Lab1 answers

Exercise 3

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

We can find that information in the Xv6 ebook :

> Once it has loaded the GDT register, the boot loader enables protected mode by setting the 1 bit (CR0_PE) in register %cr0 (9292-9294). Enabling protected mode does not immediately change how the processor translates logical to physical addresses; it is only when one loads a new value into a segment register that the processor reads the GDT and changes its internal segmentation settings. One cannot directly modify %cs, so instead **the code executes an ljmp (far jump) instruction (9303), which allows a code segment selector to be specified. The jump continues execution at the next line (9306) but in doing so sets %cs to refer to the code descriptor entry in gdt.** That descriptor describes a 32-bit code segment, so the processor switches into 32-bit mode.

Summarized : the boot loader sets CR0_PE in %cr0 (description here, then does a trick (using `ljmp`) to set %cs which causes the switch to 32-bit mode.

The relevant code from bootasm.S

```
  # 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
```

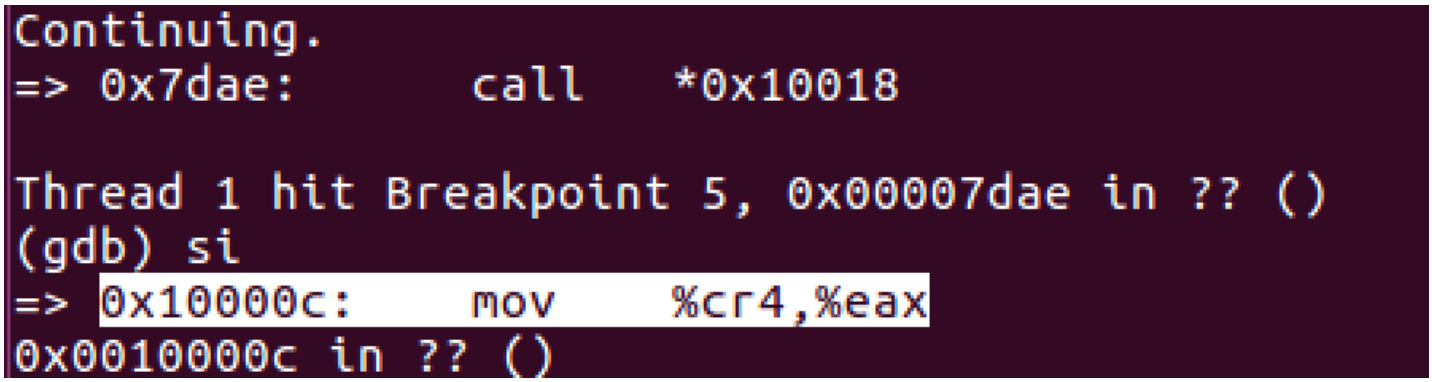
> What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

The last instruction of the boot loader is the call to the kernel : `call *0x10018` (in bootblock.asm).

The first instruction of the kernel it just loaded is `mov %cr4,%eax` (in kernel.asm).

> Where is the first instruction of the kernel ?

I'm not sure whether we're asked in which file, or at which address.
The file is kernel.asm.
The physical address is 0x10000c (virtual : 0x8010000c), as we can see from simulating and using gdb.



> 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?

This is I believe answered later in the lab :

> The boot loader uses the ELF program headers to decide how to load the sections. The program headers specify which parts of the ELF object to load into memory and the destination address each should occupy. [...] Other information for each program header is given, such as the virtual address (“vaddr”), the physical address (“paddr”), and the size of the loaded area (“memsz” and “filesz”).

In practice, we can see the C code doing that in bootmain.c :

```
1  elf = (struct elfhdr*)0x10000;  // scratch space
2
3  // Read 1st page off disk
4  readseg((uchar*)elf, 4096, 0);
5
6  // Is this an ELF executable?
7  if(elf->magic != ELF_MAGIC)
8    return;  // let bootasm.S handle error
9
10 // Load each program segment (ignores ph flags).
11 ph = (struct proghdr*)((uchar*)elf + elf->phoff);
12 eph = ph + elf->phnum;
13 for(; ph < eph; ph++){
14   pa = (uchar*)ph->paddr;
15   readseg(pa, ph->filesz, ph->off);
16   if(ph->memsz > ph->filesz)
17     stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
18 }
```

The informations given by `elf` that we need come from the first page off disk.

Exercise 6

> Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint?

