CS 344 : Operating Systems Lab **Lab Assignment 0**

Name: Vignesh Ravichandra Rao **Roll Number**: 190101109

Exercise 1)

The modified code is as follows:

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

    asm("incl %0":"=r"(x):"0"(x));

    printf("Hello x = %d after increment\n", x);

    if(x == 2){
        printf("OK\n");
    }
    else{
        printf("ERROR\n");
    }
}
```

Output:

```
vigneshrrao:Desktop$ gcc vig_ex1.c
vigneshrrao:Desktop$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
```

Explanation:

The inline assembly follows the syntax :

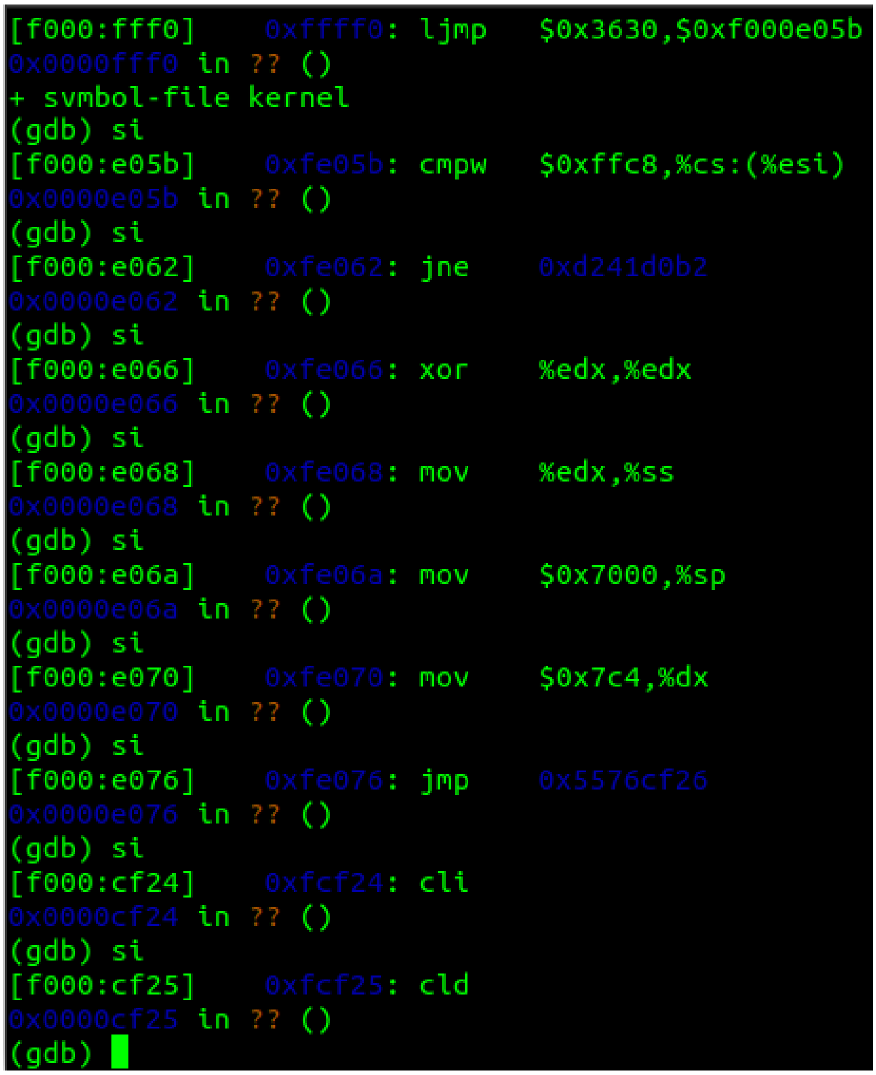
`asm ("statements":output_registers:input_registers);`

- **Output Registers**: We specify the output register as "**`=r`**"**`(x)`**. The "**`r`**"indicates dynamic register allotment. The "**`=`**" writes the final value of the register into the variable **`x`** mentioned in parentheses.
- **Input Registers**: We specify the input register as `"0"`**`(x)`**. The "`0`" tells the compiler to use the register allocated as output previously. The value to be inserted in the register is that present in the value **`x`** mentioned in parentheses.
- **Statement**: The assembly instruction we use is "**`incl %0`**". **`incl`** increments the value of the 32 bit (long type denoted by the **`l`** in **`incl`**) register that follows it by 1. `%0` indicates the register alloted as the output register

---

Exercise 2)

Below is the screenshot of the first few instructions that are run as a part of ROM BIOS.



