目 录

1 从 Loader 到内核

Loader 要做的两项工作为：

1. 加载内核到内存。
2. 跳入保护模式。

1.1 加载内核到内存

想要将内核加载到内存需要以下步骤：

1. 寻找内核所在位置。
2. 将内核读入内存。

这两个步骤与加载 Loader 入内存的步骤类似，具体细节在《加载 Loader 入内存》的文档中有讲清楚，这里就不再展开。

以下是将内核加载到内存的实现代码：

```
1   org 0100h
2   ; 堆栈基址
3   BaseOfStack equ 0100h
4   ; 内核被加载到的位置
5   BaseOfKernelFile equ 0800h
6   OffsetOfKernelFile equ 0h
7
8       jmp LABEL_START
9       nop
10
11  ; FAT12磁盘的头
12  BS_OEMName DB 'ForrestY'
13
14  BPB_BytePerSec DW 512    ; 每扇区字节数
15  BPB_SecPerClus DB 1      ; 每簇多少扇区
16  BPB_RsvdSecCnt DW 1      ; Boot记录占用多少扇区
17  BPB_NumFATs DB 2         ; 共有多少个FAT表
18  BPB_RootEntCnt DW 224    ; 根目录文件数最大值
19  BPB_TotSec16 DW 2880     ; 逻辑扇区总数
20  BPB_Media DB 0xF0        ; 媒体描述符
21  BPB_FATSz16 DW 9         ; 每FAT扇区数
22  BPB_SecPerTrk DW 18      ; 每磁道扇区数
23  BPB_NumHeads DW 2        ; 磁头数
24  BPB_HiddSec DD 0         ; 隐藏扇区数
25  BPB_TotSec32 DD 0        ; 如果wTotalSectorCount是0，由这个值记录扇区数
26
27  BS_DrvNum DB 0           ; 中断13的驱动器号
28  BS_Reserved1 DB 0        ; 未使用
29  BS_BootSig DB 29h        ; 扩展引导标记
30  BS_VolID DD 0            ; 卷序列号
```

```
31      BS_VolLab DB 'OrangeS0.02' ; 卷标，必须11字节
32      BS_FileSysType DB 'FAT12   ' ; 文件系统类型，必须8字节
33
34      FATSz equ 9
35      RootDirSectors equ 14
36      SectorNoOfRootDirectory equ 19
37      SectorNoOfFAT1 equ 1
38      DeltaSectorNo equ 17
39
40      LABEL_START:
41          mov ax, cs
42          mov ds, ax
43          mov es, ax
44          mov ss, ax
45          mov sp, BaseOfStack
46
47          mov dh, 0
48          call DispStr
49
50          mov word [wSectorNo], SectorNoOfRootDirectory
51          xor ah, ah
52          xor dl, dl
53          int 13h
54
55      LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
56          cmp word [wRootDirSizeForLoop], 0
57          jz LABEL_NO_KERNELBIN
58          dec word [wRootDirSizeForLoop]
59          mov ax, BaseOfKernelFile
60          mov es, ax
61          mov bx, OffsetOfKernelFile
62          mov ax, [wSectorNo]
63          mov cl, 1
64
65          mov si, KernelFileName
66          mov di, OffsetOfKernelFile
67          cld
68          mov dx, 10h
69
70      LABEL_SEARCH_FOR_KERNELBIN:
71          cmp dx, 0
72          jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR
73          dec dx
74          mov cx, 11
75
76      LABEL_CMP_FILENAME:
77          cmp cx, 0
78          jz LABEL_FILENAME_FOUND
79          dec cx
80          lodsb
81          cmp al, byte [es:di]
82          jz LABEL_GO_ON
83          jmp LABEL_DIFFERENT
84
85      LABEL_GO_ON:
86          inc di
```

```
87          jmp LABEL_CMP_FILENAME
88
89      LABEL_DIFFERENT:
90          and di, 0FFE0h
91          add di, 20h
92          mov si, KernelFileName
93          jmp LABEL_SEARCH_FOR_KERNELBIN
94
95      LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
96          add word [wSectorNo], 1
97          jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN
98
99      LABEL_NO_KERNELBIN:
100         mov dh, 2
101         call DispStr
102
103     %ifdef _LOADER_DEBUG_
104         mov ax, 4c00h
105         int 21h
106     %else
107         jmp $
108     %endif
109
110     LABEL_FILENAME_FOUND:
111         mov ax, RootDirSectors
112         and di, 0FFF0h
113
114         ; 记录kernel.bin的大小
115         push eax
116         mov eax, [es:di+01Ch]
117         mov dword [dwKernelSize], eax
118         pop eax
119
120         add di, 01Ah
121         mov cs, word [es:di]            ; cs保存着簇号
122         push cx
123         add cx, ax
124         add cx, DeltaSectorNo           ; cs保存着Kernel.bin的扇区号
125         mov ax, BaseOfKernelFile
126         mov es, ax
127         mov bx, OffsetOfKernelFile
128         mov ax, cx
129
130     LABEL_GOON_LOADING_FILE:
131         ; 打印“.”
132         push ax
133         push bx
134         mov ah, 0Eh
135         mov al, '.'
136         mov bl, 0Fh
137         int 10h
138         pop bx
139         pop ax
140
141         mov cl, 1
142         call ReadSector
```

```
143         pop ax
144         call GetFATEntry
145         cmp ax, 0FFFh
146         jz LABEL_FILE_LOADED
147         push ax
148         mov dx, RootDirSectors
149         add ax, dx
150         add ax, DeltaSectorNo
151         add bx, [BPB_BytePerSec]
152         jmp LABEL_GOON_LOADING_FILE
153
154     LABEL_FILE_LOADED:
155         call KillMotor
156
157         mov dh, 1
158         call DispStr
159
160         jmp $
161
162     MessageLength equ 9
163     LoadMessage db "Loading  "
164     Message1    db "Ready    "
165     Message2    db "No Kernel"
166
167     DispStr:
168         mov ax, MessageLength
169         mul dh
170         add ax,
171
172     KillMotor:
173         push dx
174         mov dx, 03F2h
175         mov al, 0
176         out dx, al
177         pop dx
178         ret
```

