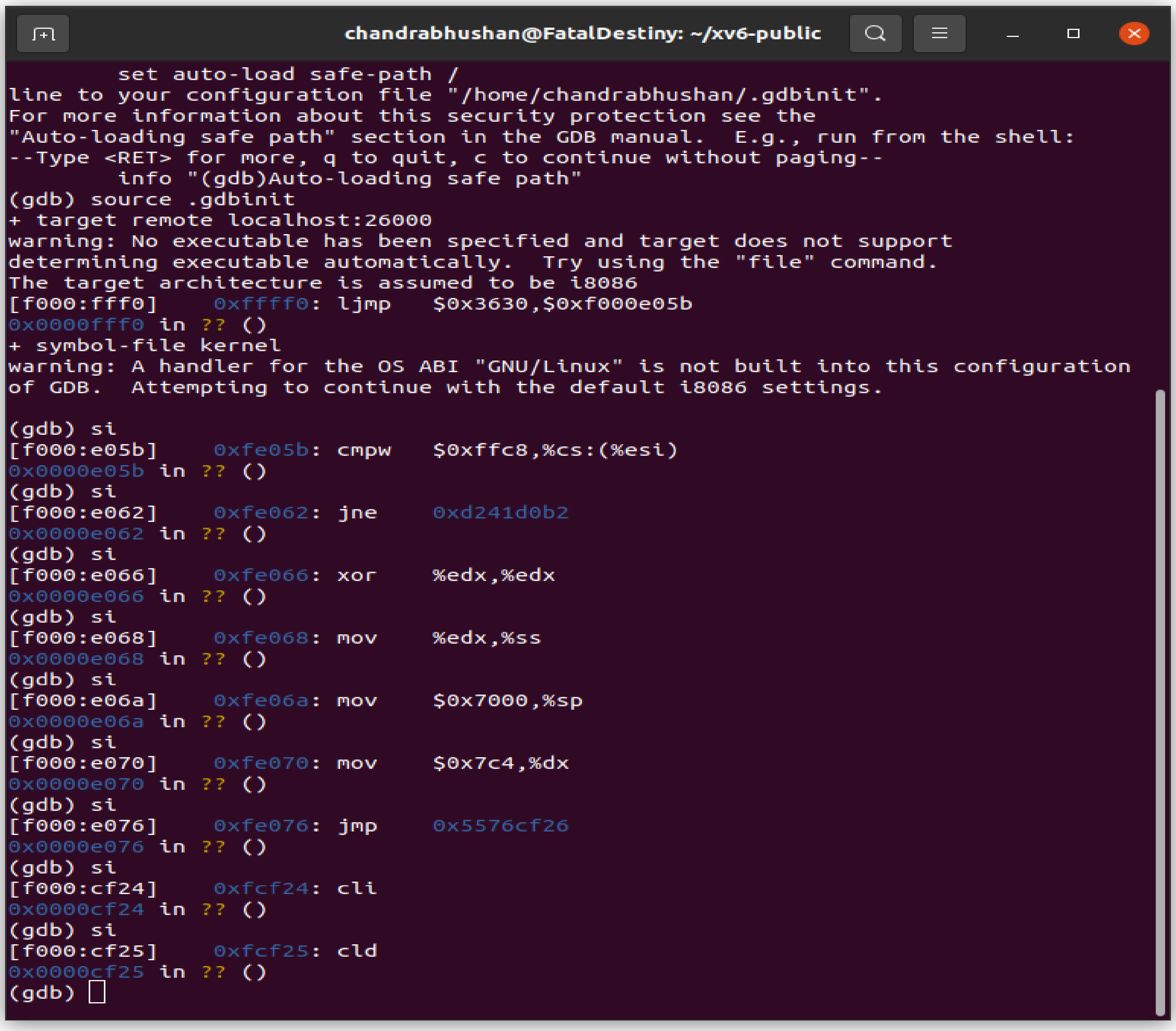
Assignment 1

Name : Chandrabhushan Reddy

Roll Number : 200101027

Exercise 1: Modified code has been submitted in the zip file. Name of the file is exercise1.c

Exercise 2: The following image depicts few instructions of ROM BIOS



```
        set auto-load safe-path /
line to your configuration file "/home/chandrabhushan/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
--Type <RET> for more, q to quit, c to continue without paging--
        info "(gdb)Auto-loading safe path"
(gdb) source .gdbinit
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(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. Hence, the first thing that the BIOS does is jmp backwards to an earlier location in the BIOS i.e. fe05b

2. [f000:e05b] 0xfe05b: cmpw $0xffc8, %cs: (%esi)

- Compares **ffc8** with the contents of location pointed by code segment and instruction pointer whose values are in cs register and esi register respectively

3. [f000:e062] 0xfe062: jne 0xd241d0b2

- If the previous comparison results in an inequality then jump to location **0xd241d0b2.** Else go to the next instruction

4. [f000:e066] 0xfe066: xor %edx, %edx

- Sets the value of edx to zero i.e., %edx is 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 **0x7000**

7. [f000:e070] 0xfe070: mov $0x7c4, %dx

- Sets the value of dx register to **0x7c4**

8. [f000:e076] 0xfe076: jmp 0x5576cf26

- Jumps to the location **0x5576cf26**

9. [f000:cf24] 0xfcf24: cli

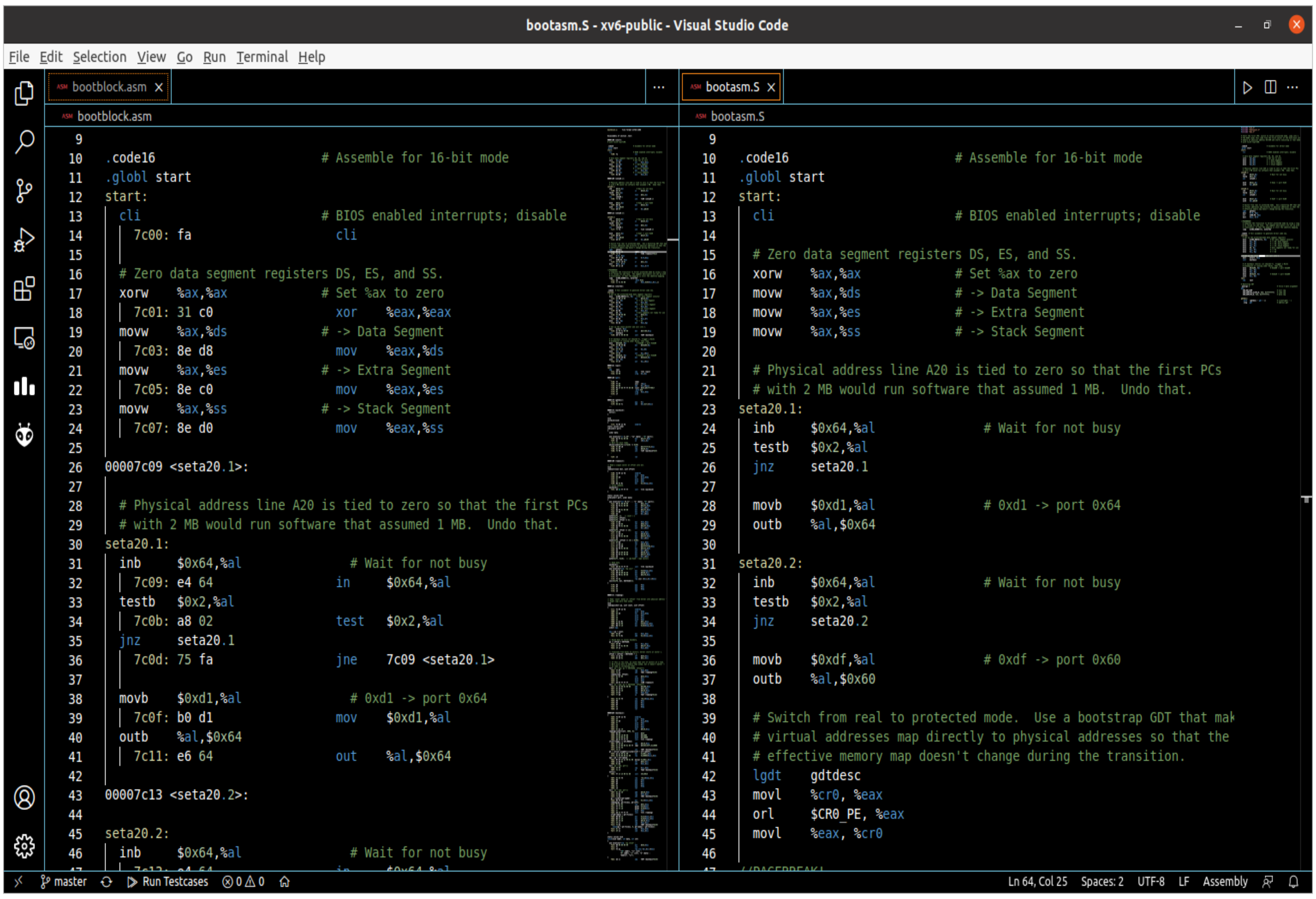
◦ Clears the interrupt flag. This is done because the CPU should not be interrupted while boot loading.

10. [f000:cf25] 0xfcf25: cld

◦ Clears the direction flag. This is done for the proper execution of subsequent instructions.

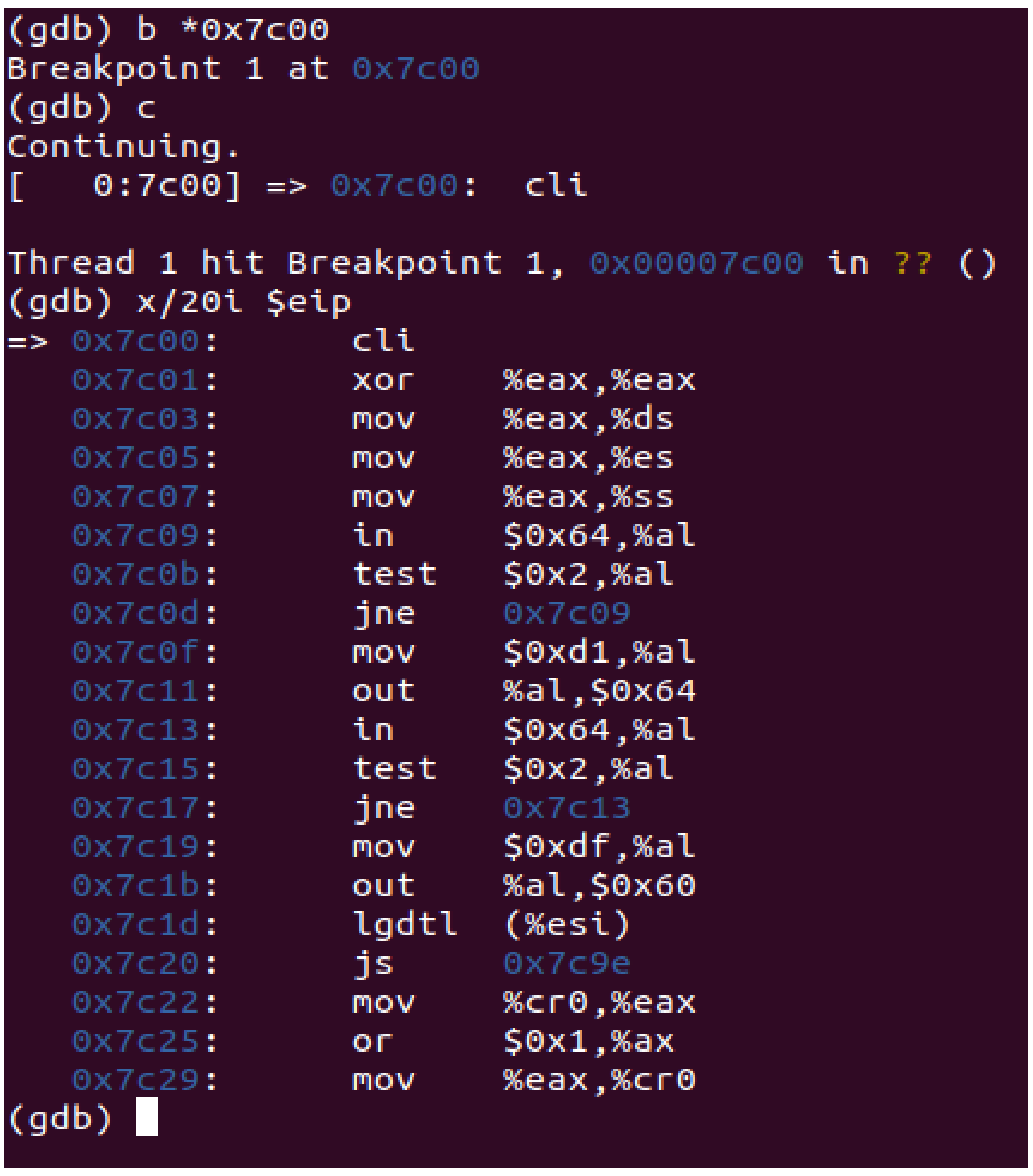
Exercise 3 :

The below figures depict the comparisions between bootasm.S, bootblock.asm and GDB disassembly


bootasm.S - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
bootblock.asm
bootblock.asm
bootasm.S
bootasm.S
master
Run Testcases
0 0
Ln 64, Col 25 Spaces: 2 UTF-8 LF Assembly

