

Assignment 0A

OS344 - Operating Systems Laboratory

Pranshu Pandya

Assignment 0A:-

Exercise 1: Becoming familiar with inline assembly by writing a simple program

The screenshot shows a Visual Studio Code window with a C file named ex1.c. The code includes a main function that prints 'Hello x = 1' and then uses inline assembly ('asm') to increment the value of x by 1. It then prints the result and checks if x is 2. If x is 2, it prints 'OK', otherwise, it prints 'ERROR'. The right side of the screen shows a 'CPH JUDGE: RESULTS' panel for a testcase named 'ex1'. The status is '1 Failed 0ms'. The received output is 'Hello x = 1' followed by 'Hello x = 2 after increment OK'. Below the panel are buttons for 'Run All', 'Stop', 'Help', and 'Delete'.

```
File Edit Selection View Go Run Terminal Help
C ex1.c
C ex1.c > main(int, char **)
1 // Simple inline assembly example
2 //
3 #include <stdio.h>
4 int main(int argc, char **argv)
5 {
6     int x = 1;
7     printf("Hello x = %d\n", x);
8
9     //
10    // Put in-line assembly here to increment
11    // the value of x by 1 using in-line assembly
12    //
13    asm("inc %%eax":=r"(x):"r" (x));
14    printf("Hello x = %d after increment\n", x);
15    if (x == 2)
16    {
17        printf("OK\n");
18    }
19    else
20    {
21        printf("ERROR\n");
22    }
23 }
```

The above image shows the code with the output as below.

The screenshot shows a terminal window with the following session:

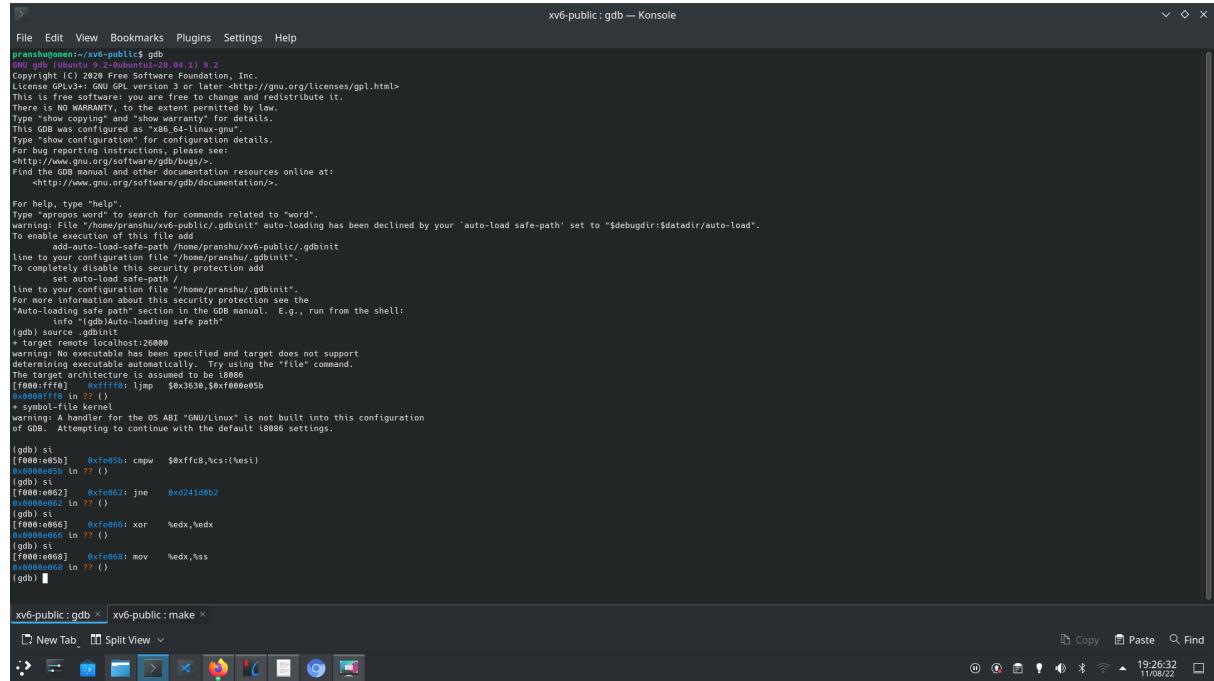
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
pranshu@openen:~/Desktop/code/oslabs$ gcc ex1.c
pranshu@openen:~/Desktop/code/oslabs$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
pranshu@openen:~/Desktop/code/oslabs$
```

The code I have added:

```
asm("inc %%eax":=r"(x):"r" (x));
```

In this x is both input as well as output operand. The code increases the value of x by 1.

Exercise 2: Tracing into the ROM BIOS



```
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This program was compiled as "x86_64-Linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<http://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: No executable file specified and target does not support
dynamically-loading symbols automatically. Try using the "file" command.
The target architecture is assumed to be i86pc
[fe00:ffff]: ljmp $0x3630,$0xf000e05b
[0x0000e05b]: ??()
symbol-table-load
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i86pc settings.