```
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) si
[f000:e070]    0xfe070: mov    $0x7c4,%dx
0x0000e070 in ?? ()
(gdb) si
[f000:e076]    0xfe076: jmp    0x5576cf26
0x0000e076 in ?? ()
(gdb) si
[f000:cf24]    0xfcf24: cli
0x0000cf24 in ?? ()
(gdb) si
[f000:cf25]    0xfcf25: cld
0x0000cf25 in ?? ()
(gdb)
```

Explanation:

1. `[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b`
   - The current location `[f000:fff0] = ffff0` is 16 bytes from the end of BIOS ROM region of the memory and too less to fit all required instructions.
   - Hence, the first step is to long jump to a previous location i.e. `fe05b`
2. `[f000:e05b] 0xfe05b: cmpw $0xffc8,%cs:(%esi)`
   - Compares the constant hexadecimal value `ffc8` with the contents of location pointed by segment = value in cs register = `f000` and offset = value in esi register
3. `[f000:e062] 0xfe062: jne 0xd241d0b2`
   - If the previous comparison results in an inequality (i.e. zero flag is not set) then jump to location `0xd241d0b2`
4. `[f000:e066] 0xfe066: xor %edx,%edx`
   - Since there was no jump to `0xd241d0b2`, it means the comparison set the zero flag. The current instruction effectively clears out the contents of the edx registers by using xor operation. Hence, contents of `edx = 0`.
5. `[f000:e068] 0xfe068: mov %edx,%ss`
   - move contents of `edx` register to stack segment register (`ss`). Hence, content of `ss` register is now 0.
6. `[f000:e06a] 0xfe06a: mov $0x7000,%sp`
   - Initialised the stack pointer which points to the top of stack to hexadecimal value 7000
7. `[f000:e070] 0xfe070: mov $0x7c4,%dx`
   - Sets the value of dx register to hexadecimal value `7c4`

8. `[f000:e076] 0xfe076: jmp 0x5576cf26`
    - jumps to a different location
9. `[f000:cf24] 0xfcf24: cli`
    - Disables interrupts by clear interrupt instruction. This is done because the CPU should not be interrupted while boot loading which is a critical process.
10. `[f000:cf25] 0xfcf25: cld`
    - Clear direction flag which is required for proper execution of subsequent instructions.

Exercise 3)

Code tracing: Comparison of bootasm.S, bootblock.asm and GDB disassembly

Figure 3.1: bootasm.S

```
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60

  # 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
```

Figure 3.2: GDB disassembly

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/20i $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
   0x7c1d:      lgdtl  (%esi)
   0x7c20:      js     0x7c9e
   0x7c22:      mov    %cr0,%eax
   0x7c25:      or     $0x1,%ax
   0x7c29:      mov    %eax,%cr0
```

