Jos Lab1 Answer

5140379020 路旭

Exercise3：

（1）

从 ljmp $PROT_MODE_CSEG, $protcseg 这条指令开始执行 32-bit 代码；

```
movl %cr0, %eax
orl  $CR0_PE_ON, %eax
movl %eax, %cr0
```

这些指令将执行的代码从 16-bit 换成 32-bit.

（2）

Boot loader 执行的最后一句指令是

```
((void (*)(void)) (ELFHDR->e_entry))();
```

Kernel 的第一条指令是

```
movw    $0x1234,0x472;
```

（3）

Boot loader 通过读取 ELF 文件的文件头来判断应该读取多少 sector，代码位于 boot/main.c.从 0x00100000 开始的内核代码包含了 ELF 的头部。

Exercise5：

第一次查看时 0x00100000 处没有有效信息(全为 0),因为 boot loader 仅占用了 0x000A0000 到 0x00100000 这段地址，而第二次查看时，kernel 被加载到从 0x00100000 开始的地方，所以第二次与第一次信息不同（这时里面应该是 ELF 文件的内容）。

Exercise6：

将无法成功进入 kernel，程序停留在将要进入内核的地方。

Exercise8：

对于 case 'o'，要求是将数字按 8 进制打印，并在前面加上字符‘0’，打印‘0’只需要

```
putch('0',putdat);
```

按八进制打印数字，仿照 case'x'的情况，代码为：

```
num=getuint(&ap,lflag);
base=8;
goto number;
```

注意的是进制 base 改为 8，并加上 goto number 指令。

Exercise9：

对于本题，我的理解是当 cprintf 的参数中的字符串格式中出现%+d 时，要注意判断要打印的整数的正负，是正数则要在打印时先打印‘+’符号（负数时无论字符串格式中是%+d 还是%d 都要打印‘-’符号）。

这里我在 vprintfmt 函数中设置了一个 posflag 标记，当 case‘+’时，posflag 被设置为 1，之后程序经 reswitch 到达 case‘d'，在 case‘d'中根据 posflag 是否被设置以及整数是否大于零判断是否打印‘+’符号。

case '+':

```
    posflag=1;
    goto reswitch;
case 'd':
    num = getint(&ap, lflag);
    if ((long long) num < 0) {
        putch('-', putdat);
        num = -(long long) num;
    }
    else if(posflag==1)
    {
        putch('+',putdat);
    }
    base = 10;
    goto number;
```

1.
console.c 提供的接口函数为：

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

printf.c 通过以下函数调用此接口：

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

2.
当前输出行被打印满后，换行打印.
3.
fmt 指向 cprintf 函数的参数中的格式化字符串，ap 指向格式化字符串的参数.
4.
输出 He110 world
5.
y=后面是一串不确定的内容，因为没有给出具体参数.

Exercise10：
调用 va_arg(ap,char*)获取格式化字符串“%n”的参数指针。
当指针为 NULL 时循环调用 putch 函数打印 null_error，
当当前 cprintf 函数已打印字符越界（小于零或大于 127）时打印 overflow_error.
将当前已打印的字符数目值赋给参数指针指向的 char.

Exercise11：

对于在数字前打印 0 来补齐的情况，原函数代码即可做到。

对于在数字后打印空格来补齐的情况，先用 while 循环计算出 num 在对应 base 位制下各位的值，然后将这些值按高位到低位的顺序打印，最后不够的字符数目用空格补齐：

```
    int magnitude=0;
    int nums[32];
    unsigned long long num1=num;
    nums[magnitude]=num1%base;
    num1=num1/base;
    magnitude+=1;
    while(num1>0)
    {
        nums[magnitude]=num1%base;
        num1=num1/base;
        magnitude+=1;
    }

    int count=magnitude;
    while(count>0)
    {
        putch("0123456789abcdef"[nums[count-1]], putdat);
        count-=1;
    }

    int width1=width-magnitude;
    int padc1= ‘ ';
    while(width1>0)
    {
        putch(padc1, putdat);
        width1-=1;
    }
```

Exercise12、13、14、15（test_backtrace）：

（1）

在 mon_backtrace 中，首先调用 read_ebp()函数获取当前 ebp，

然后根据此 ebp 指向的内容获取上一级函数的 ebp、eip、调用本函数时的参数（地址 ebp 中存储的是上层函数的 ebp,0x4(ebp)中是上一层的 eip,0x8(ebp)中是 arg0...），并依次按格式打印.

然后调用 debuginfo_eip(eip, &eipinfo)获取本层 eip 对应的代码文件、函数名、line number（eipinfo.eip_file,eipinfo.eip_line,eipinfo.eip_fn_name,eip-eipinfo.eip_fn_addr）.

（2）

对于 debuginfo_eip 的补全，仿照 debuginfo_eip 里之前对 SO/FUN 的查询，

用 stab_binsearch 找到对应的行数，然后取出行号：

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

（3）

对于 backtrace 命令的添加，仿照 help 指令，在 commands 数组中添加注册：

```
{ "backtrace","Stack backtrace",mon_backtrace}
```

并在 monitor.h 中注册：

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

Exercise16：

overflow 的思路主要是：

（1）

将 0x4（ebp）指向的 return address 更改为 do_overflow 函数的代码地址，这样函数在返回时即可返回到 do_overflow 函数中执行相应操作。

（2）

为使执行完 do_overflow 中操作后还能跳转回正确的函数，我们在 return address 中填写的代码地址应该是在 push %ebp；movl %esp %ebp；指令之后的地址，所以应是 do_overflow 函数地址+3，这样 do_overflow 的 return address 就是 overflow_me 函数的上一层函数的地址，即 mon_backtrace 中的代码地址.

（3）

运用 cprintf 的“%n“字符串格式来更改地址内容代码如下：

```
    pret_addr=(char*)(read_pretaddr());
    void (*funcp)()=do_overflow;
    uint32_t funcaddr1=((uint32_t)funcp)+3;
    char* funcaddr=(char*)(&funcaddr1);
    int i=0;
    while(i<4)
    {
        int j=*funcaddr;
        j=j&0xff;
        memset(str, 0xd, j);
        str[j]='\0';
        cprintf("%s%n",str,pret_addr);
        funcaddr+=1;
        pret_addr+=1;
        i+=1;
    }
```

（4）

要注意的是因为 overflow_me 函数只执行了调用 start_overflow 操作，编译器可能会将其 inline 掉,从而导致汇编代码效果为 mon_backtrace 函数直接调用了 start_overflow 函数。为避免这种问题出现，我们强制 overflow_me 不被 inline：

```
void __attribute__((noinline)) overflow_me(void)
{
        start_overflow();
}
```

Exercise 17：

仿照 backtrace 命令，在 commands 数组及 monitor.h 中注册 time 命令：

```
{ "time","Display time",mon_time}
int mon_time(int argc, char **argv, struct Trapframe *tf);
```

在 monitor.c 中编写 mon_time 函数：

```
int mon_time(int argc, char **argv, struct Trapframe *tf)
{
    if(argc==1)
    {
        cprintf("Usage: time [command]\n");
        return 0;
    }
    int i=0;
    while(i<NCOMMANDS)
    {
        if(strcmp(argv[1],commands[i].name)==0)
        {
            unsigned long long time1=read_tsc();
            commands[i].func(argc-1,argv+1,tf);
            unsigned long long time2=read_tsc();
            cprintf("%s cycles: %llu\n",argv[1],time2-time1);
            return 0;
        }
        i+=1;
    }
    cprintf("Unknown command\n",argv[1]);
    return 0;
}
```