Operating System Seminar

Bootloader

冯吕，张旭

University of Chinese Academy of Sciences

2017 年 9 月 20 日

Outline

Task2 任务

通过向串口输出字符串实现开发板 *BIOS* 调用。

实现方法

不断向串口输出地址 0x*bfe*48000 处写字符。

代码实现

开始时在 *.data* 段定义了一个字符串变量:

str: .asciiz "Welcome to OS!"

```
20
21      la $8, str
22      li $10, 0xbfe48000
23
24 putstr:
25      lb $9, ($8)
26      beq $9, $0, end
27      sb $9, ($10)
28      addiu $8, 1
29      j putstr
30
31 end:
32      nop
```

task3 任务

实现一个操作系统启动程序。
启动程序被放在了启动设备的第一个扇区，系统启动时该扇区被 *loadboot* 自动加载在内存地址 0xa0800000 处，然后从它的入口地址 0xa0800030 开始执行。启动程序需要将剩余的操作系统程序加载的内存中，即内核。

实现

1. 向读盘函数传递参数。
   - 参数 1：内存存放地址 (0xa0800200)
   - 参数 2：*SD* 卡偏移量 (512)
   - 参数 3：需要读取的字节数 (512)
2. 调用读盘函数。
3. 跳到内核的开始处执行（实际是调用内核的 *main* 函数）

代码实现


```
18
19      li $4, 0xa0800200
20      li $5, 512
21      li $6, 512
22      jal 0x8007b1a8
23      jal 0xa080026c
24      #need add code
25      #read kernel
26
```

错误分析

刚开始的时候跳到内核开始处执行使用了 *J* 指令，导致不停输出 *It's kernel*。
原因 (个人理解)：在上一次调用读盘函数的时候，返回地址保存到了 31 号寄存器，之后使用 *J* 指令跳转到内核处执行的时候不是函数调用，因此 31 号寄存器的值没有更新，所以从内核返回的时候又回到了之前的地方，导致不断循环调用内核中的 *main* 函数。

任务

实现一个 *Linux* 工具，将启动程序和内核编译为一个 *MIPS* 架构支持的操作系统镜像。

代码实现
read_exec_file 函数


```
10 Elf32_Phdr * read_exec_file(FILE **execfile, Elf32_Ehdr **ehdr)
11 {
12     //*execfile = fopen(filename, "rb");
13     //assert(execfile);
14     *ehdr = (Elf32_Ehdr *) malloc (512 * sizeof(uint8_t));
15     fread( *ehdr, 512, 1, *execfile );
16     //printf( "%d\n", (*ehdr)->e_phoff );
17     Elf32_Phdr *phdr;
18     phdr = (Elf32_Phdr *) malloc ((512 - (*ehdr)->e_phoff) * sizeof(uint8_t));
19     fseek(*execfile, (*ehdr)->e_phoff, SEEK_SET);
20     fread (phdr, 512 - (*ehdr)->e_phoff, 1, *execfile );
21     //printf( "%d\n", phdr->p_memsz );
22     return phdr;
23 }
```

调用函数： *malloc*, *fread*, *fseek*.

首先分配一个缓冲区，读取 512 个字节进去，然后便可知道文件头表。之后重新分配一个缓冲区，将文件指针移动到程序段的开始处，读取内容，然后便获得了程序头表。

代码实现
count_kernel_sectors


```
64 int count_kernel_sectors(Elf32_Ehdr *kernel_header, Elf32_Phdr *kernel_phdr){
65     int i, count = 0;
66     for ( i = 0; i < kernel_header->e_phnum; i++ ) {
67         if ( kernel_phdr[i].p_type == PT_LOAD )
68           count += kernel_phdr[i].p_filesz;
69     }
70     return (count + 511) >> 9;
71 }
72
```

直接看代码!