(gdb) si
[fe00:ffff]: 0xfe05b: cmpw $0xfc8,%cs:(%esi)
[0x0000e05b] in ??()
(gdb) si
[fe00:ffff]: 0xfe062: jne 0xd241d0b2
[0x0000e062] in ??()
(gdb) si
[fe00:ffff]: 0xfe066: xor %edx,%edx
[0x0000e066] in ??()
(gdb) si
[fe00:ffff]: 0xfe068: mov %edx,%ss
[0x0000e068] in ??()
(gdb) l
```

The `si` instruction in `gdb` is used to step through a single x86 instruction. Steps into calls. The above screenshot shows the first 5 instructions of the xv6 operating system. Following is the instruction and their explanation.

1. [fe00: fffe] 0xfffff0: ljmp \$0x3630, \$0xf000e05b
Here,
The Starting Code Segment: f000
The Starting Instruction Pointer: fff0
The physical address of the instruction: 0xfffff0
The destination code segment: 0x3630
The destination instructor pointer: 0xf000e05b
While `ljmp` is the instruction.
2. The `cmp` instruction is used to perform comparison. It sets the Zero flag to one if both the operands are equal.
3. The `jnz` (or `jne`) instruction is a conditional jump that follows a test. It jumps to the specified location if the Zero Flag (ZF) is unset.
4. The `xor` instruction performs a logical XOR (exclusive OR) operation. This is the equivalent to the "`^`" operator in C++.

5. The mov instruction is used to move the value of edx to ss.

Part 2: The Boot Loader

The screenshot shows a dual-pane debugger interface. The left pane is a terminal window titled "xv6-public:gdb — Konsole" displaying GDB session logs. The right pane is a code editor titled "bootblock.asm - xv6-public - Visual Studio Code" showing the assembly code for the bootblock. The assembly code includes instructions like cli, xorw, movw, and movb, along with comments explaining their purpose. A vertical scrollbar is visible between the two panes.

```

xv6-public:gdb — Konsole
File Edit View Bookmarks Plugins Settings Help
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 i86pc
[1000:ffff] >0xffffffff: ljmp $0x360, $0xf000e05b
0x0000ffff 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 i86pc settings.

(gdb) si
[1000:e05b] 0xfe05b: cmpw $0xffff,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[1000:e062] 0xfe062: jne 0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[1000:e066] 0xfe066: xor %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[1000:e068] 0xfe068: mov %edx,%ss
0x0000e068 in ?? ()
(gdb) si
Breakpoint 1 at 0x7c00
(gdb) continue
Continuing.
[ 0:7c00] => 0x7c00: cli
Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/10i
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 0x7c0f
0x7c0f: mov $0xd1,%al
0x7c11: out %al,$0x64
0x7c13: in $0x64,%al
(gdb) si
[ 0:7c01] => 0x7c01: xor %eax,%eax
0x00007c01 in ?? ()
(gdb) 
xv6-public:gdb x New Tab Split View < Copy < Paste < Find < 
File Edit Selection View Go Run Terminal Help
bootmain.c bootasm.S bootblock.asm
bootblock.asm
13 clri                                # BIOS enabled interrupts;
14 7c00: fa
15
16 # Zero data segment registers DS, ES, and SS.
17 xorw %ax,%ax                         # Set %ax to zero
18 7c01: 31 c0
19 movw %ax,%ds                          # -> Data Segment
20 7c03: 8e d8
21 movw %ax,%es                          # -> Extra Segment
22 7c05: 8e c0
23 movw %ax,%ss                          # -> Stack Segment
24 7c07: 8e d0
25
26 00007c09 <seta20.1>
27
28 # Physical address line A20 is tied to zero so that the
29 # with 2 MB would run software that assumed 1 MB. Undo
30 seta20.1:
31 inb $0x64,%al                         # Wait for not busy
32 7c09: e4 64
33 testb $0x2,%al
34 7c0b: a8 02
35 jnz seta20.1
36 7c0d: 75 fa
37 jne 7c09 <seta20.1>
38 movb $0xd1,%al                         # $0xd1 -> port 0x64
39 7c0f: b0 d1
40 outb %al,$0x64
41 7c11: e6 64
42
43 00007c13 <seta20.2>

```

Here I have set breakpoint at address 0x7c00 using b *0x7c00. Then I have used the continue statement to continue execution till the next breakpoint which is 0x7c00. In the image you can see the right hand side window where I have shown the bootblock.asm where you can see the instruction at 7c00. I have also examined the next 10 instructions using the x/10i command.

bootmain.c

```

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(0xF0, dst, SECTSIZE/4);
74 }
75
76 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
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 }
```

bootblock.asm

