CS 344: Operating Systems Lab

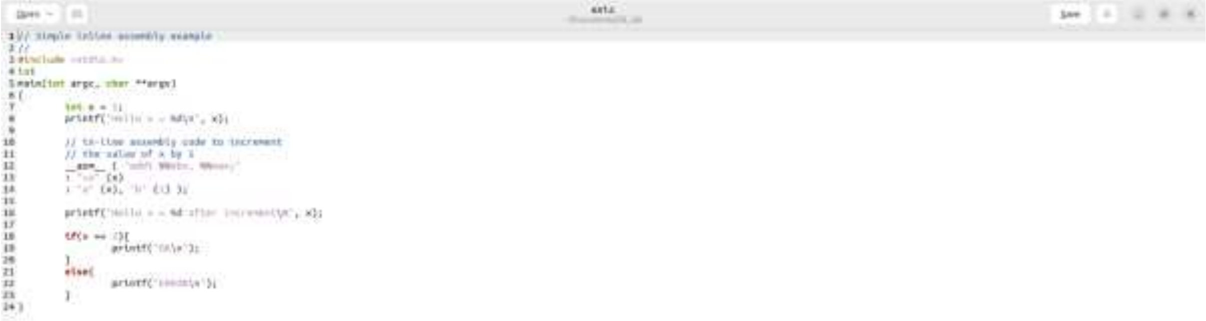
ASSIGNMENT 0A

Name: Pragyan Banerjee

Roll no.: 200123080

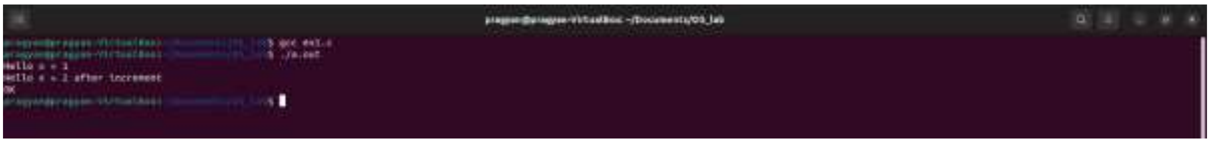
Question 1

ex1.c code



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

        // in-line assembly code to increment
        // the value of x by 1
        __asm__ ( "addl %%ebx, %%eax;"
        : "=a" (x)
        : "a" (x), "b" (1) );

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

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

Output



```
pragyan@pragyan-VirtualBox:~/Documents/OS_lab$ gcc ex1.c
pragyan@pragyan-VirtualBox:~/Documents/OS_lab$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
pragyan@pragyan-VirtualBox:~/Documents/OS_lab$ 
```

Question 2

1st instruction: [f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b

- Jump to CS = $0xf000 & IP = 0xe05b

- 0x3630 is jump to this CS (earlier in the BIOS)

- 0xf000e05b is the IP which is different from the lab because it is 32 bits rather than 16 bits and that is all the way into the top of the extended memory location but before the memory mapped PCI device location reserved by the BIOS

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

- Compare content at 0xffc8 & with content at code segment offset with value at esi.

- esi:- 32-bit source index register

3rd Instruction: [f000:e062] 0xfe062: jne 0xd241d0b0

- Jump to 0xd241d0b0 if the above comparison does not set ZF

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

- ZF was set thus jump of previous instruction doesn’t occur

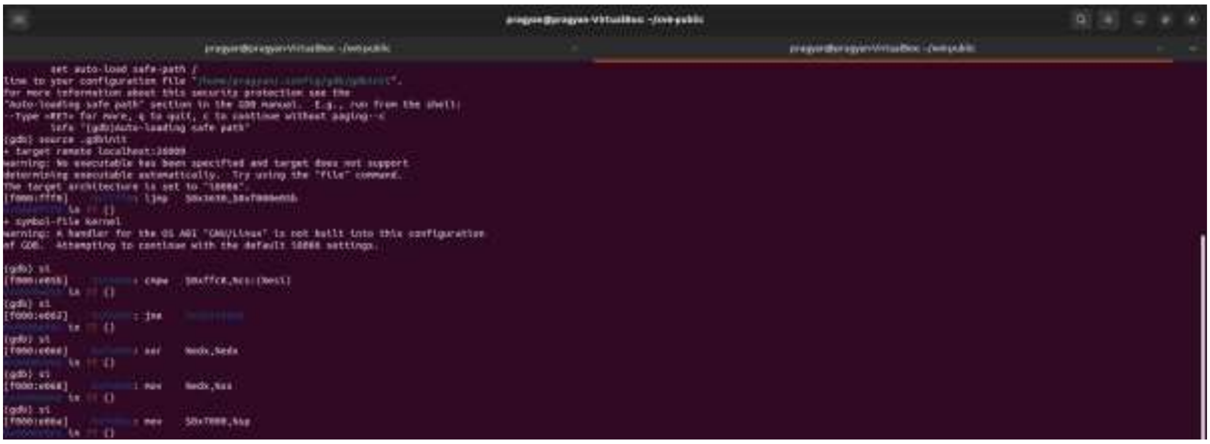
- It set edx to zero, edx is 32-bit general-purpose register.

5th instruction: [f000:e068] 0xfe068: mov %edx,%ss

- Move content of stack segment register(ss) to edx

6th instruction: [f000:e06a] 0xfe06a: mov $0x7000,%sp

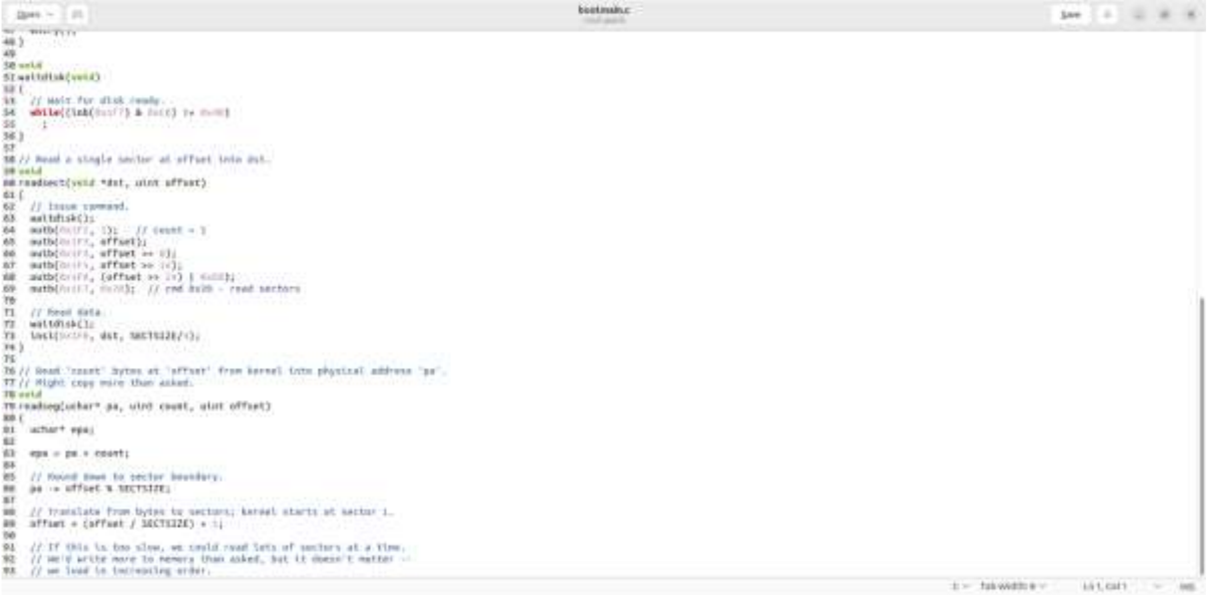
- Move content at the location pointed 16-bit stack pointer(sp) to $0x7000


pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public

```
        set auto-load safe-path /
line to your configuration file "/home/pragyan/.config/gdb/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--c
        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 set to "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    [illegible]
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 ?? ()
```

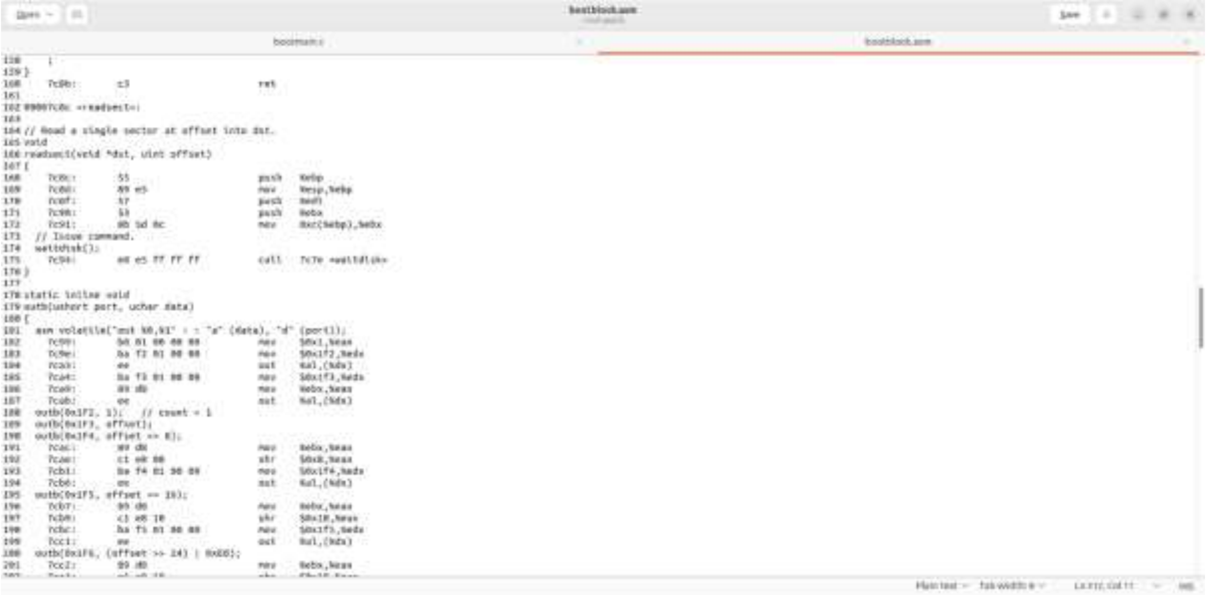
Question 3

The code for readsect() is given below

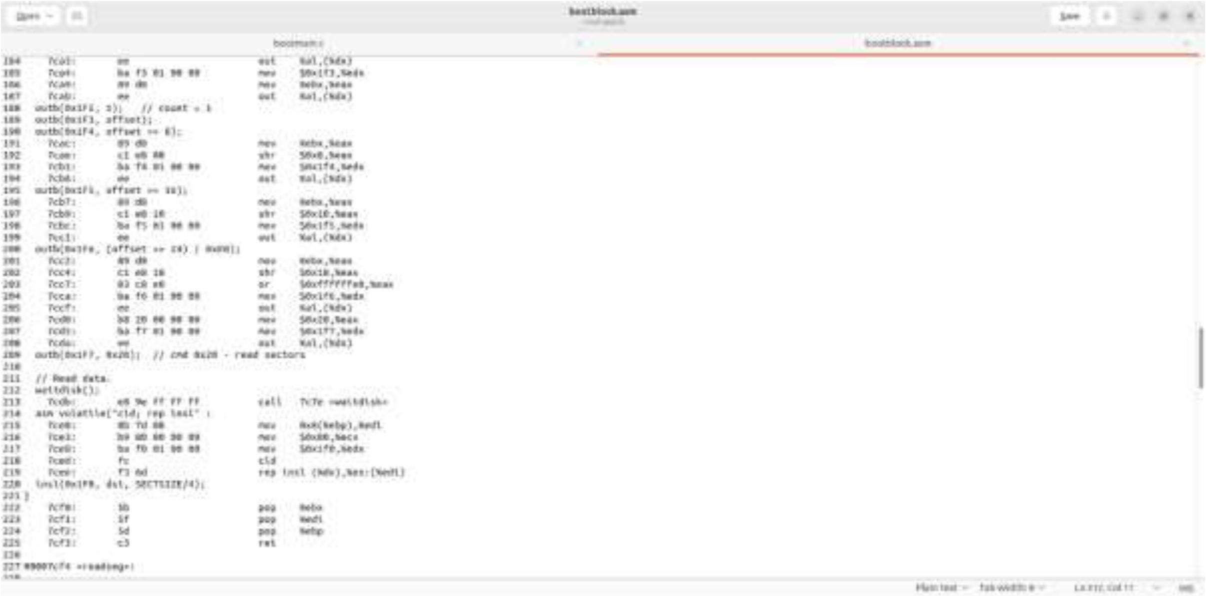


```c
}

void
waitdisk(void)
{
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40)
    ;
}

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

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

// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
  uchar* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
```

