Lab1 Solution

516030910006 方俊杰

Exercise 1.

浏览《 PC Assembly Language 》，注意其汇编是 NASM assembler 的 Intel 格式，而Linux 里的 GNU assembler 是 AT&T 格式。两者在寄存器/立即数的前缀、操作数方向、内存单元操作数格式、间接寻址方式、操作码后缀上存在区别。

Exercise 2.

用 gdb 调试 qemu，开启后用 si 步进，首先进行了一个比较，若 cs:esi 不等于 0x48 就会jmp走，继续步进在设置 ss、sp、dx寄存器，然后跳转去执行了一些 IO 操作，继而 load 了 GDT，并设置了 cr0，然后又是一个ljmp改变了架构为 i386，然后似乎陷入了循环。查阅了解到 BIOS 在 设置 中断描述符寄存器，初始化总线和必要的设备，搜索可加载的boot loader，并从硬盘的第一个 sector 读取运行之，转交控制权给它。

Exercise 3.

boot loader sector 将 load 到 0x7c00 物理地址的内存上，设置断电执行到那，然后步进查看汇编运，查看运行 readsect、readseg 的一些情况。

1. 在设置完 cr0 开启虚拟地址映射，并用ljmp设置 cs ip之后开始执行 32位代码，`boot/boot.S`中的 `ljmp $PROT_MODE_CSEG, $protcseg` 导致切换。
2. boot loader 执行的最后一条代码是 `((void (*)(void)) (ELFHDR->e_entry))()`，由此在 `boot/boot.asm` 中找到 **boot loader 最后执行的指令** 是

```
((void (*)(void)) (ELFHDR->e_entry))();
    7d71:       ff 15 18 00 01 00       call   *0x10018
```

利用 gdb 调试发现下一条指令即 **kernel 的第一条指令**在 `obj/kern/kernel`中

```
f010000c <entry>:
f010000c:       66 c7 05 72 04 00 00    movw   $0x1234,0x472
```

3. `boot/main.c` 中的先读取了第一个 segment，是一个 ELF 文件的 header，其中包括了应加载扇区长度的信息：`ELFHDR->e_phnum`

Exercise 4.

输出如下图：

```
1: a = 0x7fffd3b19760, b = 0x564d736f2260, c = 0x7f6a22834a95
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
test: c = 0x7fffd3b19760, 3[c] = 0x7fffd3b1976c
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 = 0x7fffd3b19760, b = 0x7fffd3b19764, c = 0x7fffd3b19761
```

1. 第一行打印了 a、b、c 三个指针的地址，a和c在栈上，b由 malloc 分配，在堆里

2. 然后使 c 指向 数组 a 的起始地址，然后给数组 a 的 4 个元素赋值，由于指针 c 指向的元素与数组 a 内元素相同，可用 c[0] 访问数组 a 的第一个元素，继而改变其赋值，因此第二行打印出的 a[0] 为 200，其余为循环赋值的结果

3. 然后用三种不同的方式通过整数指针 c 访问 a 数组的第 2、3、4 元素，改变其赋值，因此第 3 行打印出用 c 赋值的结果

4. 然后使得整数指针自增 1 ，这样 c 就指向了 数组 a 的第二个元素，再改变此时 c 指向的值，相当于改变 a[1]，因此第 4 行打印出的 a[1] 为新赋的 400

5. 之后的 `c = (int *) ((char *) c + 1)` 导致了 指针 c 指向了相对 a 不对齐的地址，在 (char *)a + 5 的地方，因而 corrupted。考虑在 `*c = 500` 执行前后，原本 a[1] = 400 = 0x190， a[2] = 301 = 0x12D，而 500 = 0x1F4，在内存中为小端排列

addr	**&a[1]**	**c**	**-**	**-**	**&a[2]**	**-**	**-**	**-**
前	90	01	00	00	2D	01	00	00
后	90	F4	01	00	00	01	00	00

   因此，之后第5行打印的 a[1] = 128144 = 0x01F490, a[2] = 128 = 0x0100

6. 第 6 行打印 b 在**整数指针**下自增1 与 c 转换为**字符指针**自增 1 的结果，进行比较也说明了 corrupt 的原因

Exercise 5.