```

167 // Read a single sector at offset into dst.
168 void
169 readsect(void *dst, uint offset)
170 {
171     7c90: f3 0f 1e fb    endbr32
172     7c94: 55             push %ebp
173     7c95: 89 e5           mov %esp,%ebp
174     7c97: 57             push %edi
175     7c98: 53             push %ebx
176     7c99: 8b 5d 0c         mov 0xc(%ebp),%ebx
177     // Issue command.
178     7c9a: e8 dd ff ff ff   call 7c7e <waitdisk>
179
180 }
181
182 static inline void
183 outb(ushort port, uchar data)
184 {
185     asm volatile("out %0,%1" : : "a" (data), "d" (port));
186     7ca0: b0 01 00 00         mov $0x1, %eax
187     7ca1: ba f2 01 00 00         mov $0x1f5, %edx
188     7ca2: ee               out %al, (%dx)
189     7ca3: ba f3 01 00 00         mov $0x1f6, %edx
190     7ca4: 89 d8             mov %ebx, %eax
191     7ca5: ee               out %al, (%dx)
192     outb(0x1F2, 1); ← // count = 1
193     outb(0x1F3, offset); ←
194     outb(0x1F4, offset > 8); ←
195     7cb4: 89 d8             mov %ebx, %eax
196     7cb5: c1 e8 08           shr $0x8, %eax
197     7cb6: ba f4 01 00 00         mov $0x1f4, %edx
198     7cb7: ee               out %al, (%dx)
199     outb(0x1F5, offset > 16); ←
200     7cb8: 89 d8             mov %ebx, %eax
201     7cc1: c1 e8 10           shr $0x10, %eax
202     7cc2: ba f5 01 00 00         mov $0x1f5, %edx
203     7cc3: ee               out %al, (%dx)
204     outb(0x1F6, (offset >> 24) | 0xE0); ←
205     7cc4: 89 d8             mov %ebx, %eax
206     7cc5: c1 e8 18           shr $0x18, %eax
207     7cc6: 83 c8 e0           or $0xfffffe0, %eax
208     7cd2: ba f6 01 00 00         mov $0x1f6, %edx
209     7cd7: ee               out %al, (%dx)
210     7cd8: b0 20 00 00 00         mov $0x20, %eax
211     7cd9: ba f7 01 00 00         mov $0x1f7, %edx
212     7ce2: ee               out %al, (%dx)
213     outb(0x1F7, 0x20); // cmd 0x20 - read sectors ←
214
215     // Read data.
216     7ce3: e8 96 ff ff         call 7c7e <waitdisk>
217
218     asm volatile("clld; rep insl :"
219     7ce8: 8b 7d 08           mov 0x8(%ebp), %edi
220     7ceb: b9 00 00 00 00         mov $0x80, %ecx
221     7cf0: ba f0 01 00 00         mov $0x1f0, %edx
222     7cf1: fc               clld
223     7ff6: f3 6d             rep insl (%dx), %es:(%edi)
224     insl(0xF0, dst, SECTSIZE/4); ←
225
226     7cf8: 5b               pop %ebx
227     7cf9: 5f               pop %edi
228     7cf0: 5d               pop %ebp
229     7cf1: c3               ret
230
231     00007fcf <readseg>;
232
233 // Read 'count' bytes at 'offset' from kernel into phycial address 'pa'
```

bootmain.c

```

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(0xF0, dst, SECTSIZE/4); ←
74 }
75
76 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
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 }
```

bootblock.asm

```

195 7c4: 89 d8             mov %ebx, %eax
196 7c5: c1 e8 08           shr $0x8, %eax
197 7c6: ba f4 01 00 00         mov $0x1f4, %edx
198 7c7: ee               out %al, (%dx)
199     outb(0x1F5, offset > 16); ←
200 7cb: 89 d8             mov %ebx, %eax
201 7cc1: c1 e8 10           shr $0x10, %eax
202 7cc2: ba f5 01 00 00         mov $0x1f5, %edx
203 7cc3: ee               out %al, (%dx)
204     outb(0x1F6, (offset >> 24) | 0xE0); ←
205 7cc4: 89 d8             mov %ebx, %eax
206 7cc5: c1 e8 18           shr $0x18, %eax
207 7cc6: 83 c8 e0           or $0xfffffe0, %eax
208 7cd2: ba f6 01 00 00         mov $0x1f6, %edx
209 7cd7: ee               out %al, (%dx)
210 7cd8: b0 20 00 00 00         mov $0x20, %eax
211 7cd9: ba f7 01 00 00         mov $0x1f7, %edx
212 7ce2: ee               out %al, (%dx)
213     outb(0x1F7, 0x20); // cmd 0x20 - read sectors ←
214
215     // Read data.
216     7ce3: e8 96 ff ff         call 7c7e <waitdisk>
217
218     asm volatile("clld; rep insl :"
219     7ce8: 8b 7d 08           mov 0x8(%ebp), %edi
220     7ceb: b9 00 00 00 00         mov $0x80, %ecx
221     7cf0: ba f0 01 00 00         mov $0x1f0, %edx
222     7cf1: fc               clld
223     7ff6: f3 6d             rep insl (%dx), %es:(%edi)
224     insl(0xF0, dst, SECTSIZE/4); ←
225
226     7cf8: 5b               pop %ebx
227     7cf9: 5f               pop %edi
228     7cf0: 5d               pop %ebp
229     7cf1: c3               ret
230
231     00007fcf <readseg>;
232
233 // Read 'count' bytes at 'offset' from kernel into phycial address 'pa'
```

In the above image I have traced through the readsect function in bootmain.c and pointed out the exact assembly instructions corresponding to each of the statements.