The assembly code for readsect() is given below

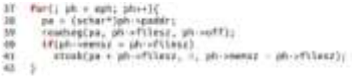


```
    }
}
    7c8b:	c3                   	ret

00007c8c <readsect>:

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
    7c8c:	55                   	push   %ebp
    7c8d:	89 e5                	mov    %esp,%ebp
    7c8f:	57                   	push   %edi
    7c90:	53                   	push   %ebx
    7c91:	8b 5d 0c             	mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c94:	e8 e5 ff ff ff       	call   7c7e <waitdisk>
}

static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
    7c99:	b8 01 00 00 00       	mov    $0x1,%eax
    7c9e:	ba f2 01 00 00       	mov    $0x1f2,%edx
    7ca3:	ee                   	out    %al,(%dx)
    7ca4:	ba f3 01 00 00       	mov    $0x1f3,%edx
    7ca9:	89 d8                	mov    %ebx,%eax
    7cab:	ee                   	out    %al,(%dx)
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cac:	89 d8                	mov    %ebx,%eax
    7cae:	c1 e8 08             	shr    $0x8,%eax
    7cb1:	ba f4 01 00 00       	mov    $0x1f4,%edx
    7cb6:	ee                   	out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cb7:	89 d8                	mov    %ebx,%eax
    7cb9:	c1 e8 10             	shr    $0x10,%eax
    7cbc:	ba f5 01 00 00       	mov    $0x1f5,%edx
    7cc1:	ee                   	out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cc2:	89 d8                	mov    %ebx,%eax
```



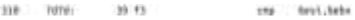
```
    7ca3:	ee                   	out    %al,(%dx)
    7ca4:	ba f3 01 00 00       	mov    $0x1f3,%edx
    7ca9:	89 d8                	mov    %ebx,%eax
    7cab:	ee                   	out    %al,(%dx)
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cac:	89 d8                	mov    %ebx,%eax
    7cae:	c1 e8 08             	shr    $0x8,%eax
    7cb1:	ba f4 01 00 00       	mov    $0x1f4,%edx
    7cb6:	ee                   	out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cb7:	89 d8                	mov    %ebx,%eax
    7cb9:	c1 e8 10             	shr    $0x10,%eax
    7cbc:	ba f5 01 00 00       	mov    $0x1f5,%edx
    7cc1:	ee                   	out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cc2:	89 d8                	mov    %ebx,%eax
    7cc4:	c1 e8 18             	shr    $0x18,%eax
    7cc7:	83 c8 e0             	or     $0xffffffe0,%eax
    7cca:	ba f6 01 00 00       	mov    $0x1f6,%edx
    7ccf:	ee                   	out    %al,(%dx)
    7cd0:	b8 20 00 00 00       	mov    $0x20,%eax
    7cd5:	ba f7 01 00 00       	mov    $0x1f7,%edx
    7cda:	ee                   	out    %al,(%dx)
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
    7cdb:	e8 9e ff ff ff       	call   7c7e <waitdisk>
  asm volatile("cld; rep insl" :
    7ce0:	8b 7d 08             	mov    0x8(%ebp),%edi
    7ce3:	b9 80 00 00 00       	mov    $0x80,%ecx
    7ce8:	ba f0 01 00 00       	mov    $0x1f0,%edx
    7ced:	fc                   	cld
    7cee:	f3 6d                	rep insl (%dx),%es:(%edi)
  insl(0x1F0, dst, SECTSIZE/4);
}
    7cf0:	5b                   	pop    %ebx
    7cf1:	5f                   	pop    %edi
    7cf2:	5d                   	pop    %ebp
    7cf3:	c3                   	ret

00007cf4 <readseg>:
```

The for loop that reads the sectors of the kernel from the disk is given below



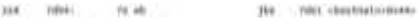
```
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }
```

The first instruction of the loop



```
    7d7d:	39 f3                	cmp    %esi,%ebx
```

The last instruction of the loop

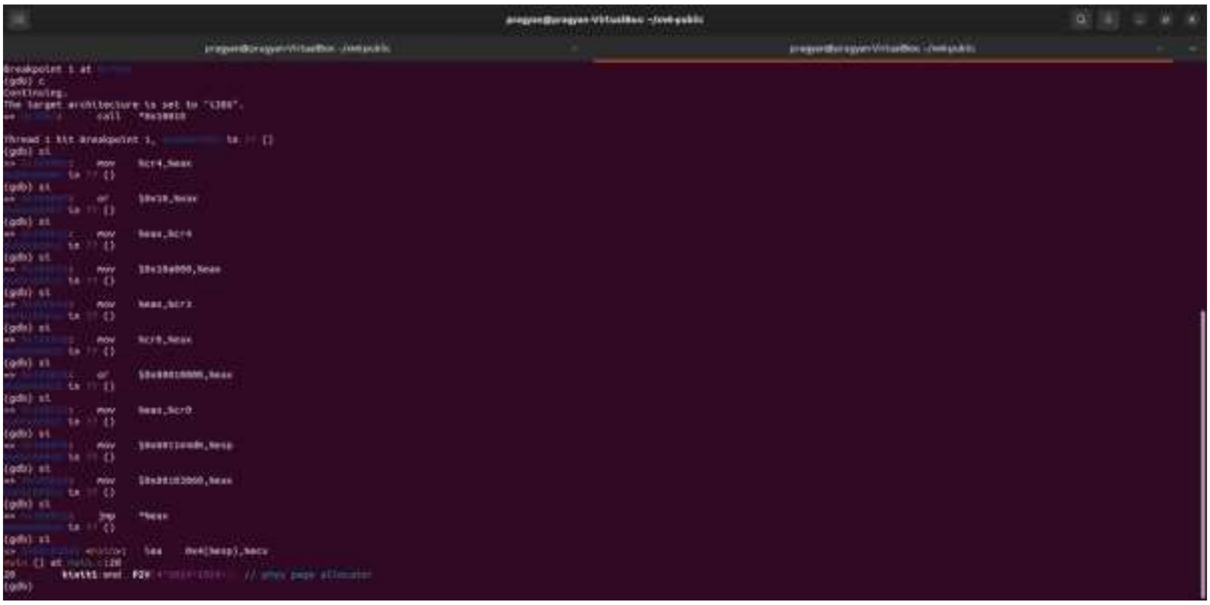


```
    7d84:	76 eb                	jbe    7d71 <bootmain+0x44>
```

The explanation for the first instruction is that the first operation on entering the for loop will be comparison between the values of ph and eph because the loop will run only when ph < eph. The explanation of last instruction is that the loop ends when the values of ph and eph become equal and hence the loop jumps to the next instruction at 0x7d91. Hence the jump instruction will be the last instruction of the for loop. The next instruction after the for loop is

```
313    7d81:    ff 15 18 00 01 00       call   *0x10018
```