```
.code16                       # Assemble for 16-bit mode
.globl start
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

00007c13 <seta20.2>:

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

```
.code16                       # Assemble for 16-bit mode
.globl start
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

  # 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
  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 mak
  # 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

```

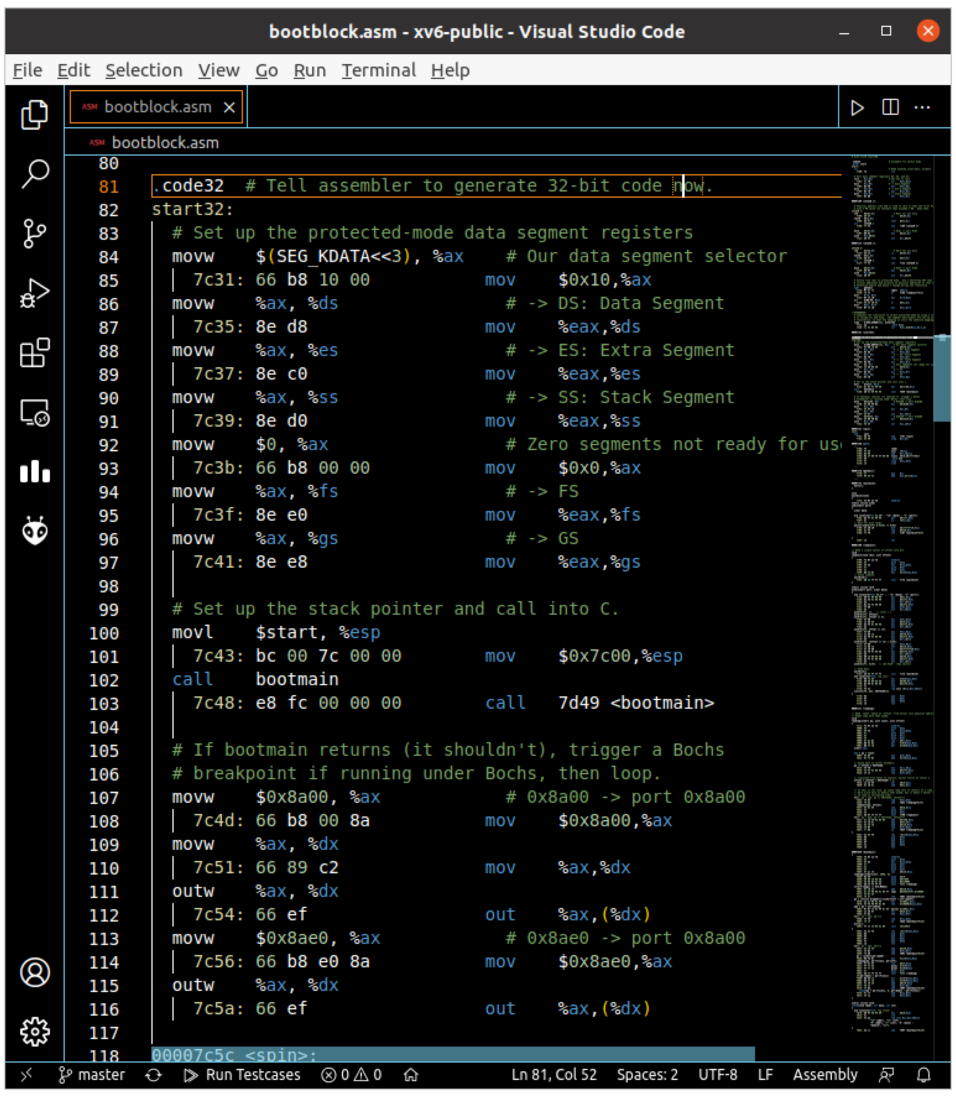


```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

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
(gdb) 
```

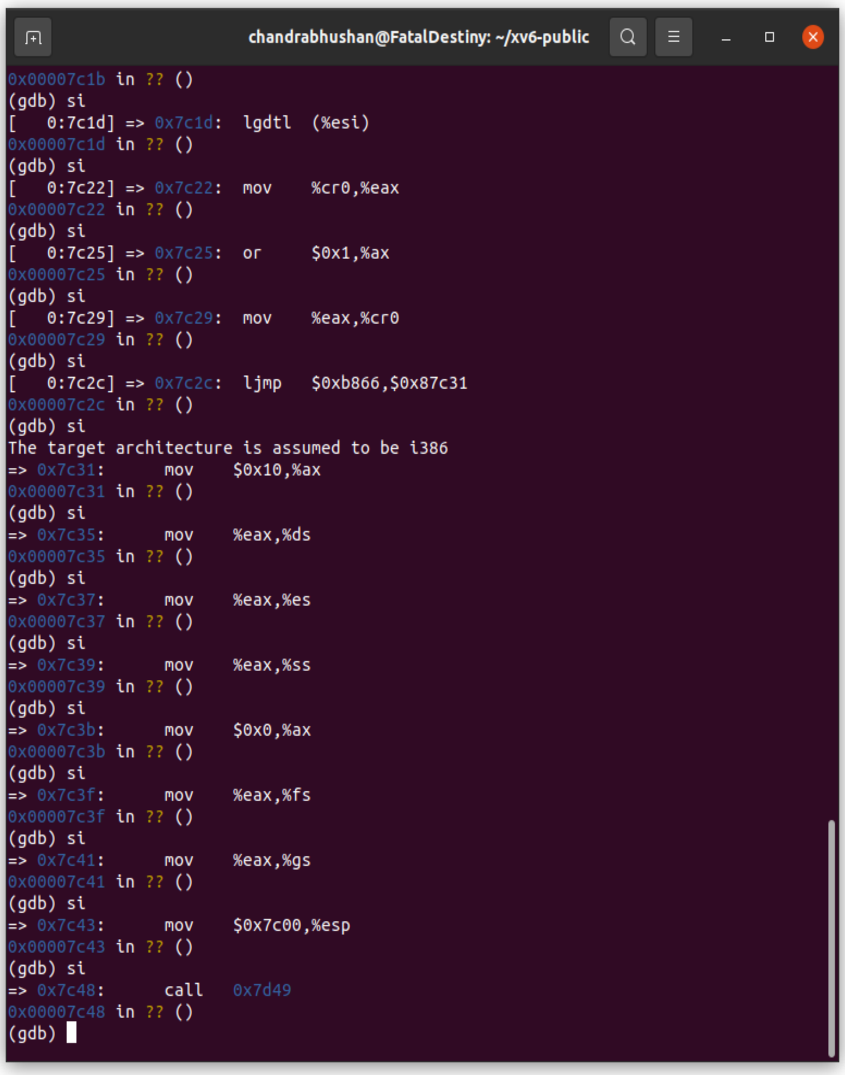
Main differences between bootasm.S and the disassembly on GDB/ bootblock.asm are that:

