Jos Lab 1

Deng Wen 17307130171

Exercise 1

This exercise is to read the 《PC Assembly Language》 book and the 《Brennan's Guide to Inline Assembly》guide to know more information about PC assembly language and the most important features of GNU assembler syntax.

Exercise 2

I have got the following instructions by using si:

```
[f000:fff0]     0xffff0: ljmp   $0x3630,$0xf000e05b
```

CS（CodeSegment）and IP（Instruction Pointer）are used to get the address of next instruction. Formula： physical address = 16 * segment + offset.

When PC begin to work，CS = 0x3630，IP = 0xf000e05b，the frist instruction does a jmp operation，jump to address of 16 * segment + offset.

```
[f000:e05b]     0xfe05b: cmpw   $0xffc8,%cs:(%esi)
[f000:e062]     0xfe062: jne    0xd241d416
[f000:e066]     0xfe066: xor    %edx,%edx
[f000:e068]     0xfe068: mov    %edx,%ss
[f000:e06a]     0xfe06a: mov    $0x7000,%sp
[f000:e070]     0xfe070: mov    $0x2d4e,%dx
[f000:e076]     0xfe076: jmp    0x5575ff02
[f000:ff00]     0xfff00: cli
[f000:ff01]     0xfff01: cld
```

CLI：Clear Interupt，which prohibits interrupt. STL: set interrupt, which allows interrupts to occur. CLI and STI are used to mask interrupts and recover interrupts. For example, CLI is required when setting stack base address SS and offset address SP, because if these two instructions are separated, it is likely that SS has been modified, but because of interrupts and code jumps to other places for execution, if SP has not yet been modified, errors may occur .

CLD: Clear Director。 STD：Set Director。 Used in line block transfers, they determine the direction of block transfers. CLD makes the transmission direction from low address to high address, while STD is the opposite.

LIDT: load interrupt descriptor. Lgdt: load global descriptor.

```
[f000:ff02]    0xfff02: mov    %ax,%cx
[f000:ff05]    0xfff05: mov    $0x8f,%ax
[f000:ff0b]    0xfff0b: out    %al,$0x70
[f000:ff0d]    0xfff0d: in     $0x71,%al
```

When CPU communicates with external devices, it is usually realized by accessing and modifying registers in device controller. The registers in the device controller are also called IO ports. For the convenience of management, 80x86 CPU adopts the way of IO port addressing separately, that is, all the ports of devices are named into an IO port address space. This space is independent of the memory address space. Therefore, it is necessary to use different instructions to access the port from the instructions to access the memory.

In %al , portaddress writes a value to the port with the port address of portaddress, which is the value in the Al register
Out portaddres,%al reads the value in the port whose port address is portaddress into register al

```
[f000:ff0f]    0xfff0f: in     $0x92,%al
[f000:ff11]    0xfff11: or     $0x2,%al
[f000:ff13]    0xfff13: out    %al,$0x92
[f000:ff15]    0xfff15: mov    %cx,%ax
[f000:ff18]    0xfff18: lidtl  %cs:(%esi)
[f000:ff1e]    0xfff1e: lgdtl  %cs:(%esi)
[f000:ff24]    0xfff24: mov    %cr0,%ecx
[f000:ff27]    0xfff27: and    $0xffff,%cx
[f000:ff2e]    0xfff2e: or     $0x1,%cx
[f000:ff32]    0xfff32: mov    %ecx,%cr0
[f000:ff35]    0xfff35: ljmpw  $0xf,$0xff3d
```

We need to know first that when the PC starts, the CPU runs in real mode, and when it enters the operating system kernel, it will run in protected mode.
The instructions before are mainly to set the value of %ss and %esp registers, open A20 gate, and enter a protect model.

As the first program running after PC startup, its most important function is to import the operating system from disk into memory, and then transfer the control to the operating system. So at the end of running, BIOS will check which device can find the operating system from the current system, usually our disk. Maybe it's a U-disk and so on. When the BIOS determines that the operating system is located in the disk, it will load the first sector of the disk, usually called boot sector, into the memory first. This boot sector includes a very important program -- boot loader, which is responsible for the import of the entire operating system from the disk into memory, as well as some other very important configurators Do. Finally, the operating system will start to run.

Exercise 3

First we look at file boot.s:

```
#include <inc/mmu.h>

# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
```

These instructions are the first few sentences of boot. S, among which cli is boot. S and the first instruction of boot loader. This command is used to turn off all interrupts. Because it is possible that the interrupt was turned on during BIOS operation. At this time, the CPU works in real mode.

```
  cld                         # String operations increment
```

This instruction specifies the direction of pointer movement for subsequent string processing operations.

```
  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

These commands are mainly to clear all three segment registers, DS, ES, SS. Because of BIOS, the operating system can not guarantee what number is stored in these three registers. So this is also to prepare for entering the protection mode later.

```
  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
```

```
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
```

Seta20.1 and seta20.2 realize the function of opening A20 door. Seta20.1 sends 0x61 command to 0x64 port of keyboard controller, which means to write data to P2 port of keyboard controller; seta20.2 writes data to P2 port of keyboard controller. The way to write data is to write data through the 0x60 port of the keyboard controller. The data to be written is 0xdf, because A20 gate is included in P2 port of keyboard controller. With the writing of 0xdf, A20 gate is opened.

test performs and logical operation on two parameters (target, source), and sets flag register according to the result. The result itself will not be saved.

```
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
```

```
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain

  # If bootmain returns (it shouldn't), loop.
spin:
  jmp spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL              # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)           # data seg

gdtdesc:
  .word   0x17                            # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

GDT is the global descriptor table and gdtr is the global descriptor table register. In order to address memory in "protected mode", GDT is required. Each item in GDT table is called "segment descriptor", which is used to record some attribute information of each memory segment. Each segment descriptor takes up 8 bytes. The CPU uses the gdtr register to hold our GDT position in memory and the length of GDT. Lgdt gdtdesc loads the value of the source operand (stored in the gdtdesc address) into the global descriptor table register.

What does an operating system do after the computer starts up :
1. The computer is powered on, and the operating environment is 1MB addressing limit band "winding" mechanism
2. Open A20 gate to make the computer break through 1MB addressing limit
3. Create a GDT global descriptor table in memory and tell the CPU the location and size of the created GDT table
4. Set the control register and enter the protection mode
5. Continue to execute according to memory addressing mode of protection mode

Now we look at file main.c:

```
#include <inc/x86.h>
```

```
#include <inc/elf.h>

/**********************************************************************
 * This a dirt simple boot loader, whose sole job is to boot
 * an ELF kernel image from the first IDE hard disk.
 *
 * DISK LAYOUT
 *  * This program(boot.S and main.c) is the bootloader.    It should
 *    be stored in the first sector of the disk.
 *
 *  * The 2nd sector onward holds the kernel image.
 *
 *  * The kernel image must be in ELF format.
 *
 * BOOT UP STEPS
 *  * when the CPU boots it loads the BIOS into memory and executes it
 *
 *  * the BIOS intializes devices, sets of the interrupt routines, and
 *    reads the first sector of the boot device(e.g., hard-drive)
 *    into memory and jumps to it.
 *
 *  * Assuming this boot loader is stored in the first sector of the
 *    hard-drive, this code takes over...
 *
 *  * control starts in boot.S -- which sets up protected mode,
 *    and a stack so C code then run, then calls bootmain()
 *
 *  * bootmain() in this file takes over, reads in the kernel and jumps to it.
 **********************************************************************/

#define SECTSIZE    512
#define ELFHDR      ((struct Elf *) 0x10000) // scratch space

void readsect(void*, uint32_t);
void readseg(uint32_t, uint32_t, uint32_t);
```

ELF file: elf is a file format used to store programs on disk. It is created after the program has been compiled and linked. An ELF file contains multiple segments. For an executable program, it usually contains the text section for storing code, the data section for storing global variables, and the rodata section for storing string constants. The header of an ELF file is used to describe how the ELF file is stored in memory. It should be noted that if your file is a linkable file or an executable file, there will be different elf header formats.

```
void
bootmain(void)
{
	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);

	// call the entry point from the ELF header
	// note: does not return!
	((void (*)(void)) (ELFHDR->e_entry))();

bad:
	outw(0x8A00, 0x8A00);
	outw(0x8A00, 0x8E00);
	while (1)
		/* do nothing */;
}
```

This function is easy to understand with the help of the comment in them.

