lab1

exercise 1

x86-64 属于**CISC**，有以下一些特点：

- 指令长度变长
- 寻址方式较多，比较灵活
- CPU可以不经由寄存器对内存进行操作

而ARM指令集属于**RISC**,其：

- 指令长度固定
- 寻址方式单一
- 计算基本都在寄存器中完成，寄存器与内存的通信由load/store来完成
- 寄存器相比前者富余得多，前者返回地址存储在栈中，而ARM则有LR寄存器来存储当前函数的返回地址

exercise 2

第一个函数为`_start()`，其地址为`0x0000000000080000`

```
0x0000000000080000 in ?? ()
(gdb) where
#0  0x0000000000080000 in _start ()
Backtrace stopped: not enough registers or memory available to unwind further
```

exercise 3

`init`段的地址为`0x0000000000080000`,对应上题的`_start`,该函数位于`boot/start.S`,

```
There are 9 section headers, starting at offset 0x20cf0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] init              PROGBITS         0000000000080000  00010000
       000000000000b5b0  0000000000000008 WAX       0     0     4096
  [ 2] .text             PROGBITS         ffffff000008c000  0001c000
       0000000000001474  0000000000000000  AX       0     0     8
  [ 3] .rodata           PROGBITS         ffffff0000090000  00020000
       0000000000000118  0000000000000001 AMS       0     0     8
  [ 4] .bss              NOBITS           ffffff0000090120  00020118
       0000000000008000  0000000000000000  WA       0     0     16
  [ 5] .comment          PROGBITS         0000000000000000  00020118
       0000000000000032  0000000000000001  MS       0     0     1
  [ 6] .symtab           SYMTAB           0000000000000000  00020150
       0000000000000858  0000000000000018           7    46     8
  [ 7] .strtab           STRTAB           0000000000000000  000209a8
       0000000000000308  0000000000000000           0     0     1
  [ 8] .shstrtab         STRTAB           0000000000000000  00020cb0
```

```
       000000000000003c  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

```
    mrs x8, mpidr_el1
    and x8, x8, #0xFF
    cbz x8, primary

  /* hang all secondary processors before we intorduce multi-processors */
secondary_hang:
    bl secondary_hang
```

`mpidr_el1`记录了`cpu_id`,如果其不为0，则会跳转到`secondary_hang`不断循环，从而实现挂起

exercise 4

```
build/kernel.img:     file format elf64-little

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 init          0000b5b0  0000000000080000  0000000000080000  00010000  2**12
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         00001474  ffffff000008c000  000000000008c000  0001c000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata       00000118  ffffff0000090000  0000000000090000  00020000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00008000  ffffff0000090120  0000000000090120  00020118  2**4
                  ALLOC
  4 .comment      00000032  0000000000000000  0000000000000000  00020118  2**0
                  CONTENTS, READONLY
```

除了`init`段的VMA和LMA相同外，其余的section的VMA和LMA都有`ffffff0000000000`的offset.

`.comment`为注释信息，在可执行文件中应不占空间。

`.init`中是bootloader的代码，内存空间还没有初始化，所以使用LMA寻址。在bootloader准备函数栈和异常向量，初始化UART,以及初始化页表并开启MMU后，进行到位于`.text`的kernel部分，至此开始使用VMA寻址 。

exercise 5

```
static int printk_write_num(char **out, long long i, int base, int sign,
                int width, int flags, int letbase)
{
    char print_buf[PRINT_BUF_LEN];
    char *s;
    int t, neg = 0, pc = 0;
    unsigned long long u = i;

    if (i == 0) {
```

```
        print_buf[0] = '0';
        print_buf[1] = '\0';
        return prints(out, print_buf, width, flags);
    }

    if (sign && base == 10 && i < 0) {
        neg = 1;
        u = -i;
    }
    // TODO: fill your code here
    // store the digitals in the buffer `print_buf`:
    // 1. the last postion of this buffer must be '\0'
    // 2. the format is only decided by `base` and `letbase` here

