Ex:2:

Further instructions into BIOS

Clear interrupts flag cli

Clear direction flag cld

Outputs bytes in CMOS registers(registers in which mem. stored even during power-off)-i.e. i/o port 0x70

take input from 0x71

A20 handler

Sets up GDTR and IDTR

.

.

.

Ex:3

outb(0x1F2, 1); // count = 1

```
mov    $0x1f2,%edx
mov    $0x1,%al
out    %al,(%dx)
```

outb(0x1F3, offset);

```
movzbl %bl,%eax          //moves 1byte into a 4-byte reg. with 24 leading zeroes
mov    $0xf3,%dl
out    %al,(%dx)
```

outb(0x1F4, offset >> 8);

```
movzbl %bh,%eax
mov    $0xf4,%dl
out    %al,(%dx)
```

outb(0x1F5, offset >> 16);

```
mov    %ebx,%eax
mov    $0xf5,%dl
shr    $0x10,%eax
movzbl %al,%eax
out    %al,(%dx)
```

outb(0x1F6, (offset >> 24) | 0xE0);

```
shr    $0x18,%ebx
mov    $0xf6,%dl
mov    %bl,%al
or     $0xffffffe0,%eax
out    %al,(%dx)
```

outb(0x1F7, 0x20);

```
mov    $0x20,%al
mov    $0xf7,%dl
out    %al,(%dx)
```

The begin and end of for-loop are at 0x7d47 and 0x7d5f

Sub-q1)

At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

Ans:

At 0x7c2d, where there is a long jump, the processor switches from real to protected i.e.16 to 32 bit.

sub-q2)

What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?

Ans:

The last instruction of Bootloader is jump to where kernel is loaded i.e.(0x10018)

0x7d61 => call *0x10018

The first instr. executed by kernel is movw $0x1234, 0x472

sub-q3)

Where is the first instruction of the kernel?

Ans:

First instruction of kernel at 0x10000c

sub-q4)

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

Ans:

Bootloader decides on number of sectors it must read by finding end of kernel and it is found in ELF format's program header.

Ex-4

C-pointers

Ex-5

The Failed instr. line is 0x7c2d: ljmp $0x8,$0x7d32

i.e. activating protected mode.

program receives SIGTRAP

Qemu has triple fault

The reason for failure is that GDT is loaded with garbage at 0x8 (cs).

So trying to access this will result in fault in fault which goes to handler again resulting in double fault and going to handle double fault results in triple fault.

Ex-6

At the point where BIOS enters the boot loader, at addr. 0x00100000, the value 0x00000000 is present, since it has nothing stored.

At the point where bootloader transfers control to kernel(i.e. 0x7d61), the value at addr. 0x00100000 is 0x1badb002.

The reason for containing some value in 0x00100000 in second point is that kernel is loaded at 0x00100000. (Load address of kernel)

Ex-7

Before running mov %eax, %cr0, addr. 0x00100000 contains 0x1badb002 and 0xf0100000 contains 0x00000000

After running this, both have 0x1badb002.

Reason paging enabled and 0xf0100000 v.a. is mapped to p.a. 0x00100000

If the instr. enabling paging is commented out, the instr. at 0x10002a => jmp *%eax (where eax contains 0xf010002c) fails because that addr. is not mapped.

Ex-8

Code unfilled found at Line 207 in lib/printfmt.c

Filled.

sub-q1)

Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

Ans:

console.c exports { void cputchar(int c) } fn. This function is used as subroutine by printf to print (i.e. display) a specific character on console display.(VGA).

sub-q2)

Explain the following from console.c:

Ans:

The code mentioned i.e.

```
if (crt_pos >= CRT_SIZE) {
int i;


memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
crt_buf[i] = 0x0700 | ' ';
crt_pos -= CRT_COLS;
}
```

Actually does the following.

1) If the text/char to be printed in current line exceeds the size of console i.e. I have exceeded the line limit, then it first transfers the current buffer into next line.

2) Fills the space occupied by Overflown buffer on prev. line with white spaces.

sub-q3)

For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

Trace the execution of the following code step-by-step:

int x = 1, y = 3, z = 4; cprintf("x %d, y %x, z %d\n", x, y, z);

- In the call to cprintf(), to what does fmt point? To what does ap point?
- List (in order of execution) each call to cons_putc, va_arg, and vcprintf. For cons_putc, list its argument as well. For va_arg, list what ap points to before and after the call. Forvcprintf list the values of its two arguments.

Ans:

In the call to cprintf("x %d, y %x, z %d\n", x, y, z);

fmt points to "x %d, y %x, z %d\n"

ap points to a list i.e. [x,y,z] of arguments passed down to function where x,y,z are replaced with args passed to fn.

cprintf's detailed calls

- vcprintf(fmt, [1,3,4])

- va_arg before [1,3,4] after [3,4] //called by getint()

- cons_putc(49); //called by printnum()

- va_arg before [3,4] after [4]

- cons_putc(51); //numbers as ASCII vals

- va_arg before [4] after []

- cons_putc(52);

sub-q4)

Run the following code.

unsigned int i = 0x00646c72; cprintf("H%x Wo%s", 57616, &i);

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

Ans:

Output: He110 World

Explanation: 57616 in hexadecimal is e110 &

0x72 - r, 0x6c - l, 0x64 - d, 0x0 - null forming 'rld/0'

cprintf's calls

- vcprintf(fmt, [57616, 0x00646c72])

- va_arg before [57616, 0x00646c72] after [0x00646c72]

- cons_putc(1);

- cons_putc(1);

- cons_putc(0);

- va_arg before [0x00646c72] after []

- cons_putc(114);

- cons_putc(108);

- cons_putc(100);

This output for little Indian since most significant byte is stored at lesser mem. location (i.e. 0x72 0x6c 0x64 0x00)

For big Indian the arg. should be 0x726c6400

sub-q5)

In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

cprintf("x=%d y=%d", 3);

Ans:

Output: x=3 y=<some Random Num.>

Reason: va_arg call for y's argument returns the the value in mem. location next to the x's argument in list which is arb.

sub-q6)

Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

Ans:

One method is to keep another stack and keep popping from one and pushing to other and use this stack as argument stack.

Ex-9:

Kernel initializes stack just before calling C-code (i.e. at 0xf010002f => mov $0x0,%ebp at 0xf0100034 => mov $0xf0110000,%esp)

Using paging 0xf0110000 translates to 0x00110000 the memory at which esp points to (physical mem.) Kernel reserves space for stack by setting esp to point to a location. The sapce between end of kernel's instr. space and esp is reserved for stack.

Ex-10:

8 words are pushed into stack

They are caller function's arguments, return address to which fn. Must return to, state of ebp, and some other value.

Ex-11 & Ex-12:

Implemented