- **lgdt gdtdesc** instruction in bootasm.S has been expanded to two constituent instructions: **lgdtl (%esi)** and **js 0x7c9e**
- All the instructions starting from cli all the way up to mov %eax, %cr0 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)

Code Tracing bootmain() and readsect() in bootmain.c :

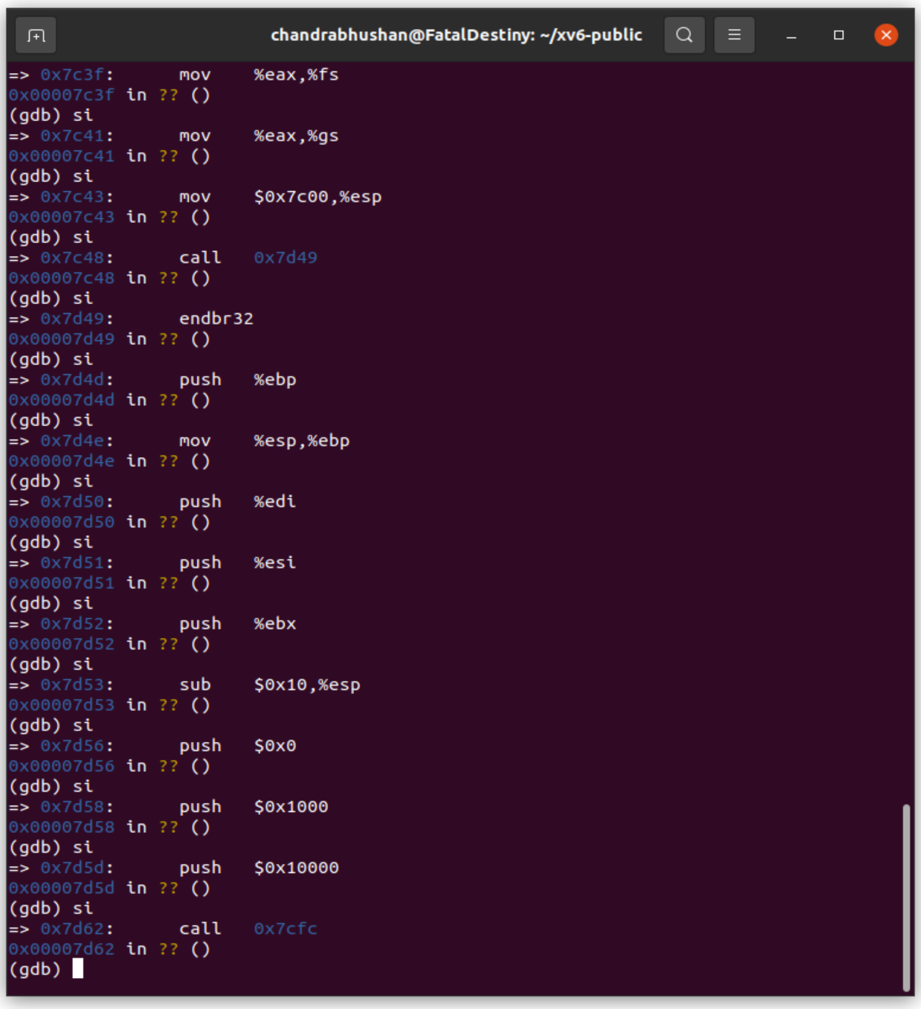

bootblock.asm - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
bootblock.asm
bootblock.asm
master
Run Testcases
⊗ 0 ⚠ 0
Ln 81, Col 52
Spaces: 2
UTF-8
LF
Assembly

```
 80
 81  .code32  # Tell assembler to generate 32-bit code now.
 82  start32:
 83    # Set up the protected-mode data segment registers
 84    movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
 85    | 7c31: 66 b8 10 00            mov    $0x10,%ax
 86    movw    %ax, %ds                # -> DS: Data Segment
 87    | 7c35: 8e d8                  mov    %eax,%ds
 88    movw    %ax, %es                # -> ES: Extra Segment
 89    | 7c37: 8e c0                  mov    %eax,%es
 90    movw    %ax, %ss                # -> SS: Stack Segment
 91    | 7c39: 8e d0                  mov    %eax,%ss
 92    movw    $0, %ax                 # Zero segments not ready for us
 93    | 7c3b: 66 b8 00 00            mov    $0x0,%ax
 94    movw    %ax, %fs                # -> FS
 95    | 7c3f: 8e e0                  mov    %eax,%fs
 96    movw    %ax, %gs                # -> GS
 97    | 7c41: 8e e8                  mov    %eax,%gs
 98    |
 99    # Set up the stack pointer and call into C.
100    movl    $start, %esp
101    | 7c43: bc 00 7c 00 00         mov    $0x7c00,%esp
102    call    bootmain
103    | 7c48: e8 fc 00 00 00         call   7d49 <bootmain>
104    |
105    # If bootmain returns (it shouldn't), trigger a Bochs
106    # breakpoint if running under Bochs, then loop.
107    movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
108    | 7c4d: 66 b8 00 8a            mov    $0x8a00,%ax
109    movw    %ax, %dx
110    | 7c51: 66 89 c2               mov    %ax,%dx
111    outw    %ax, %dx
112    | 7c54: 66 ef                  out    %ax,(%dx)
113    movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
114    | 7c56: 66 b8 e0 8a            mov    $0x8ae0,%ax
115    outw    %ax, %dx
116    | 7c5a: 66 ef                  out    %ax,(%dx)
117
118  00007c5c <spin>:
```

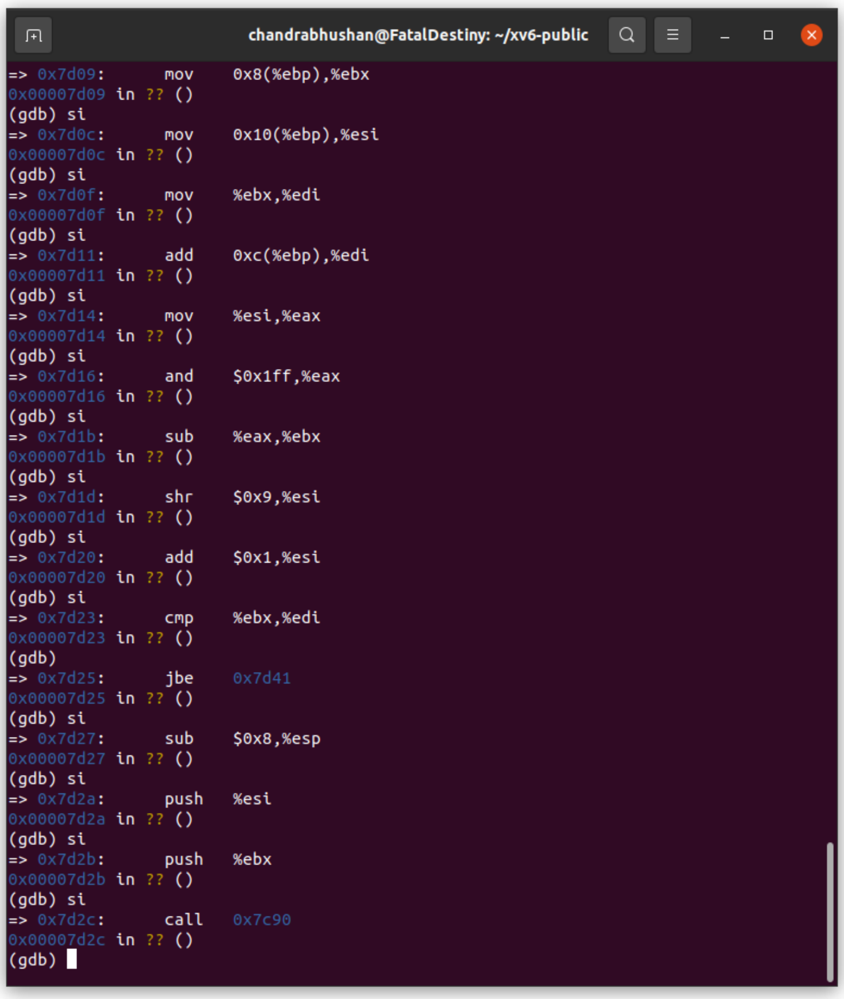


```
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  lgdtl  (%esi)
0x00007c1d in ?? ()
(gdb) si
[   0:7c22] => 0x7c22:  mov    %cr0,%eax
0x00007c22 in ?? ()
(gdb) si
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb) si
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$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 ?? ()
(gdb) si
=> 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)
```