Figure 3.3: bootblock.asm

```
start:
  cli                         # BIOS enabled interrupts; disable
    7c00:       fa                      cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
    7c01:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    7c03:       8e d8                   mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    7c05:       8e c0                   mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    7c07:       8e d0                   mov    %eax,%ss

00007c09 <seta20.1>:

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
    7c09:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c0b:       a8 02                   test   $0x2,%al
  jnz     seta20.1
    7c0d:       75 fa                   jne    7c09 <seta20.1>

  movb    $0xd1,%al               # 0xd1 -> port 0x64
    7c0f:       b0 d1                   mov    $0xd1,%al
  outb    %al,$0x64
    7c11:       e6 64                   out    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
    7c13:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c15:       a8 02                   test   $0x2,%al
  jnz     seta20.2
    7c17:       75 fa                   jne    7c13 <seta20.2>

  movb    $0xdf,%al               # 0xdf -> port 0x60
    7c19:       b0 df                   mov    $0xdf,%al
  outb    %al,$0x60
    7c1b:       e6 60                   out    %al,$0x60

  # 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
    7c1d:       0f 01 16                lgdtl  (%esi)
    7c20:       78 7c                   js     7c9e <readsect+0xe>
  movl    %cr0, %eax
    7c22:       0f 20 c0                mov    %cr0,%eax
  orl     $CR0_PE, %eax
    7c25:       66 83 c8 01             or     $0x1,%ax
  movl    %eax, %cr0
    7c29:       0f 22 c0                mov    %eax,%cr0
```

This section of the boot sector is performing the following three tasks:

- Initialising data, extra and stack segment to zero and disabling interrupts.
- Setting A20 address line by probing ports 0x64,0x60 of the keyboard controller.
- Enable 32-bit protected mode by setting the global description table (GDT).

Main difference between bootasm.S and the disassembly on GDB/ bootblock.asm is that:

- `lgdt gdtdesc` instruction in bootasm.S has been expanded to two constituent instructions: `lgdtl (%esi)` and `js 0x7c9e`

Coding tracing: bootmain() and readsect() in bootmain.c

Figure 3.4: bootblock.asm

```
51  ljmp    $(SEG_KCODE<<3), $start32
52
53 .code32  # Tell assembler to generate 32-bit code now.
54 start32:
55  # Set up the protected-mode data segment registers
56  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
57  movw    %ax, %ds                # -> DS: Data Segment
58  movw    %ax, %es                # -> ES: Extra Segment
59  movw    %ax, %ss                # -> SS: Stack Segment
60  movw    $0, %ax                 # Zero segments not ready for use
61  movw    %ax, %fs                # -> FS
62  movw    %ax, %gs                # -> GS
63
64  # Set up the stack pointer and call into C.
65  movl    $start, %esp
66  call    bootmain
67
68  # If bootmain returns (it shouldn't), trigger a Bochs
69  # breakpoint if running under Bochs, then loop.
70  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
71  movw    %ax, %dx
72  outw    %ax, %dx
73  movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
74  outw    %ax, %dx
75 spin:
76  jmp     spin
```

Figure 3.5: GDB call to bootmain

```
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is assumed to be i8086
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
(gdb) si
=> 0x7c39:      mov    %eax,%ss
0x00007c39 in ?? ()
(gdb) si
=> 0x7c3b:      mov    $0x0,%ax
0x00007c3b in ?? ()
(gdb) si
=> 0x7c3f:      mov    %eax,%fs
0x00007c3f in ?? ()
(gdb) si
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb) si
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb) si
=> 0x7c48:      call   0x7d49
0x00007c48 in ?? ()
(gdb) si
=> 0x7d49:      endbr32
0x00007d49 in ?? ()
```

In Figure 3.4 line 66 contains the call to bootmain() This corresponds to the instruction at location 0x7c48 in Figure 3.5 (highlighted)

Figure 3.5: Tracing into readsect() From bootmain()

bootmain()

```
(gdb) x/12i $eip
=> 0x7d49:      endbr32
   0x7d4d:      push   %ebp
   0x7d4e:      mov    %esp,%ebp
   0x7d50:      push   %edi
   0x7d51:      push   %esi
   0x7d52:      push   %ebx
   0x7d53:      sub    $0x10,%esp
   0x7d56:      push   $0x0
   0x7d58:      push   $0x1000
   0x7d5d:      push   $0x10000
   0x7d62:      call   0x7cfc
   0x7d67:      add    $0x10,%esp
```