Readseg is to read the contents of the first page of the kernel (4MB = 4096 = sectsize * 8 = 512 * 8) to the memory address elfhdr (0x10000). In fact, after these are completed, it is equivalent to reading the elf header of the operating system image file and putting it into memory.

```
// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
	uint32_t end_pa;
```

```
	end_pa = pa + count;

	// round down to sector boundary
	pa &= ~(SECTSIZE - 1);

	// translate from bytes to sectors, and kernel starts at sector 1
	offset = (offset / SECTSIZE) + 1;

	// If this is too slow, we could read lots of sectors at a time.
	// We'd write more to memory than asked, but it doesn't matter --
	// we load in increasing order.
	while (pa < end_pa) {
		// Since we haven't enabled paging yet and we're using
		// an identity segment mapping (see boot.S), we can
		// use physical addresses directly.  This won't be the
		// case once JOS enables the MMU.
		readsect((uint8_t*) pa, offset);
		pa += SECTSIZE;
		offset++;
	}
}

void
waitdisk(void)
{
	// wait for disk reaady
	while ((inb(0x1F7) & 0xC0) != 0x40)
		/* do nothing */;
}

void
readsect(void *dst, uint32_t offset)
{
	// wait for disk to be ready
	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

	// wait for disk to be ready
```

```
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE/4);
}
```

This function mainly does three things: wait for the disk (waitdisk), output the number of sectors and address information to the port (out), and read the sector data (Insl).

1. Wait for the disk. The function implementation of waitdisk is as follows. In fact, it does one thing: read bit 7 and bit 6 of port 0x1fc until bit 7 = 0 and bit 6 = 1. According to reference 1, port 1f7 is used as a status register when it is read, where bit 7 = 0 means the controller is idle, bit 6 = 1 means the driver is ready. Therefore, waitdisk does not end the wait until the controller is idle and the drive is ready. `
2. Output data to the port. According to reference 1, IDE defines 8 registers to operate the hard disk. The PC architecture maps the first hard disk controller to port 1f0-1f7, while the second hard disk controller is mapped to port 170-177. The out function outputs the sector count, LBA address and other information to port 1f2-1f6, and then writes the 0x20 command to 1f7, indicating that the sector reading operation is to be performed.
3. Read sector data. Insl function is mainly used. Its implementation is an inline assembly statement. Insl function essentially reads 128 dwords (512 bytes, that is, the number of bytes in a sector) from 0x1F0 port to the destination address. Among them, 0x1F0 is the data register, which must be used to read and write hard disk data.

For the three questions in the lab pdf:

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- 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?

1. In the boot.S file, the computer first works in real mode, and at this time it works in 16bit mode. When the "ljmp $ PROT_MODE_CSEG, $ protcseg" statement is run, it officially enters the 32-bit working mode. The root cause is that the CPU is working in protected mode at this time.
2. The last statement executed by the boot loader is the last statement in the bootmain subroutine "((void (*) (void)) (ELFHDR-> e_entry)) ();", which is the jump to the beginning of the operating system kernel program Instruction. First sentence movw $ 0x1234, 0x472.
3. First, information about how many segments are in the operating system and how many sectors are in each segment is located in the Program Header Table in the operating system file. Each table entry in this table corresponds to a segment of the operating system. And the contents of each table item include the size of the

segment, the offset of the starting address of the segment, and so on. So if we can find this table, we can use the information provided by the table items to determine how many sectors the kernel occupies.The information about where this table is stored is in the ELF header information of the operating system kernel image file.

Exercise 4

The questions consist of two parts: read The C Programming Language, and run pointers. C and understand The output.

After review the knowledge of C progamming language I ran the pointers.c file and get the following result:

```
1: a = 0x7fff150d9540, b = 0x5617ec351260, c = 0xf0b6ff
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fff150d9540, b = 0x7fff150d9544, c = 0x7fff150d9541
```

Analysis:

```
#include <stdio.h>
#include <stdlib.h>

void
f(void)
{
	int a[4];
	int *b = malloc(16);
	int *c;
	int i;

	printf("1: a = %p, b = %p, c = %p\n", a, b, c);

	c = a;
	for (i = 0; i < 4; i++)
	a[i] = 100 + i;
	c[0] = 200;
	printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
	    a[0], a[1], a[2], a[3]);

	c[1] = 300;
	*(c + 2) = 301;
```

```
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
        a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
        a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
        a[0], a[1], a[2], a[3]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}

int
main(int ac, char **av)
{
    f();
    return 0;
}
```

1. The first line print the address that a, b, c pointers are refering to, a refers to the memory address of array a; b refers to a area which is newly assigned. C refers to a uncertain area.Note: in this file, not usual to the online blog, the first line's output is "a, b, c" not "&a, &b, &c". And if add &, the result is:

```
1: &a = 0x7ffc53262a30, &b = 0x7ffc53262a20, &c = 0x7ffc53262a28
```

In which &a > &b > &c.

2. The second line print the value in array a. The values were set in the previous codes by letting c point to the first element of array a, using a for loop to set array a with 100 101 102 103 and then setting c[0] equal to 200(also set a[0] equal to 200). It is easy to understand.

3. Based on the how pointers work in C language: c[1] represent the value of a[1], *(c+2) represent the value of a[2], 3[c] represents the value of a[3]. So the codes before the print function is just doing assignment operation. The third line is very clear then.

4. Before print the fourth line, the program set c equal to c+1 which means c points to the next element. That is means c actually now refers to the address of the second element of array a. So *c = a[1]. Then set a[1] equal to 400. So the fourth line is reasonable.

5. In the beginning, *c represent the value ： 0000 0000 0000 0000 0000 0001 1101 0000

*(c+1) represent the value : 0000 0000 0000 0000 0000 0001 0010 1101
Now doing “c = (int *) ((char *) c + 1);” is to make c point to part of a[1] and part of a[2]:

xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
|--------------------a[2]-------------------------|-------------------------a[1]----------------------|
|-------------------------*c-------------------------|

And 500 is 0000 0000 0000 0000 0000 0001 1111 0100.
So after set *c = 500, the result is:
a[1]: 0000 0000 0000 0001 1111 0100 1001 0000 = 128144
a[2]: 0000 0000 0000 0000 0000 0001 0000 0000 = 256
6. Since I have explained in the 5th line, the reason why the 6th line print like this is similar to above. Just the 6th line the to print the address not the value in the address. The both tell us the memory offset for Pointers plus or minus an integer depends on the type of pointer.

This is the a full list of the names, sizes, and link addresses of all the sections in the kernel executable.

```
obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001af9  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000764  f0101b00  00101b00  00002b00  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003ccd  f0102264  00102264  00003264  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000019e5  f0105f31  00105f31  00006f31  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00009300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000008  f0111300  00111300  00012300  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .got.plt      0000000c  f0111308  00111308  00012308  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.local 00001000  f0112000  00112000  00013000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  8 .data.rel.ro.local 00000044  f0113000  00113000  00014000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          00000648  f0113060  00113060  00014060  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 10 .comment      0000002b  00000000  00000000  000146a8  2**0
                  CONTENTS, READONLY
```

This is the memory address in the program's text section at which the program should begin executing.