In the above bootblock.asm image, line 102 contains the call to bootmain(). This corresponds to the instruction at location 0x7c48 in the GDB call to bootmain (as clearly depicted in the above right image)

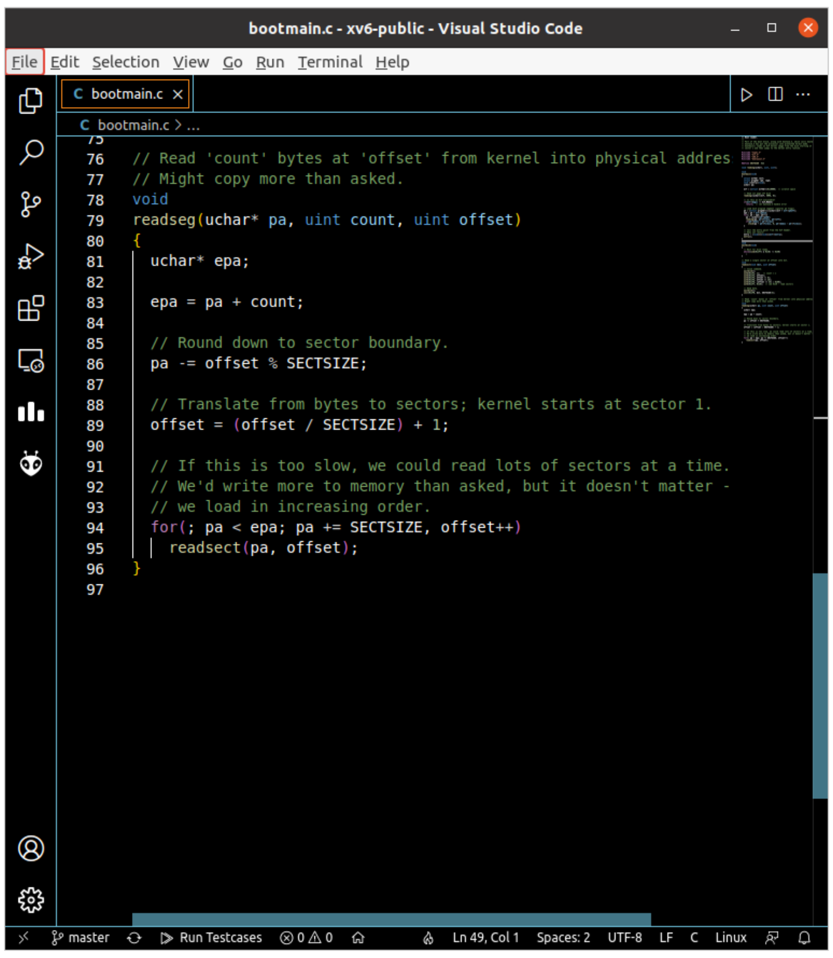


```
=> 0x7c3f:      mov    %eax,%fs
0x00007c3f in ?? ()
(gdb) si
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb) si
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb) si
=> 0x7c48:      call   0x7d49
0x00007c48 in ?? ()
(gdb) si
=> 0x7d49:      endbr32
0x00007d49 in ?? ()
(gdb) si
=> 0x7d4d:      push   %ebp
0x00007d4d in ?? ()
(gdb) si
=> 0x7d4e:      mov    %esp,%ebp
0x00007d4e in ?? ()
(gdb) si
=> 0x7d50:      push   %edi
0x00007d50 in ?? ()
(gdb) si
=> 0x7d51:      push   %esi
0x00007d51 in ?? ()
(gdb) si
=> 0x7d52:      push   %ebx
0x00007d52 in ?? ()
(gdb) si
=> 0x7d53:      sub    $0x10,%esp
0x00007d53 in ?? ()
(gdb) si
=> 0x7d56:      push   $0x0
0x00007d56 in ?? ()
(gdb) si
=> 0x7d58:      push   $0x1000
0x00007d58 in ?? ()
(gdb) si
=> 0x7d5d:      push   $0x10000
0x00007d5d in ?? ()
(gdb) si
=> 0x7d62:      call   0x7cfc
0x00007d62 in ?? ()
(gdb)
```

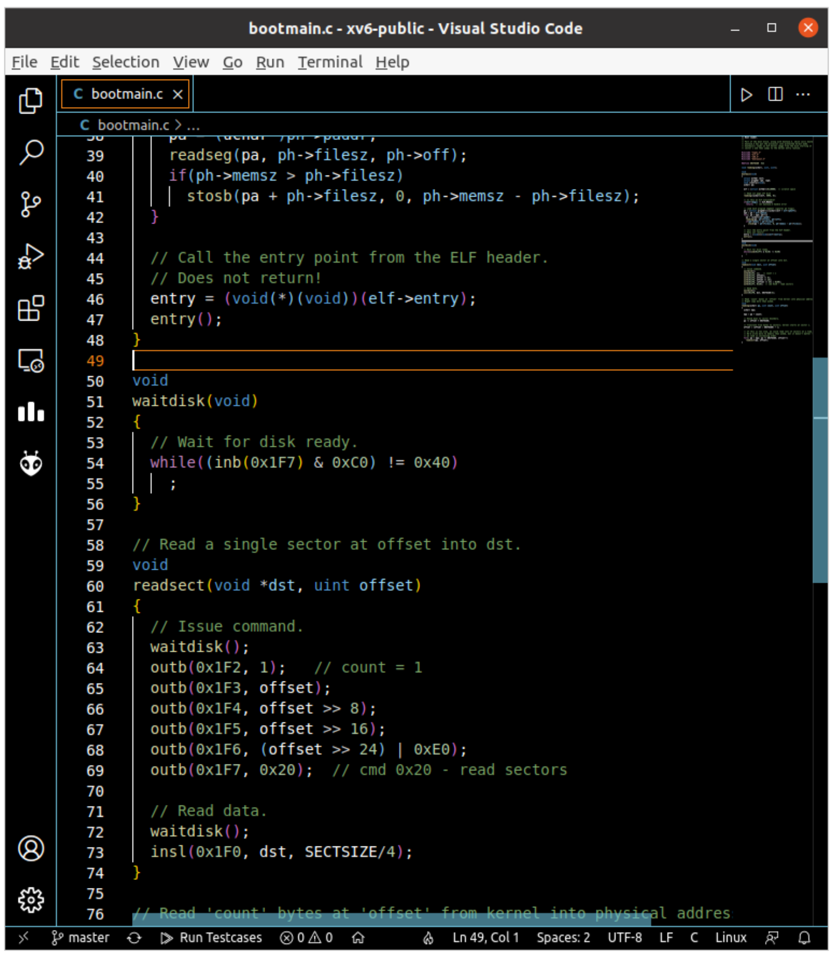


```
=> 0x7d09:      mov    0x8(%ebp),%ebx
0x00007d09 in ?? ()
(gdb) si
=> 0x7d0c:      mov    0x10(%ebp),%esi
0x00007d0c in ?? ()
(gdb) si
=> 0x7d0f:      mov    %ebx,%edi
0x00007d0f in ?? ()
(gdb) si
=> 0x7d11:      add    0xc(%ebp),%edi
0x00007d11 in ?? ()
(gdb) si
=> 0x7d14:      mov    %esi,%eax
0x00007d14 in ?? ()
(gdb) si
=> 0x7d16:      and    $0x1ff,%eax
0x00007d16 in ?? ()
(gdb) si
=> 0x7d1b:      sub    %eax,%ebx
0x00007d1b in ?? ()
(gdb) si
=> 0x7d1d:      shr    $0x9,%esi
0x00007d1d in ?? ()
(gdb) si
=> 0x7d20:      add    $0x1,%esi
0x00007d20 in ?? ()
(gdb) si
=> 0x7d23:      cmp    %ebx,%edi
0x00007d23 in ?? ()
(gdb)
=> 0x7d25:      jbe    0x7d41
0x00007d25 in ?? ()
(gdb) si
=> 0x7d27:      sub    $0x8,%esp
0x00007d27 in ?? ()
(gdb) si
=> 0x7d2a:      push   %esi
0x00007d2a in ?? ()
(gdb) si
=> 0x7d2b:      push   %ebx
0x00007d2b in ?? ()
(gdb) si
=> 0x7d2c:      call   0x7c90
0x00007d2c in ?? ()
(gdb)
```

The above left image depicts the call to **readseg()** at location **0x7d62** and the right above image depicts the call to **readsect()** at location **0x7d2c .** These images together shows the process of tracing through bootmain() into readseg() and finally into readsect().


bootmain.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
bootmain.c
bootmain.c > ...
master
Run Testcases
Ln 49, Col 1 Spaces: 2 UTF-8 LF C Linux

```c
76  // Read 'count' bytes at 'offset' from kernel into physical addres
77  // Might copy more than asked.
78  void
79  readseg(uchar* pa, uint count, uint offset)
80  {
81    uchar* epa;
82
83    epa = pa + count;
84
85    // Round down to sector boundary.
86    pa -= offset % SECTSIZE;
87
88    // Translate from bytes to sectors; kernel starts at sector 1.
89    offset = (offset / SECTSIZE) + 1;
90
91    // If this is too slow, we could read lots of sectors at a time.
92    // We'd write more to memory than asked, but it doesn't matter -
93    // we load in increasing order.
94    for(; pa < epa; pa += SECTSIZE, offset++)
95      readsect(pa, offset);
96  }
97
```

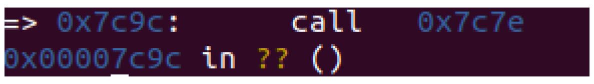

bootmain.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
bootmain.c
bootmain.c > ...
master
Run Testcases
Ln 49, Col 1 Spaces: 2 UTF-8 LF C Linux

```c
39      readseg(pa, ph->filesz, ph->off);
40      if(ph->memsz > ph->filesz)
41        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
42    }
43
44    // Call the entry point from the ELF header.
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)
55      ;
56  }
57
58  // Read a single sector at offset into dst.
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  }
75
76  // Read 'count' bytes at 'offset' from kernel into physical addres
```

The above images show **readseg()** and **readsect()** in bootmain.c at line numbers 79 and 60 respectively.