调试结果如图：

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/4x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/4x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
```

在进入 `boot loader` 时， 0x100000地址为 0， 而进入 kernel 后，该地址存在内容如图，在 `kern/kernel.ld`中可以看到

```
/* AT(...) gives the load address of this section, which tells
    the boot loader where to load the kernel in physical memory */
.text : AT(0x100000) {
    *(.text .stub .text.* .gnu.linkonce.t.*)
}
```

0x100000 是指定的 kernel load address，加载后会把 .text 写入该地址

Exercise 6.

修改 `boot/Makefrag` 文件 `-Ttext` 值 从 0x7c00到别的值，重新编译运行失败

Exercise 7.

在 0x10000c 打断点并步进，同时用 x 命令 查看 0x00100000 和 0xf0100000 两处的地址，发现在 `kern/entry.S` 的 `movl %eax, %cr0` 指令后，两者的内容一致了，之前是不一致的，因此该指令使得 虚拟地址映射 生效。若注释之，后面的 `jmp *%eax` 会 Fail，因为跳转到的目标地址是一个 高地址，但是没有地址映射的话，就会访问非法地址发送异常。

Exercise 8.

完善 `lib/printfmt.c` 中 `vprintfmt` 对于 八进制 的实现即可

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

Exercise 9.

字符匹配时增加对于 '+' 的检查，设置一个新的 flag，然后在输出十进制时检查 flag，若成立则先输出一个前导 '+' 即可

```
    case '+':
            preceflag = 1;
            goto reswitch;
    ......
    // (signed) decimal
    case 'd':
        num = getint(&ap, lflag);
        if ((long long) num < 0) {
            putch('-', putdat);
            num = -(long long) num;
        }
        else if (preceflag) {
            putch('+', putdat);
        }

        base = 10;
        goto number;
```

- 回答问题

1. `console.c` 处理与显示字符的 IO 交互，提供 `cputchar`、`getchar`、`iscons` 三个接口。
   `printf.c` 仅调用了 `cputchar` 用于实现格式化输出。
2. CRT 指的是 一种显示器，这段话判断 crt_pos 是否大于 设定的 CRT_SIZE，当条件成立时进行操作，操作完使得 crt_pos 减少 一行大小 CRT_COLS。
   显然这段话作用是在终端输出满时向下滚动一行
3. 这里实现变长参数函数
   - `fmt` 指向格式化字符串"x %d, y %x, z %d\n"，`ap`指向其他参数
   - 依次为：
     `vcprintf`，第一个参数指向 "x %d, y %x, z %d\n"，第二个参数指向 x 即"\001"
     `cons_putc`('x') `cons_putc`(' ')
     `va_arg`，之前指向 x, 之后指向 y "\003"
     `cons_putc`(1) `cons_putc`(',') `cons_putc`(' ') `cons_putc`('y')`cons_putc`(' ')
     `va_arg`，之前指向 y, 之后指向 z "\004"
     `cons_putc`(3) `cons_putc`(',') `cons_putc`(' ') `cons_putc`('z')`cons_putc`(' ')
     `va_arg`，之前指向 z, 之后指向 null
     `cons_putc`(4) `cons_putc`('\n')
4. 输出 `He110 World`
   %x 打印 16 进制的 57616 = 0xe110时即为 `e110`
   %s LE打印 0x00646c72 对应的 ASCII码分别为 0x72-'r', 0x6c-'l', 0x64-'d', 0x00-'\0'结束字符串
   若是BE，则 i 需要改为 0x726c6400， 57616 不用修改，数的表示和输出还是一样的

5. 根据可变参数函数的实现，ap 更根据栈来获取参数，'y='后面会接上 3 所在栈的后 4 bytes 数据按十进制输出
6. 将接口的参数倒着指定即可，或者修改关于可变参数函数的实现 va_arg、va_start

Exercise 10.

将putdat的值写入 va_arg 指定的地址即可，注意错误检查

```
    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";

    // Your code here
    // store putdat to va_arg
    char *num_ch = va_arg(ap, char *);
    if (num_ch == NULL)
        printfmt(putch, putdat, "%s", null_error);
    else if (*(int *)putdat >= 256 - 1) {
        //printfmt(putch, putdat, "how much %d\n", *(int *)putdat);
        printfmt(putch, putdat, "%s", overflow_error);
        *num_ch = -1;
    }
    else
        *num_ch = *(char *)putdat;
```

Exercise 11.

用 %- 实现数字靠左对齐，在 `printnum` 增加对于 `padc` 为 '-' 时的新实现即可