```

File Edit Selection View Go Run Terminal Help
C bootmain.c x C:\x86.h bootasm.S
C bootmain.c > readsect(void *, uint)
17 void
18 bootmain(void)
19 {
20     struct elfhdr *elf;
21     struct proghdr *ph, *eph;
22     void (*entry)(void);
23     uchar* pa;
24
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             stobs(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)

```

bootblock.asm

```

308     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
309     7d7e: 8d 98 00 00 01 00    mov    0x1000(%eax),%eax
310     7d7f: a1 1c 00 01 00    lea    0x1000(%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    %esi-0x5d<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     7da0: 39 de    cmp    %ebx,%esi
330     7da1: 76 eb    jbe    7d91 <bootmain+0x48>
331     pa = (uchar*)ph->paddr;
332     7da2: 8b 7b 8c    mov    0xc(%ebx),%edi
333     readseg(pa, ph->filesz, ph->off);
334     7da3: 83 ec 04    sub    $0x4,%esp
335     7da4: ff 73 04    pushl  0x4(%ebx)
336     7da5: ff 73 10    pushl  0x10(%ebx)
337     7da6: 57    push    %edi
338     7da7: e9 44 ff ff ff    call   7fcf <readseg>
339     if(ph->memsz > ph->filesz)
340     7da8: 8b 4b 14    mov    0x14(%ebx),%ecx
341     7da9: 8b 43 10    mov    0x10(%ebx),%eax
342     7daa: 83 c4 10    add    $0x10,%esp
343     7da1: 39 c1    cmp    %eax,%ecx
344     7da2: 76 da    jbe    7d9f <bootmain+0x56>
345     stobs(pa + ph->filesz, 0, ph->memsz - ph->filesz);
346

```

Ln 321, Col 17 Spaces:2 UTF-8 LF nasm x86 assembly Go Live

Tracing through the `readsect` and denoting the beginning and end of the for loop. It is marked in the image on the right side. The for loop is starting at address 0x7d8d and ending at address 0x7da4.

The first instruction of the for loop:

7d8d: 39 f3 cmp %esi,%ebx

The Last instruction executed by the for loop:

7da4: 76 eb jbe 7d91 <bootmain+0x48>

The first instruction is a comparison between the values of `ph` and `eph` and that is exactly what we should do while entering the loop as the loop will run only when `ph < eph`.

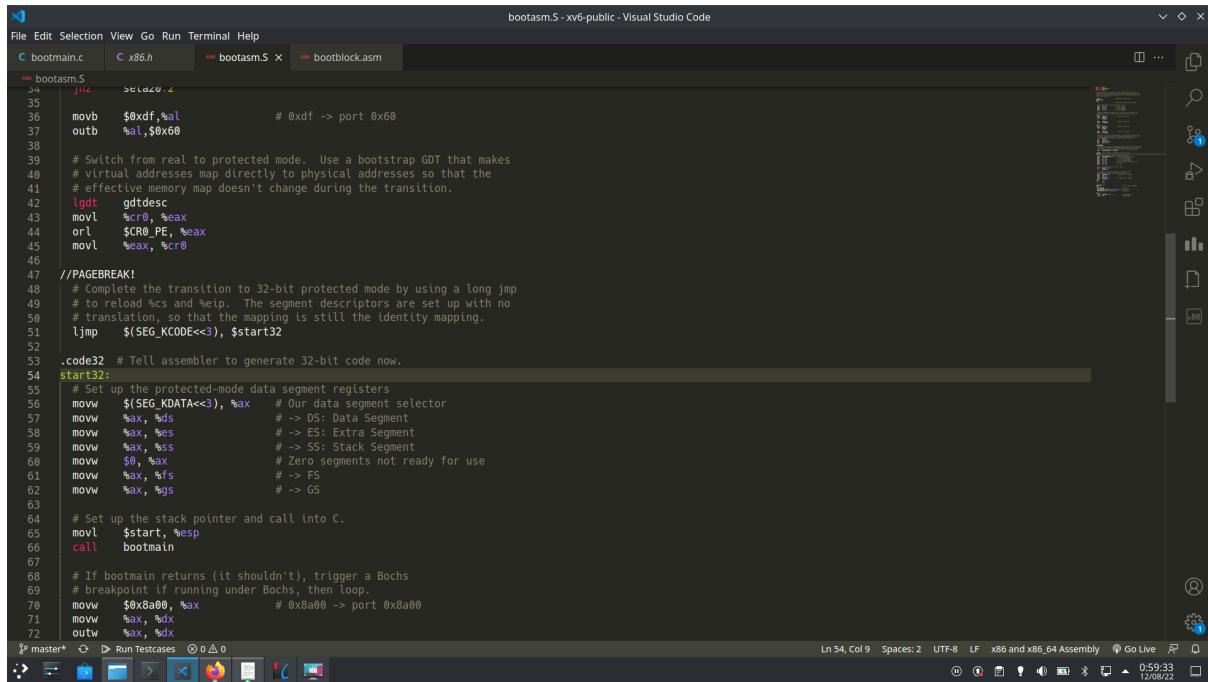
In the last instruction, we know that the loop ends when `ph` and `eph` attain equal values and thus the loop jumps to the next instruction at 0x7d91.

The screenshot shows a terminal window titled "xv6-public : gdb — Konsole". The window contains a GDB session transcript. The user has set a breakpoint at address 0x7d91 and stepped through the assembly code, which includes instructions involving registers %cr4, %eax, %cr3, %cr0, and %esp, along with memory locations like 0x0000000c, 0x0000000f, and 0x00000012. The session ends with the command "endbr32". The terminal also shows the system tray with icons for battery, signal, volume, and time (04:35, 12/08/22).

