Exercise 3. Set a breakpoint at address 0x7c00 and Continue execution until that breakpoint.

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

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[   0:7c01] => 0x7c01:  cld
0x00007c01 in ?? ()
```

Trace into bootmain() in boot/main.c, and then into readsect().
identify the exact assembly instructions that correspond to each of the statements in readsect().

Bootmain address is 0x7d15 ,
Readsect address is 0x7c7c

```
bootmain(void)
{
    7d15:       55                      push   %ebp
    7d16:       89 e5                   mov    %esp,%ebp
```

```
00007c7c <readsect>:

void
readsect(void *dst, uint32_t offset)
{
```

void
readsect(void *dst, uint32_t offset)
{

waitdisk(); // that is just a function call

```
0x7c83:       call    0x7c6a
```

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

```
0x7c88:       mov     $0x1f2,%edx
0x7c8d:       mov     $0x1,%al
0x7c8f:       out     %al,(%dx)
```

outb(0x1F3, offset);

```
0x7c90:      mov    $0x1f3,%edx
0x7c95:      mov    %cl,%al
0x7c97:      out    %al,(%dx)
```

outb(0x1F4, offset >> 8);

```
0x7c98:      mov    %ecx,%eax
0x7c9a:      mov    $0x1f4,%edx
0x7c9f:      shr    $0x8,%eax
0x7ca2:      out    %al,(%dx)
```

outb(0x1F5, offset >> 16);

```
0x7ca3:      mov    %ecx,%eax
0x7ca5:      mov    $0x1f5,%edx
0x7caa:      shr    $0x10,%eax
0x7cad:      out    %al,(%dx)
```

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

```
0x7cae:      mov    %ecx,%eax
0x7cb0:      mov    $0x1f6,%edx
0x7cb5:      shr    $0x18,%eax
0x7cb8:      or     $0xffffffe0,%eax
0x7cbb:      out    %al,(%dx)
```

outb(0x1F7, 0x20); // cmd 0x20 - read sectors

```
0x7cbc:      mov    $0x1f7,%edx
0x7cc1:      mov    $0x20,%al
0x7cc3:      out    %al,(%dx)
```

waitdisk();

```
0x7cc4:      call   0x7c6a
```

insl(0x1F0, dst, SECTSIZE/4);

```
0x7cc9:      mov    0x8(%ebp),%edi
0x7ccc:      mov    $0x80,%ecx
0x7cd1:      mov    $0x1f0,%edx
0x7cd6:      cld
0x7cd7:      repnz insl (%dx),%es:(%edi)
```

}

Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk.

The starting of the loop is the address 0x7d51 and the end of the loop is at address 7d69. The loop will be exited when the last jmp fails.

```
0x7d51:      cmp    %esi,%ebx
0x7d53:      jae    0x7d6b
0x7d55:      pushl  0x4(%ebx)
0x7d58:      pushl  0x14(%ebx)
0x7d5b:      add    $0x20,%ebx
0x7d5e:      pushl  -0x14(%ebx)
0x7d61:      call   0x7cdc
0x7d66:      add    $0xc,%esp
0x7d69:      jmp    0x7d51
```

Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

When the loop is finished, the last instruction in the bootmain will call the first instruction in entry.s which is part of the kernel

Note that what stored in address 0x10018 is 0x10000c which where the first instruction of the kernel is stored

```
(gdb) b *0x7d6b
Breakpoint 1 at 0x7d6b
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d6b:      call   *0x10018

Breakpoint 1, 0x00007d6b in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
```

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

The following code is what does the conversion from the real mode (16 bit) to the protected mode (32 bit)

Ljmp : it is used to change the values of CS register [code segment]

```
0x7c1e:      lgdtl  (%esi)
0x7c21:      fs jl  0x7c33
0x7c24:      and    %al,%al
0x7c26:      or     $0x1,%ax
0x7c2a:      mov    %eax,%cr0
0x7c2d:      ljmp   $0xb866,$0x87c32
0x7c34:      adc    %al,(%eax)
0x7c36:      mov    %eax,%ds
0x7c38:      mov    %eax,%es
0x7c3a:      mov    %eax,%fs
0x7c3c:      mov    %eax,%gs
0x7c3e:      mov    %eax,%ss
```

- **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 in the boot loader is the last instruction in bootmain.c which is :

```
        // call the entry point from the ELF header
        // note: does not return!
        ((void (*)(void)) (ELFHDR->e_entry))();
    7d6b:       ff 15 18 00 01 00       call   *0x10018
```

This function jumps to whatever in the address 0x10018

```
(gdb) x/x 0x10018
0x10018:        0x0010000c
```

- ***Where* is the first instruction of the kernel?**

The first instruction in the kernel is in the Assembly file entry.s, and it is located at address 0x10000c

```
Breakpoint 1, 0x00007d6b in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
```

- **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?**

It happens when the bootloader loads the first 4096 bytes of kernel ELF into the memory from disk.and this information is found in ELF header.

