instruction (first instruction of the kernel) and following conclusions are made based on the screenshot below:
     - Last instruction of boot loader: `call *0x10018`
     - First instruction of kernel: `mov %cr4, %eax`

```
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d91:      call   *0x10018

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

**Fig 3.6: GDB terminal showing last instruction of boot loader and the first instruction of kernel**

  3. **Fig 2.9** tells that the boot loader runs a **for** loop **starting at *ph*** and **ending at *eph – 1***. The value of **ph** is obtained **from** the **ELF header** and the number of **iterations** are given by ***elf->phnum***, thus ***eph = ph + elf->phnum***.

```
// 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);
}
```

Fig 3.7: For loop in boot loader which loads all the sectors

## Exercise 4

- The **first part** requires to understand line by line the output of the file **pointers.c**. Therefore, below is the line by line explanation of each line of output.

1. Statement: `printf("1: a = %p, b = %p, c = %p\n", a, b, c);`
   Reason for output of line 1: The variables **a** and **c** belong to **stack** memory address and thus store **close addresses**. **Malloc** is used to **dynamically** allocate address in the **heap** space and so the value stored in **b** is having a **totally different** address.

```
4   void f(void)
5   {
6       int a[4];
7       int *b = malloc(16);
8       int *c;
9       int i;
10
11      printf("1: a = %p, b = %p, c = %p\n", a, b, c);
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/Assgn_0$ gcc pointers.c
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/Assgn_0$ ./a.out
1: a = 0x7ffc80316d40, b = 0x559f095252a0, c = 0x7ffc80316d67
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 = 0x7ffc80316d40, b = 0x7ffc80316d44, c = 0x7ffc80316d41
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/Assgn_0$
```

Fig 4.1: Output and source code for line 1

2. The **statement** is attached as a **screenshot**.
   Reason for output of line 2: After **line 13**, the pointer **c** stores the **address of array a**. Thus **line 15** changes the value of previously assigned **a[0]** of 100 to **200** and so in the output the value of a[0] is 200 whereas the **other values** of the array are **same** as before.

Fig 4.2: Source code for line 2

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

3. Reason for output of line 3: **Line 20** changes the value of **a[1]** to **300** (as c is still storing the address of array a). **Lines 21** and **22** are just two other ways of **dereferencing** the elements of an array and change **a[2]** and **a[3]** to **301** and **302** respectively. Thus the only **difference** between output of line 2 and 3 is the values of a[1], a[2] and a[3].

```
20      c[1] = 300;
21      *(c + 2) = 301;
22      3 [c] = 302;
23      printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
24             a[0], a[1], a[2], a[3]);
```

Fig 4.3: Source code for line 3

4. Reason for output of line 4: **Line 26** increases the address in **c by 4 bytes** (since **c** is of the type **int***). Thus c now points to address of a[1] and **Line 27** changes **a[1] to 400**. Hence the output in line 4 is same as that of line 2 except the value of a[1] is now 400.

Fig 4.4: Source code for line 4

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

5. Reason for output of line 5: **Line 31** increases the address in **c by 1 byte** (since c is changed to **char*** before incrementing). Now c doesn't point to any specific element in a and thus the value of **500** is assigned **partially between a[1] and a[2]** and it changes both the values in a **corrupted** way.

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

Fig 4.5: Source code for line 5

6. Reason for output of line 6: **Line 36** stores the address of **a[1] in b** and **line 37** stores the address of **a incremented** by **1 byte** (similar to line 31). Thus the value of a is same as output of line 1 and value of **b is a + 4 bytes** and value of **c is a + 1 byte**.

Fig 4.6: Source code for line 6

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

- The next part asks us to solve the **K-splice** pointer challenge from which following **conclusions** are drawn:
  - Pointers and arrays are two **different** things, although **sometimes** arrays **decay** into pointers.
  - If x is an array of int, then the type of &x is a pointer to an array and **&x + 1** increments the address by **4*n bytes** (where n is size of array x).
  - On the other hand **x + 1** increments the address by **4 bytes** (since in this case x decays into &x[0]).

**Result: printf("%p\n", &x) prints 0x7fffdfbf7f00.**

Aside: what is the type of &x[0]? Well, x[0] is an int, so &x[0] is "pointer to int". That feels right.

```
printf("%p\n", &x+1);
```

What will this print? [Guess]
Ok, now for the coup de grace. x may be an array, but &x is definitely a pointer. So what's &x+1?

First, another aside: what is the type of &x? Well... &x is a pointer to an array of 5 ints. How would you declare something like that?

Let's fire up cdecl and find out:

```
cdecl> declare y as array 5 of int;
int y[5]
cdecl> declare y as pointer to array 5 of int;
int (*y)[5]
```

Confusing syntax, but it works:
int (*y)[5] = &x; compiles without error and works the way you'd expect.

But back to the question at hand. Pointer arithmetic tells us that &x+1 is going to be the address of x + sizeof(x). What's sizeof(x)? Well, it's an array of 5 ints. On this system, each int is 4 bytes, so it should be 20 bytes, or 0x14.

**Result &x+1 prints 0x7fffdfbf7f14.**

Fig 4.7: K-Splice pointer challenge

## Exercise 5 (pre – tasks):

- Different **program sections** (especially .text, .data and .rodata) were analysed in the **ELF** binary file (**kernel.ld**). Screenshot of some of the sections is attached along with a brief explanation of observations:

```
.text : AT(0x100000) {
    *(.text .stub .text.* .gnu.linkonce.t.*)
}

PROVIDE(etext = .); /* Define the 'etext' symbol to this value */

.rodata : {
    *(.rodata .rodata.* .gnu.linkonce.r.*)
}
```

Fig 5.1: Important program sections of kernel.ld

  - **.text**: This section consists of program's **executable** instructions.
  - **.data**: This section consists of program's **initialized data** like global variables.
  - **.rodata**: This section consists of program's **read-only** data.

- The commands **objdump –h kernel** and **objdump –h bootblock.o** were executed to the inspect various code sections in the **kernel** file and **text section** of the boot loader. The screenshot of the output is attached below.