When we enter the boot loader, only itself has been loaded in memory (512 bytes), and there is thus nothing (all 0's) at

0x100000 :

```
[   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
```

However, when we enter the kernel, the kernel itself has been loaded by the bootloader starting at address 0x100000. Therefore when we look into the memory, there is now something :

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

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x1000b8e0      0x220f0012      0xc0200fd8
```

Exercise 7

> Make sure you understand what just happened.

To my understanding, `movl %eax, %cr0` turns on the mapping from virtual address to physical address by setting the CR0_PG bit.

Hence 0x80100000 that we before interpreted as a huge physical address is now considered a virtual address and thus mapped to 0x100000. That's why after this instruction, if we examine the memory at 0x100000 and at 0x80100000, we see the same data.

> What is the first instruction after the new mapping is established that would fail to work properly if the mapping weren't in place?

The first instruction failing is the first instruction to execute after the jmp (first instruction in main). We have no problem doing the jump because a jump simply updates the value of %eip, but does not try to fetch what is there (yet). However, when we try to execute the next instruction, we'll have to look at what is really at *physical* address 0x80103919 (main) and that causes a failure since we don't have that much memory accessible. It is what qemu tells us before exiting too :

> fatal: Trying to execute code outside RAM or ROM at 0x80103919

Exercise 8

1.

> Explain the the three functions putch, vcprintf, and cprintf in console.c and what their relationship to the rest of console.c functions is.

putch outputs a single character to the console. vcprintf writes to the console, according to its argument format and a variable length arguments list. cprintf does the same as vprintf but initializes the va_list before calling vprintf.

Complete

2.

> Explain the following from `console.c` :

```
if((pos/80) >= 24){
  memmove(crt, crt+80, sizeof(crt[0])*23*80);
  pos -= 80;
  memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
}
```

First note in console.c this comment `// Cursor position: col + 80*row.` suggests that a row in the console is of width 80. pos/80 (the integer division of pos by 80) gives us the row number at which we the cursor stands. Thus `if((pos/80) >= 24)` means "if the cursor is on row 24 or more". What we need to do in that case is "scroll up" (by one line). `memmove` relocates the lines that we still want to see (ie. all but the first one) to the beginning of the CGA memory. The cursor position is adjusted accordingly (sent back by 80 characters) and finally we make sure to zero-out the remaining characters (those which are now after the cursor, since we scrolled up) with a `memset` .

3.

> In the call to `cprintf()` , to what does `fmt` point? To what does `ap` point?

`fmt` points to the (beginning of the) formating string, ie. to "x %d, y %x, z %d\n".

`ap` is of type `va_list` , ie. it's a variable which holds a variable length argument list. It originally "points" to the first argument (though the implementation of va_list is a bit more complicated from what I've read).

4.

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

The output is `He110 World` . The explanation is simple :

- The `%x` means that 57616 will be printed in hex format and 57616 in hex is e110.
- The `%s` means that 0x00646c72 will be interpreted as a string (ie. consecutive bytes interpreted as characters according to the ASCII table). Using the ASCII table, we see that 0x72 is 'r', 0x6c is 'l' and 0x64 is 'd'. We also make sure to put the 0 symbol to terminate the string.

All together, we get the concatenation :

> H + e110 + _Wo + rld => He110 World

If the x86 were big-endian, we would need to put each byte of i in the reverse order : 0x726c6400. We would not need to change the value of 57616 (provided a correct big-endian implementation of %x).

5.

> 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);

Arguments are "passed" by setting the memory in a specific place to a given value (the arg. value). cprintf isn't aware of the fact that the memory hasn't been set to a specific value for the second `%d` symbol, and thus will print whatever is in memory at the place it usually looks up the value for the second argument.

6.

> 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 (Edit on 8/27/18: or the macros va*start, va*arg) so that it would still be possible to pass it a variable number of arguments?

We could probably "undo" this change of calling convention by having either cprintf or the va*start, va*arg macros reverse the order of the arguments before doing the same it does here.

Exercise 9

> Determine where the kernel initializes its stack, and exactly where in memory its stack is located.

The kernel initializes the stack pointer at instruction 0x7c43 and locates it to 0x7c00 (which is in "low memory", according to the given memory diagram)

> How does the kernel reserve space for its stack?

From the xv6 book (p. 36):

> Now exec allocates and initializes the user stack. It allocates just one stack page. Exec copies the argument strings to the top of the stack one at a time, recording the pointers to them in ustack. It places a null pointer at the end of what will be the argv list passed to main. The first three entries in ustack are the fake return PC, argc, and argv pointer. Exec places an inaccessible page just below the stack page, so that programs that try to use more than one page will fault.
>
> And at which "end" of this reserved area is the stack pointer initialized to point to?

The stack grows towards lower addresses, so the stack pointer must be initialized to point at the higher "end" of the reserved area.

Exercise 10

> How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

Each recursive level test_backtrace pushes 8 32-bit words on the stack.