```
struct Proghdr *ph, *eph;

	// read 1st page off disk
	readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

	// is this a valid ELF?
	if (ELFHDR->e_magic != ELF_MAGIC)
		goto bad;

	// load each program segment (ignores ph flags)
	ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
	eph = ph + ELFHDR->e_phnum;
	for (; ph < eph; ph++)
		// p_pa is the load address of this segment (as well
		// as the physical address)
		readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

Exercise 6. 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?

The address when BIOS enters the Boot loader is 0x7c00
The address when Boot loader enters the Kernel is 0x1000c

Here the 8 words at these 2 times

```
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

```
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x0000b812      0x220f0011      0xc0200fd8
(gdb)
```

They are different because when the BIOS enters the boot loader, the Kernel hasn’t
Been loaded into memory so there is no useful data at that address.
However, after the boot loader loads the Kernel in memory, and since the kernel starts executing at address 0x10000c then this is the region where the kernel has been loaded

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the movl %eax, %cr0. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the **stepi** GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

Before the instruction

```
Breakpoint 1, 0x00100025 in ?? ()
(gdb) x/x 0x00100000
0x100000:       0x1badb002
(gdb) x/x 0xf0100000
0xf0100000:     0x00000000
```

After the instruction

```
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/x 0xf0100000
0xf0100000:     0x1badb002
(gdb) x/x 0x00100000
0x100000:       0x1badb002
```

Before the instruction there was no virtual addressing, so the address oxf0100000 has no meaning. After that instruction, the virtualization started and that high address could be mapped/translated into the physical address 0x0100000. That is why both of the addresses have the same content

Once it has loaded the GDT register , the bootloader enables protected mode by setting the 1 bit (CR0_PE) in register %cr0.Enabling protected mode doesnot immediately change how the processor translates logical to physical addresses; it isonly 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 m

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the movl %eax, %cr0 in kern/entry.S, trace into it, and see if you were right.

The first instruction that will fail is the instruction that will try to jump into a virtual address which is the jmp instruction below:

```
=> 0x100028:    mov    $0xf010002f,%eax
   0x10002d:    jmp    *%eax
```

The qemu failed and exited

```
qemu: fatal: Trying to execute code outside RAM or ROM at 0x00000000f010002c

EAX=f010002c EBX=00010094 ECX=00000000 EDX=0000009d
```

```
Breakpoint 1, 0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
Remote connection closed
(gdb)
```

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

Determine where the kernel initializes its stack?

The Kernel initialize its stack used the instruction below, so the top of the stack is at a virtual address 0xf0110000.

```
        movl    $0x0,%ebp                       # nuke frame pointer
f010002f:       bd 00 00 00 00          mov    $0x0,%ebp

        # Set the stack pointer
        movl    $(bootstacktop),%esp
f0100034:       bc 00 00 11 f0          mov    $0xf0110000,%esp
```

And these are the other areas of memory as defined in the program header

```
oha@moha:~/6.828/lab$ objdump -h obj/kern/kernel

bj/kern/kernel:     file format elf32-i386

ections:
dx Name          Size      VMA       LMA       File off  Algn
 0 .text         00001861  f0100000  00100000  00001000  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata       00000730  f0101880  00101880  00002880  2**5
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .stab         000038b9  f0101fb0  00101fb0  00002fb0  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .stabstr      000018c6  f0105869  00105869  00006869  2**0
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .data         0000a300  f0108000  00108000  00009000  2**12
                 CONTENTS, ALLOC, LOAD, DATA
 5 .bss          00000644  f0112300  00112300  00013300  2**5
                 ALLOC
```

.data is below the stack and ends at 0xf0108a300
.bss is above the stack and begins at address 0xf0112300
The area in between (as far as I can tell) could be used by the stack

where in memory its stack is located ?
The top of the stack is 0xf0110000, so the stack could use the memory space below this address,
Virtual address : 0xf0110000
Physical address: 0x00110000

How does the kernel reserve space for its stack?At which "end" of this reserved area is the stack pointer initialized to point to?
It reserves space by defining the Top of the stack using %esp register and then each time we push some data, it will decrease.
The stack pointer (which is in register %esp) points to the highest address, and each time we push some data, the %esp will decrease by 4.

```
.data
###################################################################
# boot stack
###################################################################
```

```
    .p2align    PGSHIFT     # force page alignment
    .globl      bootstack
bootstack:
    .space      KSTKSIZE
    .globl      bootstacktop
Bootstacktop:

movl    $(bootstacktop),%esp
```

Here the size of the kernel stack is KSTKSIZE which is equal to 8*4096
Here at first the stack pointer points to the bootstack then allocates KSKSIZE bytes below it and then bootstacktop label which points to the end of the stack.