2 跳入保护模式

在将内核加载进入内存之后，我们将跳入保护模式。

2.1 定义 GDT 表

首先我们将创建 GDT 表，其中存放三个段描述符，分别是 0 ～ 4GB 的可执行段、0 ～ 4GB 的可读写段和指向显存开始地址的段。

定义代码如下所示：

```
1   %macro Descriptor 3
2       dw %2 & 0FFFFh
```

```
        dw %1 & 0FFFFh
        db (%1 >> 16) & 0FFh
        dw ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh)
        db (%1 >> 24) & 0FFh
    %endmacro

    LABEL_GDT:   Descriptor 0, 0, 0
    LABEL_DESC_FLAT_C:   Descriptor 0, 0fffffh, DA_CR | DA_32 | DA_LIMIT_4K
    LABEL_DESC_FLAT_RW:  Descriptor 0, 0fffffh, DA_DRW | DA_32 | DA_LIMIT_4K
    LABEL_DESC_VIDEO:    Descriptor 0B8000h, 0ffffh, DA_DRW | DA_DPL3

    GdtLen equ $ - LABEL_GDT
    GdtPtr dw GdtLen - 1
           dd BaseOfLoaderPhyAddr + LABEL_GDT ; BaseOfLoaderPhyAddr是Loader的实际物
               理地址，加上LABEL_GDT后，是GDT的实际物理地址

    SelectorFlatC equ LABEL_DESC_FLAT_C - LABEL_GDT
    SelectorFlatRW equ LABEL_DESC_FLAT_RW - LABEL_GDT
    SelectorVideo equ LABEL_DESC_VIDEO - LABEL_GDT + SA_RPL3
```

需要知道的是，Loader 的段基址是 BaseOfLoader，所以 Loader 中标号的物理地址可以用“BaseOfLoader*10h+ 标号的偏移”算出。

2.2 进入保护模式

进入保护模式的代码如下:

```
    [SECTION .s32]
    ALIGN 32
    [BITS 32]
    LABEL_PM_START:
        mov ax, SelectorVideo
        mov gs, ax
        mov ah, 0Fh
        mov al, 'P'
        mov (gs:((80*0+39)*2)), ax
        jmp $

    LABEL_FILE_LOADED:
        call KillMotor

        mov dh, 1
        call DispStrRealMode

        lgdt [GdtPtr]

        cli

        in al, 92h
        or al, 00000010b
        out 92h, al

        mov eax, cr0
```

```
27          or eax, 1
28          mov cr0, eax
29
30          jmp dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
```

2.3 保护模式下的操作

在保护模式下，我们可以做如下操作：

1. 初始化各个寄存器的值。
2. 获得可使用内存的情况。
3. 打开分页机制。

2.3.1 初始化各个寄存器的值

代码如下所示：

```
1     StackSpace: times 1024 db 0
2     TopOfStack equ BaseOfLoaderPhyAddr + $
3
4     [SECTION .s32]
5     ALIGN 32
6     [BITS 32]
7     LABEL_PM_START:
8         mov ax, SelectorVideo
9         mov gs, ax
10
11        mov ax, SelectorFlatRW
12        mov ds, ax
13        mov cs, ax
14        mov es, ax
15        mov fs, ax
16        mov ss, ax
17        mov esp, TopOfStack
```