```
xv6-public : gdb — Konsole
File Edit View Bookmarks Plugins Settings Help
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) continue
Continuing.
The target architecture is assumed to be i386
=> 0x7d91: call *0x10018
Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x00010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x00010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x000100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x109000,%eax
0x000100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x00010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x00010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010000,%eax
0x000100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x000100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0x8010b5c0,%esp
0x000100028 in ?? ()
(gdb) si
=> 0x10002d:    mov    $0x80103040,%eax
0x00010002d in ?? ()
(gdb) si
=> 0x100032:    jmp   *%eax
0x000100032 in ?? ()
(gdb) si
=> 0xb0103040 <main>: endbr32
main () at main.c:19
19
(gdb) ■
```

In the above image I have set breakpoint at the last instruction of the for loop and then stepped into further instructions.

Exercise 3)



The screenshot shows the Visual Studio Code interface with the file 'bootasm.S' open. The code is written in x86 assembly and includes comments explaining the transition from real mode to protected mode. Key instructions include 'movb \$0xd0,%al' (port 0x60), 'lgdt gdtesc', and 'ljmp \$(SEG_KCODE<<3), \$start32'. The assembly code is preceded by '# PAGEBREAK!' and followed by '# Tell assembler to generate 32-bit code now.' The code block ends with a 'start32:' label and a series of 'movw' instructions setting up segment registers. The final part of the code includes a call to 'bootmain' and a 'jmp' instruction.

```
bootasm.S - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C bootmain.c C x86.h bootasm.S bootblock.asm
bootasm.S
34     j14  SetZero
35
36     movb $0xd0,%al          # 0xd0 -> port 0x60
37     outb %al,$0x60
38
39     # Switch from real to protected mode. Use a bootstrap GDT that makes
40     # virtual addresses map directly to physical addresses so that the
41     # effective memory map doesn't change during the transition.
42     lgdt gdtesc
43     movl %cr0,%eax
44     orl $CR0_PE,%eax
45     movl %eax,%cr0
46
47 //PAGEBREAK!
48     # Complete the transition to 32-bit protected mode by using a long jmp
49     # to reload %cs and %esp. The segment descriptors are set up with no
50     # translation, so the mapping is still the identity mapping.
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 $0xa000,%ax            # 0xa000 -> port 0xa00
71     movw %ax,%dx
72     outw %ax,%dx
Ln 54, Col 9 Spaces: 2 UTF-8 LF x86 and x86_64 Assembly Go Live 12/08/22 0:59:33
```

3.1) In the bootasm.S, after analysis and reading the comments we can see that "movw \$(SEG_KDATA<<3), %ax" is the first instruction to be executed in 32-bit mode. The instruction "ljmp \$(SEG_KCODE<<3), \$start32" causes the transition to 32-bit mode.

3.2) As it is clear from the bootmain function of bootmain.c the last line of the bootloader is call to the entry function, so the last instruction to be executed is the following:

```
7d91: ff 15 18 00 01 00      call    *0x10018
```

The instruction is shifting the control to the address which is stored at 0x10018 since * operator is used for dereferencing. Now the first instruction can be found by looking into the kernel.asm file or we can see the contents of the memory location 0x10018 by using the x/1x 0x10018 command. We will get 0x0010000c as the address stored at memory location 0x10018. using x/1i 0c0010000c , we can see the instruction stored at that address.

So the first instruction of kernel is:

```
0x10000c: 0f 20 e0          mov     %cr4,%eax
```

3.3) The bootloader finds the information about how many sectors it must read in the information stored in the ELF header.

```

bootmain.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C bootmain.c ✘ bootasm.S ✘ bootblock.asm ✘ kernel.asm
C bootmain.c > bootmain(void)
21     struct proghdr *ph, *eph;
22     void (*entry)(void);
23     uchar* pa;
24
25     elf = (struct elfhdr*)0x1000; // 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 + ph->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

```

Ln 41, Col 57 Spaces: 2 UTF-8 LF C ⚡ Go Live Linux 2:23:33 12/08/22

As we can see in the above image the code from line 35 to 42 is used to load the kernel. ELF stands for Executable and Linkable format and ELF headers are loaded by xv6 into a memory location which is pointed by an “elf” pointer. The starting address of the first segment of the kernel to be loaded in “ph”- program header by adding an offset (“elf->phoff”) to the starting address(elf). End pointer eph is also maintained which points to the memory location after the end of the last segment. Then in the for loop it iterates over all the segments. For each segment, the segment which has to be loaded is pointed by pa. Then the current segment is loaded at the location by passing pa, ph->filesz and ph->off parameters to readseg. It then checks if the file size is less than the memory which has to be read; if so then it will append zeros at the end of the extra memory. So the bootloader will keep loading segments while the condition “ph < eph” is true. The value of ph and eph are determined using attributes phoff and phnum of the ELF header.

Exercise 4)

```

try.c - Desktop - Visual Studio Code
File Edit Selection View Go Run Terminal Help
try.c > (void)
C try.c > f(void)
4 void
5 {
6     int a[4];
7     int *b = malloc(16);
8     int *c;
9     int i;
10
11    printf(*: a = %p, b = %p, c = %p\n", a, b, c);
12
13    c = a;
14    for (i = 0; i < 4; i++)
15        a[i] = 100 + i;
16    c[0] = 200;
17    printf(*: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
18           a[0], a[1], a[2], a[3]);
19
20    c[1] = 300;
21    *(c + 2) = 301;
22    3[c] = 302;
23    printf(*: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
24           a[0], a[1], a[2], a[3]);
25
26    c = c + 1;
27    *c = 400;
28    printf(*: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
29           a[0], a[1], a[2], a[3]);
30
31    c = (int *) ((char *) c + 1);
32    *c = 500;
33    printf(*: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
34           a[0], a[1], a[2], a[3]);
35
36    b = (int *) a + 1;
37    c = (int *) ((char *) a + 1);
38    printf(*: a = %p, b = %p, c = %p\n", a, b, c);
39
40 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