readseg()

```
(gdb) x/25i $eip
=> 0x7cfc:      endbr32
   0x7d00:      push   %ebp
   0x7d01:      mov    %esp,%ebp
   0x7d03:      push   %edi
   0x7d04:      push   %esi
   0x7d05:      push   %ebx
   0x7d06:      sub    $0xc,%esp
   0x7d09:      mov    0x8(%ebp),%ebx
   0x7d0c:      mov    0x10(%ebp),%esi
   0x7d0f:      mov    %ebx,%edi
   0x7d11:      add    0xc(%ebp),%edi
   0x7d14:      mov    %esi,%eax
   0x7d16:      and    $0x1ff,%eax
   0x7d1b:      sub    %eax,%ebx
   0x7d1d:      shr    $0x9,%esi
   0x7d20:      add    $0x1,%esi
   0x7d23:      cmp    %ebx,%edi
   0x7d25:      jbe    0x7d41
   0x7d27:      sub    $0x8,%esp
   0x7d2a:      push   %esi
   0x7d2b:      push   %ebx
   0x7d2c:      call   0x7c90
   0x7d31:      add    $0x200,%ebx
```

Call to readsect()

Figure 3.5 shows the process of tracing through bootmain() into readseg() and finally into readsect().

Figure 3.6: readsect() in bootmain.c

```
59 void
60 readsect(void *dst, uint offset)
61 {
62   // Issue command.
63   waitdisk();
64   outb(0x1F2, 1);   // count = 1
65   outb(0x1F3, offset);
66   outb(0x1F4, offset >> 8);
67   outb(0x1F5, offset >> 16);
68   outb(0x1F6, (offset >> 24) | 0xE0);
69   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
70
71   // Read data.
72   waitdisk();
73   insl(0x1F0, dst, SECTSIZE/4);
74 }
```

Assembly level instructions corresponding to each instruction in readsect():

Note: Line numbers are with respect to Figure 3.6

Line 63:

```
(gdb) si
=> 0x7c9c:      call   0x7c7e
```

Line 64:

```
(gdb) x/3i $eip
=> 0x7ca1:      mov    $0x1,%eax
   0x7ca6:      mov    $0x1f2,%edx
   0x7cab:      out    %al,(%dx)
```

Line 65:

```
=> 0x7cac:      mov    $0x1f3,%edx
   0x7cb1:      mov    %ebx,%eax
   0x7cb3:      out    %al,(%dx)
```

Line 67:

```
=> 0x7cb4:      mov    %ebx,%eax
   0x7cb6:      shr    $0x8,%eax
   0x7cb9:      mov    $0x1f4,%edx
   0x7cbe:      out    %al,(%dx)
```

Line 67:

```
=> 0x7cbf:      mov    %ebx,%eax
   0x7cc1:      shr    $0x10,%eax
   0x7cc4:      mov    $0x1f5,%edx
   0x7cc9:      out    %al,(%dx)
```

Line 68:

```
=> 0x7cca:      mov    %ebx,%eax
   0x7ccc:      shr    $0x18,%eax
   0x7ccf:      or     $0xffffffe0,%eax
   0x7cd2:      mov    $0x1f6,%edx
   0x7cd7:      out    %al,(%dx)
```

Line 69:

```
=> 0x7cd8:      mov    $0x20,%eax
   0x7cdd:      mov    $0x1f7,%edx
   0x7ce2:      out    %al,(%dx)
```

Line 72:

```
=> 0x7ce3:        call    0x7c7e
```

Line 73:

```
=> 0x7ce8:        mov     0x8(%ebp),%edi
   0x7ceb:        mov     $0x80,%ecx
   0x7cf0:        mov     $0x1f0,%edx
   0x7cf5:        cld
   0x7cf6:        rep insl (%dx),%es:(%edi)
```

Details about for loop that reads remaining sectors of kernel into memory

Figure 3.7: bootmain shown in bootblock.asm

```
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 }
349
350 static inline void
351 stosb(void *addr, int data, int cnt)
352 {
353   asm volatile("cld; rep stosb" :
354     7dc9:       b8 00 00 00 00          mov    $0x0,%eax
355     7dce:       fc                      cld
356     7dcf:       f3 aa                   rep stos %al,%es:(%edi)
357                 "=D" (addr), "=c" (cnt) :
358                 "0" (addr), "1" (cnt), "a" (data) :
359                 "memory", "cc");
360 }
361     7dd1:       eb cc                   jmp    7d9f <bootmain+0x56>
```

Figure 3.7 shows the for loop that reads remaining sectors of the kernel into the memory. The for loop extends from memory location **0x7d8d** to **0x7dd1**. Upon termination of the loop, the instruction at **0x7d91– call *0x10018** is run. This is the last instruction executed by the bootloader after which control is passed onto the kernel. Here the pointer to location ***0x10018** is the pointer to the entry field of the ELF header.

