操作系统 lab1 报告

马晓彬 2012011402

一、 理解通过 make 生成执行文件的过程

1.理解 makefile 生成 ucroe.img 的过程

①Makefile 文件整体进行了宏、变量的声明以及命令的执行，整体分为三大步骤，使用 make “V=” 命令编译，得到编译器执行指令，以下为三大步骤内容使用 gcc 将.c 和.s 文件编译为.o 的二进制文件

```
# create bootblock
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

bootblock = $(call totarget,bootblock)

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
        @echo + ld $@
        $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
        @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
        @$(OBJDUMP) -t $(call objfile,bootblock) | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call
symfile,bootblock)
        @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
        @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

$(call create_target,bootblock)
```

如上图所示，通过对 function.mk 文件中 Makefile 函数的调用，Makefile 代码生成了如下图的一系列命令，最终生成了 bootblock 二进制文件。

```
+ cc kern/init/init.c
gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/ -I
kern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
```

其中包括了对很多相同文件的操作，以生成 init.o 为例，命令解释如下：

编译选项	含义
-I<dir>	添加头文件的搜索路径
-fno-builtin	不接受不是两个下划线开头的内建函数
-Wall	打开 gcc 所有警告
-ggdb	生成可供 gdb 使用的调试信息
-m32	生成适用于 32 位环境的代码
-gstabs	生成 stabs 格式的调试信息
-nostdinc	不在标准系统目录中搜索头文件，只在-I 的路径中搜搜索
-fno-stack-protector	不生成用于检测缓冲区溢出的代码
-c	只编译不链接
-o	表示输出文件名

②使用 ld 命令将.o 的二进制文件链接生成大的二进制程序

③最终将二进制程序使用 dd 命令按顺序拷贝到 ucore.img 文件，其它区域用零文件填充。对应的 Makefile 代码如下，基本是直接调用 linux 命令执行的。

```
# create ucore.img
UCOREIMG        := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel) $(bootblock)
        $(V)dd if=/dev/zero of=$@ count=10000
        $(V)dd if=$(bootblock) of=$@ conv=notrunc
        $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create_target,ucore.img)
```

2.一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

一个系统认为符合规范的磁盘主引导扇区的大小为 512 字节，并且以 0x55AA 结束。

二、 使用 qemu 执行并调试 lab1 中的软件。

1. 在/tools/gdbinit 文件中添加如下命令：

```
set architecture i8086
target remote :1234
break *(0x7c00)
define hook-stop
x/3i $pc
end
```

就可以通过 stepi 命令以单条汇编代码单步调试，可以观察 BIOS 启动过程而且可以看到当前的和下一条指令。

2. 使用 make debug 命令开始执行，会看到 qemu 和 gdb 的窗口，之所以在调试窗口中显示 3 条指令，是因为有些等待的代码会下图所示会在一条汇编指令循环上千次，可以直接再添加一个断点使用 continue 命令绕过：

```
0x7ccc:      repnz insl (%dx),%es:(%edi)
0x7cce:      pop    %edi
```

最终启动程序进入 while(1)循环等待中断。

3. 右图是调试中的信息，右图是 bootblock.asm，下图是 bootasm.s

```
Breakpoint 1 at 0x7c00
=> 0x7c00:      cli
   0x7c01:      cld
   0x7c02:      xor    %ax,%ax
```

```
.code16
    cli
    7c00:       fa                      cli
    cld
    7c01:       fc                      cld

    # Set up the important data segment registers (
    xorw %ax, %ax
    7c02:       31 c0                   xor    %eax
```

```
start:
.code16
    cli
    cld

    # Set up the important data segment
    xorw %ax, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss
```

可见三者相同。

4. 在 0x7c00 处设置断点，成功停止并看到了盖内存储的汇编代码。

三、 分析 bootloader 进入保护模式的过程。

#在实模式中，计算机从 0x7c00 处开始执行命令，也即汇编代码的 start 处

.globl start

start:

.code16 # 以 16 位模式编译

cli # 禁止中断

```
cld
# 初始化各个段寄存器(DS, ES, SS).
xorw %ax, %ax                                   #段寄存器为 0
movw %ax, %ds                                   # -> 数据段
movw %ax, %es                                   # -> 额外段，和 DI 搭配
movw %ax, %ss                                   # -> 栈段寄存器

# 初始化 A20，将键盘控制器上所有地址线置高，32 位地址线都可用.
seta20.1:
    inb $0x64, %al                              #等待 8042 控制器空闲.
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                 # 参数 0xd1 表示写数据至 8042 的端口
    outb %al, $0x64                                 # 写数据 0x64
seta20.2:
    inb $0x64, %al                  #等待 8042 控制器空闲.
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                             # 参数 0xdf 表示
    outb  %al,  $0x60                                       # 0xdf =
11011111, means set P2's A20 bit(the 1 bit) to 1
    # 从实模式转换为保护模式，使用 GDT 和段寄存器实现虚地址至实地址的转换，同时在转换中当前的地址仍然有效
    lgdt gdtdesc                                #装载 gdt 表头至寄存器
     #将 cr0 寄存器 PE 位置 1 开启保护模式
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0

    #通过长跳转更新 cs 的基地址
    ljmp $PROT_MODE_CSEG, $protcseg

.code32                                         # 表明以下代码以 32 位编译
protcseg:
    # 初始化段寄存器
    movw $PROT_MODE_DSEG, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %fs
    movw %ax, %gs
    movw %ax, %ss
    # 设置栈顶指针以及返回地址并跳转至启动处
```

```
movl $0x0, %ebp
movl $start, %esp
call bootmain
```

四、 分析 bootloader 加载 ELF 格式的 OS 的过程

bootloader 使用两层封装的读取硬盘的代码将

readsect 函数的作用是从设备的第 secno 扇区读取数据到内存的 dst 处

```
static void
readsect(void *dst, uint32_t secno) {
    waitdisk();
    //outb 操作端口，是与外设通信的汇编代码
    outb(0x1F2, 1);      //设置读取扇区的数目为 1，即 512KB 的 bootblock
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF); //以上三行发送 1 到 24 位
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0); //发 25 到 29 位并将 30-32 位置为 1

    outb(0x1F7, 0x20);                    // 0x20 命令是读取扇区
    waitdisk();
    insl(0x1F0, dst, SECTSIZE / 4);       // 读取值内存 dst 处，单位转换
    }
```

readseg 对 readsect 函数进行了封装，使其能够从内核的 offset 处读取 count 个字节的信息至内存（虚地址）va 处 static void readseg(uintptr_t va, uint32_t count, uint32_t offset)。

Bootmain 函数通过调用 readseg 函数进行了操作系统到内存的读取，并通过调用其中的入口函数将控制权交给操作系统。

```
void bootmain(void) {
    // 读取硬盘中的第一页（扇区）
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 通过其中的结束段确定其合法性
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // 读取每个程序段到系统内存中 (ignores ph flags)
    // ph 为偏移量
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
```

```
        for (; ph < eph; ph ++) {
            readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
        }//读取至指定的内存处

        // 调用系统代码中的入口函数转移设备控制权，且不返回
        ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

    bad://遇到错误的处理
        outw(0x8A00, 0x8A00);
        outw(0x8A00, 0x8E00);

        /* do nothing */
        while (1);
    }
```

五、 实现函数调用堆栈跟踪函数。

```
print_stackframe(void) {
    uint32_t ebp = read_ebp(); //读取两个寄存器
    uint32_t eip = read_eip();
    int i;
    for(i = 0; i < STACKFRAME_DEPTH; i++)//直到栈底
    {
        cprintf("ebp = 0x%x ",ebp);
        cprintf("eip = 0x%x\n", eip);//输出当前函数块栈指针和ebp返回地址
        int j;
        for(j = 0; j < 4; j++)//打印函数的四个参数
            cprintf("arg %d:0x%x ", j, *((uint32_t* )ebp) +2+j);
        cprintf("\n");
        print_debuginfo(eip-1);//打印函数名和汇编代码所在位置
        ebp = *((uint32_t* )ebp);//将ebp转为调用函数ebp
        eip = *((uint32_t* )ebp+1);//将返回地址指针更新为调用函数的返回地址（存储于ebp+1处）

    }
```