pranshu@pranshu-OptiPlex-5090:~/Desktop$ ./a.out
1: a = 0x7ffc40e7e017, b = 0x55e8c7c812a0, c = 0x7ffc40e7e017
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
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 = 0x7ffc40e7e017, b = 0x7ffc40e7df74, c = 0x7ffc40e7dff1
pranshu@pranshu-OptiPlex-5090:~/

```

D Run Testcases ⌂ D 3 32 Spaces: 4 UTF-8 LF C Go Live Linux R 3:22:38 12/08/22

In line 1: the address of pointers a, b , c is printed.

In the line 2: as c is equal to a and we are modifying value of c afterwards the a[0] = c[0] = 200 and a[1]=101, a[2]=102, a[3]=103 as filled by the loop

In the line 3: c[1] = a[1] = 300 , *(c+2)=c[2]=a[2]=301, 3[c]=c[3]=a[3]= 302

In the line 4: we have incremented c to c+1 so *c = c[1]=a[1]=400 which is reflected in the output

In the line 5: as we are typecasting c as character before adding 1 to it according to pointer arithmetic its address will be incremented by 1 and then again typecast as integer. So now c is pointing at the 2nd byte of 1th index of the array so it will change the last 3 byte of c[1] and first byte of c[2] thus the value of a[1] and a[2] is changed.

In the line 6: as for b we have incremented the pointer of integer pointer type by 1 the value of b will be addressof(a) + 1*sizeof(int). And for c as a is typecasted to character pointer before adding 1 value of c will be addressof(a) + 1*sizeof(char) and which are exactly the case as you can see in the image.

Displaying a full list of the names, sizes, and link addresses of all the sections in the kernel using objdump command

Assgn_0A.pdf — Okular

File View Edit Go Bookmarks Tools Settings Help

Browse A Text Selection Annotations ^ 9 of 12 Zoom Out 90% Zc

Contents

Search...

Part... X86... Exerc... Simu... For... The... The... The... The... The... Exerc... Exerc... Part... Exerc... Load... Exerc... \$obj... \$obj... Exerc... \$obj... Exerc... Link... The... GD... Ctrl-c b fu... set... info...

loaded into memory. In the ELF object, this is stored in the ph->p_pa field (in this case, it really is a physical address, though the ELF specification is vague on the actual meaning of this field).

The link address of a section is the memory address from which the section expects to execute. The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for. (It is possible to generate *position-independent* code that does not contain any such absolute addresses. This is used extensively by modern shared libraries, but it has performance and complexity costs, so we won't be using it in this course.)

Typically, the link and load addresses are the same. For example, look at the .text section of the boot loader:

```
$ objdump -h bootblock.o
```

The BIOS loads the boot sector into memory starting at address 0x7c00, so this is the boot sector's load address. This is also where the boot sector executes from, so this is also its link address. We set the link address by passing -Ttext 0x7C00 to the linker in Makefile, so the linker will produce the correct memory addresses in the generated code.

Exercise 5.

Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in Makefile to something wrong, run **make clean**, recompile the job with **make**, and trace into the boot loader again to see what happens. Don't

xv6-public : bash — Konsole

pranshu@omen:~/xv6-public\$ objdump -h kernel

kernel: file format elf32-i386

Sections:	Idx	Name	Size	VMA	LMA	File off	Align
0 .text			000070da	80100000	00100000	00001000	2**4
1 .rodata			000009cb	801070e0	001070e0	000000e0	2**5
2 .data			00000000	8010a000	0010a000	00000000	2**12
3 .bss			0000a788	8010a520	0010a520	00000516	2**5
4 .debug_line			00006cb5	00000000	00000000	00000516	2**0
5 .debug_info			000121ce	00000000	00000000	000121cb	2**0
6 .debug_abbrev			000031d7	00000000	00000000	00024399	2**0
7 .debug_aranges			000003a8	00000000	00000000	00028370	2**3
8 .debug_str			000009ec	00000000	00000000	00028718	2**0
9 .debug_loc			00000621	00000000	00000000	00029254	2**0
10 .debug_ranges			00000089	00000000	00000000	0002fd62	2**0
11 .comment			0000002b	00000000	00000000	00030aea	2**0

pranshu@omen:~/xv6-public\$

xv6-public : gdb x xv6-public : make x xv6-public : bash x

New Tab Split View Copy Paste Find 14:29:40 12/08/22

We can see the result of the command in the above image. Note that the VMA and LMA of .text of the kernel are different so it is loaded and executed from different addresses.

Assgn_0A.pdf — Okular

File View Edit Go Bookmarks Tools Settings Help

Browse A Text Selection Annotations ^ 9 of 12 Zoom Out 90% Zc

Contents

Search...

Part... X86... Exerc... Simu... For... The... The... The... The... The... Exerc... Exerc... Part... Exerc... Load... Exerc... \$obj... \$obj... Exerc... \$obj... Exerc... Link... The... GD... Ctrl-c b fu... set... info...

loaded into memory. In the ELF object, this is stored in the ph->p_pa field (in this case, it really is a physical address, though the ELF specification is vague on the actual meaning of this field).

The link address of a section is the memory address from which the section expects to execute. The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for. (It is possible to generate *position-independent* code that does not contain any such absolute addresses. This is used extensively by modern shared libraries, but it has performance and complexity costs, so we won't be using it in this course.)

Typically, the link and load addresses are the same. For example, look at the .text section of the boot loader:

```
$ objdump -h bootblock.o
```

The BIOS loads the boot sector into memory starting at address 0x7c00, so this is the boot sector's load address. This is also where the boot sector executes from, so this is also its link address. We set the link address by passing -Ttext 0x7C00 to the linker in Makefile, so the linker will produce the correct memory addresses in the generated code.

Exercise 5.

Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in Makefile to something wrong, run **make clean**, recompile the job with **make**, and trace into the boot loader again to see what happens. Don't

OS Lab Assignment 0 — Google Docs

File Edit View Insert Format Tools Extensions Help

pranshu@omen:~/xv6-public\$ objdump -h bootblock.o

bootblock.o: file format elf32-i386

Sections:	Idx	Name	Size	VMA	LMA	File off	Align
0 .text			000001d3	00007c00	00007c00	00000074	2**2
1 .eh_frame			000000b0	00007d04	00007d04	00000248	2**2
2 .comment			0000002b	00000000	00000000	000002f8	2**0
3 .debug_aranges			00000040	00000000	00000000	00000328	2**3
4 .debug_info			000005d7	00000000	00000000	00000368	2**0
5 .debug_abbrev			0000022c	00000000	00000000	0000093a	2**0
6 .debug_line			00000029a	00000000	00000000	00000666	2**0
7 .debug_str			0000002b	00000000	00000000	00000909	2**0
8 .debug_loc			000002bb	00000000	00000000	00001021	2**0
9 .debug_ranges			00000078	00000000	00000000	000012dc	2**0

pranshu@omen:~/xv6-public\$

xv6-public : gdb x xv6-public : make x xv6-public : bash x

New Tab Split View Copy Paste Find 14:35:14 12/08/22

As we can see for bootblock.o the VMA and LMA are the same showing that the link and the load addresses are the same.

Exercise 5)

Here I have changed the bootloader's link address in Makefile from 0x7c00 to 0x7c10 and then after running make clean and make again. I started debugging it step by step.

The screenshot shows two windows side-by-side. On the left is a code editor with the Makefile for xv6-public. On the right is a terminal window titled 'xv6-public : gdb' showing the GDB debugger interface. The GDB session shows assembly code and registers. A breakpoint is set at 0x7c00, and the assembly shows instructions starting at 0x7c00. The terminal window also displays configuration files and command-line arguments used for the build.

```

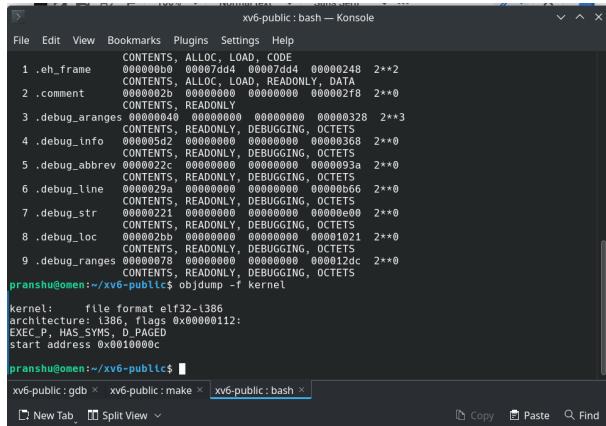
Makefile - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C bootmain.c M Makefile Go bootasm.S bootblock.asm
M Makefile
83 LDFLAGS += -m $(shell $(LD) -V | grep elf.i386 2>/dev/null | head -n 1)
84
85 # Disable PIE when possible (for Ubuntu 16.10 toolchain)
86 ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '^f[!f]no-pie'),)
87 CFLAGS += -fno-pie -no-pie
88 endif
89 ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e 'I[!f]nopie'),)
90 CFLAGS += -fno-pie -no-pie
91 endif
92
93 xv6.img: bootblock kernel
94 dd if=/dev/zero of=xv6.img count=10000
95 dd if=bootblock of=xv6.img conv=notrunc
96 dd if=kernel of=xv6.img seek=1 conv=notrunc
97
98 xv6memfs.img: bootblock kernelmemfs
99 dd if=/dev/zero of=xv6memfs.img count=10000
100 dd if=bootblock of=xv6memfs.img conv=notrunc
101 dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc
102
103 bootblock: bootasm.S bootmain.c
104 $(GCC) $(CFLAGS) -fno-pic -nostdinc -I . -c bootmain.c
105 $(GCC) $(CFLAGS) -fno-pic -nostdinc -I . -c bootasm.S
106 $(LD) $(LDFLAGS) -N -e start -Ttext 0x7c10 -o bootblock.o bootblock.asm
107 $(OBJCOPY) -S bootblock.o > bootblock.asm
108 $(OBJCOPY) -S 0 binary -j .text bootblock.o bootblock
109 ./sign.pl bootblock
110
111 entryther: entryther.S
112 $(GCC) $(CFLAGS) -fno-pic -nostdinc -I . -c entryther.S
113 $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryther.o
114 $(OBJCOPY) -S 0 binary -j .text bootblockother.o entryther
115 $(OBJCOPY) -S bootblockother.o > entryther.asm
116
117 initcode: initcode.S
118 $(GCC) $(CFLAGS) -nostdinc -I . -c initcode.S
119 $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o
120 $(OBJCOPY) -S 0 binary initcode.out initcode
121 $(OBJCOPY) -S initcode.o > initcode.asm
122
123 kernel: $(OBJS) entry.o entryther initcode kernel.ld
124 $(LD) $(LDFLAGS) -T kernel.ld -o kernel.entry.o $(OBJS) -b binary initcode entryth
125 $(OBJCOPY) -S kernel > kernel.ase
126 $(OBJCOPY) -t kernel | sed '1,*/SYMBOL TABLE/d; s/.* //; /^$/d' > kernel.sys
127
128 # kernelmemfs is a copy of kernel that maintains the
129 # disk Image in memory instead of writing to a disk.

```

One observation we can see is that changing the bootloader's link address will not affect BIOS and thus it will run smoothly and the control will be given to the boot loader. So I placed a breakpoint at the bootloader's previous link address that is 0x7c00 as the change is made after it so till that point both versions are guaranteed to behave identical. Now I run step into command for around 150 steps and start observing for change in both versions by keeping them side by side. The first change which I found is shown in the image below.

The screenshot shows two GDB sessions side-by-side. Both sessions show assembly code and registers. The left session has a breakpoint at 0x7c11, and the right session has a breakpoint at 0x7c10. The assembly code is identical up to the point where the link address is set. In the left session, the instruction at 0x7c31 is 'mov \$0x10,%ax'. In the right session, the instruction at 0x7c31 is 'mov \$0fe05,%es1'. This indicates a change in the assembly code due to the different link address.

The left side is where the link address is correctly set to be 0x7c00 and the right side is where the link address is changed to be 0x7c10.