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

Exercise 5

We can set the breakpoint at 0x7c00 because this is where the BIOS enters the boot loader. And then examie the 8 words in 0x100000. Then set breakpoint at 0x00100000c where the the boot loader has entered the kernel. Again examie the 8 words in 0x100000 The following is the process:

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

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb) x/8i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
   0x10000d:    movl   $0xb81234,0x472
   0x100017:    and    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)

   0x10001b:    and    %al,%bl
```

We should know:
0x7c00，is the address where the BIOS loads the boot sector.
0x100000 is the address where the Kernel is finally loaded.

Thus, the kernel is loaded by the boot loader, initially when the BIOS switches to the boot loader, it has not started the corresponding load, so at this time all 8 words are 0. When the boot loader enters the kernel to run, the kernel has been loaded at this time, so starting from 0x1000000 is the file content of the kernel ELF file. (ELF HEADER)

Exercise 6

This problem hopes that we can modify the link address of boot loader, the author introduced two concepts, one is the link address, one is the loading address. A link address can be understood as the address of an instruction in an executable program that is processed by the compiler linker, the logical address. The loading address is the physical address at which the executable is actually loaded into memory.

So first we modify the file boot/Makefrag to set -Ttext equal to 0x7c40 instead of 0x7c00. Then run make clean and make, to see what happens:

First we look at the obj/boot/boot.asm file to see that the link address has be modified from 0x7c00 to 0x7c40.

Modified:

```
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
    7c40:       fa                    cli
  cld                         # String operations increment
    7c41:       fc                    cld
```

Original:

```
.globl start
start:
  .code16                      # Assemble for 16-bit mode
  cli                          # Disable interrupts
    7c00:       fa                    cli
  cld                          # String operations increment
    7c01:       fc                    cld
```

We can see the link address has been changed.

Then run make gdb , set breakpoint at 0x7c2d (where the boot loader jump to the kernel) and do c or si command:

```
(gdb) b *0x7c2d
Breakpoint 1 at 0x7c2d
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c72

Breakpoint 1, 0x00007c2d in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c72

Breakpoint 1, 0x00007c2d in ?? ()
```

We can see that the Boot Loader could not jump into the kernel and get struck in a loop because the link address is not the same as the load address which makes the jump instruction jumps to a wrong address.

Then we modify the value of -Ttest to the original value 0x7c00 and make clean and make, to see what happens:

```
(gdb) b *0x7c2d
Breakpoint 1 at 0x7c2d
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c32

Breakpoint 1, 0x00007c2d in ?? ()
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
The target architecture is assumed to be i386
=> 0xf010029c <serial_proc_data+24>:    mov    $0xffffffff,%eax
serial_proc_data () at kern/console.c:54
54                      return -1;
```

We can see that Boot Loader jumps into the kernel correctly.
So the inconformity between load address and link address may cause the program goes wrong in this JOS.

Exercise 7

In this part we already know, In this experiment, a paging management approach is used to implement the address mapping described above. But instead of the usual paging mechanism used on computers, the designers wrote a program, lab kern\entrygdir.c, to do the mapping. Only the address range of the virtual address space, 0xf0000000~0xf0400000, can be mapped to the physical address range, 0x00000000~0x00400000. The virtual address range: 0x00000000~0x00400000 can also be mapped to the physical address range: 0x00000000~0x00400000. Any address that is no longer in the range of these two virtual addresses will cause a hardware exception. Although it can only map these two small Spaces, it is enough to start the program.

To find where the mapping takes effect, first we see the kern/entry.S file and the obj/kern/kernel.asm file to see how they are doing.
entry.S:

```
.globl entry
entry:
        movw    $0x1234,0x472                   # warm boot

        # We haven't set up virtual memory yet, so we're running from
        # the physical address the boot loader loaded the kernel at: 1MB
        # (plus a few bytes).  However, the C code is linked to run at
        # KERNBASE+1MB.  Hence, we set up a trivial page directory that
        # translates virtual addresses [KERNBASE, KERNBASE+4MB) to
        # physical addresses [0, 4MB).  This 4MB region will be
        # sufficient until we set up our real page table in mem_init
        # in lab 2.

        # Load the physical address of entry_pgdir into cr3.  entry_pgdir
        # is defined in entrypgdir.c.
        movl    $(RELOC(entry_pgdir)), %eax
        movl    %eax, %cr3
        # Turn on paging.
        movl    %cr0, %eax
        orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
        movl    %eax, %cr0

        # Now paging is enabled, but we're still running at a low EIP
        # (why is this okay?).  Jump up above KERNBASE before entering
        # C code.
        mov     $relocated, %eax
        jmp     *%eax
```

This file is similar to the following asmemble file kernel.asm.
And we can see the address of every instruction for us to set breakpoint.

```
f010000c <entry>:
f010000c:       66 c7 05 72 04 00 00    movw   $0x1234,0x472
f0100013:       34 12
        # sufficient until we set up our real page table in mem_init
        # in lab 2.

        # Load the physical address of entry_pgdir into cr3.  entry_pgdir
        # is defined in entrypgdir.c.
        movl    $(RELOC(entry_pgdir)), %eax
f0100015:       b8 00 20 11 00          mov    $0x112000,%eax
        movl    %eax, %cr3
f010001a:       0f 22 d8                mov    %eax,%cr3
        # Turn on paging.
        movl    %cr0, %eax
f010001d:       0f 20 c0                mov    %cr0,%eax
        orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
f0100020:       0d 01 00 01 80          or     $0x80010001,%eax
        movl    %eax, %cr0
f0100025:       0f 22 c0                mov    %eax,%cr0

        # Now paging is enabled, but we're still running at a low EIP
        # (why is this okay?).  Jump up above KERNBASE before entering
        # C code.
        mov     $relocated, %eax
f0100028:       b8 2f 00 10 f0          mov    $0xf010002f,%eax
        jmp     *%eax
f010002d:       ff e0                   jmp    *%eax
```

From this code we can see the boot loader defines the GDT (Global Descirptor Table) during initialization, replacing the original GDT, which is moved to physical memory [0,4MB] and placed in cr3.

It is the code

```
f0100020:       0d 01 00 01 80          or     $0x80010001,%eax
        movl    %eax, %cr0
f0100025:       0f 22 c0                mov    %eax,%cr0
```

So we run gdb again and set breakpoint at 0x1000c cause this is the address of the kernel entrance, and do si instruction until we reach the “mov %eax, %cr0” instruction

```
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
(gdb) si
=> 0x100015:    mov    $0x112000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4bx 0x00100000
0x100000:       0x02    0xb0    0xad    0x1b
(gdb) x/4bx 0xf0100000
0xf0100000 <_start+4026531828>: 0x00    0x00    0x00    0x00
```

We can see that before doing the instruction the two address 0x00100000 and 0xf0100000 are mapping to differet address.

Then do si and check again:

```
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4bx 0xf0100000
0xf0100000 <_start+4026531828>: 0x02    0xb0    0xad    0x1b
(gdb) x/4bx 0x00100000
0x100000:       0x02    0xb0    0xad    0x1b
```

We can see that now, the two address are mapping into same address. That's where the mapping taking effect exactly as what we have guessed before.

Then comment out the code in kernel/entry.S file:

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

And make clean, make , doing the same operation like before:

```
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
(gdb) si
=> 0x100015:    mov    $0x112000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    mov    $0xf0100027,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    jmp    *%eax
0x00100025 in ?? ()
(gdb) si
=> 0xf0100027 <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
```

We can see at this time, the program went wrong because there is no paging management, and virtual to physical address conversion is not performed at this point. In the picture the 0xf01000027 is beyond the memory.

Exercise 8 & 9

In this exercise, we will first read the code of the following three source files and figure out the relationship between them.

\kern\printf.c，\kern\console.c, \lib\printfmt.c

Befor doing so we can answer question in later part first:

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`?

After a brief scan, we found that:

1.cprintf, vcprintf subroutines in printf.c called vprintfmt in printfmt.c

2.putch subroutines in printf.c called cputchar in console.c

3.some programs in printfmt.c also called cputchar in console.c

So we first look into console.c:

```
// `High'-level console I/O.  Used by readline and cprintf.

void
cputchar(int c)
{
        cons_putc(c);
}
```

```
// output a character to the console
static void
cons_putc(int c)
{
        serial_putc(c);
        lpt_putc(c);
        cga_putc(c);
}
```

So we can know from the comment that cputchar is to output a character to the screen.

Then we look into file printfmt.c:

First take a look at the comment at the beginning of the file:

```
// Stripped-down primitive printf-style formatting routines,
// used in common by printf, sprintf, fprintf, etc.
// This code is also used by both the kernel and user programs.
```

From this comment, we know that the subroutine defined in this file is the key to being able to print information directly to the screen using printf functions at programming time.

The function we are interested is “vprintfmt”:

(this function’s code is too long, so I don’t put it here.)

This function has four parameters:

(1) void (*putch)(int, void*):

This parameter is a function pointer. This type of function contains two input parameters int, void *, and the int parameter represents the value of a character to be output. Void * represents the address of the location where the character is to be output, but the value of the void * parameter is not the address, but the address of the storage unit where the value of the address is stored. For example, I want to output a character with a character value of 0x30 ('0 ') to the address 0x01. At this time, our program should be as follows:

```
1 int addr = 0x01;
2 int ch = 0x30;
3 putch(ch, &addr);
```

The reason for this is that this subroutine can realize that after the value is stored in this address, the address number will automatically increase by 1, that is, after the above code is executed, the value at 0x01 memory will change to 0x30, and the value of addr will change to 0x02

(2) void *putdat:

This parameter is the pointer to the memory address where the input character is to be

stored, which is the same meaning as the second input parameter of the putch function above.

(3) const char *fmt:

This parameter represents that when you write a formatted output program like printf, you specify the format string, that is, the first input parameter of the printf function, such as printf ("this is% d test", n). In this subprogram, fmt is "this is %d test".

(4) va_list ap:

This parameter represents multiple input parameters, that is, the parameters in the printf subroutine starting from the second parameter, such as ("these are %d test and %d test", n, m). Then ap refers to n, m.

The execution process of this function is mainly a while loop, which is divided into the following steps:

1.First of all, all the characters before '%' in the output format string FMT of one will be output directly, such as' this is %d test '. Of course, if you encounter the end character '\ 0' in the output of these characters one by one, the output will be ended.

2. The rest of the code deals with the formatted output after the '%' symbol. For example, if it is% D, the corresponding parameter will be output in decimal. In addition, there are other special characters such as '%5d 'for display of 5 digits, of which 5 need special processing.

And this program is just what the exercise 8 and 9 let us add. In of the source program, here is the code to process the display of octal format:

For exercise 8, we can implement it based on the hex situation:

```
case 'o':
        // Replace this with your code.
        putch('0', putdat);
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

For exercise 9, we can add the following code to realize the function:

```
    switch (ch = *(unsigned char *) fmt++) {
    case '+':
            padc = '+';
            altflag = 1;
            goto reswitch;

    // flag to pad on the right
```

and

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

Then we have finished exercise 8 and 9.

Finally we look into file printf.c:
This file defines the top-level formatted output subroutines we will use in programming, such as printf, sprintf, etc.
Now we check the cprintf subprogram:

```
int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);
        cnt = vcprintf(fmt, ap);
        va_end(ap);

        return cnt;
```

In cprintf, we find that by using the operations of va_list, va_arg(), va_start(), va_end(), the input parameters after fmt of cprintf are all converted to a parameter of va_list type, and then fmt and the newly generated ap are passed to vcprntf as parameters.
Then for the putch function:

```
static void
putch(int ch, int *cnt)
{
        cputchar(ch);
        (*cnt)++;
}
```

It calls the subroutine we first analyzed, cputchar, which can output characters to the screen. So this pull subroutine meets the requirements of vprintfmt subroutine.
Finally, look at the second parameter. This parameter does not have the meaning of memory address here. We can see that in the pull, it just outputs the characters to the screen, and then adds 1 to this CNT. It does not store the characters to the address that CNT points to, so this CNT becomes a counter. Record how many characters have been output.

For these questions in the jos01.pdf, we now answer it one by one:
Question 1 has already be answered in the previous pages, so we skip this.

2. Explain the following from `console.c`:

```
1       if (crt_pos >= CRT_SIZE) {
2               int i;
3               memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4               for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                       crt_buf[i] = 0x0700 | ' ';
6               crt_pos -= CRT_COLS;
7       }
```

crt_buf: This is a character array buffer, which contains the characters to be displayed on the screen.
crt_pos: This indicates the position of the last character displayed on the screen. Before introducing this variable, we need to know some knowledge. This is what I inquired on the Internet.
Early computers could only display information to users through text mode, for example, when you turn on the computer now, before entering the desktop, all information is displayed on the screen through text. So this mode is called text mode, so this console. C source program considers a very common text mode, 80x25 text mode, that is, the whole screen can display up to 25 lines of characters, and each line can display up to 80 characters. So a total of 80x25 positions. When we want to display a specific character to a position on the screen, we must specify the display position and display character to the screen driver cga.
In the console. c file, the subroutine cga_putc (int c) completes this function and displays the character c to the next position of the current display on the screen. For example, three lines of data (line 0, line 1, line 2) have been displayed in the current screen, and the third line has displayed 40 characters. At this time, execute cga_putc (0x65), and the corresponding character 'a' of 0x65 will be displayed in the 41st character of line 2. So cga_putc needs two variables, crt_buf, a character array pointer, which is all the characters currently displayed on the screen. crt_pos - indicates the position of the next character to be displayed in the array. In fact, through this value, we can deduce its position on the screen. For example, if crt_pos = 85, it should be displayed at line 2 (i.e. line 1) and character 6 (character 5). Therefore, the value range of crt_pos should be 0 ~ (80 * 25-1).
The code to be analyzed in the above topic is located in cga_putc. cga_putc is divided into three parts. The first part is to determine what to display according to the character value int C. The second part is the above code. The third part is to display the characters you decide to display to the designated position on the screen. Let's analyze the second part,
When crt_pos > = CRT_SIZE, where CRT_SIZE = 80 * 25, because we know that the value range of crt_pos is 0 ~ (80 * 25-1), if this condition is true, then the output content on the screen has exceeded one page. So at this time, you need to scroll the page up one line, that is, put the original line 1-79 on the current line 0-78, and then change the line 79 into a line of space (of course, not all of them are spaces, and the character int c you input should be displayed on the character 0). So the memcpy operation is to copy

the contents of lines 1-79 in the crt_buf character array to the positions of lines 0-78. The next for loop turns the last line, line 79, into spaces. Finally, you need to change the value of crt_pos.

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

Recall cprintf function:

```
int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);
        cnt = vcprintf(fmt, ap);
        va_end(ap);

        return cnt;
}
```

To answer the first question, fmt naturally points to the format string for displaying information. In this code, it points to the string "x %d, y %x, z %d\n". The ap is of the va_list type. As we have shown before, the number of input parameters of this type is variable. So ap points to a collection of all input parameters.
Then enter vprintfmt subroutine. We have analyzed this subroutine. No more details here. The working process of this subroutine is to continuously analyze the format string FMT. The way of analysis is to divide the format string into several parts. Each part has at most one parameter to be displayed. For example, the format string in our problem can be divided into four parts:

"x %d", ", y %x" , ", z %d", "\n"

Then analyze the string before % in each part and output it directly. For example, "X" in "x %d". Then analyze the content after the% number. For example, the analysis result in "x %d" is to display a parameter in decimal. Every time the content after the% number is analyzed, the program will perform different operations according to the analysis results. After analyzing "x %d", the code begins to take the following branch:

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

First of all, this branch is a subfunction getint. The contents of this subfunction are as follows:

```
static long long
getint(va_list *ap, int lflag)
{
        if (lflag >= 2)
                return va_arg(*ap, long long);
        else if (lflag)
                return va_arg(*ap, long);
        else
                return va_arg(*ap, int);
}
```

It can be seen that according to different parameter types, the next parameter will be taken out from the ap parameter list by using the va_arg method, and the code in line 9 will be executed in our example. Here, a call is made to va_arg. Before the call, ap includes three parameters: x, y, z: 1, 3, 4. After the call, only y and z are left: 3, 4.

