操作系统研讨课实验报告

Project1-Bootloader

- 王华强
- 2016K8009929035

实验简述

实验要求: 调用BIOS函数完成bootblock, 打印字符串并般移kernel. 编写kernel调用BIOS函数打印字符串. 编写creatimage来生成启动镜像.

Bootblock是写在磁盘头部的一段特殊程序, BIOS在完成启动自检等等流程之后会将PC跳转到Bootblock处. 其主要作用是将操作系统内核从外存搬运到内存之中, 之后将控制权移交给操作系统(跳转到操作系统内核之中).

Createimage是镜像生成程序, 它的作用是从ELF文件中取出实际的可执行汇编代码, 并按照位置要求生成反映实际外存内容的文件: 磁盘镜像.

实验具体细节

Stage1 调用BIOS函数

```
        la  $a0, msg
        jal 0x8007b980
```

遵循MIPsO32函数调用规则, 调用函数分成两个部分: 准备参数, 用`jal`指令跳转到地址.

--

Stage2

在bootblock.s中进行函数调用, 将kernel读取到内存中.

基本的函数调用如下:

```
        li  $a0, 0xa0800200
        li  $a1, 0x00000200
        li  $a2, 0x00000200
        jal 0x8007b1cc
```

--

对于kernel大小较大的情况, createimage已经将kernel大小写在固定的位置. 这里我们将kernel大小写在外存首个扇区的末尾(0x800001FF). (详见Stage3实现) 这个首个扇区中的数值随bootloader一同被加载到内存中. 因此可以从内存中指定地址读取这个数值, 将之转换成以Byte为单位之后作为数据搬运BIOS函数的参数.

对应代码如下:

```
#       target position
        li  $a0, 0xa0800000
#       data source
        li  $a1, 0x00000200
#       data size
#       calculate data size (byte)
#       imagecrater saved kernel size in sector number here:
        lb      $a3, 0x000001ff
        li      $t0, 0x00000200
        mult    $a3,$a0
        mflo    $a2
```

--

kernel的编写

在kernel中利用强制类型转换, 使用函数地址直接调用函数, 完成字符串的打印.

```
        char hello_os[]="Hello OS!\n";
        void (*call_printstr)(char* string) = bios_printstr;
        call_printstr(hello_os);
        return;
```

跳转至kernel入口

在bootloader的最后使用`jal 0x80000200`跳转到kernel入口.

解决附加题时有所区别.

--

Stage2 附加题

Idea1

设计如下:

```
# prepare data for func 0x8007b1cc:
# ......
# jump kernel:
        li  $ra, 0xa0800000
        j   0x8007b1cc #copy data
```

在使用BIOS函数将kernel移动到内存中之后, 函数将会返回`$ra`寄存器中的地址. 这里我们在调用函数之前修改`$ra`寄存器的值到`0`, 这样在复制函数返回之后就会自动从kernel的开头开始执行.

在调用BIOS中的数据复制(外存到内存)函数时, 使用`j`来代替`jal`, 这样`jal`就不会向`$ra`中写入数据. 在函数返回时会返回到之前写入`$ra`中的地址`0`, 亦即附加题中所要求的kernel起始地址.

--

这种解法的原理如下:

函数`read_sd_card`(0x8007b1cc)是由BIOS提供的, 在内存中的位置(0x8007b1cc)不会被kernel覆盖, 因此能正常执行. 作为一个函数调用其中必定有`ret`指令, 执行完毕之后返回`$ra`寄存器的地址, 即`0x80000000`, 即可实现跳转到kernel.

--

修改 `$ra` 也会引起一个问题, 在kernel返回的时候返回地址是 `0` , 可能引发问题. 为了解决这个问题可以考虑在kernel中嵌入汇编. 比如, 将kernel的第一条指令写成 `li $ra, 0x000xxxxxx(返回地址))`

又或者, 可以在kernel结尾设置 `while(1);` 来避免返回.