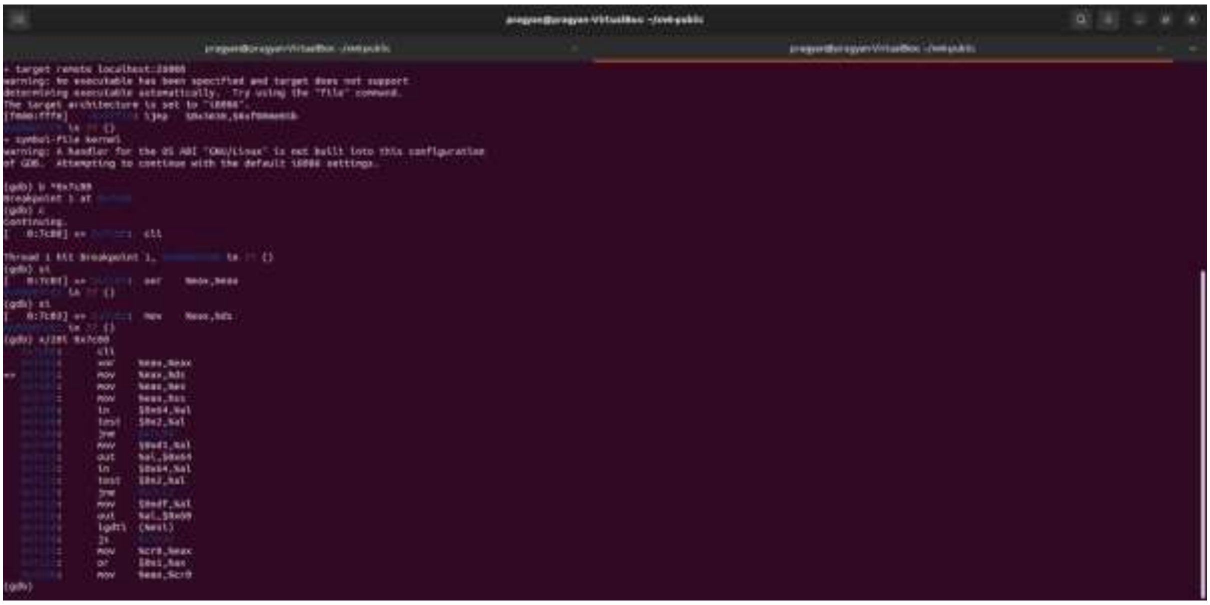
Making a breakpoint at that address and then stepping into further instructions gives the following output.


pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public

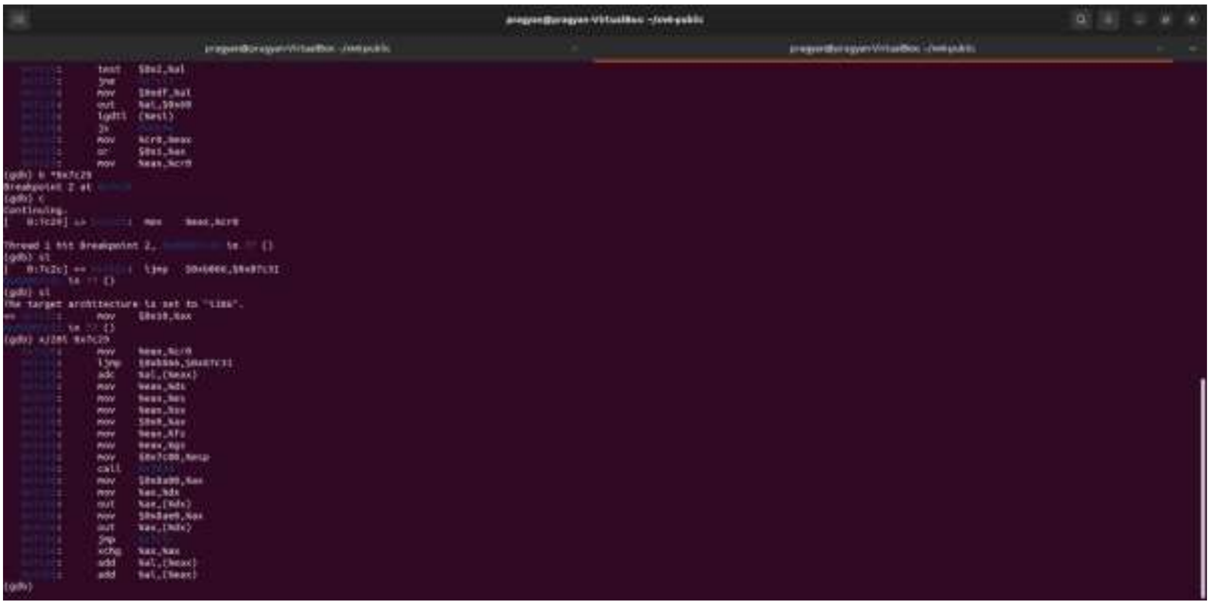
```
Breakpoint 1 at [illegible]
(gdb) c
Continuing.
The target architecture is set to "i386".
=> [illegible]:    call   *0x10018

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    %cr4,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:    or     $0x10,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    %eax,%cr4
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    $0x10a000,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    %eax,%cr3
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    %cr0,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:    or     $0x80010000,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    %eax,%cr0
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    $0x801154d0,%esp
[illegible] in ?? ()
(gdb) si
=> [illegible]:    mov    $0x80103040,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:    jmp    *%eax
[illegible] in ?? ()
(gdb) si
=> [illegible] <main>:    lea    0x4(%esp),%ecx
main () at main.c:20
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb)
```

(a)

The command line $(SEG_KCODE<<3), $start32 causes the switch from 16 to 32-bit mode in bootasm.S


pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public

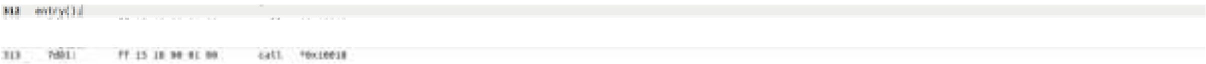
```
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is set to "i8086".
[f000:fff0]    [illegible]: ljmp   $0x3630,$0xf000e05b
[illegible] in ?? ()
+ symbol-file 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 [illegible]
(gdb) c
Continuing.
[   0:7c00] => [illegible]:  cli

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb) si
[   0:7c01] => [illegible]:  xor    %eax,%eax
[illegible] in ?? ()
(gdb) si
[   0:7c03] => [illegible]:  mov    %eax,%ds
[illegible] in ?? ()
(gdb) x/20i 0x7c00
   [illegible]:      cli
   [illegible]:      xor    %eax,%eax
=> [illegible]:      mov    %eax,%ds
   [illegible]:      mov    %eax,%es
   [illegible]:      mov    %eax,%ss
   [illegible]:      in     $0x64,%al
   [illegible]:      test   $0x2,%al
   [illegible]:      jne    [illegible]
   [illegible]:      mov    $0xd1,%al
   [illegible]:      out    %al,$0x64
   [illegible]:      in     $0x64,%al
   [illegible]:      test   $0x2,%al
   [illegible]:      jne    [illegible]
   [illegible]:      mov    $0xdf,%al
   [illegible]:      out    %al,$0x60
   [illegible]:      lgdtl  (%esi)
   [illegible]:      jl     [illegible]
   [illegible]:      mov    %cr0,%eax
   [illegible]:      or     $0x1,%ax
   [illegible]:      mov    %eax,%cr0
(gdb)
```



```
[illegible]:      test   $0x2,%al
[illegible]:      jne    [illegible]
[illegible]:      mov    $0xdf,%al
[illegible]:      out    %al,$0x60
[illegible]:      lgdtl  (%esi)
[illegible]:      js     [illegible]
[illegible]:      mov    %cr0,%eax
[illegible]:      or     $0x1,%ax
[illegible]:      mov    %eax,%cr0
(gdb) b *0x7c29
Breakpoint 2 at [illegible]
(gdb) c
Continuing.
[   0:7c29] => [illegible]:  mov    %eax,%cr0

Thread 1 hit Breakpoint 2, [illegible] in ?? ()
(gdb) si
[   0:7c2c] => [illegible]:  ljmp   $0xb866,$0x87c31
[illegible] in ?? ()
(gdb) si
The target architecture is set to "i386".
=> [illegible]:   mov    $0x10,%ax
[illegible] in ?? ()
(gdb) x/20i 0x7c29
[illegible]:      mov    %eax,%cr0
[illegible]:      ljmp   $0xb866,$0x87c31
[illegible]:      adc    %al,(%eax)
[illegible]:      mov    %eax,%ds
[illegible]:      mov    %eax,%es
[illegible]:      mov    %eax,%ss
[illegible]:      mov    $0x0,%ax
[illegible]:      mov    %eax,%fs
[illegible]:      mov    %eax,%gs
[illegible]:      mov    $0x7c00,%esp
[illegible]:      call   [illegible]
[illegible]:      mov    $0x8a00,%ax
[illegible]:      mov    %ax,%dx
[illegible]:      out    %ax,(%dx)
[illegible]:      mov    $0x8ae0,%ax
[illegible]:      out    %ax,(%dx)
[illegible]:      jmp    [illegible]
[illegible]:      xchg   %ax,%ax
[illegible]:      add    %al,(%eax)
[illegible]:      add    %al,(%eax)
(gdb)
```

(b)

By analysing the contents of bootasm.S, bootmain.c and bootblock.asm, we conclude that bootasm.S switches the OS into 32-bit mode and then calls bootmain.c which first loads the kernel using ELF header and the enters the kernel using entry(). Hence the last instruction of bootloader is entry(). Looking for the same in bootblock.asm, we find out the instruction to be



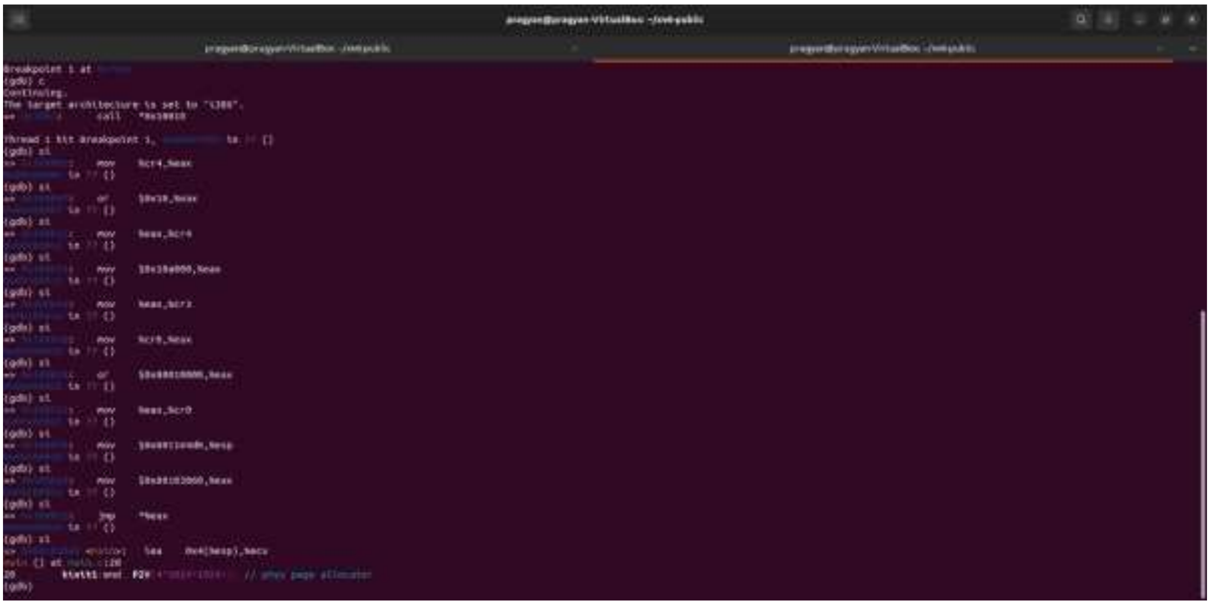
```
  entry();
    7d81:       ff 15 18 00 01 00       call   *0x10018
```

which is a call instruction which shifts control to the address stored at 0x10018 since dereferencing operator (*) has been used. Now we need to know the starting address of the kernel. We can find this by two methods:

(i) By looking at the first word of memory stored at 0x10018 (by using the command "x/1x 0x10018")
(ii) By looking at the contents of "objdump -f kernel"

After getting the starting address of kernel, we need to see what is the instruction stored at that address to get the first instruction of kernel. We can do this by two methods:

(i) By using "x/1i 0x0010000c"
(ii) By looking into kernel.asm

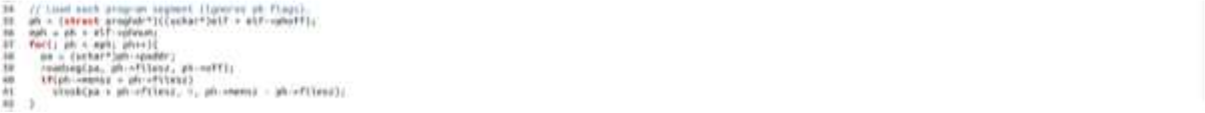


```
Breakpoint 1 at [illegible]
(gdb) c
Continuing.
The target architecture is set to "i386".
=> [illegible]:     call   *0x10018

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    %cr4,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:     or     $0x10,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    %eax,%cr4
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    $0x10a000,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    %eax,%cr3
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    %cr0,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:     or     $0x80010000,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    %eax,%cr0
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    $0x8010c5d0,%esp
[illegible] in ?? ()
(gdb) si
=> [illegible]:     mov    $0x80103060,%eax
[illegible] in ?? ()
(gdb) si
=> [illegible]:     jmp    *%eax
[illegible] in ?? ()
(gdb) si
=> [illegible] <main>:    lea    0x4(%esp),%ecx
main () at main.c:20
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb)
```

Hence, the first instruction of kernel is

0x10000c: mov   %cr4,%eax

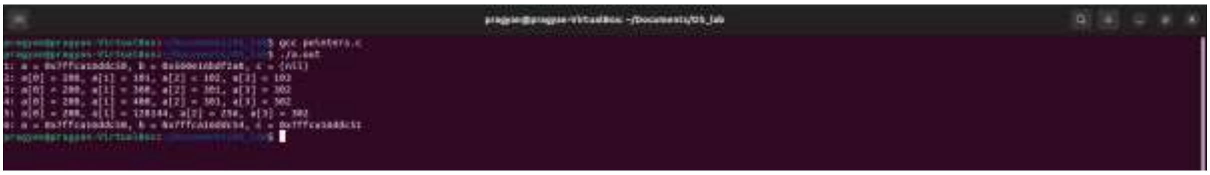
(c)



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

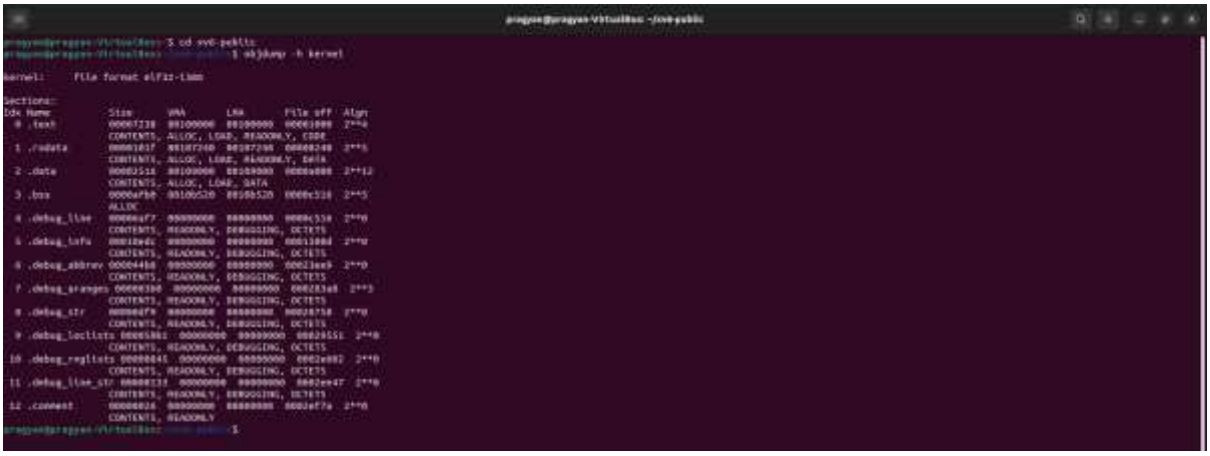
Question 4

pointers.c



```
pragyan@pragyan-VirtualBox:~/Documents/OS_lab$ gcc pointers.c
pragyan@pragyan-VirtualBox:~/Documents/OS_lab$ ./a.out
1: a = 0x7ffca1addc10, b = 0x5600e1dbf2a0, c = (nil)
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 = 0x7ffca1addc10, b = 0x7ffca1addc14, c = 0x7ffca1addc11
pragyan@pragyan-VirtualBox:~/Documents/OS_lab$
```

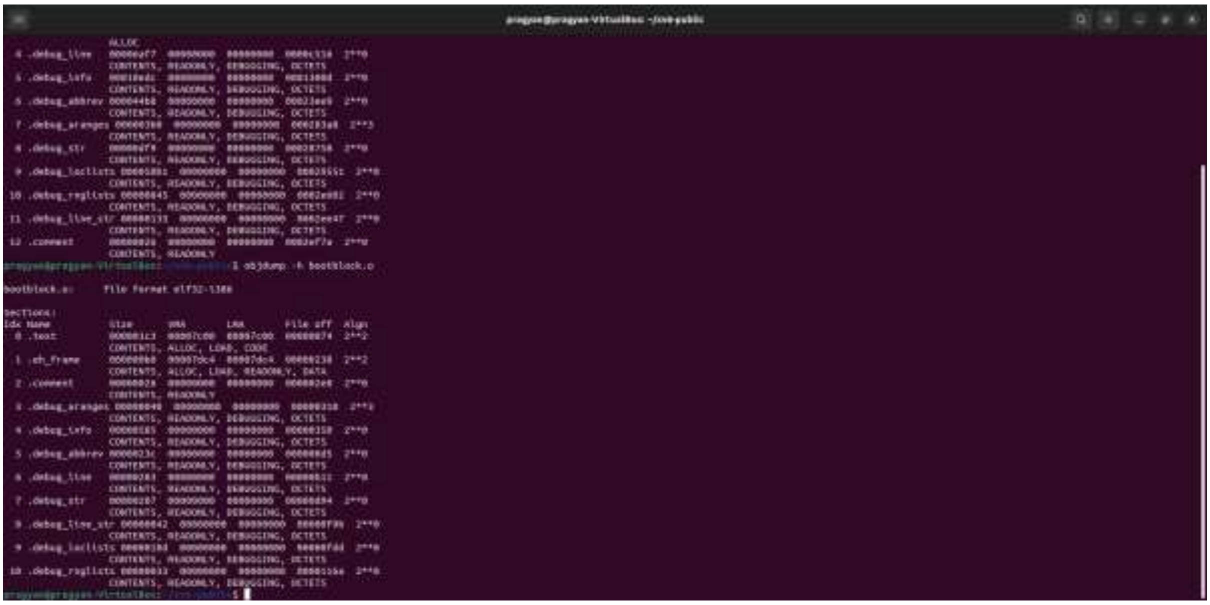
objdump -h kernel



```
pragyan@pragyan-VirtualBox:~$ cd xv6-public
pragyan@pragyan-VirtualBox:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00007238  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000101f  80107240  00107240  00008240  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80109000  00109000  0000a000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000afb0  8010b520  0010b520  0000c516  2**5
                  ALLOC
  4 .debug_line   00006af7  00000000  00000000  0000c516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00010edc  00000000  00000000  0001300d  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 000044b8  00000000  00000000  00023ee9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000300  00000000  00000000  000283a8  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000df9  00000000  00000000  000286a8  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 00005861  00000000  00000000  000295a1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002ee02  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 00000133  00000000  00000000  0002ee47  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      00000026  00000000  00000000  0002ef7a  2**0
                  CONTENTS, READONLY
pragyan@pragyan-VirtualBox:~/xv6-public$
```

