Exercise 1

Skip it, already gained from CS:APP.

Exercise 2

Use `make qemu-gdb` to compile and boot up the OS, and use `make gdb` to track the instructions.

The first 24 instructions are:

```
1  [f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
2  [f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6c48
3  [f000:e062]    0xfe062: jne    0xfd2e1
4  [f000:e066]    0xfe066: xor    %dx,%dx
5  [f000:e068]    0xfe068: mov    %dx,%ss
6  [f000:e06a]    0xfe06a: mov    $0x7000,%esp
7  [f000:e070]    0xfe070: mov    $0xf3691,%edx
8  [f000:e076]    0xfe076: jmp    0xfd165
9  [f000:d165]    0xfd165: mov    %eax,%ecx
10 [f000:d168]    0xfd168: cli
11 [f000:d169]    0xfd169: cld
12 [f000:d16a]    0xfd16a: mov    $0x8f,%eax
13 [f000:d170]    0xfd170: out    %al,$0x70
14 [f000:d172]    0xfd172: in     $0x71,%al
15 [f000:d174]    0xfd174: in     $0x92,%al
16 [f000:d176]    0xfd176: or     $0x2,%al
17 [f000:d178]    0xfd178: out    %al,$0x92
18 [f000:d17a]    0xfd17a: lidtw  %cs:0x6c38
19 [f000:d180]    0xfd180: lgdtw  %cs:0x6bf4
20 [f000:d186]    0xfd186: mov    %cr0,%eax
21 [f000:d189]    0xfd189: or     $0x1,%eax
22 [f000:d18d]    0xfd18d: mov    %eax,%cr0
23 [f000:d190]    0xfd190: ljmpl  $0x8,$0xfd198
24 The target architecture is assumed to be i386
25 => 0xfd198:     mov    $0x10,%eax
```

- 1: jump to 0xfe05b
- 2,3: jump to 0xfd2e1 if memory %cs:0x6c48 is not equal to zero; here it didn't jump
- 4-7: set registers
- 10: clear interrupt, avoiding input from peripheral devices to interrupt the booting procedure.
- 11: clear director, set the block to be transferred from low address to high address.
- 13-17: CPU uses I/O port to communicate with outside. 0x70 and 0x71 are for CMOS operation.
- 18: load Interrupt Descriptor Table
- 19: load Global Descriptor Table
- 23: long jump and come into protected mode from real mode.

......

Exercise 3

`boot/boot.S` started in 16-bit real mode. The bootloader starts from 0x7c00, which is a historical legacy from IBM.

- It first sets up the important data segment registers (DS, ES, SS).
- Then `seta20.1` and `seta20.2` opens A20 gate to unwrap the addresses higher than 1MB.
- load GDT and switch from real mode to protected mode.
- Then call `bootmain` in `boot/main.c`

`boot/main.c` reads the kernel's ELF from hard disk and calls the kernel.

Question 1: At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

Answer: The processor enters protected mode when the PE bit of cr0 register is 1. So it enters 32-bit mode after `orl  $CR0_PE_ON,  %eax`.

Question 2: What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?

Answer: Last instruction for boot loader is `0x7d6b:  call  *0x10018`, which is execute `((void (*)(void)) (ELFHDR->e_entry))();` in `boot/main.c`. First instruction of kernel is `0x10000c:  movw  $0x1234,0x472`.

Question 3: 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?

Answer: The information is obtained from the ELF program header. Each program header table records `p_pa` (physical address), `p_memsz` (memory size it will take), `p_offset` (the offset to the kernel file). So the boot loader can base on the information, and for each segment, it will read `p_memsz` of bytes from `p_offset` of the ELF into address `p_pa`.

Exercise 4

Skip it, already have good command of C.

Exercise 5

The 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader:

```
(gdb) x/16x 0x00100000
0x100000:   0x00000000  0x00000000  0x00000000  0x00000000
0x100010:   0x00000000  0x00000000  0x00000000  0x00000000
0x100020:   0x00000000  0x00000000  0x00000000  0x00000000
0x100030:   0x00000000  0x00000000  0x00000000  0x00000000
```

The 8 words of memory at 0x00100000 at the point the boot loader enters the kernel.

```
(gdb) x/16x 0x00100000
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010:   0x34000004  0x0000b812  0x220f0011  0xc0200fd8
0x100020:   0x0100010d  0xc0220f80  0x10002fb8  0xbde0fff0
```

```
0x100030:   0x00000000  0x110000bc  0x0056e8f0  0xfeeb0000

(  0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
   0x10000d:    movl   $0xb81234,0x472
   0x100017:    add    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)
   0x10001b:    and    %al,%bl
   0x10001d:    mov    %cr0,%eax
   0x100020:    or     $0x80010001,%eax
   0x100025:    mov    %eax,%cr0
   0x100028:    mov    $0xf010002f,%eax
   0x10002d:    jmp    *%eax
   0x10002f:    mov    $0x0,%ebp
   0x100034:    mov    $0xf0110000,%esp
   0x100039:    call   0x100094             )
```

This is different because the boot loader reads the kernel to 0x100000. Obviously the instructions there are the kernel's instructions.