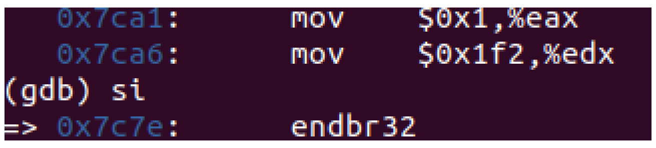
Assembly level instructions corresponding to each instruction in readsect():

Line 63 :



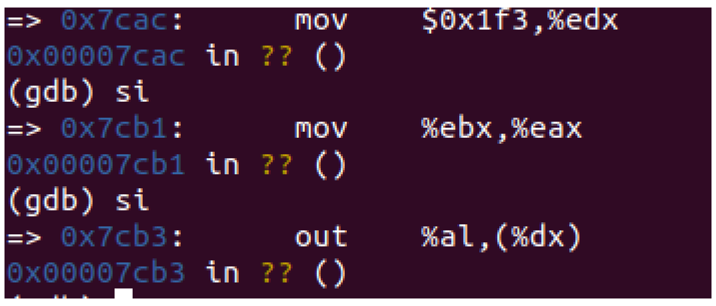
```
=> 0x7c9c:      call   0x7c7e
0x00007c9c in ?? ()
```

Line 64 :



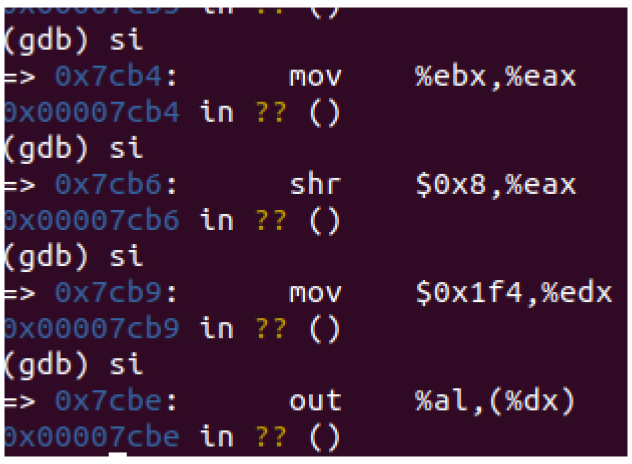
```
   0x7ca1:      mov    $0x1,%eax
   0x7ca6:      mov    $0x1f2,%edx
(gdb) si
=> 0x7c7e:      endbr32
```

Line 65 :



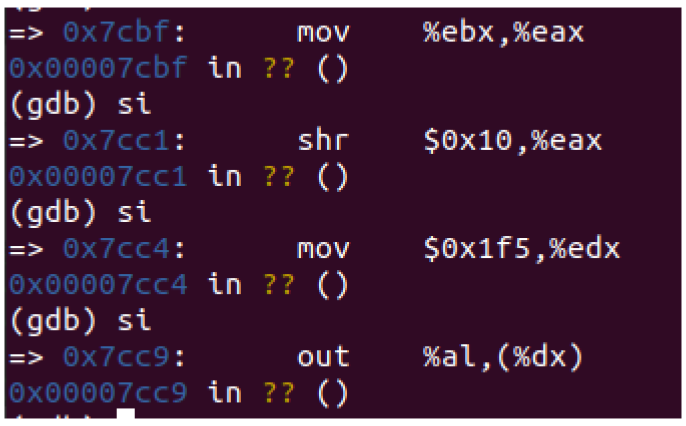
```
=> 0x7cac:      mov    $0x1f3,%edx
0x00007cac in ?? ()
(gdb) si
=> 0x7cb1:      mov    %ebx,%eax
0x00007cb1 in ?? ()
(gdb) si
=> 0x7cb3:      out    %al,(%dx)
0x00007cb3 in ?? ()
```

Line 66 :



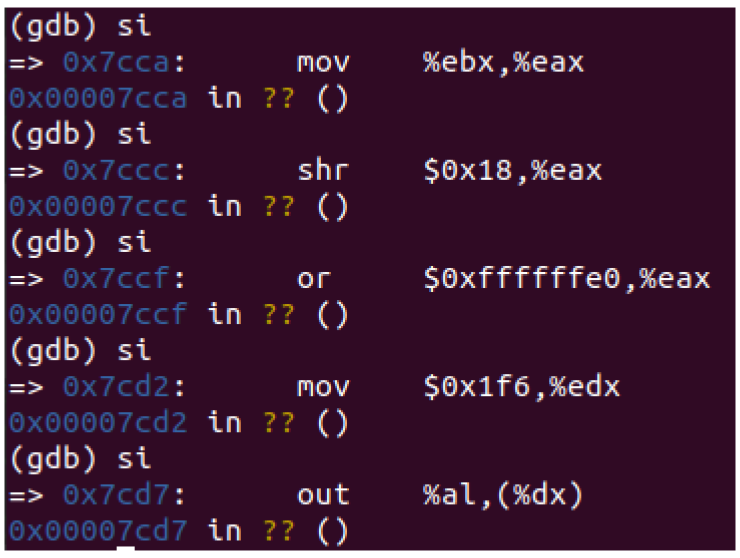
```
(gdb) si
=> 0x7cb4:      mov    %ebx,%eax
0x00007cb4 in ?? ()
(gdb) si
=> 0x7cb6:      shr    $0x8,%eax
0x00007cb6 in ?? ()
(gdb) si
=> 0x7cb9:      mov    $0x1f4,%edx
0x00007cb9 in ?? ()
(gdb) si
=> 0x7cbe:      out    %al,(%dx)
0x00007cbe in ?? ()
```

Line 67 :



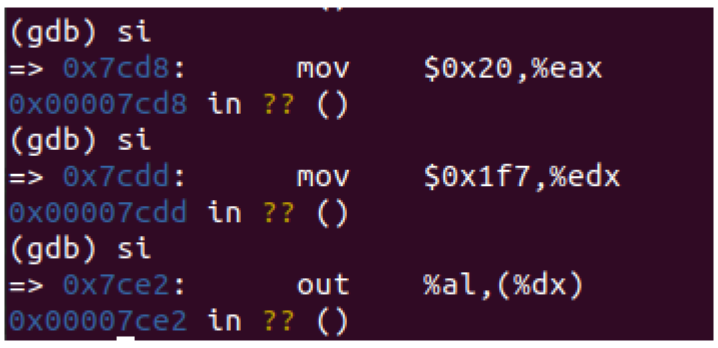
```
=> 0x7cbf:      mov    %ebx,%eax
0x00007cbf in ?? ()
(gdb) si
=> 0x7cc1:      shr    $0x10,%eax
0x00007cc1 in ?? ()
(gdb) si
=> 0x7cc4:      mov    $0x1f5,%edx
0x00007cc4 in ?? ()
(gdb) si
=> 0x7cc9:      out    %al,(%dx)
0x00007cc9 in ?? ()
```

Line 68 :



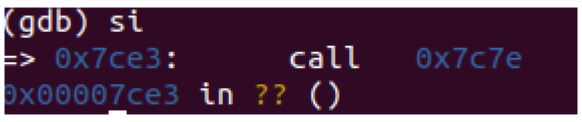
```
(gdb) si
=> 0x7cca:      mov    %ebx,%eax
0x00007cca in ?? ()
(gdb) si
=> 0x7ccc:      shr    $0x18,%eax
0x00007ccc in ?? ()
(gdb) si
=> 0x7ccf:      or     $0xffffffe0,%eax
0x00007ccf in ?? ()
(gdb) si
=> 0x7cd2:      mov    $0x1f6,%edx
0x00007cd2 in ?? ()
(gdb) si
=> 0x7cd7:      out    %al,(%dx)
0x00007cd7 in ?? ()
```

Line 69 :



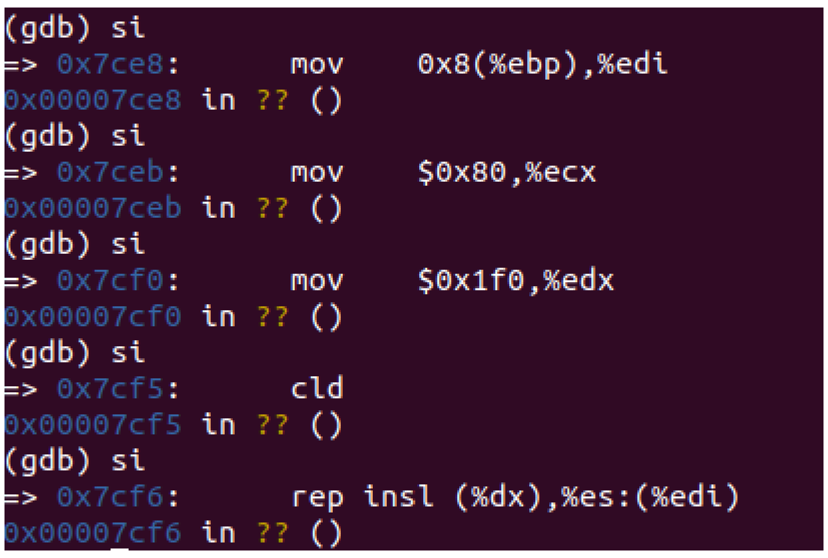
```
(gdb) si
=> 0x7cd8:      mov    $0x20,%eax
0x00007cd8 in ?? ()
(gdb) si
=> 0x7cdd:      mov    $0x1f7,%edx
0x00007cdd in ?? ()
(gdb) si
=> 0x7ce2:      out    %al,(%dx)
0x00007ce2 in ?? ()
```