As we can see in the above screenshot, VMA and LMA of .text section is different indicating that it loads and executes from different addresses.

objdump -h bootblock.o

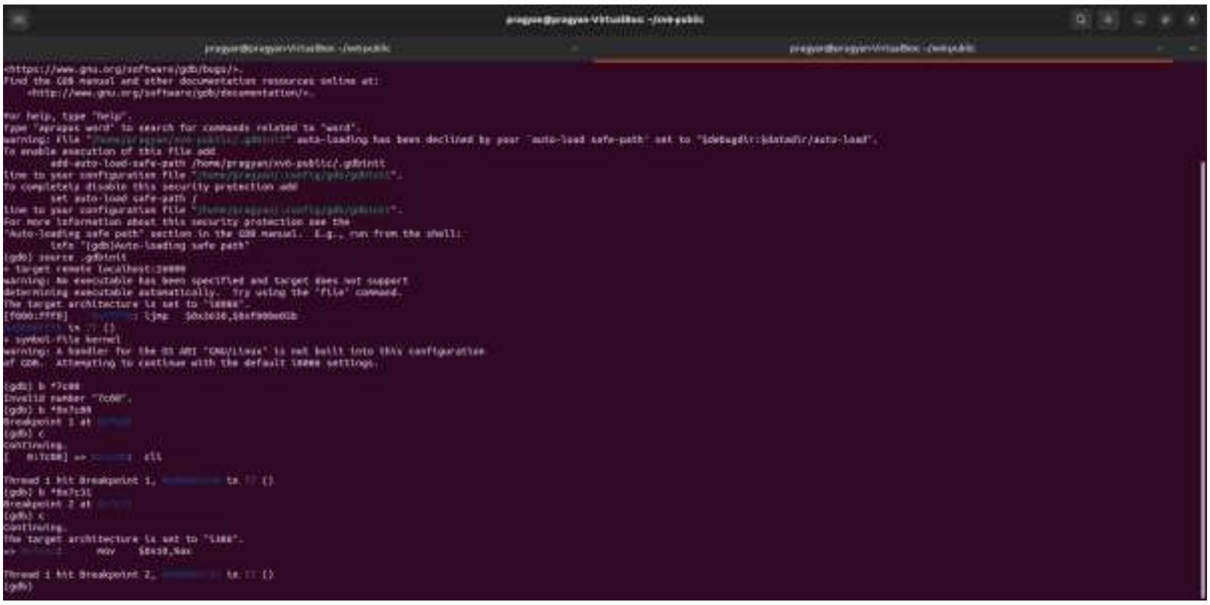


```
                  ALLOC
  4 .debug_line   0000eaf7  00000000  00000000  0000c538  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   0001dedc  00000000  00000000  0001b088  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 000044b8  00000000  00000000  00023ee9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000388  00000000  00000000  000283a8  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000df8  00000000  00000000  00028758  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 00001881  00000000  00000000  00029551  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002ae81  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 00000133  00000000  00000000  0002ee47  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      00000026  00000000  00000000  0002ef7a  2**0
                  CONTENTS, READONLY
pragyan@pragyan-VirtualBox:~/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dc4  00007dc4  00000238  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000026  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000318  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000358  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008dd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b11  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000207  00000000  00000000  00000d94  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 00000042  00000000  00000000  00000f9b  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fdd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  0000116a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
pragyan@pragyan-VirtualBox:~/xv6-public$ 
```

As we can see in the above screenshot, VMA and LMA of .text section is same indicating that it loads and executes from the same address.

Question 5

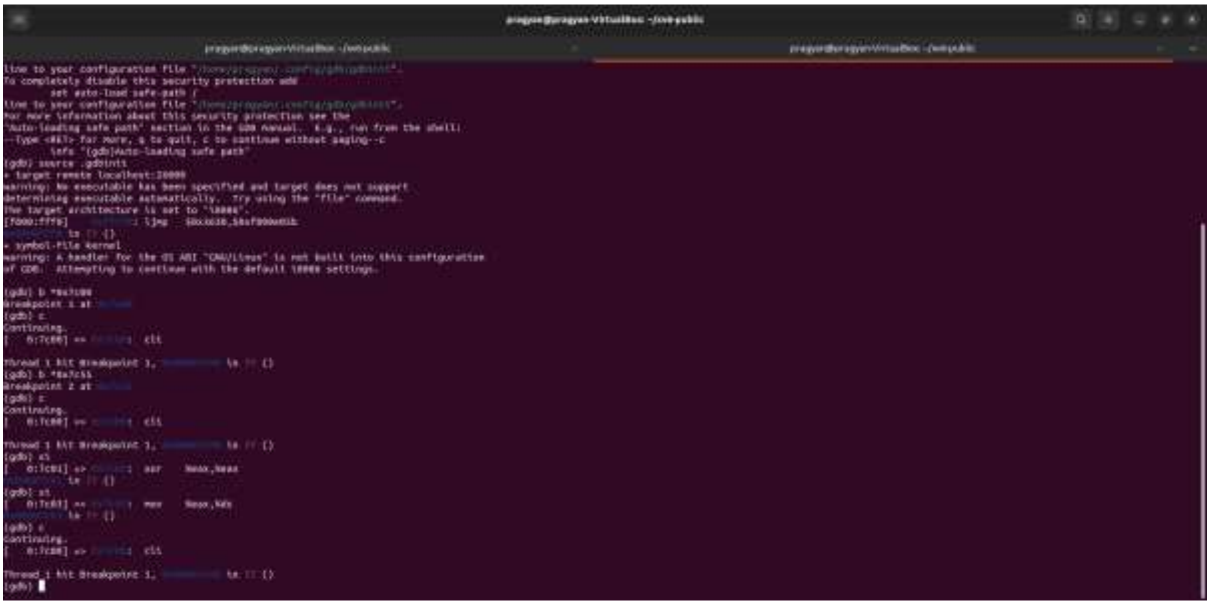
When boot loader's link address is 0x7C00 then commands are running properly and transition from 16 to 32 bit was occurring at 0x7C31 address location as seen below:



pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public
pragyan@pragyan-VirtualBox: ~/xv6-public

```
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: File "/home/pragyan/xv6-public/.gdbinit" auto-loading has been declined 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/pragyan/xv6-public/.gdbinit
line to your configuration file "/home/pragyan/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/pragyan/.config/gdb/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:
        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 set to "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 *7c00
Invalid number "7c00".
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) b *0x7c31
Breakpoint 2 at 0x7c31
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax

Thread 1 hit Breakpoint 2, 0x00007c31 in ?? ()
(gdb)
```

But when the boot loader's link address is changed to any other address (we took 0x7C24 in this case), after running make clean make and restarting gdb and continuing from address location 0x7C00, then the boot loader is restarting again and again after running some instructions in the gdb.