实现之后得到的输出如图：

```
ebp = 0x7b08 eip = 0x1009a6 arg0:0x7b1a arg1:0x7b1b arg2:0x7b1c arg3:0x7b1d
    kern/debug/kdebug.c:294: print_stackframe+21
ebp = 0x7b18 eip = 0x100092 arg0:0x7b3a arg1:0x7b3b arg2:0x7b3c arg3:0x7b3d
    kern/init/init.c:48: grade_backtrace2+33
ebp = 0x7b38 eip = 0x1000bb arg0:0x7b5a arg1:0x7b5b arg2:0x7b5c arg3:0x7b5d
    kern/init/init.c:53: grade_backtrace1+38
ebp = 0x7b58 eip = 0x1000d9 arg0:0x7b7a arg1:0x7b7b arg2:0x7b7c arg3:0x7b7d
    kern/init/init.c:58: grade_backtrace0+23
ebp = 0x7b78 eip = 0x1000fe arg0:0x7b9a arg1:0x7b9b arg2:0x7b9c arg3:0x7b9d
    kern/init/init.c:63: grade_backtrace+34
ebp = 0x7b98 eip = 0x100055 arg0:0x7bca arg1:0x7bcb arg2:0x7bcc arg3:0x7bcd
    kern/init/init.c:28: kern_init+84
ebp = 0x7bc8 eip = 0x7d68 arg0:0x7bfa arg1:0x7bfb arg2:0x7bfc arg3:0x7bfd
    <unknow>: -- 0x00007d67 --
ebp = 0x7bf8 eip = 0x7c4f arg0:0x2 arg1:0x3 arg2:0x4 arg3:0x5
    <unknow>: -- 0x00007c4e --
ebp = 0x0 eip = 0xf000ff53 arg0:0xf000ff55 arg1:0xf000ff56 arg2:0xf000ff57 arg3:0xf000ff58
    <unknow>: -- 0xf000ff52 --
ebp = 0xf000ff53 eip = 0x0 arg0:0x2 arg1:0x3 arg2:0x4 arg3:0x5
    <unknow>: -- 0xffffffff --
ebp = 0x0 eip = 0xf000ff53 arg0:0xf000ff55 arg1:0xf000ff56 arg2:0xf000ff57 arg3:0xf000ff58
    <unknow>: -- 0xf000ff52 --
```

与参考答案的区别：

答案中停止条件多了 ebp！=0，就避免了我 ebp=0 之后所有的错误输出。

六、 完善中断初始化和处理

```
void
idt_init(void) {
    extern uintptr_t __vectors[]; //中断向量表指针
    int i;
    for(i = 0; i < 256; i++)
    {
        if(i < IRQ_OFFSET)//对于编号小于IRQ的视为中断，其它视为异常
        {
            SETGATE(idt[i], 1, GD_KTEXT, __vectors[i],3); //设置段地址为为内核代码段的段号，还有系统处理程序的入口地址
        }
        else
        {
            SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], 3);
        }
    }
    SETGATE(idt[0x80], 0, GD_KTEXT, __vectors[0x80],0);//设置系统中断调用int 0x80权限为用户权限
    lidt(&idt_pd);
```

```
case IRQ_OFFSET + IRQ_TIMER:
    ticks ++;
    if(ticks % 100 == 0)
        print_ticks();
```

与答案不同之处：

答案没有对中断门的类型进行区分，对所有的中断都视为陷阱门，而且系统中断号是 121 和 120，不知有何区别，但是仍然能智能工程运行。

七、 实验中重要的知识点：

1. 理解了 makefile 的复杂应用
2. 系统的初始化（bootloader）的工作流程
3. 操作系统和硬盘外设交互的流程（ELF）和控制权转向操作系统的流程
4. C 函数调用栈的结构