2.3.2 获得可使用内存

在之前说过，使用 BIOS 中断“int 15h”可以获得内存信息。代码如下所示：

```
1         mov ebx, 0
2         mov di, _MemChkBuf
3     .MemChkLoop:
4         mov eax, 0R820h
5         mov eax, 20
6         mov edx, 053D4150h
7         int 15h
```

```
        jc .MemChkFail
        add di, 20
        inc dword [_dwMCRNumber]
        cmp ebx, 0
        jne .MemChkLoop
        jmp .MemChkOk
    .MemChkFail:
        mov dword [_dwMCRNumber], 0
    .MemChkOk:
        jmp $
```

为了让启动过程更多信息，我们还可以添加打印内存信息的函数，代码如下：

```
    DispMemInfo:
        push esi
        push edi
        push ecx

        mov esi, MemChkBuf
        mov ecx, [_dwMCRNumber]

    .loop:
        mov edx, 5
        mov edi, ARDStruct
    .1:
        push dword [esi]
        call DispInt
        pop eax

        stosd

        add esi, 4
        dec edx
        cmp edx, 0
        jnz .1
        call DispReturn
        cmp dword [dwType], 1
        jne .2
        mov eax, [dwBaseAddrLow]
        add eax, [dwLengthLow]
        cmp eax, [dwMemSize]
        jb .2
        mov [dwMemSize], eax

    .2:
        loop .loop

        call DispReturn
        push szRAMSize
        call DispStr
        add esp, 4

        push dword [dwMemSize]
        call DispInt
        add esp, 4

```

```
44          pop ecx
45          pop edi
46          pop esi
47          ret
48
49      DispAL:
50          push ecx
51          push edx
52          push edi
53
54          mov edi, [dwDispPos]
55
56          mov ah, 0Fh
57          mov dl, al
58          shr al, 4
59          mov ecx, 2
60      .begin:
61          and al, 01111b
62          cmp al, 9
63          ja .1
64          add al, '0'
65          jmp .2
66      .1:
67          sub al, 0Ah
68          add al, 'A'
69      .2:
70          mov [gs:edi], ax
71          add edi, 2
72
73          mov al, dl
74          loop .begin
75
76          mov [dwDispPos], edi
77
78          pop edi
79          pop edx
80          pop ecx
81
82          ret
83
84      ; 使用堆栈传递参数
85      DispInt:
86          mov eax, [esp + 4]
87          shr eax, 24
88          call DispAL
89
90          mov eax, [esp + 4]
91          shr eax, 16
92          call DispAL
93
94          mov eax, [esp + 4]
95          shr eax, 8
96          call DispAL
97
98          mov eax, [esp + 4]
99          call DispAL
```

```
100
101         mov ah, 07h
102         mov al, 'h'
103         push edi
104         mov edi, [dwDispPos]
105         mov [gs:edi], ax
106         add edi, 4
107         mov [dwDispPos], edi
108         pop edi
109
110         ret
111
112     ; 使用堆栈传递参数
113     DispStr:
114         push ebp
115         mov ebp, esp
116         push ebx
117         push esi
118         push edi
119
120         mov esi, [esp + 8]
121         mov edi, [dwDispPos]
122         mov ah, 0Fh
123     .1:
124         lodsb
125         test al, al
126         jz .2
127         cmp al, 0Ah
128         jnz .3
129         push eax
130         mov eax, edi
131         mov bl, 160
132         div bl
133         and eax, 0FFh
134         inc eax
135         mov bl, 160
136         mul bl
137         mov edi, eax
138         pop eax
139         jmp .1
140     .3:
141         mov [gs:edi], ax
142         add edi, 2
143         jmp .1
144     .2:
145         mov [dwDispPos], edi
146
147         pop edi
148         pop esi
149         pop ebx
150         pop ebp
151
152         ret
153
154     DispReturn:
155         push szReturn
```

```
156         call DispStr
157         add esp, 4
158 
159         ret
160 
161     LABEL_DATA:
162     ; 实模式下使用的符号
163     _szMemChkTitle: db "BaseAddrL BaseAddrH LengthLow LengthHigh   Type", 0Ah, 0
164     _szRAMSize: db "RAM size:", 0
165     _szReturn: db 0Ah, 0
166 
167     _dwMCRNumber: dd 0
168     _dwDispPos: dd (80 * 6 + 0) * 2
169     _dwMemSize: dd 0
170     _ARDStruct:
171         _dwBaseAddrLow: dd 0
172         _dwBaseAddrHigh: dd 0
173         _dwLengthLow: dd 0
174         _dwLengthHigh: dd 0
175         _dwType: dd 0
176     _MemChkBuf: times 256 db 0
177 
178     ; 保护模式下使用的符号
179     szMemChkTitle equ BaseOfLoaderPhyAddr + _szMemChkTitle
180     szRAMSize equ BaseOfLoaderPhyAddr + _szRAMSize
181     szReturn equ BaseOfLoaderPhyAddr + _szReturn
182     dwDispPos equ BaseOfLoaderPhyAddr + _dwDispPos
183     dwMemSize equ BaseOfLoaderPhyAddr + _dwMemSize
184     dwMCRNumber equ BaseOfLoaderPhyAddr + _dwMCRNumber
185     ARDStruct equ BaseOfLoaderPhyAddr + _ARDStruct
186         dwBaseAddrLow equ BaseOfLoaderPhyAddr + _dwBaseAddrLow
187         dwBaseAddrHigh equ BaseOfLoaderPhyAddr + _dwBaseAddrHigh
188         dwLengthLow equ BaseOfLoaderPhyAddr + _dwLengthLow
189         dwLengthHigh equ BaseOfLoaderPhyAddr + _dwLengthHigh
190         dwType equ BaseOfLoaderPhyAddr + _dwType
191     MemChkBuf equ BaseOfLoaderPhyAddr + _MemChkBuf
```