```
File Edit View Bookmarks Plugins Settings Help
.xv6-public: bash — Konsole
CONTENTS, ALLOC, LOAD, CODE
1 .eh_frame 000000b0 00007dd4 000007dd4 00000248 2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .comment 0000002b 00000000 00000000 00000278 2**0
CONTENTS, READONLY
3 .debug_aranges 00000000 00000000 00000328 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
4 .debug_info 000005d2 00000000 00000000 00000368 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
5 .debug_abbrev 0000022c 00000000 00000000 0000093 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
6 .debug_line 0000029a 00000000 00000000 00000b66 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
7 .debug_str 00000221 00000000 00000000 00000000 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
8 .debug_loc 000002bb 00000000 00000000 00001021 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
9 .debug_ranges 00000078 00000000 00000000 000012dc 2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
pranshu@omen:~/xv6-public$ objdump -f kernel
kernel: file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
pranshu@omen:~/xv6-public$
```

I have also tried the objdump command to see the entry point as shown in the above image.

Exercise 6)

In the below image I have restarted QEMU and GDB and first set the breakpoint at 0x7c00 where the bootloader gets the control. I continued execution till the breakpoint. Then I used the x/8x 0x00100000 command to examine the 8 words of memory at 0x00100000. Then I set the breakpoint at 0x0010000c which is the point where the bootloader enters the kernel. Then I continue the execution till the breakpoint. Then I again examine the 8 words of memory at 0x00100000.

The screenshot shows a terminal window titled "xv6-public : gdb — Konsole <2>". The terminal displays a GDB session on the xv6-public kernel. The session starts with a warning about auto-loading being declined by the kernel. It then sets a breakpoint at address 0x7c00 and continues execution. The output shows assembly code and memory dump at address 0x00100000. The assembly code includes instructions like ljmp \$0x3d8, \$0xf000e05b and mov %cr4,%eax. The memory dump shows multiple 0x00000000 entries. The terminal also shows the system clock as 17:06:21 and the date as 12/08/22.

```
xv6-public : gdb — Konsole <2>
File Edit View Bookmarks Plugins Settings Help
For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: File "/home/pranshu/xv6-public/.gdbinit" auto-loading has been declined by your `auto-load-safe-path' configuration file. See the "auto-load-safe-path" section in the GDB manual.
To enable execution of this file add
    add-auto-load-safe-path /home/pranshu/xv6-public/.gdbinit
line to your configuration file "/home/pranshu/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/home/pranshu/.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 assumed to be i386
[1000:ffff] 0xfffffe: ljmp $0x3d8,$0xf000e05b
$0000ffff 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) break *0x7c00
Breakpoint 1 at 0x7c00
(gdb) continue
Continuing.
[ 0:7c00] => 0x7c00: cli
Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x00000000: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000010: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) break *0x0010000c
Breakpoint 2 at 0x10000c
(gdb) continue
Continuing.
The target architecture is assumed to be i386
<- 0x10000c: mov %cr4,%eax
Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x00000000: 0x1badb002 0x00000000 0xe4524ffe 0x83e0200f
0x00000010: 0x220f10c8 0x900000e0 0x220f0010 0xc0200fd8
(gdb)
```

As it can be seen in the above image at first the value at the 0x00100000 is changed after the bootloader entered the kernel. The reason behind this is that the address 0x00100000 is actually 1MB which is the address from where the kernel is loaded into the memory. Before the kernel was loaded into the memory, this address didn't have any data, i.e. garbage values. By default, all the uninitialized values are set to 0 in xv6. Hence, at the first breakpoint we got all zeroes as data hasn't been loaded at that time. But when we check at the second breakpoint, the kernel has been loaded into the memory and thus this address now contains meaningful data instead of zeroes.