操作系统实验报告

author:高梓源 Stu. Num: 2019K8009929026

实验一：引导、内核镜像和ELF文件

task1: Bootloader输出

过程很简单，只需适当了解汇编语言，在此有多种写法，比如：

```
# Way 1:
lui a0,%hi(msg)
addi a0,%lo(msg)

# Way 2:
la a0,msg

SBI_CALL SBI_CONSOLE_PUTSTR
```

阅读过头文件后发现具体过程：

1. 首先准备寄存器，将字符串首址移动到`a0`，作为调用函数的第一个参数`arg0`
2. 由宏定义，`SBI_CONSOLE_PUTSTR`转换为`9`
3. 由宏定义，`SBI_CALL 9`转换为三条语句：

```
li a7,9
ecall
nop
```

准备好寄存器与系统调用号，正好对应`sbi.h`中C语言描述的过程

注意在运行过程会遇到很多问题，去`faq.pdf`中寻求解答即可，比如从网上下载`createimage`，修改`Makefile`，不让它编译`createimage.c`和清理`createimage`，新建`run_qemu.sh`复制粘贴`faq.pdf`中提供的命令，`chmod +x` 命令更改文件权限，将`Makefile`中镜像路径改为`/dev/sdb`后制作SD卡中镜像

task2: 制作kernel内核镜像

S core

bootblock.S更改

```
la a0,kernel
la a1,os_size_loc
lh a1,(a1)
```

```
addi a2,zero,1
SBI_CALL SBI_SD_READ

// Jump to kernel_main
la a3,kernel_main
jr a3
```

仍然利用汇编完成。读取磁盘操作从a2代表的扇区开始读a1个扇区到a0处即可，根据需求与已知调用函数

跳转执行最好不用j指令，因为其面对长距离跳转编译时会优化出未初始化的gp寄存器，选择利用寄存器存值跳转更方便

head.S更改

清空.BSS段内容，只需将__bss_start和__BSS_END之间的所有内存字清零即可，运用一个循环：

```
 /* Clear BSS for flat non-ELF images */
  la a0,__bss_start
  la a1,__BSS_END__
clear_BSS_loop:
  addi a0,a0,4
  beq a0,a1,set_up_C    # see below
  sw zero,(a0)
  j clear_BSS_loop
```

建立运行环境，设置栈指针sp，kernel_stack已经给出，作为大地址赋值即可：

```
set_up_C:
  lui sp,%hi(KERNEL_STACK)
  addi sp,sp,%lo(KERNEL_STACK)
```

跳转执行kernel.c直接跳转至main()函数即可

kernel.c更改

更改监听键盘输入，无限循环getchar，将读取的返回值返回即可，在主函数里调用进行打印，注意合法字符的判别即可 打印几个字符串，根据sbi.h的模板，调用函数即可：

```
int getch()
{
    //Waits for a keyboard input and then returns.
    int input;
    while(1){
        input = sbi_console_getchar();
        if(input>=0){
            return input;
```

```
        }
    }
    return -1;
}

int main(void)
{
    sbi_console_putstr("Hello OS\n\r");
    sbi_console_putstr(buf);

    int input;
    while(1){
        input = getch();
        if(input>=0){
            sbi_console_putchar(input);
        }
    }
}
```

A-C core

实现方法是将bootblock拷贝到一个新的地址然后跳转至新程序对应的下一条指令运行，注意刷新`I-cache`

先空出`jump`的地址，利用objdump进行反汇编，查看下一地址和函数首个语句的偏移量，在笔者程序中为`0x26`，于是更改`bootblock.S`如下：

```
// 1. kernel address (move kernel to here ~)
.equ kernel, 0x50200000

// 2. kernel main address (jmp here to start kernel main!)
.equ kernel_main, 0x50200000

// 3. move bootblock to 0x5e000000
.equ bbl_new_addr, 0x5e000000
.equ bbl_new_entrance,0x5e000026
# see by objdump

main:
    // move bootblock
    la a0,bbl_new_addr
    addi a1,zero,1 #unkown block number
    addi a2,zero,0
    SBI_CALL SBI_SD_READ

    fence.i    #refresh I-cache

    // jump to new address
    // calculate entrance
    la a0,bbl_new_entrance
    jr a0
```

```
    fence.i

    // below are the same with S-core
```