Line 72 :



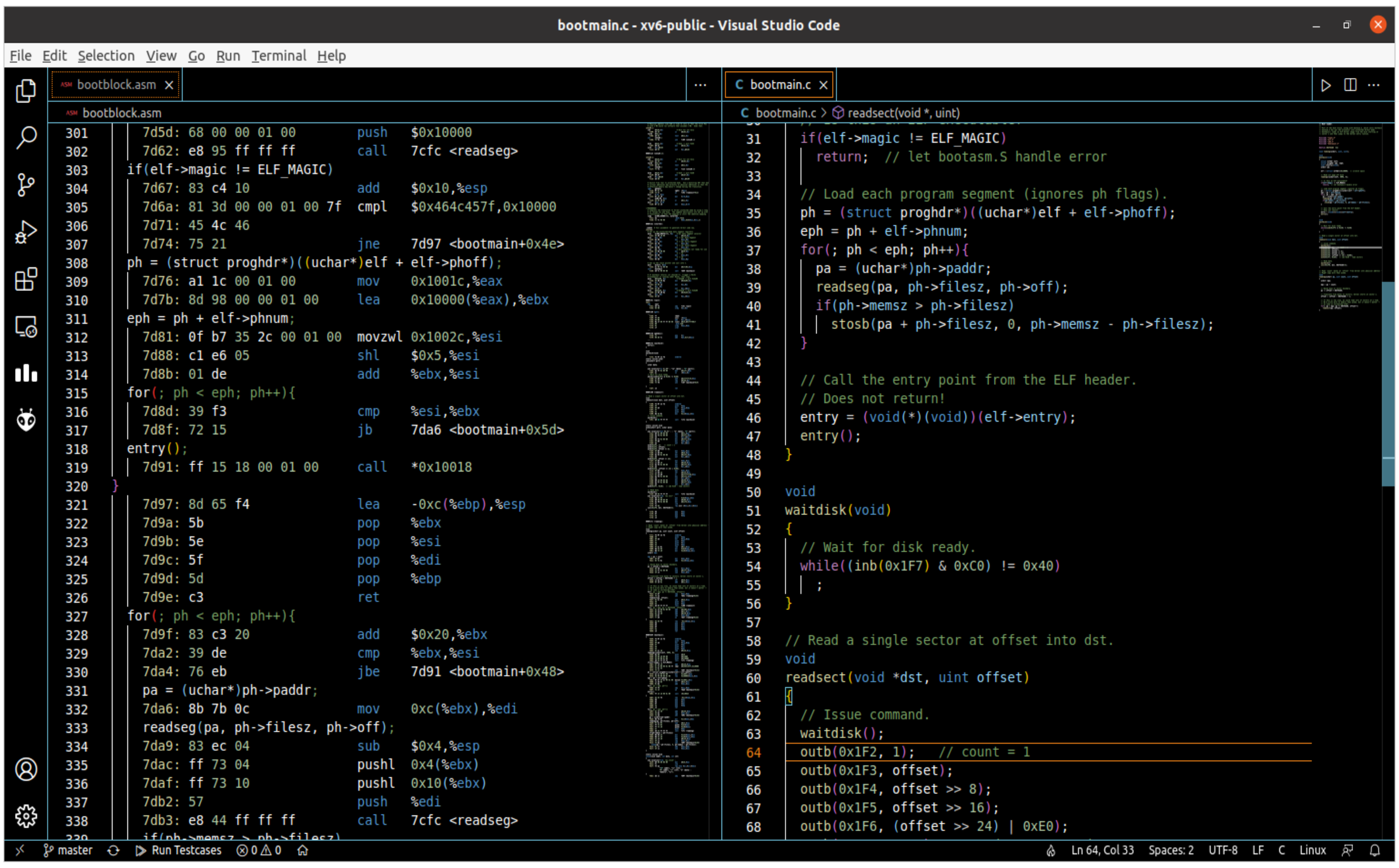
```
(gdb) si
=> 0x7ce3:      call   0x7c7e
0x00007ce3 in ?? ()
```

Line 73 :



```
(gdb) si
=> 0x7ce8:      mov    0x8(%ebp),%edi
0x00007ce8 in ?? ()
(gdb) si
=> 0x7ceb:      mov    $0x80,%ecx
0x00007ceb in ?? ()
(gdb) si
=> 0x7cf0:      mov    $0x1f0,%edx
0x00007cf0 in ?? ()
(gdb) si
=> 0x7cf5:      cld
0x00007cf5 in ?? ()
(gdb) si
=> 0x7cf6:      rep insl (%dx),%es:(%edi)
0x00007cf6 in ?? ()
```

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



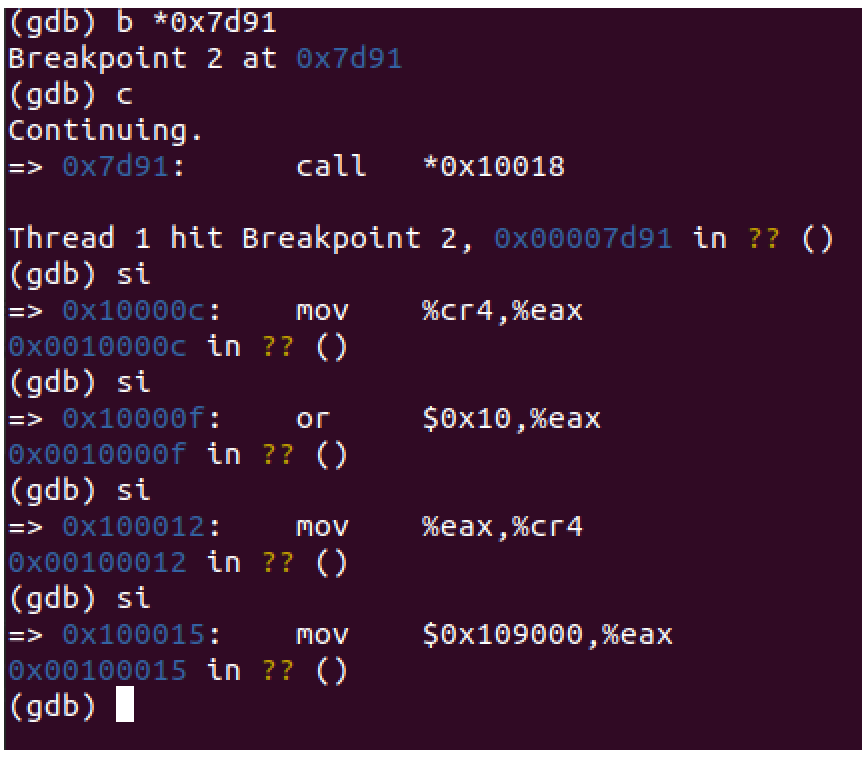
bootmain.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
bootblock.asm
bootblock.asm
bootmain.c
bootmain.c > readsect(void *, uint)
master
Run Testcases
0 0
Ln 64, Col 33 Spaces: 2 UTF-8 LF C Linux

```
301      7d5d: 68 00 00 01 00        push   $0x10000
302      7d62: e8 95 ff ff ff        call   7cfc <readseg>
303    if(elf->magic != ELF_MAGIC)
304      7d67: 83 c4 10              add    $0x10,%esp
305      7d6a: 81 3d 00 00 01 00 7f  cmpl   $0x464c457f,0x10000
306      7d71: 45 4c 46
307      7d74: 75 21                 jne    7d97 <bootmain+0x4e>
308    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
309      7d76: a1 1c 00 01 00        mov    0x1001c,%eax
310      7d7b: 8d 98 00 00 01 00     lea    0x10000(%eax),%ebx
311    eph = ph + elf->phnum;
312      7d81: 0f b7 35 2c 00 01 00  movzwl 0x1002c,%esi
313      7d88: c1 e6 05              shl    $0x5,%esi
314      7d8b: 01 de                 add    %ebx,%esi
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)
```

```
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    }
43
44    // Call the entry point from the ELF header.
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)
55      ;
56  }
57
58  // Read a single sector at offset into dst.
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);
```

The above image shows the for loop that reads remaining sectors of the kernel into the memory. The left image shows the assembly code and the right image shows the source code.

The code on line 46-47 (in right side source file) whose assembly is on line 318-319 (left side assembly file) 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.