代码实现


```
25 void write_boot(FILE **imagefile, FILE *boot_file, Elf32_Ehdr *boot_header,
26         Elf32_Phdr *boot_phdr)
27 {
28     int i;
29     int count = 0;
30     for ( i = 0; i < boot_header->e_phnum; i++ ) {
31         if( boot_phdr[i].p_type == PT_LOAD ){//need write to image
32             count += boot_phdr[i].p_filesz;
33             int test = fseek(boot_file, boot_phdr[i].p_offset, SEEK_SET);//seek
34             assert(!test);
35             uint8_t buf[boot_phdr[i].p_filesz];
36             fread(buf, boot_phdr[i].p_filesz, 1, boot_file);
37             fwrite(buf, boot_phdr[i].p_filesz, 1, *imagefile);
38         }
39     }
40     uint8_t zero[512 - count - 2];
41     for ( i = 0; i < 512 - count -2; i++ ) {
42         zero[i] = 0;
43     }
44     uint8_t end[2] = {85,170};
45     fwrite(zero, 512 - count - 2, 1, *imagefile);
46     fwrite(end, 2, 1, *imagefile);
47 }
```

write_boot 和 *write_kernel*：二者区别不大。不同的地方是在第一个扇区的结束的最后两个字节应写上 0x55 和 0xaa。
看代码。

main 函数


```
92 int main(int argc, char *argv[]){
93     if (argc < 3)
94       exit(0);
95     Elf32_Ehdr *ehdr, *ehdrk;
96     Elf32_Phdr *phdr, *phdrk;
97     FILE *fp, *fpk;
98     FILE *imfp = fopen("image", "ab");
99     fp = fopen(argv[2], "rb");
100    assert(fp);
101    fpk = fopen(argv[3], "rb");
102    assert(fpk);
103    phdr = read_exec_file(&fp,  &ehdr);
104    phdrk = read_exec_file(&fpk,  &ehdrk);
105    int sec_boot = count_kernel_sectors (ehdr, phdr); //count b
106    int sec_kernel = count_kernel_sectors(ehdrk, phdrk);//count
107
108    write_boot(&imfp, fp, ehdr, phdr);//write bootblock to imag
109    write_kernel(&imfp, fpk, ehdrk, phdrk);
110
111    fclose(fp);
112    fclose(fpk);
113    fclose(imfp);
114 }
 NORMAL  createimage.c      | main()
"createimage.c" 114L, 3359C
```

首先调用 *read_exec_file* 函数读取可执行文件，然后将相应的内容写入镜像文件中。

extended_opt 函数

根据之前的读取信息和计算扇区数的函数即可打印出相关的信息。

```
void extended_opt(Elf32_Ehdr *boot_ehdr, Elf32_Ehdr *kernel_ehdr,
            Elf32_Phdr *boot_phdr,  Elf32_Phdr *kernel_phdr, int cb, int ck){
    printf("length_of_bootblock = %d\n", boot_ehdr->e_shentsize * boot_ehdr->e_shnum);
    printf( "p_offset = %d, p_filesz = %d\n",boot_ehdr->e_phoff, boot_ehdr->e_phentsize );
    printf( "length_of_kernel = %d\n", kernel_ehdr->e_shentsize * kernel_ehdr->e_shnum );

    int ker_sec = count_kernel_sectors(kernel_ehdr, kernel_phdr);
    printf( "kernel_sectors: = %d", ker_sec );
    printf( "p_offset = %d, p_filesz = %d\n",kernel_ehdr->e_phoff, kernel_ehdr->e_phentsize );
    printf( "kernel_phdr->p_offset = %d\n\n", kernel_phdr->p_offset );

    printf( "bootblock image info\n" );
    int boot_sec = count_kernel_sectors(boot_ehdr, boot_phdr);
    printf( "sectors = %d\n", boot_sec );
    printf( "offset of segment in the file: %x", boot_phdr->p_offset );
    printf("the image's virtural address of segment in memory: %x\n", boot_phdr->p_vaddr);
    printf("the file image size of segment: %x\n", boot_ehdr->e_phentsize * boot_ehdr->e_phnum);
    printf("the memory image size of segment: %x\n", boot_phdr->p_memsz);
    printf("size of write to the OS image: %x\n", cb);
    printf("padding up to %x\n", boot_phdr->p_align);

    printf( "kernel image info\n" );
    printf( "sectors = %d\n", ker_sec );
    printf( "offset of segment in the file: %x", kernel_phdr->p_offset );
    printf("the image's virtural address of segment in memory: %x\n", kernel_phdr->p_vaddr);
    printf("the file image size of segment: %x\n", kernel_ehdr->e_phentsize * kernel_ehdr->e_phnum);
    printf("the memory image size of segment: %x\n", kernel_phdr->p_memsz);
    printf("size of write to the OS image: %x\n", ck);
    printf("padding up to %x\n", kernel_phdr->p_align);

}
```

未实现函数

record_kernel_sectors 函数还未实现。

如何向 *bootblock* 传参？嵌入式汇编？