以上为1.0方案，在找到蒋老师design review之后换了一种新思路，如果kernel过大，使用固定地址存放bootblock有可能仍然存在冲突，因此灵活起见，根据kernel本身的大小决定放置位置，进跟着kernel放置即可，此处默认kernel大小考虑了栈空间

变换思路，首先读取用户输入，索引到想要加载的kernel，然后根据kernel stack栈索引该kernel的便宜量和大小，根据大小放置bootblock，跳转至对应的下一条语句进行执行，然后从sd卡中读入kernel，跳转执行kernel即可

这套方案支持大核与多核的加载

task3:制作createimage

直接叙述C-core整体的实现方式

create_image函数

```
static void create_image(int nfiles, char *files[])
{
    int ph, nbytes = 0, first = 1, os_size_offset = 2, is_bootblock = 1,
os_num = nfiles-1;
    FILE *fp, *img;
    Elf64_Ehdr ehdr;
    Elf64_Phdr phdr;

    /* open the image file */
    img = fopen("image","w+");
    if(!img){
        printf("Failure: cannot open image file!\n");
    }

    /* for each input file */
    while (nfiles-- > 0) {

        /* open input file */
        fp = fopen(files[0],"r+");

        /* read ELF header */
        read_ehdr(&ehdr, fp);
        printf("0x%04lx: %s\n", ehdr.e_entry, *files);

        /* for each program header */
        for (ph = 0; ph < ehdr.e_phnum; ph++) {

            printf("\tsegment %d\n",ph);

            /* read program header */
```

```
            read_phdr(&phdr, fp, ph, ehdr);

            /* write segment to the image */
            write_segment(ehdr, phdr, fp, img, &nbytes, &first);
        }
        fclose(fp);
        files++;
        if(!is_bootblock){
            write_os_size(nbytes,img,os_size_offset);
            os_size_offset+=2;
        }
        else{
            is_bootblock = 0;
        }
        nbytes = 0;
    }
    write_kernel_num(img,os_num);
    fclose(img);
}
```

整体流程如上，首先将读入的命令行参数传递至函数，而后打开镜像文件写入 对于目标的所有文件，首先读取ELF头，获取`ehdr`信息，对于`ehdr.e_phnum`的所有程序段，读取程序段的`header`，按照`header`提供的`offset, filesz`的信息写入`memsz`大小的`image`扇区，而后计算这一块`kernel`的大小写入`image`特定位置

对于目标中所有`kernel`块如此操作，只需要将`image`扇区依次累积写入，在所有块大小的内存区域最前面写入总计多少`kernel`，方便读取选择。

因为需要在加入`--extend`选项时打印详细信息，而在`option`结构体中可以查看是否输入这一参数，直接用判断进行输出即可，认为不加额外的函数反而简单，在过程中打印输出即可

```
static void read_ehdr(Elf64_Ehdr * ehdr, FILE * fp)
{
    if(!fread(ehdr,sizeof(Elf64_Ehdr),1,fp)){
        error("Warning: the format of input file is not ELF64\n");
    }
}
```

此处只需要将`fp`开头的结构体使用`read_ehdr`读入其大小的字节数即可，可以加一个`error`判断

```
static void read_phdr(Elf64_Phdr * phdr, FILE * fp, int ph,
                      Elf64_Ehdr ehdr)
{
    fseek(fp,ehdr.e_phoff+ph*ehdr.e_phentsize,SEEK_SET);
    if(!fread(phdr,sizeof(Elf64_Phdr),1,fp)){
        error("Warning: fail to read program header\n");
    }
}
```