    //now pointer s should point to the last element of char array `print_buf`
    s = print_buf + PRINT_BUF_LEN - 1;
    //set the last position to '\0'
    *s = '\0';

    if (neg) {
        if (width && (flags & PAD_ZERO)) {
            simple_outputchar(out, '-');
            ++pc;
            --width;
        } else {
            *--s = '-';
        }
    }
    //from back to front
    while(u){
        --s;
        unsigned int curBit = u % base;
        u = u / base;
        if(curBit <= 9){
            *s = '0' + curBit;
        }else{
            if(letbase)
                *s = 'a' + curBit - 10;
            else
                *s = 'A' + curBit - 10;
        }
    }

    return pc + prints(out, s, width, flags);
}
```

exercise 6

```
    /* Prepare stack pointer and jump to C. */
    adr     x0, boot_cpu_stack
    add     x0, x0, #0x1000
    mov     sp, x0
```

内核栈初始化的函数位于`boot/start.S`,其通过`sp`指向`boot_cpu_stack[0]`的第4096`(0x1000)`字节处为内核栈保留了4096字节的空间

exercise 7

```
0x0000000000080000 in ?? ()
(gdb) b stack_test
Breakpoint 1 at 0xffffff000008c020
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0xffffff000008c020 in stack_test ()
(gdb) x/10ga $x29
0xffffff0000092110 <kernel_stack+8176>: 0x0 0xffffff000008c018
0xffffff0000092120 <kernel_stack+8192>: 0x0 0x0
0xffffff0000092130 <kernel_stack+8208>: 0x0 0x0
0xffffff0000092140 <kernel_stack+8224>: 0x0 0x0
0xffffff0000092150 <kernel_stack+8240>: 0x0 0x0
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0xffffff000008c020 in stack_test ()
(gdb) x/10ga $x29
0xffffff00000920f0 <kernel_stack+8144>: 0xffffff0000092110 <kernel_stack+8176>
0xffffff000008c0d4 <main+72>
0xffffff0000092100 <kernel_stack+8160>: 0x0 0xffffffc0
0xffffff0000092110 <kernel_stack+8176>: 0x0 0xffffff000008c018
0xffffff0000092120 <kernel_stack+8192>: 0x0 0x0
0xffffff0000092130 <kernel_stack+8208>: 0x0 0x0
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0xffffff000008c020 in stack_test ()
(gdb) x/10ga $x29
0xffffff00000920d0 <kernel_stack+8112>: 0xffffff00000920f0 <kernel_stack+8144>
0xffffff000008c070 <stack_test+80>
0xffffff00000920e0 <kernel_stack+8128>: 0x5 0xffffffc0
0xffffff00000920f0 <kernel_stack+8144>: 0xffffff0000092110 <kernel_stack+8176>
0xffffff000008c0d4 <main+72>
0xffffff0000092100 <kernel_stack+8160>: 0x0 0xffffffc0
0xffffff0000092110 <kernel_stack+8176>: 0x0 0xffffff000008c018
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0xffffff000008c020 in stack_test ()
(gdb) x/10ga $x29
0xffffff00000920b0 <kernel_stack+8080>: 0xffffff00000920d0 <kernel_stack+8112>
0xffffff000008c070 <stack_test+80>
0xffffff00000920c0 <kernel_stack+8096>: 0x4 0xffffffc0
0xffffff00000920d0 <kernel_stack+8112>: 0xffffff00000920f0 <kernel_stack+8144>
0xffffff000008c070 <stack_test+80>
0xffffff00000920e0 <kernel_stack+8128>: 0x5 0xffffffc0
0xffffff00000920f0 <kernel_stack+8144>: 0xffffff0000092110 <kernel_stack+8176>
0xffffff000008c0d4 <main+72>
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0xffffff000008c020 in stack_test ()
(gdb) x/10ga $x29
```

