ADARSH KUMAR
190101002

CS 344: Operating Systems Lab
Assignment 0

Solution 1

By using *asm* keyword we can use inline assembly code in C. I used Extended-Asm to do this exercise, in extended *asm* we can read and write C variables from assembler and also can perform jumps from assembler code to C labels, if required.

Syntax :

```
asm("Assembler Template"
    : Output Operands
    : Input Operands
    );
```

Code Added :

```
__asm__("inc %0" : "=r"(x) : "r"(x));
```

Output:

```
blackhat  mnt > d > … > Sem 5
$ gcc -o ex1 ex1.c
blackhat  mnt > d > … > Sem 5
$ ./ex1
Hello x = 1
Hello x = 2 after increment
OK
```

= is for denoting that we are updating the register value
inc is unary arithmetic operation in *x86* assembly language that increments value by 1
%0 denotes GNU to choose whatever general purpose register
Whole formatted .c code is attached in the folder

Solution 2

Numbers below denotes the instruction number according to the image attached

1: A comparison instruction of form *cmpw a, b* - sets condition codes according to b-a
2: Conditional jump instruction, with condition being jump if not equal
3: *XOR* instruction, this store 0 in *%edx*, because *A XOR A = 0*

4: Data movement instruction, that loads Stack Segment Register(ss) with *%edx*
5: Loading value *0x7000* to register *sp*

6: Loading value *0x7c4* to register *dx*

7: Unconditional jump instruction, that moves the execution to address *0x5576cf26*
8: Interrupt flag is cleared using *cli*, it disables the interrupts if flag is cleared

9: Clears the direction flag(*DF=0*), used for direction in which some instructions work
10: Loading value of *%ax* to *%cx*

```
(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) si
[f000:cf26]    0xfcf26: mov    %ax,%cx
0x0000cf26 in ?? ()
(gdb)
```

First 10 instructions of ROM BIOS, achieved this by using *si* command

Solution 3

```
.code16                     # Assemble for 16-bit mode
.globl start
start:      rsc, 15 years ago • spacing fixes: no tabs, 2-space indents (for rtm)
  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
```

bootasm.S* file showing 10 instructions starting from *0x7c00

```
.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
```

Snippet of *bootblocl.asm,* showing the disassembly of the instructions starting at *0x7c00*

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/10i $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
(gdb)
```

Disassembly of the instructions starting at address *0x7c00* using *x/Ni Addr* command of *gdb,* before that breakpoint at *0x7c00 was set*

Comparing all the above figures we can see the disassembly of the instructions and also the source code in *bootasm.S* are all similar that shows the next 10 instructions starting at address *0x7c00* this is the address where the boot sector is loaded. There are definitely some syntactical changes like change in keywords but the underlying assembly instructions are all same for all the three code snippets shown above.

```
// 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);
}
```

Function *readsect( )* in the file *bootmain.c*

```
00007c90 <readsect>:

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
    7c90: f3 0f 1e fb          endbr32
    7c94: 55                   push   %ebp
    7c95: 89 e5                mov    %esp,%ebp
    7c97: 57                   push   %edi
    7c98: 53                   push   %ebx
    7c99: 8b 5d 0c             mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c9c: e8 dd ff ff ff       call   7c7e <waitdisk>
}
```

readsect( )* in the *bootboack.asm file

```
(gdb) x/15i $eip
=> 0x7c90:      endbr32
   0x7c94:      push   %ebp
   0x7c95:      mov    %esp,%ebp
   0x7c97:      push   %edi
   0x7c98:      push   %ebx
   0x7c99:      mov    0xc(%ebp),%ebx
   0x7c9c:      call   0x7c7e
```

We can also see *readsect( )* function instructions using the *x/Ni Addr* command after setting a breakpoint at *0x7c90,* the same way what we did in the last part of the question

This code snippet taken from *bootblock.asm* shows which part is responsible for reading the remaining portion of the sectors of the kernel. The for loop starting from line number 327 is responsible for sector reading. After the for loop iteration completion the flow of the code reaches at instruction at line number 319 which has address of *0x7d91* which in turn executes *call *0x10018*.
We can step through remaining of the bootloader using *si* command in gdb.