Back to vprintfmt, now num stores the value 1 to be displayed. The next step is to determine whether the value to be displayed is negative. If it is negative, the putch function should be called first to display a negative sign on the screen. Then jump to number position.

number is a subroutine printnum (putch, putdat, num, base, width, padc). This subroutine will display the parameter 1 you just got according to the specified base and format. In this subroutine, we can see that it will display the parameter value (num = 1) you get according to your specified base (base = 10), one by one. So every time it gets the value of a bit, it will call a putch and display it on the screen. In addition, this code putch (padc, putdat) is to realize that when the display needs right alignment, the left side should be filled with space first.

So the first parameter x = 1 is displayed on the screen, and the next two are the same.

In order to run this code truly, we can find the file \kern\monitor.c, edit it with vim, and add these two instructions to the monitor subprogram, as follows:

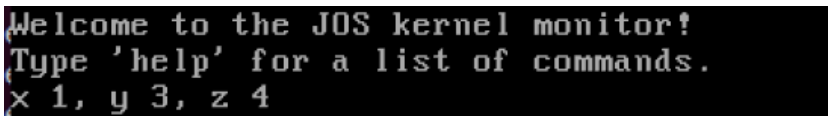
```
void
monitor(struct Trapframe *tf)
{
        char *buf;

        cprintf("Welcome to the JOS kernel monitor!\n");
        cprintf("Type 'help' for a list of commands.\n");

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


        while (1) {
                buf = readline("K> ");
```

And then do make clean, make, make qemu , we can get:



4. 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 out in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

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?

Here's a description of little- and big-endian and a more whimsical description.

Like question 3, add the following codes in the same position and repeat the process:

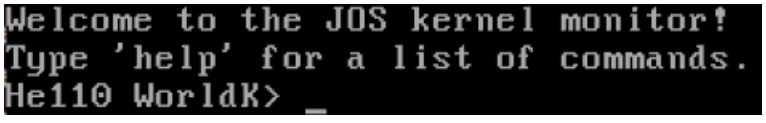
```
void
monitor(struct Trapframe *tf)
{
        char *buf;

        cprintf("Welcome to the JOS kernel monitor!\n");
        cprintf("Type 'help' for a list of commands.\n");

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


        while (1) {
                buf = readline("K> ");
                if (buf != NULL)
```

We get the output:



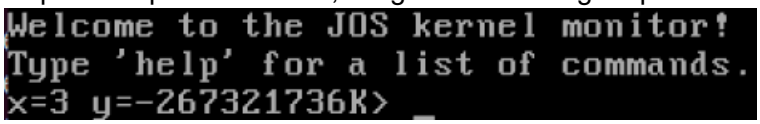
Why do you want to output such a value? First, look at the first %x, which means to output the first parameter in hexadecimal. The value of the first parameter is 57616. Its corresponding hexadecimal representation is e110, so it becomes He110.
Then look at the next %s and output the string that the parameter points to. The parameter is &i, which is the address of variable i, so what should be output is the string at the address of variable i

Before cprintf, we defined i as an int variable, so now we need to split them and output them by byte by byte..
Because x86 is a little-endian mode, the highest byte of the representative word is stored on the highest byte address. Assuming that the address of I variable is 0x00, the four bytes of I are stored in 0x00, 0x01, 0x02, 0x03. Because of small end storage, 0x72 ('r ') is stored at 0x00, 0x6c ('l') is stored at 0x01, 0x64 ('d ') is stored at 0x02, and 0x00 ('\0') is stored at 0x03
So in cprintf, I will start to traverse byte by byte from the address of I, just output "World".
If you want to print the same result on the big-endian machine, the value of i must be 0x726c6400, and the printing place of e110 does not need to be changed.

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);
```

Repeat the process above, we get the following output:



Since y has no parameters specified, an indefinite value is output.

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?

Here is the code in /inc/stdarg.h:

```
/*      $NetBSD: stdarg.h,v 1.12 1995/12/25 23:15:31 mycroft Exp $      */

#ifndef JOS_INC_STDARG_H
#define JOS_INC_STDARG_H

typedef __builtin_va_list va_list;

#define va_start(ap, last) __builtin_va_start(ap, last)

#define va_arg(ap, type) __builtin_va_arg(ap, type)

#define va_end(ap) __builtin_va_end(ap)

#endif  /* !JOS_INC_STDARG_H */
```

It can be seen that va_arg keeps growing backward to remove the variable address of the next parameter. This is equivalent to the compiler's right to left sequential push. Since the parameters of the Backstack are in the low memory position, if the variables are taken out from left to right, the compiler will push in from right to left.
If the compiler changes the stack pressing order, then in order to get all the parameters out correctly, you need to modify va_start, va_arg and adapt it to the parameter fetching order.

Exercise 10

%n in C99 standard, it is the number of characters of output string. The format of parameter should be a char * type.
For example: you input a string “abcde”, and get answer 5 by “%n”.
Then fill the code block in /lib/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 * posp ; //position pointer
                        if ((posp = va_arg(ap, char *)) == NULL){
                                printfmt(putch,putdat,"%s",null_error);
                        }else if(*((unsigned int *)putdat) > 127 ){// or
                        between ' ' to '~'
                                printfmt(putch,putdat,"%s",overflow_error);
                                *posp = -1;
                        }else{
                                *posp = *(char *)putdat;
                        }
                        break;
                        }

                // escaped '%' character
                case '%':
                        putch(ch, putdat);
                        break;
```

Remark:
1) here we will always use three variables: num, putch and putdat. num should be the extracted number, putch should be the extracted character, and putdat should be the extracted string length.
2) if the type of the parameter that can't be used for revenue is not int, but char, it means that if t in the example in 8.8.1 is to be char, the corresponding part of the code should also be changed.
And add some code into kern/monitor.c to test:

```
cprintf("Welcome to the JOS kernel monitor!\n");
cprintf("Type 'help' for a list of commands.\n");

int j=0;

char b;

cprintf("%s%n\n","TA is awesome.",&b);

cprintf("Input %d characters\n",b);

while (1) {
        buf = readline("K> ");
        if (buf != NULL)
                if (runcmd(buf, tf) < 0)
                        break;
}
```

make clean, make, and make qemu to see what happens:

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TA is awesome.
Input 14 characters
```

Thus the problem has been finished.

Exercise 11

Exercise 11. Modify the function `printnum()` in `lib/printfmt.c` to support `"%-"` when printing numbers. With the directives starting with "%-", the printed number should be left adjusted. (i.e., paddings are on the right side.) For example, the following function call:

```
cprintf("test:[%-5d]", 3)
```

, should give a result as

```
"test:[3    ]"
```

(4 spaces after '3'). Before modifying `printnum()`, make sure you know what happened in function `vprintffmt()`.

Replace the following in the printnum() function of lib/printfmt.c:

```
// your code here:
if (padc == '-') {
        int i = 0;
        int num_of_digit = 0;
        int temp = num;
        while(temp > 0) {
                num_of_digit += 1;
                temp /= base;
        }
        printnum(putch, putdat, num, base, num_of_digit, ' ');
        for (i = 0; i < width - num_of_digit; i++)
                putch(' ', putdat);
        return;
}
```

And add following code in lib/monitor.c:

```
cprintf("Welcome to the JOS kernel monitor!\n");
cprintf("Type 'help' for a list of commands.\n");

cprintf("test:[%-5d]", 3);

while (1) {
        buf = readline("K> ");
```

Then make clean, make, make qemu to see the results:

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
test:[3    ]K>
```

So we get the right answer.

Exercise 12

1. First, you need to determine which instruction the operating system kernel starts to initialize its stack space.

The running process of boot. S and main. C files has been analyzed previously. The code in this file is the first two parts of the code to be executed after PC startup and BIOS

operation. But they don't belong to the kernel of the operating system. When the bootmain function in the main. C file runs to the end, the last instruction it executes is to jump to the entry address in the entry. S file. At this time, control has been transferred to entry. S.
Before jumping to entry, the contents of %esp and %ebp registers were not modified. It can be seen that there is no statement initializing stack space in bootmain.
Next, enter entry. S. in entry. S, we can see that its last instruction is to call the i386_init() subroutine. This subroutine is located in the init. C file. In this program, some initialization work has been started on the operating system, and the self weight enters into the motor function. It can be seen that when i386 init subroutine is found, the stack of kernel should have been set. Therefore, the instruction to set the kernel stack should be the two statements in entry. S before the call i386_init instruction:

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp                       # nuke frame pointer

# Set the stack pointer
movl    $(bootstacktop),%esp

# now to C code
call    i386_init
```

That is :

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

2. Where is the stack located in memory?
To solve this problem, we need to analyze the entry. S file. The best way is naturally to read its source code, and also with the disassembly file.
First, we have configured a qemu and gdb debugging environment. Now we first set a breakpoint in GDB, just before entering the entry, and then debug step by step.