此处运用fseek找到指定偏移量的字节流，实际上是ehdr段之后第ph个程序段（从0计数），只需要将前面的若干等长(e_phentsize)程序段跳过，而后用fread读取phdr结构体大小的字段即可

```
static void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr, FILE * fp,
                          FILE * img, int *nbytes, int *first)
{
    int total_size = ((phdr.p_memsz+511)/512)*512;
    if(options.extended==1){
        printf("\t\toffset 0x%lx\t\tvaddr
0x%lx\n",phdr.p_offset,phdr.p_vaddr);
        printf("\t\tfilesz 0x%lx\t\tmemsz
0x%lx\n",phdr.p_filesz,phdr.p_memsz);
    }
    // read
    fseek(fp,phdr.p_offset,SEEK_SET);
    char *data=(char *)malloc(total_size*sizeof(char));
    fread(data,phdr.p_filesz,1,fp);
    // write
    fseek(img,(*first-1) * 512,SEEK_SET);
    fwrite(data,total_size,1,img);
    *nbytes += total_size;
    *first += total_size/512;
    if(phdr.p_filesz && options.extended==1){
        printf("\t\twriting 0x%lx bytes\n",phdr.p_filesz);
        printf("\t\tpadding up to 0x%x\n",(*first-1) * 512);
    }
}
```

总大小计算实际上是$\lcell phdr.p_memsz/512 \rcell$，而后从fp中读入整段程序段到img对应位置，由于使用指针，操作完后自动累加，下次仍然可以直接读取

```
static void write_os_size(int nbytes, FILE * img, int os_size_offset)
{
    int kernel_size = nbytes/512;   // -1 excludes bootblock
    fseek(img,OS_SIZE_LOC - os_size_offset,SEEK_SET);
    char data[2]={kernel_size & 0xff, (kernel_size>>8) & 0xff};
    fwrite(data,1,2,img);
    if(options.extended==1)
        printf("kernel_size: %d sector(s)\n",kernel_size);
}
static void write_kernel_num(FILE *img, int os_num){
    fseek(img,OS_SIZE_LOC,SEEK_SET);
    char data[2]={os_num & 0xff, (os_num>>8) & 0xff};
    fwrite(data,1,2,img);
    if(options.extended==1)
        printf("there are %d kernel(s) in image\n",os_num & 0xffff);
}
```

写入os_size在给出的0x1fc的位置偏移两位开始写，因为最开始的两位用于存放kernel的数目，方便进行索引 注意因为512为0x200，最后两个字节被写os_size，因此按照向下生长的方式，即类似栈写入os_size

修改bootblock.S

```
.equ os_num, 0x502001fc
.equ kn0_os_size_loc, 0x502001fa
.text
.global main

main:
// print number of kernels
    la a0,os_num
    lh a0,(a0)
    addi a0,a0,47
    mv t0,a0
    SBI_CALL SBI_CONSOLE_PUTCHAR

    PRINT_N

    // C core: read input char
read_input:
    SBI_CALL SBI_CONSOLE_GETCHAR
    blt a0,zero,read_input
    bgt a0,t0,read_input
    addi a1,zero,48
    sub a0,a0,a1    # calculate which kernel: a0

    // load kernel size
    mv a3,a0
    slli a3,a3,1
    la a1,kn0_os_size_loc
    sub a1,a1,a3
    lh a1,(a1)

    addi a2,zero,1
    la t0,kn0_os_size_loc
sum:
    addi a0,a0,-1
    blt a0,zero,finish_sum
    lh t0,(t0)
    add a2,a2,t0
    addi t0,t0,-2
    j sum


finish_sum:
    // 2) task2 call BIOS read kernel in SD card and jump to kernel start
    la a0,kernel
    SBI_CALL SBI_SD_READ
```

为体现交互式特点，增加了提示信息，显示当前的kernel数量，直接读取存入os_num块的数据即可

而要跳转到指定的kernel进行系统启动则需要sbi_sd_read，需要计算读取的位置和大小，大小直接为kernel_os_num栈对应下标处的值，这里直接用a3存放kernel数目，而后用最初的kernel0_size的地址减去a3两倍即可。

利用循环依次减少地址，查询该kernel之前的内核大小进行累加，即该kernel的偏移量，输入到a2即可

此后进行read和跳转即可跳转到用户指定的kernel进行操作

方便之处在于允许相当大或相当多的kernel输入

测试时需修改Makefile为最初的形式，将kernel对应命令改名、复制，输出多个内核文件