```
315    for(; ph < eph; ph++){
316      7d8d: 39 f3                  cmp    %esi,%ebx
317      7d8f: 72 15                  jb     7da6 <bootmain+0x5d>
318    entry();
319      7d91: ff 15 18 00 01 00      call   *0x10018
320  }
321      7d97: 8d 65 f4               lea    -0xc(%ebp),%esp
322      7d9a: 5b                     pop    %ebx
323      7d9b: 5e                     pop    %esi
324      7d9c: 5f                     pop    %edi
325      7d9d: 5d                     pop    %ebp
326      7d9e: c3                     ret
327    for(; ph < eph; ph++){
328      7d9f: 83 c3 20               add    $0x20,%ebx
329      7da2: 39 de                  cmp    %ebx,%esi
330      7da4: 76 eb                  jbe    7d91 <bootmain+0x48>
331      pa = (uchar*)ph->paddr;
332      7da6: 8b 7b 0c               mov    0xc(%ebx),%edi
333      readseg(pa, ph->filesz, ph->off);
334      7da9: 83 ec 04               sub    $0x4,%esp
335      7dac: ff 73 04               pushl  0x4(%ebx)
336      7daf: ff 73 10               pushl  0x10(%ebx)
337      7db2: 57                     push   %edi
338      7db3: e8 44 ff ff ff         call   7cfc <readseg>
339      if(ph->memsz > ph->filesz)
340      7db8: 8b 4b 14               mov    0x14(%ebx),%ecx
341      7dbb: 8b 43 10               mov    0x10(%ebx),%eax
342      7dbe: 83 c4 10               add    $0x10,%esp
343      7dc1: 39 c1                  cmp    %eax,%ecx
344      7dc3: 76 da                  jbe    7d9f <bootmain+0x56>
345        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
346      7dc5: 01 c7                  add    %eax,%edi
347      7dc7: 29 c1                  sub    %eax,%ecx
348  }
```

Snippet of *bootbloack.asm* responsible for reading rest of the sectors of the kernel

Answering a)

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

Snippet of code segment in *bootasm.S* file

This is the code segment that is responsible for the switch from *16-bit to 32-bit protected mode*. The instruction at line number 51 of the picture is a long jump instruction that basically triggers the change in modes and is the cause of the switch from *16-bit to 32-bit protected mode*

Answering b)

As shown in the previous part where we found out the last executing instruction after the for loop is completed.

- Therefore
  0x7d91: call *0x10018
  is the **last bootloader instruction** that's executed

By setting a breakpoint at that address and exploiting the *si* command we can get the first instruction of kernel also

- So,
  0x10000c: mov %cr4, %eax
  is the **first kernel instruction**

```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

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

Answering c)

As said before, this for loop reads sectors, in order to find how many sectors are to fetched the two pointers *ph and eph* which points to the structure named *proghdr* define the limits, *ph* is the start pointer and *eph* is the end pointer.
These values of *ph and eph* are calculated using the ELF header.

```
// 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);
}
```

Solution 4

```
blackhat  mnt > d > … > CS344 OS LAB > ass0 > xv6-public  master  $
$ 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    00000ec7  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295df  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fdfd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030b05  2**0
                  CONTENTS, READONLY
```

Output of *objdump -h kernel*

```
blackhat  mnt > d > … > CS344 OS LAB > ass0 > xv6-public  master  $
$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001d3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dd4  00007dd4  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002a  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000328  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000368  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000022c  00000000  00000000  0000093a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   0000029a  00000000  00000000  00000b66  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    0000023c  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  0000103c  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012f7  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Output of *objdump -h bootblock.o*

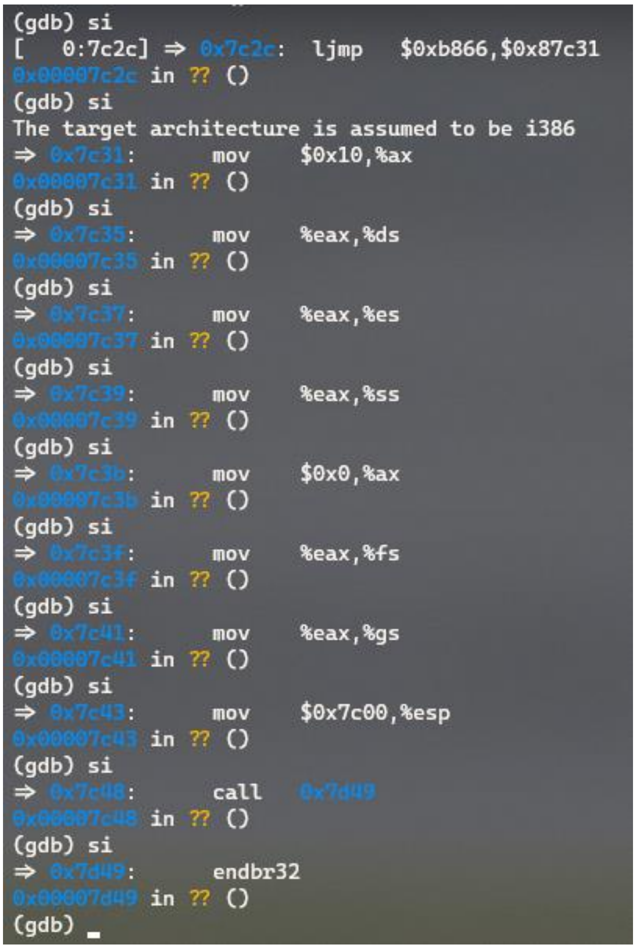
Some important column fields from these outputs :

1. **Name :** Shows the name of the section
2. **Size :** Shows the size of the section
3. **VMA :** Link address of the section, Link address is the starting memory address of the section where the execution begins
4. **LMA :** Load address of the section, Load address is the address where the section should be loaded into the memory

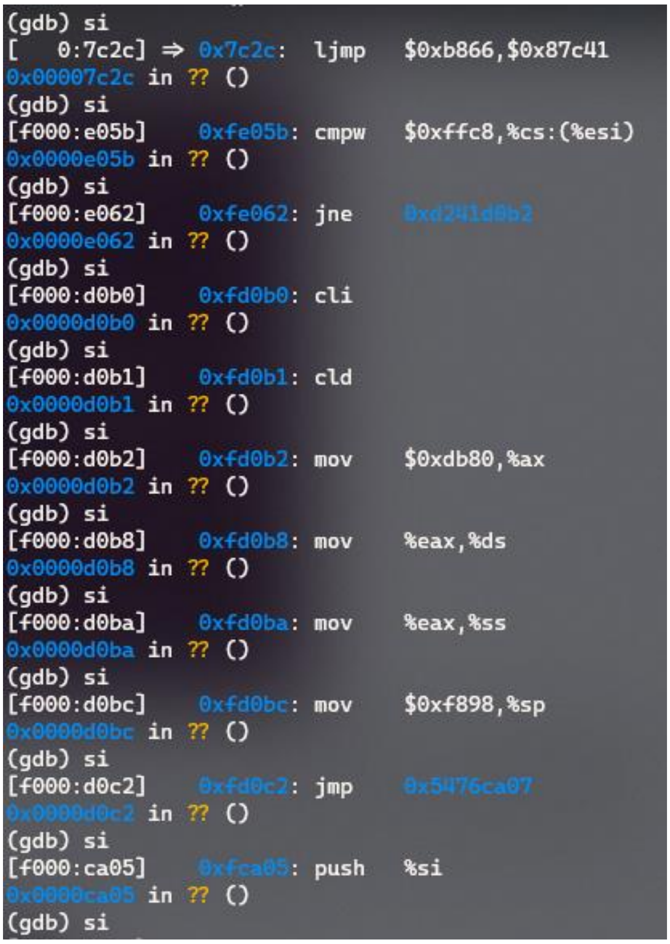
Solution 5