```
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
```

Then si until,

```
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
```

This is where the mapping take effect, and we have explained in detail before.
The following two instructions then increase the address space of the currently running program to [0xf0000000-0xf0400000].
So $relocated is equal to $0xf010002f.
The next is

```
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
=> 0xf0100034 <relocated+5>:    mov    $0xf0110000,%esp
relocated () at kern/entry.S:77
77              movl    $(bootstacktop),%esp
(gdb) si
=> 0xf0100039 <relocated+10>:   call   0xf01000a6 <i386_init>
80              call    i386_init
```

The two instructions set the values of %ebp and %esp registers respectively. Where %ebp was modified to 0. %esp is modified to the value of bootstacktop. This value is 0xf0110000. In addition, a value, bootstack, is defined at the end of entry. S. Note that before defining bootstacktop at the top of the stack in the data section, we first allocated so much storage space of kstksize for the stack. This KSTKSIZE = 8 * PGSIZE = 8 * 4096 = 32KB. So the address space used for the stack is 0xf0108000-0xf0110000, where the top pointer of the stack points to 0xf0110000. Then the stack is actually located in the physical address space of 0x00108000-0x00110000 in memory.

3. How does the kernel reserve space for its stack?
In fact, through the analysis just now, a 32KB space is declared in the data section of entry. S as the stack. This leaves a space for the kernel.

4. Which end of the reserved area does the stack pointer initialized point to?
Because the stack grows downward, the stack pointer naturally points to the highest address. The highest address is the value of bootstacktop we saw earlier. So this value will be assigned to the stack pointer register.

Exercise 13

First find the address of this subroutine and open obj/kern/kern.asm. In this file, we find that the address of the instruction calling the test_backtrace subroutine is 0xf0100040 So we set breakpoints here and start debugging.
The following is the C code of test_backtrace:

```
void
test_backtrace(int x)
{
    cprintf("entering test_backtrace %d\n", x);
    if (x > 0)
        test_backtrace(x-1);
    else
        mon_backtrace(0, 0, 0);
    cprintf("leaving test_backtrace %d\n", x);
}
```

It can be seen that this subroutine is a circular call. In each layer of the cycle, the information "entering test_backtrace x" is printed first, and then the test_backtrace is called circularly. When the circular call is completed, the message "leaving test" backtrace X "will

be printed. So the first program to enter will exit at last. Since the number of calls to this function in init is set to 5, theinformation is printed 5 times on the screen.

The assmemble code:

```
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       56                      push   %esi
f0100044:       53                      push   %ebx
f0100045:       e8 36 02 00 00          call   f0100280 <__x86.get_pc_thunk.bx>
f010004a:       81 c3 be 12 01 00       add    $0x112be,%ebx
f0100050:       8b 75 08                mov    0x8(%ebp),%esi
        cprintf("entering test_backtrace %d\n", x);
f0100053:       83 ec 08                sub    $0x8,%esp
f0100056:       56                      push   %esi
f0100057:       8d 83 b8 09 ff ff       lea    -0xf648(%ebx),%eax
f010005d:       50                      push   %eax
f010005e:       e8 01 0b 00 00          call   f0100b64 <cprintf>
        if (x > 0)
f0100063:       83 c4 10                add    $0x10,%esp
f0100066:       85 f6                   test   %esi,%esi
f0100068:       7f 2b                   jg     f0100095 <test_backtrace+0x55>
                test_backtrace(x-1);
        else
                mon_backtrace(0, 0, 0);
f010006a:       83 ec 04                sub    $0x4,%esp
f010006d:       6a 00                   push   $0x0
f010006f:       6a 00                   push   $0x0
f0100071:       6a 00                   push   $0x0
f0100073:       e8 ff 08 00 00          call   f0100977 <mon_backtrace>

f0100078:       83 c4 10                add    $0x10,%esp
        cprintf("leaving test_backtrace %d\n", x);
f010007b:       83 ec 08                sub    $0x8,%esp
f010007e:       56                      push   %esi
f010007f:       8d 83 d4 09 ff ff       lea    -0xf62c(%ebx),%eax
f0100085:       50                      push   %eax
f0100086:       e8 d9 0a 00 00          call   f0100b64 <cprintf>
}
f010008b:       83 c4 10                add    $0x10,%esp
f010008e:       8d 65 f8                lea    -0x8(%ebp),%esp
f0100091:       5b                      pop    %ebx
f0100092:       5e                      pop    %esi
f0100093:       5d                      pop    %ebp
f0100094:       c3                      ret
                test_backtrace(x-1);
f0100095:       83 ec 0c                sub    $0xc,%esp
f0100098:       8d 46 ff                lea    -0x1(%esi),%eax
f010009b:       50                      push   %eax
f010009c:       e8 9f ff ff ff          call   f0100040 <test_backtrace>
f01000a1:       83 c4 10                add    $0x10,%esp
f01000a4:       eb d5                   jmp    f010007b <test_backtrace+0x3b>
```

Now let's look at the call stack of the test_backtrace function. %esp stores the top of the stack, %ebp stores the top of the caller's stack, and eax stores the value of x. these registers need to be focused on. Therefore, I use gdb's display command to set to automatically print their values after each run. In addition, I also set to automatically print the data of the memory used in the stack, so as to clearly observe the changes of the stack. Let's go.

test_backtrace(5)

```
f01001ac:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
f01001b3:       e8 88 fe ff ff          call   f0100040 <test_backtrace>
f01001b8:       83 c4 10                add    $0x10,%esp
```

This is where the system calls test_backtrace in i386_init function. The incoming parameter x = 5. We will start to track the change of data in the stack. The data in each register and stack are shown below. It can be seen that two 4-byte integers are pushed into the stack:
1. the value of the input parameter (that is, 5).
2.The address of the next instruction of the call instruction (that is, 0xf01000dd).

```
%esp = 0xf010ffdc
%ebp = 0xf010fff8
// stack info
0xf010ffe0: 0x00000005  // input para 1st time: 5
0xf010ffdc: 0xf01000dd  // return address 1st time
```

After entering the test_backtrace function, the instructions related to data modification in the stack can be divided into three parts:
1. At the beginning of the function, stack the values of some registers so that they can be recovered before the end of the function.
2. Press the input parameters into the stack before calling cprintf.
3. Press the input parameters into the stack before the second call to test_backtrace.
test_babktrace(4):

```
%esp = 0xf010ffc0
%ebp = 0xf010ffd8
// stack info
0xf010ffe0: 0x00000005  // input para 1st time: 5
0xf010ffdc: 0xf01000dd  // return address 1st time
0xf010ffd8: 0xf010fff8  // %ebp 1st time
0xf010ffd4: 0x10094  // %esi 1st time
0xf010ffd0: 0xf0111308  // %ebx 1st time
0xf010ffcc: 0xf010004a  // Residual data
0xf010ffc8: 0x00000000  // Residual data
0xf010ffc4: 0x00000005  // Residual data
0xf010ffc0: 0x00000004  // input para 2nd time: 5
```

Similar to the test_bcktrace(4), test_backtrace(3),test_backtrace(2), test_backtrace(1) test_backtrace(0) is similar.
The mon_backtrace function is currently empty internally, so you don't need to pay attention to it.
When exiting mon_backtrace, the input parameters and reserved 4 bytes are cleared from the stack through the “add $0x10 %esp” statement. At this time,%esp = 0xf010ff30,%ebp = 0xf010ff38.
When exit test_backtrace, three consecutive pop statements stack ebx, esi and ebp registers in turn, and then return them through ret statement. The exit process of other 1-5 is similar, so we will not repeat it.

Every time I call test_backtrace, it mainly do the following things:
1. Stack the return address (the address of the next instruction of the call instruction)
2. Stack the values of ebp, esi and ebx registers to recover their values before exiting the function
3. Call cprintf function to print "entering test_backtrace x ", where x is the value of input parameter
4. Stack the input parameter (x-1), and then allocate 3 doublewords in the stack, 16 bytes in total, to facilitate stack clearing
5. Call test_backtrace (x-1)
6. Call the cprintf function to print "leaving test" backtrace x, where x is the value of the input parameter.
It will print like this:

```
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Backtrace success
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
```

There are four types of stack space used:
%ebp (occupancy 4b): the ebp at the entrance keeps the stack space used
%ebx (occupation 4b): the ebx general register used by the function is saved
%esp (occupation 20b): reduce the stack bottom pointer esp by 0x14 and store the caller function
%eip (occupy 4b): when calling the command, the eip will be pushed
32 bytes in total.

Exercise 14

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp                       # nuke frame pointer
```

We can get hint from entry.S file above, that is when %epb equal to 0, the program stops traversal.
So we can implement the function by the knowledge given by the jos01.pdf file:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        // Your code here.
        unsigned int *ebp = (unsigned int*)read_ebp();
        struct Eipdebuginfo debug_info;
        unsigned int eip = ebp[1]; // Upper level eip.
        while (ebp != NULL) {
                cprintf("  eip %08x  ebp %08x  args %08x %08x %08x %08x %08x\n",
eip, ebp, ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
                ebp = (unsigned int *)ebp[0];
                eip = ebp[1];
        }

        cprintf("Backtrace success\n");
        return 0;
}
```

At this time, make qemu will display like this:

```
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
  eip f0100078  ebp f0110df8  args 00000000 00000000 00000000 f010004a f0112308
  eip f01000a1  ebp f0110e18  args 00000000 00000001 f0110e58 f010004a f0112308
  eip f01000a1  ebp f0110e38  args 00000001 00000002 f0110e78 f010004a f0112308
  eip f01000a1  ebp f0110e58  args 00000002 00000003 f0110e98 f010004a f0112308
  eip f01000a1  ebp f0110e78  args 00000003 00000004 f0110ea8 f010004a f0112308
  eip f01000a1  ebp f0110e98  args 00000004 00000005 f0110ff8 f010004a f0112308
  eip f01001b8  ebp f0110eb8  args 00000005 00000400 fffffc00 00000000 00000000
  eip f010003e  ebp f0110ff8  args 00000003 00001003 00002003 00003003 00004003
Backtrace success
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
```

Exercise 15

1.

In `debuginfo_eip`, where do `__STAB_*` come from?

__STAB_BEGIN__, __STAB_END__, __STABSTR_BEGIN__, __STABSTR_END__ are all defined in kern/kern.ld file. They represent the starting and ending addresses of .stab and .stabstr, respectively.

```
/* Include debugging information in kernel memory */
.stab : {
        PROVIDE(__STAB_BEGIN__ = .);
        *(.stab);
        PROVIDE(__STAB_END__ = .);
        BYTE(0)         /* Force the linker to allocate space
                           for this section */
}

.stabstr : {
        PROVIDE(__STABSTR_BEGIN__ = .);
        *(.stabstr);
        PROVIDE(__STABSTR_END__ = .);
        BYTE(0)         /* Force the linker to allocate space
                           for this section */
}
```

Execute `objdump -h obj/kern/kernel`, I get :

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001d49  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000828  f0101d60  00101d60  00002d60  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003f61  f0102588  00102588  00003588  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001a12  f01064e9  001064e9  000074e9  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00009300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000008  f0111300  00111300  00012300  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .got.plt      0000000c  f0111308  00111308  00012308  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.local 00001000  f0112000  00112000  00013000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  8 .data.rel.ro.local 00000044  f0113000  00113000  00014000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          00000648  f0113060  00113060  00014060  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 10 .comment      0000002b  00000000  00000000  000146a8  2**0
                  CONTENTS, READONLY
```

Thus we know:

1.STAB_BEGIN=f0102588，STAB_END=f0102588 + 00003f61 - 1

2.STABSTR_BEGIN=f01064e9，STABSTR_END=f01064e9 + 00001a12 – 1

Next

execute objdump -G obj/kern/kernel，this is to display (in raw form) any STABS info in the file

```
obj/kern/kernel:     file format elf32-i386

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String

-1     HdrSym 0      1351   00001a11 1
0      SO     0      0      f0100000 1      {standard input}
1      SOL    0      0      f010000c 18     kern/entry.S
2      SLINE  0      44     f010000c 0
3      SLINE  0      57     f0100015 0
4      SLINE  0      58     f010001a 0
5      SLINE  0      60     f010001d 0
6      SLINE  0      61     f0100020 0
7      SLINE  0      62     f0100025 0
8      SLINE  0      67     f0100028 0
9      SLINE  0      68     f010002d 0
10     SLINE  0      74     f010002f 0
11     SLINE  0      77     f0100034 0
12     SLINE  0      80     f0100039 0
13     SLINE  0      83     f010003e 0
14     SO     0      2      f0100040 31     kern/entrypgdir.c
15     OPT    0      0      00000000 49     gcc2_compiled.
16     LSYM   0      0      00000000 64     int:t(0,1)=r(0,1);-2147483648;21474
```

There are more than 1300 lines, so we don't display them all and store them in a file.

```
jos@ubuntu:~/17307130171/jos01/jos-2019-spring$ objdump -G obj/kern/kernel > ou
tput.md
```

In the picture:

Symnum: symbol index. In other words, the whole symbol table is regarded as an array. Symnum is the subscript of the current symbol in the array. You can see that sysmnum is

arranged in ascending order from - 1.
n_type: it's symbol type. Here, I found several words with more frequent occurrence, such as SOL, OPT, SLINE, FUN, etc:

SOL: the file name it should be
OPT: GCC compilation command
SLINE: file lines
FUN: function name

n_othr: not used at present, its value is fixed to 0.
n_desc: line number in the file
n_value: indicates the address. In particular, only the address of the FUN type symbol is the absolute address, the address of the SLINE symbol is the offset, and the actual address is the function entry address plus the offset.

Next run
i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c
then check init.s file:

```
.Ltext0:
        .stabs  "gcc2_compiled.",60,0,0,0
        .stabs  "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0
        .stabs  "char:t(0,2)=r(0,2);0;127;",128,0,0,0
        .stabs  "long int:t(0,3)=r(0,3);-0;4294967295;",128,0,0,0
        .stabs  "unsigned int:t(0,4)=r(0,4);0;4294967295;",128,0,0,0
        .stabs  "long unsigned int:t(0,5)=r(0,5);0;-1;",128,0,0,0
        .stabs  "__int128:t(0,6)=r(0,6);0;-1;",128,0,0,0
        .stabs  "__int128 unsigned:t(0,7)=r(0,7);0;-1;",128,0,0,0
        .stabs  "long long int:t(0,8)=r(0,8);-0;4294967295;",128,0,0,0
        .stabs  "long long unsigned int:t(0,9)=r(0,9);0;-1;",128,0,0,0
        .stabs  "short int:t(0,10)=r(0,10);-32768;32767;",128,0,0,0
        .stabs  "short unsigned int:t(0,11)=r(0,11);0;65535;",128,0,0,0
        .stabs  "signed char:t(0,12)=r(0,12);-128;127;",128,0,0,0
        .stabs  "unsigned char:t(0,13)=r(0,13);0;255;",128,0,0,0
        .stabs  "float:t(0,14)=r(0,1);4;0;",128,0,0,0
        .stabs  "double:t(0,15)=r(0,1);8;0;",128,0,0,0
        .stabs  "long double:t(0,16)=r(0,1);16;0;",128,0,0,0
        .stabs  "_Float32:t(0,17)=r(0,1);4;0;",128,0,0,0
        .stabs  "_Float64:t(0,18)=r(0,1);8;0;",128,0,0,0
        .stabs  "_Float128:t(0,19)=r(0,1);16;0;",128,0,0,0
        .stabs  "_Float32x:t(0,20)=r(0,1);8;0;",128,0,0,0
        .stabs  "_Float64x:t(0,21)=r(0,1);16;0;",128,0,0,0
```

So it can be seen that after this command is executed, all lines with .stabs are filtered out.