Question 1:

The transition from 16 bit real to 32 bit protected mode is done by the instructions shown in Figure 3.8 of bootblock.asm

The final instruction that completes the switch to 32 bit mode is instruction at **0x7c2c – ljmp $0xb866, $0x87c31**. As seen in Figure 3.9, after this instruction is executed the target architecture is assumed to be i386 which is 32 bit.

The first instruction executed in 32 bit mode is at location **0x7c31** (in Figure 3.9)

Figure 3.8: Switch from 16 to 32 bit mode

```
61  lgdt    gdtdesc
62    7c1d:       0f 01 16                lgdtl  (%esi)
63    7c20:       78 7c                   js     7c9e <readsect+0xe>
64  movl    %cr0, %eax
65    7c22:       0f 20 c0                mov    %cr0,%eax
66  orl     $CR0_PE, %eax
67    7c25:       66 83 c8 01             or     $0x1,%ax
68  movl    %eax, %cr0
69    7c29:       0f 22 c0                mov    %eax,%cr0
70
71 //PAGEBREAK!
72  # Complete the transition to 32-bit protected mode by using a long jmp
73  # to reload %cs and %eip.  The segment descriptors are set up with no
74  # translation, so that the mapping is still the identity mapping.
75  ljmp    $(SEG_KCODE<<3), $start32
76    7c2c:       ea                      .byte 0xea
77    7c2d:       31 7c 08 00             xor    %edi,0x0(%eax,%ecx,1)
```

Figure 3.9: GDB

```
(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 assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
```

Question 2:

Figure 3.10: GDB tracing into kernel

```
(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
```

We have already seen that the last instruction of the bootloader is at location **`0x7d91`**, in the description of Figure 3.7. Now, setting a breakpoint there and continuing on GDB allows us to find the first instruction executed by the kernel.

From Figure 3.10, the first instruction of the kernel is at location **`0x10000c`** and is a **`mov`** instruction – **`mov %cr4, %eax`**.

Question 3:

Figure 3.11: ELF header in bootmain

```
25   elf = (struct elfhdr*)0x10000;  // scratch space
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 error
33
34   // Load each program segment (ignores ph flags).
35   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
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->memsz - ph->filesz);
42   }
```

Figure 3.12: ELF header printed by GDB

```
(gdb) p *((struct elfhdr *) 0x10000)
$16 = {
  magic = 1179403647,
  elf = "\001\001\001\000\000\000\000\000\000\000\000\000",
  type = 2,
  machine = 3,
  version = 1,
  entry = 1048588,
  phoff = 52,
  shoff = 212404,
  flags = 0,
  ehsize = 52,
  phentsize = 32,
  phnum = 3,
  shentsize = 40,
  shnum = 16,
  shstrndx = 15
}
```

The bootloader decides the number of sectors to read in order to fetch the entire kernel from disk by using the ELF Header.

Consider, Figure 3.11: In lines 25 and 28, the bootloader first fetches 8 sectors (4096 bytes) from the hard disk, the initial part of which is the ELF Header of the kernel image. Printing the ELF header in GDB (Figure 3.12) shows us that it contains the **`phoff`** attribute which is the offset from where program headers are fetched and **`phnum`** which is the number of program headers to fetch.

These two attributes are used to read all of the kernel from the harddisk as follows:

- The **`ph`** pointer points to the first program header (Line 35 in Figure 3.11)
- The **`eph`** pointer points to the next of the last program header to be read.