```
    if (padc == '-') {
        // print (more significant) digits
        while (num / base) {
            putch("0123456789abcdef"[num / base], putdat);
            num /= base;
            width--;
        }
        // print (the least significant) digit
        putch("0123456789abcdef"[num % base], putdat);
        // print padding on the right side
        while (--width > 0)
            putch(' ', putdat);
    }
```

Exercise 12.

在 `entry.S` 中初始化了栈，具体汇编为 `movl $(bootstacktop),%esp`
栈位置为 `bootstacktop` 的值，调试知为 `0xf0111000`，分配了 `KSTKSIZE` 大小的空间

stack point 即 esp 指向了 bootstacktop 即 `0xf0111000`地址

Exercise 13.

在调用 `test_backtrace`(0xf0100040) 的递归过程中，每个递归使用了 0x20 bytes (即 8个 32-bit word)大小的栈。压栈了很多东西，且有重复使用，依次有 进入时压栈的 ebp, 调用 `_x86.get_pc_thunk.bx` 压栈的 esi、ebx及返回地址，调用第一个 `cprintf`需要的 fmt字符串、一个参数 x 和返回地址，递归调用自身时 压栈的 x-1 和 返回地址。

- Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?
  传参是编译器在编译时判断是否正确和设置push和pop的,在汇编层面没有一个具体的括号或分界符号。一般来说参数不同于代码段，可以检查栈上值是否为代码段，去掉那些是代码段的栈值

Exercise 14.

使用 `read_ebp` 获取当前 ebp，然后依次获取 保存的上一个栈的 ebp、eip以及5个潜在参数，循环访问栈帧，知道ebp 等于 0，这是初始化栈时设置的第一个 ebp。

```
    cprintf("Stack backtrace:\n");
        uint32_t *ebp, *eip;
        uint32_t args[5];
        ebp = (uint32_t *)read_ebp();
        while ((uint32_t)ebp != 0) {
                // get return address and arguments
                eip = (uint32_t *) *(ebp + 1);
                for (int i=0; i<5; i++) {
                        args[i] = *(ebp + i + 2);
                }
                cprintf("  eip %x ebp %x args %08x %08x %08x %08x
%08x\n",
                         eip, ebp, args[0], args[1], args[2], args[3],
args[4]);
                struct Eipdebuginfo info;
                debuginfo_eip((uintptr_t)eip, &info);
                cprintf("\t%s:%d %.*s+%x\n",
                        info.eip_file, info.eip_line,
                        info.eip_fn_namelen, info.eip_fn_name,
(uint32_t)eip - (uint32_t)info.eip_fn_addr);
                ebp = (uint32_t *) *ebp;
        }
        cprintf("Backtrace success\n");
```

Exercise 15.

补全 `debuginfo_eip()` 中搜索对于代码行的代码，应用到 mon_backtrace 中

```
```

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

Exercise 16.

类似 buffer overflow，提取 `do_overflow` 函数地址分 4 次使用 cprint 中 %n 的特性，覆盖原本的返回地址，为了使得其正常返回，要把原本的返回地址 写到 栈往上 4 byte的地方使得能在 do_overflow 中正常返回。

```
    pret_addr = (char *)read_pretaddr();         // 0xf0110d9c
        target_addr = (uint32_t)do_overflow;     // 0xf010092d
        // save normal RA to RA + 4
        for (int i = 0; i < 4; i++)
        cprintf("%*s%n\n", pret_addr[i] & 0xFF, "", pret_addr + 4 + i);
        // save target to RA
        for (int i = 0; i < 4; i++)
                cprintf("%*s%n\n", (target_addr >> (8*i)) & 0xFF, "",
pret_addr + i);
```

Exercise 17.

实现类型 `time` 命令测量程序运行时间，加入 command line
使用 `rdtsc` 汇编代码测量CPU cycle，类似 `runcmd` 运行对于指令

```
uint64_t rdtsc()
{
        uint32_t lo,hi;
        __asm__ __volatile__("rdtsc":"=a"(lo),"=d"(hi));
        return (uint64_t)hi<<32 | lo;
}

int
mon_checktime(int argc, char **argv, struct Trapframe *tf)
{
        uint64_t begin = 0, end = 1;
        int res = -1;
        char *targetcmd = argv[1];
        for (int i = 0; i < ARRAY_SIZE(commands); i++) {
                if (strcmp(targetcmd, commands[i].name) == 0) {
                        begin = rdtsc();
                        res = commands[i].func(argc-1, argv+1, tf);
                        end = rdtsc();
                }
        }
```

```
        if (res < 0)
                cprintf("Unknown command '%s'\n", targetcmd);
        else
                cprintf("%s cycles: %llu\n", targetcmd, end - begin);
        return res;
}
```