CS 344 : OS Lab

Assignment 0
Anuraag Mahajan - 190101110

Exercise 1

Used the following inline assembly:

__asm__("incl %1;" : "=r" (x) : "r" (x));

Details:
__asm__ : asm or __asm__ both can be used.
"Incl %1" : assembly instruction specifying increment option on the input operand, referred to by %1.
"=r"(x) : output operand, using a register for storage with write only constraint (=)
"r"(x) : input operand, using a register for storage

Output of completed code : (ex1.c file with complete code included in the folder)

```
anuraag@DESKTOP-QEFMQFQ:/mnt/c/CP$ gcc ex1.c
anuraag@DESKTOP-QEFMQFQ:/mnt/c/CP$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
anuraag@DESKTOP-QEFMQFQ:/mnt/c/CP$
```

Exercise 2

Comments giving brief description of every instruction written in the console itself using "#".

```
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) # comparison of the two operands at the effective address
(gdb)
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) # checks if the result of previous comparison is true or false.
(gdb) # it is a conditional jump that follows a test.
(gdb)
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) # takes XOR of the two operands. Sets edx to zero in this case.
(gdb)
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb) # copies value stored in edx to ss register.
(gdb)
(gdb) si
```

```
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) # loads value 0x7000 at register sp
(gdb)
(gdb) si
[f000:e070]    0xfe070: mov    $0x7c4,%dx
0x0000e070 in ?? ()
(gdb) # loads value 0x7c4 at register dx
(gdb)
(gdb) si
[f000:e076]    0xfe076: jmp    0x5576cf26
0x0000e076 in ?? ()
(gdb) # jump to the specified address
(gdb)
(gdb) si
[f000:cf24]    0xfcf24: cli
0x0000cf24 in ?? ()
(gdb) # clears interrupt flag
(gdb)
(gdb) si
[f000:cf25]    0xfcf25: cld
0x0000cf25 in ?? ()
(gdb) # clears direction flag so that string pointers
(gdb) # auto increment after each string operation
(gdb)
(gdb) si
[f000:cf26]    0xfcf26: mov    %ax,%cx
0x0000cf26 in ?? ()
(gdb) # copies value at ax in cx
(gdb)
(gdb) si
[f000:cf29]    0xfcf29: mov    $0x8f,%ax
0x0000cf29 in ?? ()
(gdb) # loads value 0x8f into register ax
(gdb)
```

Exercise 3

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/15i $eip
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
```

>> GDB Output

Setting breakpoint at **0x7c00** and continuing through the first few instructions.

We can see that the corresponding instructions in bootblock.asm and bootasm.s are also same except for the syntactical difference.

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

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

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

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

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

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
      rsc, 15 years ago • spacing fixes: no tabs, 2-space indents (fc
```

bootasm.s

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

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

00007c09 <seta20.1>:

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

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

00007c13 <seta20.2>:

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

  movb    $0xdf,%al               # 0xdf -> port 0x60
    7c19: b0 df                mov    $0xdf,%al
  outb    %al,$0x60
```

bootblock.asm

Line to line correspondence of 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() from bootblock.asm

```
// 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);
}
```

readsect() from bootmain.c

```
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 }
```

The instructions from line 327 to line 347 read the remaining sectors of the kernel from the disk.

The instruction at line 319 **call *0x10018** is executed at the end of for loop as indicated by the conditional jump instruction at line 330

Hence, we can set the breakpoint at the address 0x7d91 and continue.

Questions:

1)

```
70
71  //PAGEBREAK!
72    # Complete the transition to 32-bit protected mode by using a long jmp
73    # to reload %cs and %eip.  The segment descriptors are set up with no
74    # translation, so that the mapping is still the identity mapping.
75    ljmp    $(SEG_KCODE<<3), $start32
76      7c2c: ea                      .byte 0xea
77      7c2d: 31 7c 08 00             xor    %edi,0x0(%eax,%ecx,1)
78
79  00007c31 <start32>:
80
81  .code32  # Tell assembler to generate 32-bit code now.
82  start32:
```

Point of transition from 16 to 32 bit code(bootblock.asm).

The long jump switch at line 75 causes the switch from 16 bit to 32 bit code.

2)