Hence, in the for loop in lines 37 through 42 (Figure 3.11), the looping condition, **`ph < eph`** ensure that the bootloader reads all sectors containing the kernel.

Figure 3.13: ELF Header Diagram

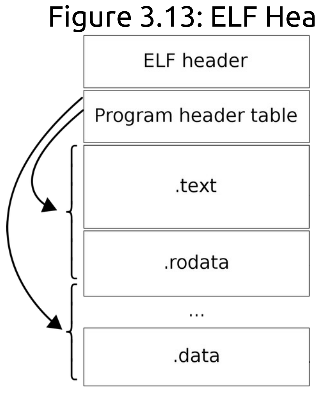


Figure 3.13 is a pictorial representation of the ELF header. It shows how the program headers referenced by ph pointer in bootmain are used to read corresponding .text and .rodata segments of the kernel.

Exercise 4)

Figure 4.1: objdump -h kernel

```
vigneshrrao: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    00000eb5  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295cd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fdeb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030af3  2**0
                  CONTENTS, READONLY
```

Figure 4.2: objdump -h bootblock.o

```
vigneshrrao: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    00000229  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001029  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012e4  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Figures 4.1 and 4.2 show various program sections of the kernel and bootblock.o binaries. The output contains the following columns:

- Name: Name of the program section. Eg: .text contains program instructions, .data contains initialized global variables, etc.
- Size: Size of the program section in bytes.
- VMA: Link address of the program section. This is the address at which the program expects to be executed.
- LMA: Load address of program section. This is the address into which the program section is actually loaded.
- File off: Offset of the section from beginning of file in disk.
- Algn: Alignment to accommodate various data types.

Exercise 5)

Figure 5.1: GDB disassembly after modifying link address

```
[   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,$0x87031
0x00007c2c in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
```

Figure 5.2: GDB disassembly with correct link address

```
[   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 assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
```

- The Makefile was edited to make the link address of .text section **`0x7000`** whereas the correct link address is **`0x7c00`**.
- As seen in Figures 5.1 and 5.2, the bootloader instruction that breaks is at location **`0x7c2c – ljmp $0xb866 , $0x87031`**. Since the long jump is to an incorrect location, the transition from 16 bit mode to 32 bit mode is not successful once the link address is modified. All instructions thereafter are erroneous as well.
- This happens because the hardcoded BIOS loads the bootloader at location **`0x7c00`**. The linker however calculates the second parameter (**`$start32`**) of ljmp instruction based on the incorrect link address. This causes the code to break.

Figure 5.3: ljmp instruction parameters. Since link address is incorrect, the $start32 absolution location is incorrectly calculated by the linker

```
  ljmp    $(SEG_KCODE<<3), $start32
    702c:       ea                      .byte 0xea
    702d:       31 70 08                xor    %esi,0x8(%eax)
        ...

00007031 <start32>:
```

Running the **`objdump -f kernel`** instruction indicates that the entry point of the kernel is at location **`0x10000c`**. This is consistent with the observations made in Exercise 3 Question 2.

Figure 5.4: objdump -f kernel

```
vigneshrrao:xv6-public$ objdump -f kernel

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

Exercise 6)

Basic hypothesis: When BIOS just enters the bootloader no useful information is present at **`0x100000`**. However, from Exercise 3 Question 3 we know that the boot loader stores the kernel (ELF Header) starting from address **`0x100000`**. Hence, when the bootloader enter the kernel, the contents at **`0x100000`** contains the kernel image.

We can confirm this using GDB as shown in Figure 6.1 (next page). We set two breakpoints- one at the entry of bootloader and another at the entry of kernel. At both breakpoints we view 8 words at location **`0x100000`**. From the GDB tracing we can conclude:

1. When the BIOS enters bootloader, the contents of 8 words at **`0x100000`** is all zero

2. Once the bootloader is executed and enters kernel, the contents of 8 words at **`0x100000`** contain different information.
3. This is because the bootloader stores the ELF header at **`0x100000`** as seen in Figure 3.11

Figure 6.1: Using GDB to compare the contents at 0x100000 before and after bootloader execution

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
```