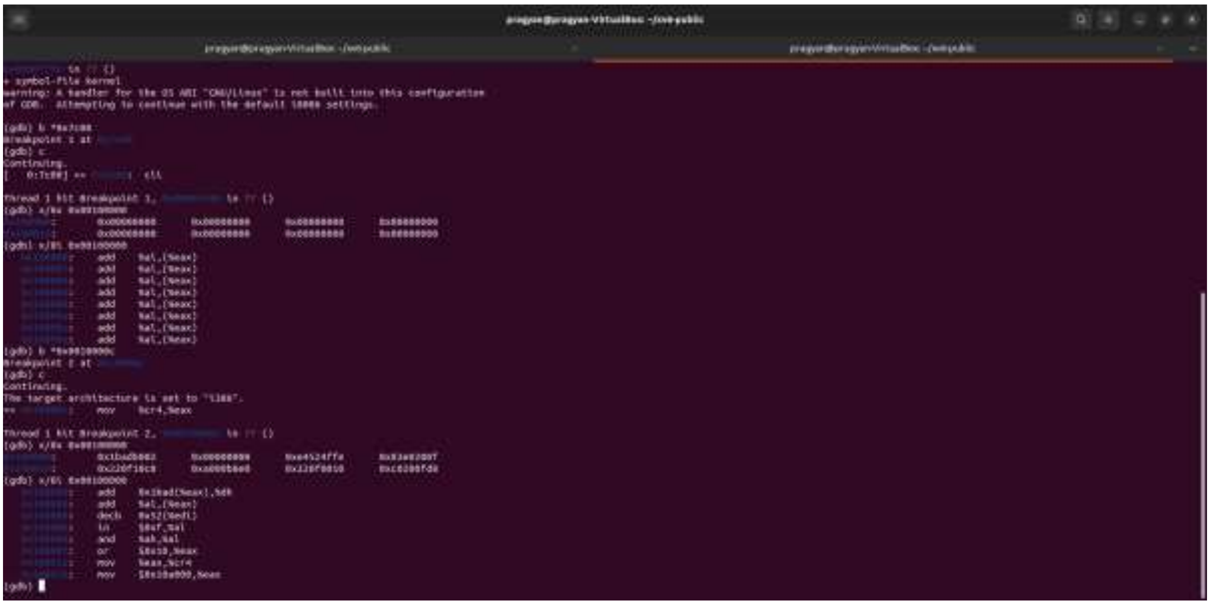


```
line to your configuration file "[illegible]".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "[illegible]".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
--Type <RET> for more, q to quit, c to continue without paging--c
        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 set to "i8086".
[f000:fff0]    [illegible]: ljmp   $0x3630,$0xf000e05b
[illegible] 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 [illegible]
(gdb) c
Continuing.
[   0:7c00] => [illegible]:  cli

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb) b *0x7c55
Breakpoint 2 at [illegible]
(gdb) c
Continuing.
[   0:7c00] => [illegible]:  cli

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb) si
[   0:7c01] => [illegible]:  xor    %eax,%eax
[illegible] in ?? ()
(gdb) si
[   0:7c03] => [illegible]:  mov    %eax,%ds
[illegible] in ?? ()
(gdb) c
Continuing.
[   0:7c00] => [illegible]:  cli

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb)
```

As seen in the image above, we tried to run commands after continuing from breakpoint at 0x7C00 address location and we always end up hitting the same breakpoint at 0x7C00. Also 16 to 32 bit architecture change didn't occurred as breakpoint b *0x7C55 is not hit which should be responsible for architecture change in this case.

Question 6

For this experiment, we have to examine the 8 words of memory at 0x00100000 at two different instances of time, the first when the BIOS enters boot loader and the second when the boot loader enters the kernel. For this, we will use the command "x/8x 0x00100000" but before that we will have to set our breakpoints. The first breakpoint will be at 0x7c00 because this is the point where the BIOS hands control over to the boot loader. The second breakpoint will be at 0x0010000c because this is the point when the kernel is passed control by the bootloader.



```
[illegible] 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 [illegible]
(gdb) c
Continuing.
[   0:7c00] => [illegible]:  cli

Thread 1 hit Breakpoint 1, [illegible] in ?? ()
(gdb) x/8x 0x00100000
[illegible]:     0x00000000      0x00000000      0x00000000      0x00000000
[illegible]:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/8i 0x00100000
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
   [illegible]:     add    %al,(%eax)
(gdb) b *0x0010000c
Breakpoint 2 at [illegible]
(gdb) c
Continuing.
The target architecture is set to "i386".
=> [illegible]:     mov    %cr4,%eax

Thread 1 hit Breakpoint 2, [illegible] in ?? ()
(gdb) x/8x 0x00100000
[illegible]:     0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
[illegible]:     0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8i 0x00100000
   [illegible]:     add    0x1bad(%eax),%dh
   [illegible]:     add    %al,(%eax)
   [illegible]:     decb   0x52(%edi)
   [illegible]:     in     $0xf,%al
   [illegible]:     and    %ah,%al
   [illegible]:     or     $0x10,%eax
   [illegible]:     mov    %eax,%cr4
   [illegible]:     mov    $0x10a000,%eax
(gdb)
```

As we can see in the diagram, we get different values at both the breakpoints. The explanation to this is as follows. The address 0x00100000 is actually 1MB which is the address from where the kernel is loaded into the memory. Before the kernel is loaded into the memory, this address contains no data (i.e. garbage value). By default, all the uninitialized values are set to 0 in xv6. Hence, when we tried to read the 8 words of memory at 0x00100000 at the first breakpoint, we got all zeroes since no data had been loaded until that point. When we check the values at the second breakpoint, the kernel has already been loaded into the memory and thus this address now contains meaningful data instead of zeroes.