```
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/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    00000ecf  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295e7  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fe05  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030b0d  2**0
                  CONTENTS, READONLY
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/xv6-public$
```

```
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/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      0000002a  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    00000244  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001044  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012ff  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/xv6-public$
```

Fig 5.2: objdump output for kernel (left) and bootblock.o (right)

## Exercise 5:

- On tracing through the first few instructions of boot loader, I concluded that the assembly instruction on **line 51** (**ljmp** instruction) would be the **first** one to **break** on **changing** the **link address** in the screenshot as shown below.

```
38
39    # Switch from real to protected mode.  Use a bootstrap GDT that makes
40    # virtual addresses map directly to physical addresses so that the
41    # effective memory map doesn't change during the transition.
42    lgdt    gdtdesc
43    movl    %cr0, %eax
44    orl     $CR0_PE, %eax
45    movl    %eax, %cr0
46
47  //PAGEBREAK!
48    # Complete the transition to 32-bit protected mode by using a long jmp
49    # to reload %cs and %eip.  The segment descriptors are set up with no
50    # translation, so that the mapping is still the identity mapping.
51    ljmp    $(SEG_KCODE<<3), $start32
```

Fig 5.3: Instruction first to break with wrong link address

- In the **line 107** the **correct** link address: **0x7c00** has been replaced with an **incorrect** link address: **0x7e00** as shown below in the screenshot. After that the commands **make clean** and **make** are run on the terminal.

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

Fig 5.4: Changing the link address in the makefile

- Fig 5.5 **compares** the **output** of GDB terminal when **correct** link address was used with the case when **wrong** link address was used. From the screenshot, following conclusions can be drawn:
    - The instructions **before** the **ljmp** instruction are **same** in the bootloader for both the cases.
    - The instructions **including** and **after** the **ljmp** instruction are **different**.
    - There is **no message** saying: "*The target architecture is set to be i386*" **in** the case with **wrong** link address (which happens after the ljmp instruction in the correct link address situation).

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

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

Fig 5.5: Differences between the correct link address (left) and wrong link address (right) cases.

After the experiment, the link address was corrected and the commands make clean and make were executed as instructed.

- Next, it is asked to **repeat** the commands of **objdump** in the terminal for kernel and for boot-loader. With reference to Fig 5.2 (new screenshot is not used here to avoid repetition), we can see that the **boot-loader** has the **VMA** (link address) and **LMA** (load address) exactly the **same** for various sections. However, in case of **kernel** the **link address** (expected execution point) is at a **higher memory address** (**shifted** by **0x8000000**) for each of the sections which are to be loaded. For ex: in case of .text program section, the LMA is 0x00100000 (i.e, **1MB**) whereas the VMA is 0x8010000.

- The **e_entry** ELF header field is stores in **struct elfhdr.entry** as shown in the **left** side of the screenshot below (**line 12**). Also, the **entry** point address (**0x0010000c**) of the kernel as recorded by execution of the command **objdump –f kernel** is shown below.

```
8    uchar elf[12];
9    ushort type;
10   ushort machine;
11   uint version;
12   uint entry;
13   uint phoff;
14   uint shoff;
```

```
sking@sking:~/Documents/FIFTH SEMESTER IITG/OS Lab/xv6-public$ objdump -
f kernel

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

Fig 5.5: e_entry ELF header (left) and entry address of kernel as given by objdump command (right)

## Exercise 6:

- After the experiment, following observations with the reasons were understood:
  - **Observation**: All the **8 words** (at 0x00100000) at the point where **BIOS enters the boot-loader** (at address 0x7c00) are **0x00000000**.
    **Reason**: The given address location (i.e 0x00100000) corresponds to an address **1 MB** in the main memory. This is the point **where** the **boot-loader** has to **load the kerne**l. As the boot-loader's **first** instruction is yet to be executed **after** the address **0x7c00** (which means that **loading** of kernel into the RAM has **not** started), **hence all** the words are having the value **0** with no meaningful data in them yet.
  - **Observation**: At the **second breakpoint** (i.e, at the point where **boot-loader** gives the control to **kernel**), the 8 words now have a **non-zero value**.
    **Reason**: This output is expected since by this time the boot-loader has **loaded** the **kernel** into the memory location starting at address 1 MB in the **RAM** and so the **words at 1 MB** location **contain meaningful** (non-zero) 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      0x000000
00
0x100010:       0x00000000      0x00000000      0x00000000      0x000000
00
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e020
0f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200f
d8
(gdb) 
```

**Fig 5.6: Output of GDB window for x/8x 0x00100000 command at breakpoints 0x7c00 and 0x7d91.**

- 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** (as described in the Exercise 3) which is **responsible** for giving the **control** to **kernel** and is the **last** instruction to execute of the **boot-loader**.