```
0xffffff0000092090 <kernel_stack+8048>: 0xffffff00000920b0 <kernel_stack+8080>
0xffffff000008c070 <stack_test+80>
0xffffff00000920a0 <kernel_stack+8064>: 0x3 0xffffffc0
0xffffff00000920b0 <kernel_stack+8080>: 0xffffff00000920d0 <kernel_stack+8112>
0xffffff000008c070 <stack_test+80>
0xffffff00000920c0 <kernel_stack+8096>: 0x4 0xffffffc0
0xffffff00000920d0 <kernel_stack+8112>: 0xffffff00000920f0 <kernel_stack+8144>
0xffffff000008c070 <stack_test+80>
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0xffffff000008c020 in stack_test ()
(gdb) x/10ga $x29
0xffffff0000092070 <kernel_stack+8016>: 0xffffff0000092090 <kernel_stack+8048>
0xffffff000008c070 <stack_test+80>
0xffffff0000092080 <kernel_stack+8032>: 0x2 0xffffffc0
0xffffff0000092090 <kernel_stack+8048>: 0xffffff00000920b0 <kernel_stack+8080>
0xffffff000008c070 <stack_test+80>
0xffffff00000920a0 <kernel_stack+8064>: 0x3 0xffffffc0
0xffffff00000920b0 <kernel_stack+8080>: 0xffffff00000920d0 <kernel_stack+8112>
0xffffff000008c070 <stack_test+80>
```

```
(gdb) x/20x $x29
0xffffff0000092090 <kernel_stack+8048>: 0xffffff00000920b0  0xffffff000008c070
0xffffff00000920a0 <kernel_stack+8064>: 0x0000000000000003  0x00000000ffffffc0
0xffffff00000920b0 <kernel_stack+8080>: 0xffffff00000920d0  0xffffff000008c070
0xffffff00000920c0 <kernel_stack+8096>: 0x0000000000000004  0x00000000ffffffc0
0xffffff00000920d0 <kernel_stack+8112>: 0xffffff00000920f0  0xffffff000008c070
0xffffff00000920e0 <kernel_stack+8128>: 0x0000000000000005  0x00000000ffffffc0
0xffffff00000920f0 <kernel_stack+8144>: 0xffffff0000092110  0xffffff000008c0d4
0xffffff0000092100 <kernel_stack+8160>: 0x0000000000000000  0x00000000ffffffc0
0xffffff0000092110 <kernel_stack+8176>: 0x0000000000000000  0xffffff000008c018
0xffffff0000092120 <kernel_stack+8192>: 0x0000000000000000  0x0000000000000000
```

FP 处的内存值为 caller function 的 FP 值，FP + 8 处的值为当前函数的 LR 值，FP - 16 处的内存值则为函数参数(没太明白FP - 8 处的值是什么)

exercise 8

```
                |           |
                |           |
                ├───────────┤
                |    LR     |
                ├───────────┤
CUR FP    ──▶   |  OLD FP   |  HIGH ADDRESS
                ├───────────┤
                |           |         |
                ├───────────┤         |
                |   ARG 1   |         |
                ├───────────┤         |
                |           |         |
                |  ......   |         |
                |           |         |
                ├───────────┤         |
                |    LR     |         |
                ├───────────┤         |
```

```
            |  OLD FP  |       ▼
            ├──────────┤
            |          |
            ├──────────┤
            |   ARG 1  |  LOW ADDRESS
            ├──────────┤
            |          |
            |          |
```

SP 的恢复只需将 FP 值重新赋给它即可

exercise 9

```
__attribute__ ((optimize("O1")))
int stack_backtrace()
{
    printk("Stack backtrace:\n");


    // Your code here.

    //fp of current function(stack_backstrce)
    u64 stackBacktraceFP = read_fp();
    //fp of the caller of current function(stack_backstrace)
    u64 callStackBackstraceFP = *(u64 *)stackBacktraceFP;

    u64 fp = callStackBackstraceFP; // value of fp;

    do{
        printk("LR %lx FP %lx Args ",*(u64 *)(fp + 8),fp);
        u64 argAdrBegin = fp - 16; // address of argList begins at fp - 16
        for(int i = 0; i < 5; ++i){
            printk("%lx ", *(u64 *)(argAdrBegin + 8 * i));
        }
        printk("\n");
        fp = *(u64 *)fp; // value of parent fp
    }while(fp);//value of fp != 0

    return 0;
}
```