Exercise 6

Now we change the *link* address in `boot/Makefrag` into `0x8C00`, which is not aligned with the *load* address, and see what will happen.

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

The boot loader is still loaded into 0x7c00. This is because what we have changed is the link address but not the load address.

Then use stepi, the problem came when `ljmp` instruction is executed. This is because the `ljmp` does not use offset to locate the instruction, so the instruction to be jumped will be wrong. This will cause the QEMU to endlessly reboot since our version is not the original version patched by MIT 6.828.

Exercise 7

Question 1: find where the new virtual-to-physical mapping takes effect.

Answer: The page is enabled after executing these instructions in `kern/entry.S`

```
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0
```

Before `movl %eax, %cr0` is executed, the memory at 0x00100000 and at 0xf0100000 are as follows.

```
(gdb) x/4x 0x00100000
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000  0x00000000  0x00000000
0x00000000
```

After it is executed, the memory at 0x00100000 and at 0xf0100000 are as follows.

```
(gdb) x/4x 0x00100000
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002  0x00000000  0xe4524ffe
0x7205c766
```

Now the virtual-to-physical mapping takes effect.

Question 2: What is the first instruction after the new mapping is established that would fail to work
properly if the old mapping were still in place?

Answer: `jmp  *%eax`

After commenting out the two lines of `orl  $(CR0_PE|CR0_PG|CR0_WP), %eax` and `movl  %eax, %cr0`, the QEMU crashes and printed out all the registers after stepi into the instruction after `jmp *%eax`, because `%eax=0xf0100027`, but the processor can not have access to the virtual memory, so it cannot execute the instruction in 0xf0100027. If we comment these two lines back, the QEMU can correctly work again.

Exercise 8

Like what has been provided in `vprintfmt` function in `lib/printfmt.c`, we just modify the `case 'u':` and insert these lines to `case 'o':` to implement the output of oct number.

```
case 'o':
    putch('0', putdat);
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

Exercise 9

Add another flag `print_plus`, set it to 1 if encounter the `+` flag.

```
int plus_sign = 0;
switch (ch = *(unsigned char *) fmt++) {
    case '+':
        plus_sign = 1;
        goto reswitch;
```

```
    case 'd':
        num = getint(&ap, lflag);
        if ((long long) num < 0) {
            putch('-', putdat);
            num = -(long long) num;
        }else if (plus_sign && num){
            putch('+', putdat);
        }
        base = 10;
        goto number;
```

Question 1: Explain the interface between `printf.c` and `console.c` . Specifically, what function does `console.c` export? How is this function used by `printf.c` ?

Answer: `console.c` deals with the hardware and provides the interface to input and output chars to console by the function `getchar` and `cputchar`. `printf.c` uses `cputchar` to implement `putch`.

Question 2: Explain the following from `kern/console.c`

```
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;
}
```

Answer: `CRT_COLS` and `CRT_SIZE` are defined in `kern/console.h`, which are the columns and the number of characters that the monitor can hold. This part of code detects that the screen is full and then drop the top line, move each line upwards, and clear the last line.

Question 3: Trace the execution of the following code step-by-step:

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

Answer: `fmt` points to the format string `"x %d, y %x, z %d\n"`. `ap` is a type of `va_list`, which is how C implement variadic function. According to the gdb outcome and also how gcc implements it, `ap` points to the address of the second parameter `x`.

3.2 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. For `vcprintf` list the
values of its two arguments.

Answer:

```
=>cprintf
=>vcprintf("x %d, y %x, z %d\n", 12(%ebp))
=>cons_putc('x')
```

```
=>cons_putc(32)
=>va_arg (uint32_t *)ebp+3->(uint32_t *)ebp+4
=>cons_putc('1')
=>cons_putc(',')
=>cons_putc(32)
=>cons_putc('y')
=>cons_putc(' ')
=>va_arg (uint32_t *)ebp+4->(uint32_t *)ebp+5
=>cons_putc('3')
=>cons_putc(',')
=>cons_putc(' ')
=>cons_putc('z')
=>cons_putc(' ')
=>va_arg (uint32_t *)ebp+5->(uint32_t *)ebp+6
=>cons_putc('4')
=>cons_putc('\n')
```

Question 4: Run the following code.

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

Answer: `He110 World`. This is because 57616=0xe110. And as we uses little-endian machine, 0x72='r', 0x6c='l', 0x64='d', 0x00='\0'. If the machine is big-endian, we just transform `i` into `0x726c6400` and we don't need to change 57616.

Question 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?

Answer: What is going to be printed is the int representation of the 4 bytes followed by `x`. This is because C uses stack to pass parameters, and the `ap` will find the supposed next parameter as there is another `%d`, regardless of whether you really put a parameter there.

Question 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 so that it would still be possible to pass it a variable number of arguments?

Answer: Method 1: modify `va_start` and `va_arg` so that it can read parameters in a reversed order.

Method 2: add a counter of the number of parameters in the `cprintf` and then do recursion.

Exercise 10

add `case 'n':` in `printfmt.c`.

```
        case 'n': {
```

```
            const char *null_error = "\nerror! writing through NULL pointer! (%n argument)\n";
            const char *overflow_error = "\nwarning! The value %n argument pointed to has been overflowed!\n";

            char* pos;
            if ((pos = va_arg(ap, char *)) == NULL){
                printfmt(putch, putdat, "%s", null_error);
            }else if (*((unsigned int *)putdat)>254){
                printfmt(putch, putdat, "%s", overflow_error);
                *pos = -1;
            }else{
                *pos = *(char *)putdat;
            }
            break;
        }
```

Exercise 11

Firstly calculate the width of the number in the specified base first. Then reuse `printnum` function to print the number on the left normally. Finally print out the padding spaces.

```
printnum(void (*putch)(int, void*), void *putdat, unsigned long long num, unsigned base, int width, int padc){
    if (padc == '-') {
        int num_width = 0;
        int temp = num;
        while(temp > 0) {
            num_width += 1;
            temp /= base;
        }
        printnum(putch, putdat, num, base, num_width, ' ');
        for (int i = 0; i < width - num_width; ++i){
            putch(' ', putdat);
        }
        return;
    }
    ...
```

Exercise 12

In `kern/entry.S`, there are two lines initializes the stack.

```
movl    $0x0,%ebp               # nuke frame pointer
movl    $(bootstacktop),%esp    # Set the stack pointer
```

Use gdb, we can find the initial value of `%esp` is 0xf0110000.

The stack grows from the high address to low address, so the stack pointer points to the higher end when it is initialized.

Exercise 13

`test_backtrace` starts at 0xf0100040. Set breakpoint there and use gdb to print out the stack.

```
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 2, test_backtrace (x=0) at kern/init.c:13
13  {
(gdb) si
=> 0xf0100041 <test_backtrace+1>:   mov    %esp,%ebp
0xf0100041  13  {
(gdb) x/48x $esp
0xf010ff58: 0xf010ff78  0xf0100068  0x00000001  0x00000002
0xf010ff68: 0xf010ff98  0x00000000  0xf010089d  0x00000003
0xf010ff78: 0xf010ff98  0xf0100068  0x00000002  0x00000003
0xf010ff88: 0xf010ffb8  0x00000000  0xf010089d  0x00000004
0xf010ff98: 0xf010ffb8  0xf0100068  0x00000003  0x00000004
0xf010ffa8: 0x00000000  0x00000000  0x00000000  0x00000005
0xf010ffb8: 0xf010ffd8  0xf0100068  0x00000004  0x00000005
0xf010ffc8: 0x00000000  0x00010094  0x00010094  0x00010094
0xf010ffd8: 0xf010fff8  0xf01000d4  0x00000005  0x00001aac
0xf010ffe8: 0x00000684  0x00000000  0x00000000  0x00000000
0xf010fff8: 0x00000000  0xf010003e  0x00111021  0x00000000
0xf0110008 <entry_pgdir+8>: 0x00000000  0x00000000  0x00000000  0x00000000
```

Each two line represents one stack frame. The first value is the `%ebp` pushed into stack. The third value is the parameter, it reduces from 5 to 0.

Question: Why can't the backtrace code detect how many arguments there actually are?

Answer: The backtrace cannot detect the number of parameter because there is no information stored on stack to show this. The function can know it because the wanted parameter is got by `offset(%ebp)`.

Exercise 14