```
(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)
```

Continuing from the breakpoint set at the end of for loop,

Last bootloader instruction :

```
=> 0x7d91:      call   *0x10018
```

First kernel instruction :

```
=> 0x10000c:    mov    %cr4,%eax
```

3)

```
    // 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);
    }
```

Boot loader decides how many sectors to read using information stored in ELF header

Starting from ph, which is set at program header with offset - *elf->phoff*, the bootloader runs a loop till eph.

Eph is set using *elf->phnum* which gives the number of program header entries. Using this information the number of sectors are calculated.

Exercise 4

```
anuraag@DESKTOP-QEFMQFQ:/mnt/c/CP/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001d3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dd4  00007dd4  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000024  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000320  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000360  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000022c  00000000  00000000  00000932  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   0000029a  00000000  00000000  00000b5e  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    0000021d  00000000  00000000  00000df8  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001015  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012d0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

objdump -h bootblock.o

```
anuraag@DESKTOP-QEFMQFQ:/mnt/c/CP/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000070da  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801070e0  001070e0  000080e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006cb5  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   000121ce  00000000  00000000  000121cb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00003fd7  00000000  00000000  00024399  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003a8  00000000  00000000  00028370  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000ea8  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fdde  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      00000024  00000000  00000000  00030ae6  2**0
                  CONTENTS, READONLY
```

objdump -h kernel

Exercise 5

bootblock.asm

```
75    //PAGEBREAK!
76      # Complete the transition to 32-bit protected mode by using a long jmp
77      # to reload %cs and %eip.  The segment descriptors are set up with no
78      # translation, so that the mapping is still the identity mapping.
79      ljmp    $(SEG_KCODE<<3), $start32
80        7d50: ea                      .byte 0xea
81        7d51: 55                      push   %ebp
82        7d52: 7d 08                   jge    7d5c <start32+0x7>
83      ...
```

The point of switch from 16-bit to 32-bit mode,i.e. Line 79 is the first instruction where wrong bootloader link address will lead to incorrect execution of further instructions.

```
bootblock: bootasm.S bootmain.c
	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
	$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
	$(OBJDUMP) -S bootblock.o > bootblock.asm
	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
	./sign.pl bootblock
```

Change the link address specified in the Makefile

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb)
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
(gdb)
=> 0x7c39:      mov    %eax,%ss
0x00007c39 in ?? ()
(gdb)
=> 0x7c3b:      mov    $0x0,%ax
0x00007c3b in ?? ()
(gdb)
=> 0x7c3f:      mov    %eax,%fs
0x00007c3f in ?? ()
(gdb)
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb)
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb)
```

Original sequence of instructions

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87d31
0x00007c2c in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb)
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb)
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
(gdb)
[f000:d0b2]    0xfd0b2: mov    $0xdb80,%ax
0x0000d0b2 in ?? ()
(gdb)
[f000:d0b8]    0xfd0b8: mov    %eax,%ds
0x0000d0b8 in ?? ()
(gdb)
[f000:d0ba]    0xfd0ba: mov    %eax,%ss
0x0000d0ba in ?? ()
(gdb)
[f000:d0bc]    0xfd0bc: mov    $0xf898,%sp
0x0000d0bc in ?? ()
(gdb)
[f000:d0c2]    0xfd0c2: jmp    0x5476ca07
0x0000d0c2 in ?? ()
(gdb)
```

Altered instructions

Change the 0x7c00 address present in makefile to some random address.

Rebuild using **make clean**, then **make.**

After setting breakpoint at 0x7c00 and continuing we can see that the GDB output is same for both executions till the long jump instruction which corresponds to line 79 mentioned above.

We can see that the further instructions are totally different and hence the wrong address leads to improper execution.

```
anuraag@DESKTOP-QEFMQFQ:/mnt/c/CP/xv6-public$ objdump -f kernel

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

objdump -f kernel

We can see the entry point address - 0x0010000c

Exercise 6

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb)
```

Starting at location 0x0010000, the boot loader loads the kernel into main memory.
There is no meaningful data at this place until the boot loader starts executing, therefore when we find no information stored at the first breakpoint.

At the second breakpoint, we see that some data has been loaded. This is due to the fact that the second breakpoint occurs near the conclusion of the boot loader, and the kernel has been fully loaded into main memory.