```

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32
```

As also shown in Solution 3, this is the code segment where there is a switch between *16-bit to 32-bit protected mode.* Here this line will be the 1st instruction where the execution will break if the address provided in Makefile would be wrong

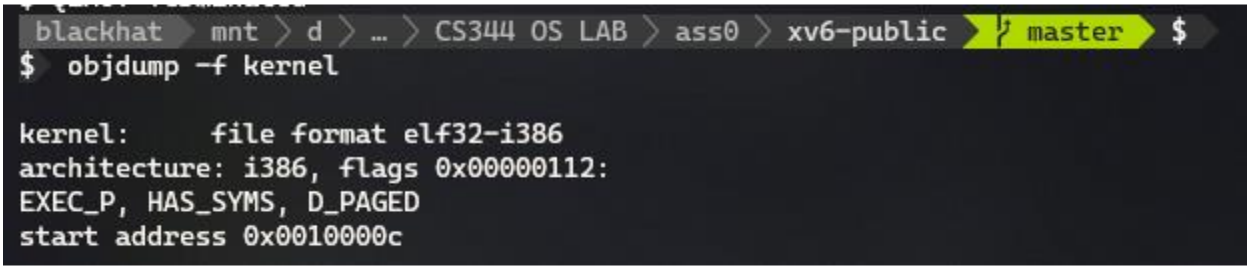


Correct link address



Incorrect link address

In both of theses pictures the next 10 instructions are shown after the 1st encounter of the *ljmp* instruction. The correct link address already written in makefile was *0x7c00.* I changed the link address to *0x7c10* i.e. something wrong and then ran *make clean* followed by *make* to rebuild all the files to load the boot loader. Then use GDB to reach the *ljmp* command at address *0x7c2c* in both cases and traced next 10 commands the wrong one has different and wrong instruction when compared to correct one due to the change in the makefile I did for manipulating the link address.



Output of the command objdump -f kernel

The link address of the *entry point,* the memory address in the program's text section at which execution begins. We can see this by running the above show command.
We can see the ***entry point address as 0x10000c***

Solution 6

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(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
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) _
```

Running the *x/8x Addr* command at two addresses, *0x7c00* and *0x7d91*

Address at which BIOS enters bootloader : *0x7c00*
Address at which boot loader enter kernel : 0x7d91

Set breakpoints at these locations and then used *x/8x Addr* command to see the 8 words of the memory at *0x100000.*
Initially before running bootloader the data at this location is all 0s but after kernel is loaded i.e. at the 2nd breakpoint we can see some data values changed in this location, this is because bootloader now have fully loaded kernel into the main memory.