```c
int mon_backtrace(int argc, char **argv, struct Trapframe *tf){
    for (uint32_t *ebp=(uint32_t *)read_ebp(); ebp!=NULL; ebp = (uint32_t *)(*ebp)){
        cprintf("eip %8x  ebp %8x  args %08x %08x %08x %08x %08x\n",
                ebp[1], ebp, ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
    }
    overflow_me();
    cprintf("Backtrace success\n");
    return 0;
}
```

The terminating condition is `ebp == 0` because in `kern/entry.S`, the `%ebp` was set to 0 by `movl    $0x0,%ebp` at first.

`ebp[1]` is the value of required `%eip` because `call` instruction will always push the return address into the stack before jump to the target function.

Exercise 15

`__STAB_*` comes from the symbol table, which is a part of the ELF binary file. This part is preserved by giving specific parameters when compiling. The boot loader do load the symbol table into the memory, which is the basis for this exercise. Otherwise, we can't have access to the symbols.

Use `stab_binsearch` to search for line number of the code in `kern/kdebug.c`'s function `debuginfo_eip`.

```
    stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
    if (lline <= rline){
        info->eip_line = stabs[lline].n_desc;
    }else{
        return -1;
    }
```

Then modify the corresponding part in `monitor.c`. Just add the several lines to print out the line number.

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf){
    struct Eipdebuginfo eipinfo;
    for (uint32_t *ebp=(uint32_t *)read_ebp(); ebp!=NULL; ebp = (uint32_t *)(*ebp)){
        cprintf("  eip %8x  ebp %8x  args %08x %08x %08x %08x %08x\n",
                ebp[1], ebp, ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
        debuginfo_eip(ebp[1], &eipinfo);
        cprintf("         %s:%d %.*s+%d\n",
            eipinfo.eip_file, eipinfo.eip_line, eipinfo.eip_fn_namelen,
            eipinfo.eip_fn_name, ebp[1] - eipinfo.eip_fn_addr);
    }
    overflow_me();
    cprintf("Backtrace success\n");
    return 0;
}
```

In order to make `backtrace` a command that can be executed in the terminal window, we must add this command into `commands[]` in `kern/monitor.c`.

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "backtrace", "Introduction of backtrace command", mon_backtrace},
};
```

Exercise 16

Examine `obj/kern/kenel.asm`, we can find the address of `do_overflow`.

```
f0100827 <do_overflow>:
    return pretaddr;
}
```

But as we are "attacking" from the source file itself, so we can just perform a coercion type conversion to get the address of the target function.

Then we simply do a "format-string attack" in the function `start_overflow`.

```
void start_overflow(void)
{
    char str[256] = {};
    int nstr = 0;

    char * pret_addr = (char *) read_pretaddr();
    uint32_t overflow_addr = (uint32_t) do_overflow;
    for (int i = 0; i < 4; ++i){
        cprintf("%*s%n\n", pret_addr[i] & 0xFF, "", pret_addr + 4 + i);
        cprintf("%*s%n\n", (overflow_addr >> (8*i)) & 0xFF, "", pret_addr + i);
    }
}
```

We should first move the return address for `start_overflow` to the return address of our to-be-executed `do_overflow`, otherwise the `%eip` will go to some random memory address. Then we modify the return address of `start_overflow` to lead our control flow go to `do_overflow`. These two steps are done byte by byte in my code. The two steps cannot be done in reversed order because once we cover the return address of `start_overflow`, we cannot find the value that we want to write into `do_overflow`'s return address.

Exercise 17

Add the statement into `kern/monitor.h`.

```
int mon_time(int argc, char **argv, struct Trapframe *tf);
```

Then implement it in `kern/monitor.c`.

```
int mon_time(int argc, char **argv, struct Trapframe *tf){
    if (argc != 2){
        cprintf("Where the fuck is command?\n");
        return -1;
    }

    uint64_t start_time, end_time;
    int inst_idx = 0;
    while (inst_idx < ARRAY_SIZE(commands) && strcmp(commands[inst_idx].name, argv[1])){
        inst_idx++;
    }
    if (inst_idx == ARRAY_SIZE(commands)){
        cprintf("What the fuck is your command?\n");
        return -1;
```

```
    }

    start_time = read_tsc();
    (commands[inst_idx].func)(1, argv+1, tf);
    end_time = read_tsc();
    cprintf("%s cycles: %d\n", commands[inst_idx].name, end_time - start_time);
    return 0;
}
```