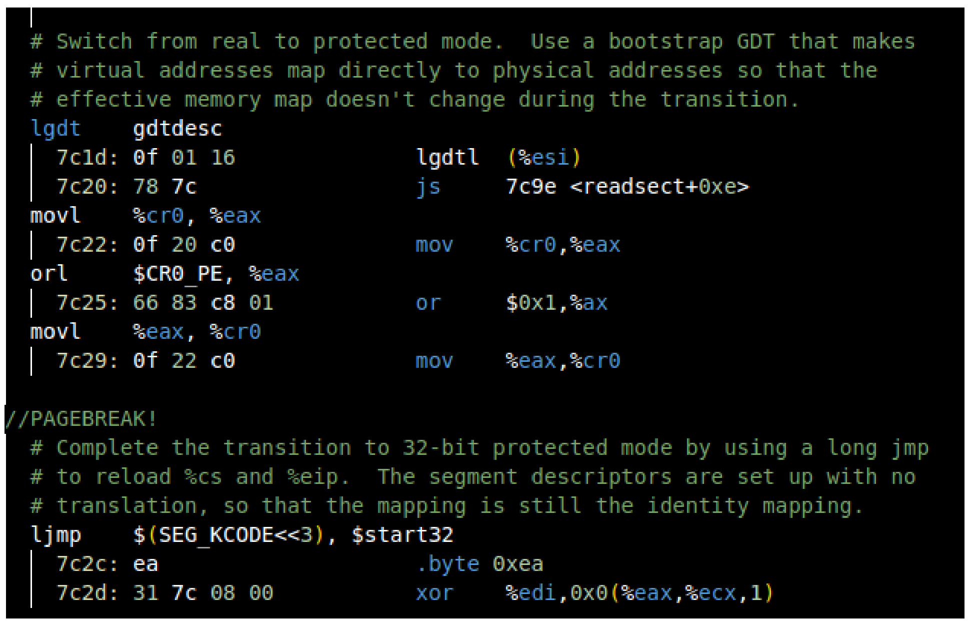


```
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(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) █
```

The breakpoint is set at 0x7d91 (line 319) as shown in the above image

Question 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 loaderdeactivates 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
   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

//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
   7c2c: ea                    .byte 0xea
   7c2d: 31 7c 08 00           xor    %edi,0x0(%eax,%ecx,1)
```

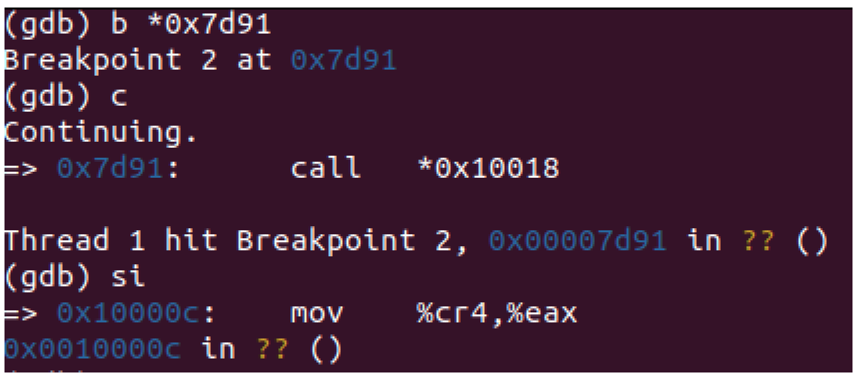
Assembly code to switch from 16 bit Real mode to 32 bit Protected mode.

Question 2 :

From the above discussion it is clear that the last instruction of boot loader would be to call the entry function (on line 47 in 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

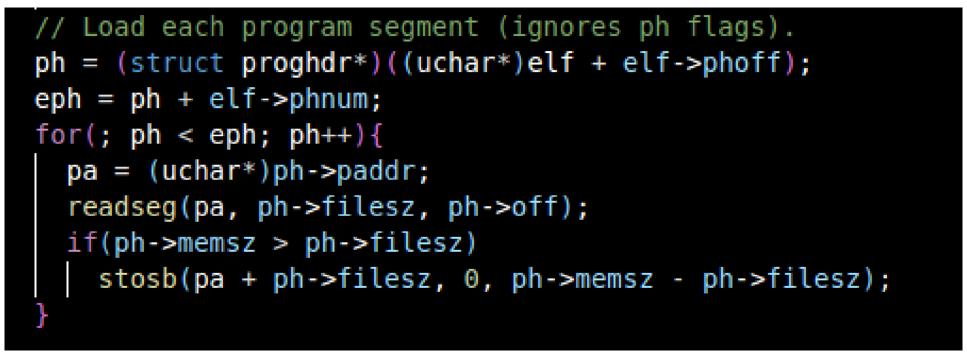
First instruction of kernel: mov %cr4, %eax



```
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
```

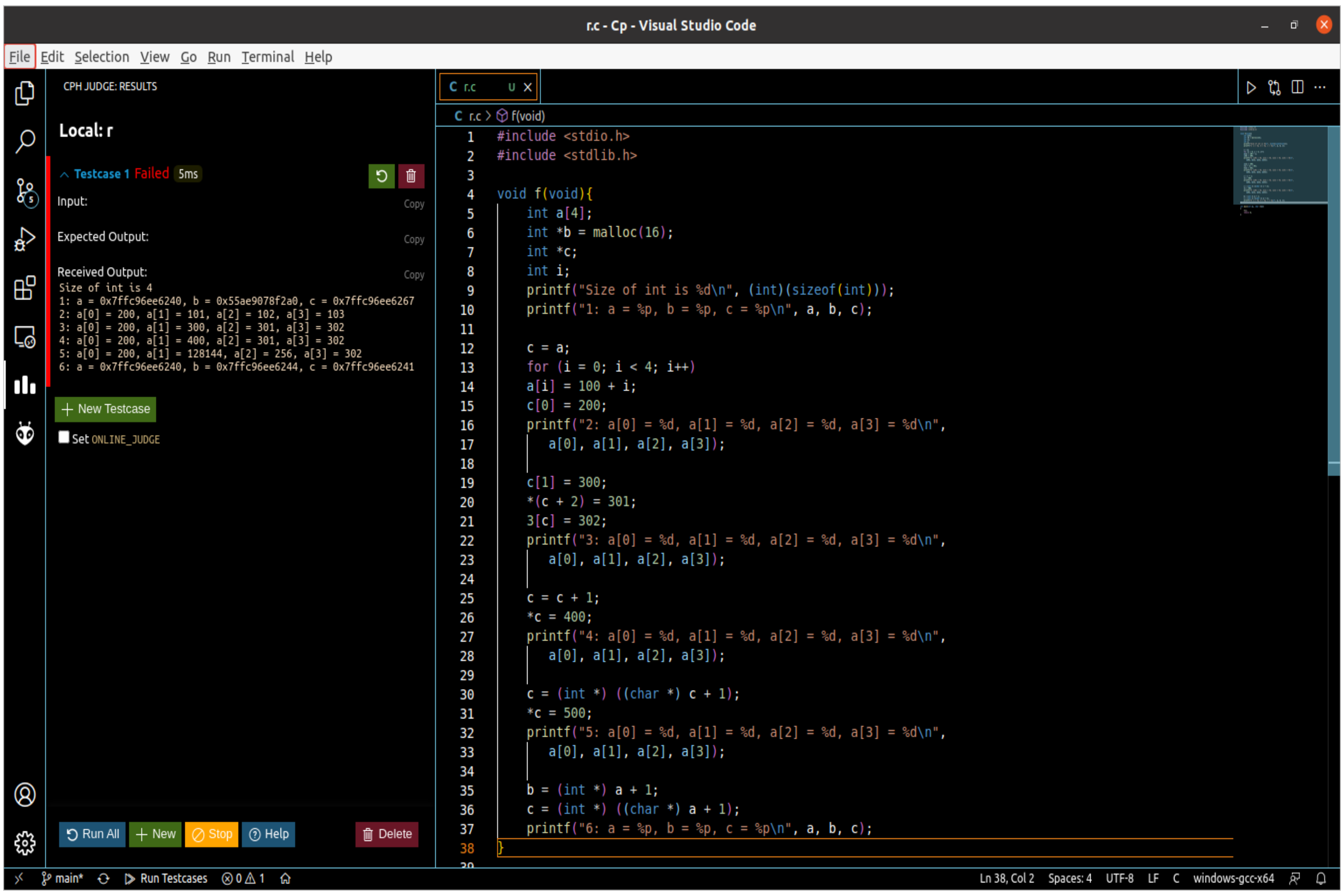
Question 3 :

It is clear from the following image 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);
}
```

Exercise 4 :



```
Size of int is 4
1: a = 0x7ffc96ee6240, b = 0x55ae9078f2a0, c = 0x7ffc96ee6267
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 = 0x7ffc96ee6240, b = 0x7ffc96ee6244, c = 0x7ffc96ee6241
```