See if the bootloader loads the symbol table in memory as part of loading the kernel binary?
Use GDB to check whether symbol information is stored in the location of symbol table. First of all, according to the output of step 3, we know that the load memory address of the. stabstr section is 0xf01064e9. Use x/8s 0xf01064e9 to print the first eight string information, as shown below.

```
(gdb) x/8s 0xf01064e9
0xf01064e9:     ""
0xf01064ea:     "{standard input}"
0xf01064fb:     "kern/entry.S"
0xf0106508:     "kern/entrypgdir.c"
0xf010651a:     "gcc2_compiled."
0xf0106529:     "int:t(0,1)=r(0,1);-2147483648;2147483647;"
0xf0106553:     "char:t(0,2)=r(0,2);0;127;"
0xf010656d:     "long int:t(0,3)=r(0,3);-2147483648;2147483647;"
```

It can be seen that the symbol table is also loaded into memory when the kernel is loaded.

After understanding the meaning of each line of stabs, you can call stab_binsearch to find the line number corresponding to an address. Since the previous code has found the function in which the address is and the function entry address, the offset can be obtained by subtracting the original address from the function entry address, and then the corresponding record can be found in the specified interval in the symbol table according to the offset. The code is as:

```
// Your code here.
stab_binsearch(stabs, &lfun, &rfun, N_SLINE, addr - info->eip_fn_addr);

if (lfun <= rfun)
{
        info->eip_line = stabs[lfun].n_desc;
}
```

Then

1. Add the backtrace command to the kernel emulator

Just add it in the kern/monitor.c file, mimicking the existing commands.

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

2. Increase the print file name, function name, and line number in mon_backtrace:

After the above exploration, the problem can be easily solved. debuginfo_eip is called in mon_backtrace to get the file name, function name, and line number.

And the function I implemented is in the following picture:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        // Your code here.
        unsigned int *ebp = (unsigned int*)read_ebp();
        struct Eipdebuginfo debug_info;
        unsigned int eip = ebp[1]; // Upper level eip.
        while (ebp != NULL) {
                cprintf("  eip %08x  ebp %08x  args %08x %08x %08x %08x %08x\n",
eip, ebp, ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
                debuginfo_eip(eip, &debug_info);
                cprintf("        %s:%d ", debug_info.eip_file,
debug_info.eip_line);
                for(int i = 0; i < debug_info.eip_fn_namelen; i++)
                        cprintf("%c", debug_info.eip_fn_name[i]);
                cprintf("+%d\n", eip - debug_info.eip_fn_addr);
                ebp = (unsigned int *)ebp[0];
                eip = ebp[1];
        }

        cprintf("Backtrace success\n");
        return 0;
}
```

This is the reuslt:

```
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
  eip f0100078 ebp f0110df8 args 00000000 00000000 00000000 f010004a f0112308
     kern/init.c:16 test_backtrace+56
  eip f01000a1 ebp f0110e18 args 00000000 00000001 f0110e58 f010004a f0112308
     kern/init.c:16 test_backtrace+97
  eip f01000a1 ebp f0110e38 args 00000001 00000002 f0110e78 f010004a f0112308
     kern/init.c:16 test_backtrace+97
  eip f01000a1 ebp f0110e58 args 00000002 00000003 f0110e98 f010004a f0112308
     kern/init.c:16 test_backtrace+97
  eip f01000a1 ebp f0110e78 args 00000003 00000004 f0110ea8 f010004a f0112308
     kern/init.c:16 test_backtrace+97
  eip f01000a1 ebp f0110e98 args 00000004 00000005 f0110ff8 f010004a f0112308
     kern/init.c:16 test_backtrace+97
  eip f01001b8 ebp f0110eb8 args 00000005 00000400 fffffc00 00000000 00000000
     kern/init.c:52 i386_init+274
  eip f010003e ebp f0110ff8 args 00000003 00001003 00002003 00003003 00004003
     <unknown>:0 <unknown>+0
Backtrace success
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
```

Exercise 16

So this is an experiment that we're going to do in the Attack Lab in CSAPP to replace the function address, except the previous experiment with CSAPP was to find the function address in assembly, and then change the function address, and now we're going to replace the stack address of the previous function in the form of %n.

Then our task will be divided into several steps:First, find the stack address where you entered the function.Second, find the address of the function you want to replace.Third,

change the stack address.The %n in exercise 10 is used to place a string into a pointer to a char, using the %*s technique to swap addresses. pret_addr[i] divides the entire address into four segments. Note that it is just a swap of small end addresses.

```
void
start_overflow(void)
{
        // You should use a techique similar to buffer overflow
        // to invoke the do_overflow function and
        // the procedure must return normally.

        // And you must use the "cprintf" function with %n specifier
        // you augmented in the "Exercise 9" to do this job.

        // hint: You can use the read_pretaddr function to retrieve
        //       the pointer to the function call return address;

        char str[256] = {};
        int nstr = 0;
        char *pret_addr;

        // Your code here.
        pret_addr = (char *)read_pretaddr();
        uint32_t maladdr = (uint32_t)do_overflow;
        for (int i = 0; i != 4; i++) {
                cprintf("%*s%n", pret_addr[i] & 0xff, "\x0d", pret_addr + 4 +
i);
        }
        for (int i = 0; i != 4; i++) {
                cprintf("%*s%n", (maladdr >> (i * 8)) & 0xff, "\x0d", pret_addr

+ i);
        }
}
```

Exercise 17

The command line of this function has two parts, the first is time, and the second is another command, so argc must be equal to 2.

Second, the following part of time must be the command we already have, namely help, helpkerninfo, or backtrace.

Then implement the function:

```
int mon_time(int argc, char **argv, struct Trapframe *tf)
{
        if (argc != 2)
                return -1;

        uint64_t before, after;
        int i;
        struct Command command;
        /* search */
        for (i = 0; i < ARRAY_SIZE(commands); i++) {
                if (strcmp(commands[i].name, argv[1]) == 0) {
                        break;
                }
        }
        if (i == ARRAY_SIZE(commands))
                return -1;
        /* run */
        before = read_tsc();
        (commands[i].func)(1, argv+1, tf);
        after = read_tsc();
        cprintf("%s cycles: %d\n", commands[i].name, after - before);
        return 0;
}
```

And to make the system call the instruction successfully we add the information into the monitor.c and monitor.h:

monitor.c

```
static struct Command commands[] = {
        { "help", "Display this list of commands", mon_help },
        { "kerninfo", "Display information about the kernel", mon_kerninfo },
        { "backtrace", "Display a backtrace of the function stack",
mon_backtrace },
        { "time", "time kerninfo",mon_time },
};
```

monitor.h

```
#ifndef JOS_KERN_MONITOR_H
#define JOS_KERN_MONITOR_H
#ifndef JOS_KERNEL
# error "This is a JOS kernel header; user programs should not #include it"
#endif

struct Trapframe;

// Activate the kernel monitor,
// optionally providing a trap frame indicating the current state
// (NULL if none).
void monitor(struct Trapframe *tf);

// Functions implementing monitor commands.
int mon_help(int argc, char **argv, struct Trapframe *tf);
int mon_kerninfo(int argc, char **argv, struct Trapframe *tf);
int mon_backtrace(int argc, char **argv, struct Trapframe *tf);
int mon_time(int argc, char **argv, struct Trapframe *tf);

#endif  // !JOS_KERN_MONITOR_H
```

And so far I have finished the lab finally under guidance of TA's blog.

Thanks to TA and god!

In the end, run make grade to check the result:

```
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/jos/17307130171/jos01/jos-2019-spring
running JOS: (1.4s)
  printf: OK
  sign: OK
  padding: OK
  overflow warning: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
  backtrace lines: OK
  overflow success: OK
Score: 90/90
```