这里的想法在实际测试中存在问题. 这里BIOS函数似乎无法正确完成kernel的复制. (实际上成功了)

经过进一步测试, 发现这种写法只有在kernel采用特定写法的情况下才能够正常工作. 具体正常工作的条件未知. (实际上是因为kernel的基地址设置有问题导致kernel中的跳转指令无法正确执行)

Idea2

在kernel之前人为添加汇编代码, 汇编代码的作用是将汇编代码之后的kernel块移动到0xa0800000处. 在将kernel移到内存之后执行这些汇编指令, 来将kernel移动到对应位置. 注意移动kernel不能覆盖这些汇编代码, 因此可以将这些代码先复制到内存中稍靠后的位置.

Idea2 实现起来比较复杂, 提交时采用第一种写法.

解决问题

在第一周的实验总结中提示了kernel代码基地址设置的问题. 为此, 修改Makefile文件将kernel的基地址设置为 `0x0a000000` . 在解决问题之后kernel得以正常运行.

核心代码如下:

```
#       target position
        li  $a0, 0xa0800000
#       data source
        li  $a1, 0x00000200
#       data size
#       calculate data size (byte)
#       imagecrater saved kernel size in sector number here:
#       lb      $a3, 0x000001ff
#       li      $t0, 0x00000200
#       mult    $a3,$a0
#   mflo    $a2
#   or
#   for small kernel:
        li  $a2, 0x00000200

#bonus question:
        li  $ra, 0xa0800000
        j   0x8007b1cc
```

--

Makefile文件改动:

```
kernel: kernel.c
        ${CC} -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800000 -N -o kernel kern
```

总结: Bootblock被载入内存后的执行流程

从上面所述很容易看到: bootblock先调用BIOS函数完成指定的工作(打印), 之后从外存读取kernel复制到内存中, 再将控制权移交到kernel(跳转到kernel入口).

--

Stage3

简述Stage3中crateimage的行为如下:

```
读取kernel;
读取bootloader;
//两者都为可执行文件
通过Elf头找出两者的`program header`, 并且确认kernel的大小;
通过`program header`找到实际可执行代码地址;
复制实际可执行代码到image中, 其中bootloader在文件开头, 而kernel在第0扇区之后;
写kernel地址;
return;
```

createimage行为描述

写入SD卡的image文件实际上是SD卡上实际内容的一个精确镜像. 而在操作系统引导阶段, 可以执行的是汇编代码而无法识别ELF文件格式. 因此, 镜像中包含bootloader的汇编代码和kernel的汇编代码. 但是, Bootblock编译后的二进制文件、Kernel编译后的二进制文件中包含ELF文件头. 因此, createimage的作用就是从ELF文件中提取出实际的汇编代码, 写到image文件的对应位置.

为了获得Bootblock和Kernel二进制文件中可执行代码的位置和大小, 只需要读取ELF文件头, 借此找到program header, 从中即可读取相关信息. 具体实现请参考代码.

在完成汇编代码的复制之后, createimage将从program header中读取的kernel大小写入到bootloader之后, 第一个扇区中的空白位置上. 这个数据随bootloader所在的扇区被一起加载到内存, 可以由bootloader直接读取.

> 注: Createimage的行为在P1反馈汇总.pdf中亦有描述.

代码实现

请参见实际代码.

细节讨论与实验心得

Stage1 常见错误

这样的写法会导致跳转到储存printstr的数据段.

```
        la  $a0, msg
        jal printstr
```

--

一个实际操作问题

前几次操作中不知道需要在开发板重启之后重新插拔SD卡. 这引发了很多不必要的问题.

--

思考题解答

- Where do you place your bootblock
- How to move kernel from disk to memory
- Where do you place your kernel in the memory
- Where is your kernel entry point
- How to create disk image

这些问题在汇报时已经解答.

实验报告要求中的问题与实验报告的对应关系, 参见`report.md`中的注释.

补充: 关于连接器脚本

研究思路: 加连接器脚本/不加连接器脚本编译, 之后比较objdump结果.

2018.9.25更新: 在P1反馈汇总中已经详细解释了连接器脚本的作用, 因此不再展开.

Huaqiang Wang (c) 2018