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void f(void){
5       int a[4];
6       int *b = malloc(16);
7       int *c;
8       int i;
9       printf("Size of int is %d\n", (int)(sizeof(int)));
10      printf("1: a = %p, b = %p, c = %p\n", a, b, c);
11
12      c = a;
13      for (i = 0; i < 4; i++)
14      a[i] = 100 + i;
15      c[0] = 200;
16      printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
17          a[0], a[1], a[2], a[3]);
18
19      c[1] = 300;
20      *(c + 2) = 301;
21      3[c] = 302;
22      printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
23          a[0], a[1], a[2], a[3]);
24
25      c = c + 1;
26      *c = 400;
27      printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
28          a[0], a[1], a[2], a[3]);
29
30      c = (int *) ((char *) c + 1);
31      *c = 500;
32      printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
33          a[0], a[1], a[2], a[3]);
34
35      b = (int *) a + 1;
36      c = (int *) ((char *) a + 1);
37      printf("6: a = %p, b = %p, c = %p\n", a, b, c);
38  }
```

Below is the line by line explanation of each line of output for above code :

Explanation for 1st output line: 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.

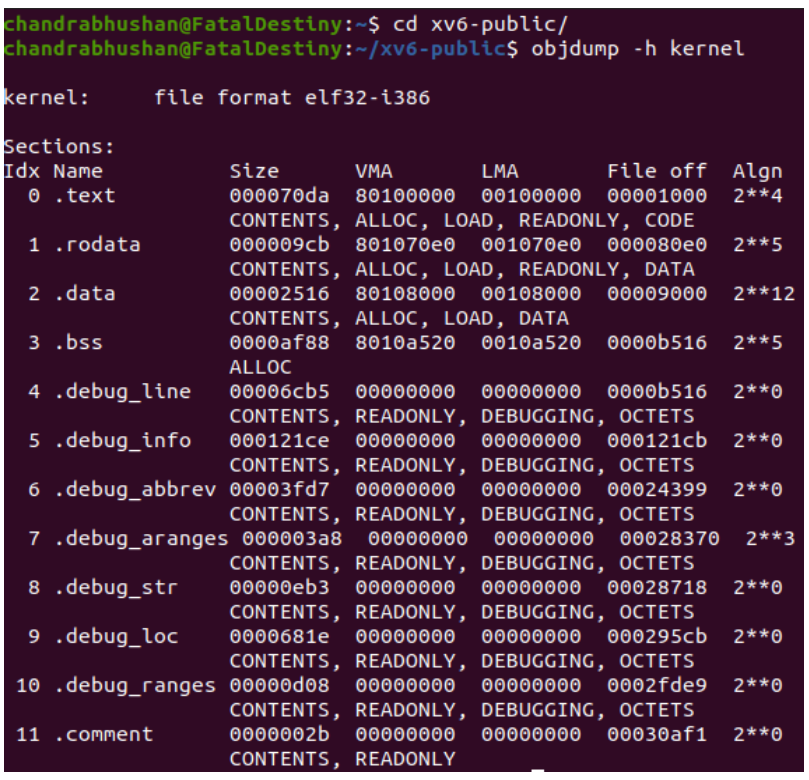
Explanation for 2nd output line: After line 13, the pointer c stores the address of array a. Thus line 15 changes the value of previously assigned a[0] of 100 to 200 and so in the output the value of a[0] is 200 whereas the other values of the array are same as before.

Explanation for 3rd output line: 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].

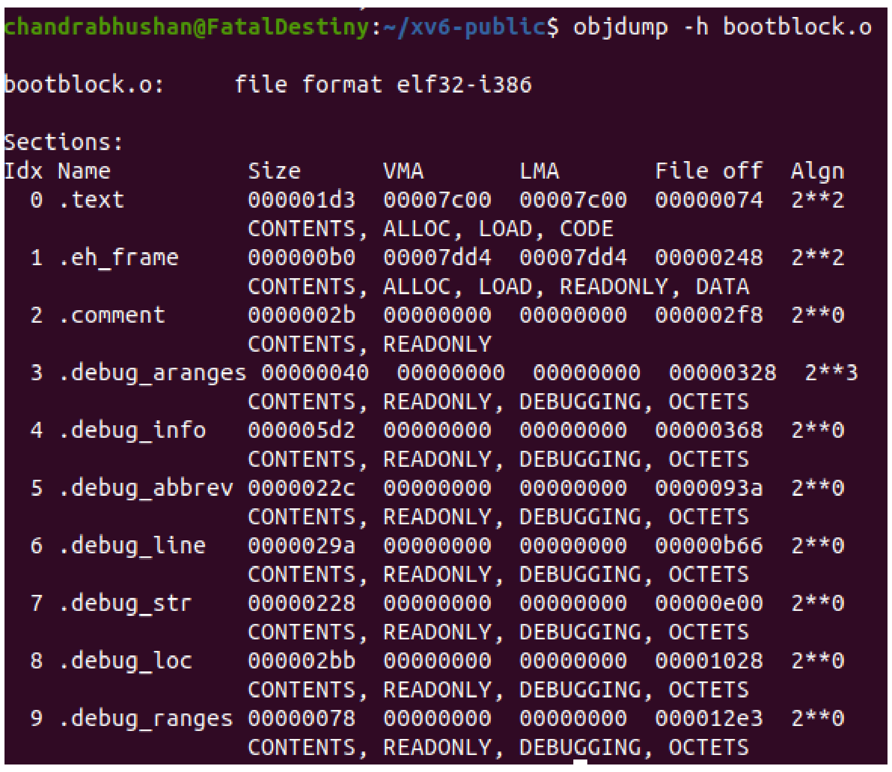
Explanation for 4th output line: Line 26 increases the address in c by 4 bytes (since c is of the type int*). Thus c now points to address of a[1] and Line 27 changes a[1] to 400. Hence the output in line 4 is same as that of line 2 except the value of a[1] is now 400.

Explanation for 5th output line: 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.

Explanation for 6th output line: 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.



```
chandrabhushan@FatalDestiny:~$ cd xv6-public/
chandrabhushan@FatalDestiny:~/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    00000eb3  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295cb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002b  00000000  00000000  00030af1  2**0
                  CONTENTS, READONLY
```



```
chandrabhushan@FatalDestiny:~/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    00000228  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001028  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012e3  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Above figures show various program sections of the kernel (left above image) and bootblock.o (right above image) 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 6 :

Below is the hypothesis made by observations:

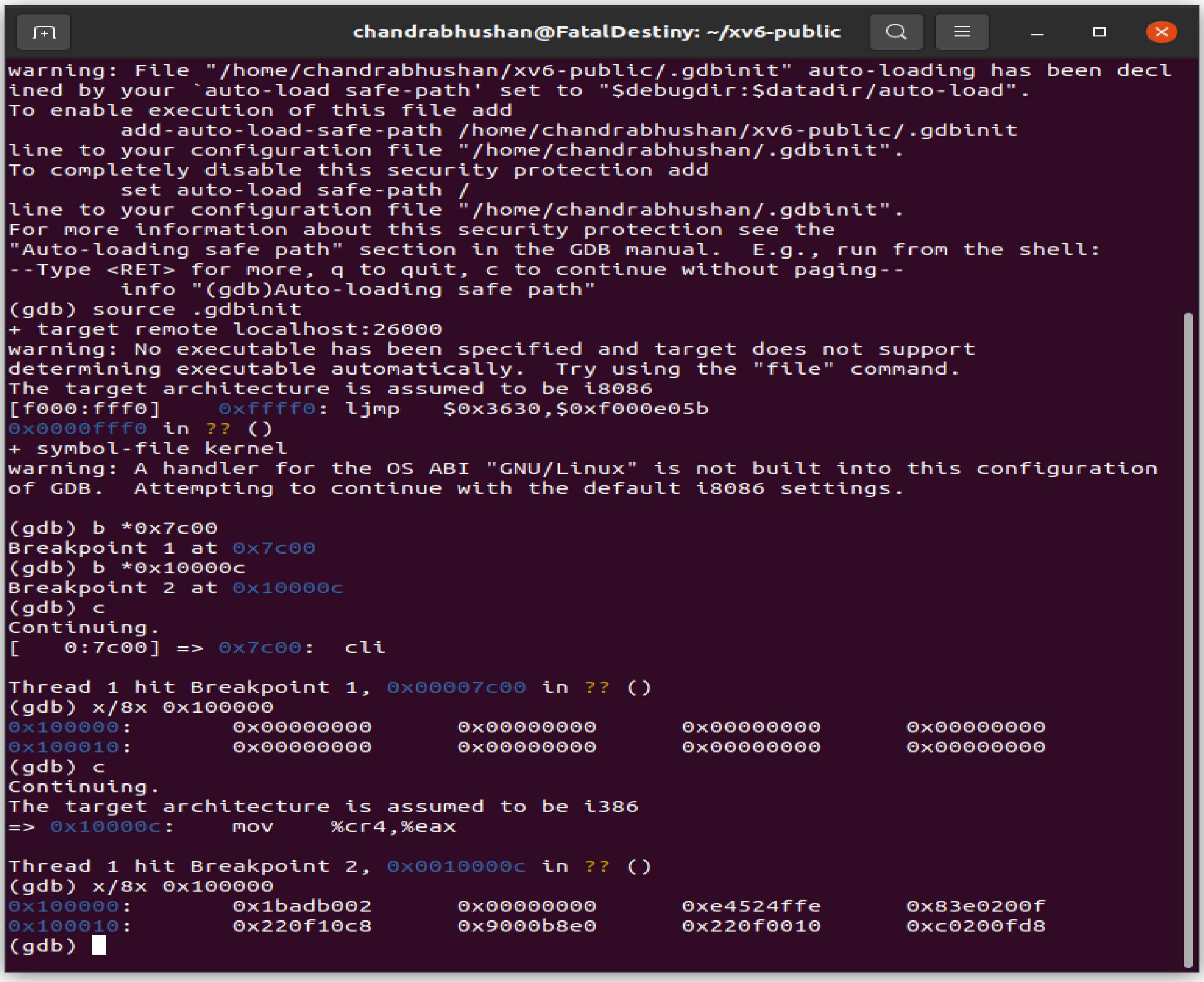
When BIOS just enters the bootloader no useful information is present at 0x100000. However, from Exercise 3 Question 3 we know that the boot loader stores the kernel (ELF Header) starting from address 0x100000. Hence, when the bootloader enter the kernel, the contents at 0x100000 contains the kernel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

We can confirm this using GDB as shown in the following figure. 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 the figure pasted in exercise 3.



chandrabhushan@FatalDestiny: ~/xv6-public

```
warning: File "/home/chandrabhushan/xv6-public/.gdbinit" auto-loading has been decl
ined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/chandrabhushan/xv6-public/.gdbinit
line to your configuration file "/home/chandrabhushan/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/chandrabhushan/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
--Type <RET> for more, q to quit, c to continue without paging--
        info "(gdb)Auto-loading safe path"
(gdb) source .gdbinit
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(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
(gdb) 
```