2.3.3 打开分页机制

启动分页的函数如下所示:

```
1     PageDirBase equ 100000h
2     PageTblBase equ 101000h
3 
4     SetupPaging:
5         xor edx, edx
6         mov edx, [dwMemSize]
7         mov ebx, 400000h
8         div ebx
9         mov ecx, eax
10        test edx, edx
11        jz .no_remainder
12        inc ecx
13    .no_remainder:
```

```
14          push ecx
15
16          mov ax, SelectorFlatRW
17          mov es, ax
18          mov edi, PageDirBase
19          xor eax, eax
20          mov eax, PageTblBase | PG_P | PG_USU | PG_RWW
21
22      .1:
23          stosd
24          add eax, 4096
25          loop .1
26
27          pop eax
28          mov ebx, 1024
29          mul ebx
30          mov ecx, eax
31          mov edi, PageTblBase
32          xor eax, eax
33          mov eax, PG_P | PG_USU | PG_RWW
34      .2:
35          stosd
36          add eax, 4096
37          loop .2
38
39          mov eax, PageDirBase
40          mov cr3, eax
41          mov eax, cr0
42          or eax, 80000000h
43          mov cr0, eax
44          jmp short .3
45      .3:
46          nop
47
48          ret
```

2.4 重新放置内核

我们将根据 ELF 文件信息将内核转移到正确的位置，也就是根据 ELF 文件中的 Program header，根据其信息进行内存复制。

2.4.1 内存复制函数

代码如下所示：

```
1   ; 使用堆栈进行参数的传递
2   MemCpy:
3       push ebp
4       mov ebp, esp
5
6       push esi
```

```
7         push edi
8         push ecx
9
10        mov edi, [ebp + 8]
11        mov esi, [ebp + 12]
12        mov ecx, [ebp + 16]
13    .1:
14        cmp ecx, 0
15        jz .2
16
17        mov al, [ds:esi]
18        inc esi
19
20        mov byte [es:edi], al
21        inc edi
22
23        dec ecx
24        jmp .1
25    .2:
26        mov eax, [ebp + 8]
27
28        pop ecx
29        pop edi
30        pop esi
31        mov esp, ebp
32        pop ebp
33
34        ret
```

2.4.2 转移内核

转移内核的函数如下:

```
1     InitKernel:
2         xor esi, esi
3         mov cx, word [BaseOfKernelFilePhyAddr + 2Ch]
4         movzx ecx, cx
5         mov esi, [BaseOfKernelFilePhyAddr + 1Ch]
6         add esi, BaseOfKernelFilePhyAddr
7     .Begin:
8         mov eax, [esi + 0]
9         cmp eax, 0
10        jz .NoAction
11        push dword [esi + 010h]
12        mov eax, [esi + 04h]
13        add eax, BaseOfKernelFilePhyAddr
14        push eax
15        push dword [esi + 08h]
16        call MemCpy
17        add esp, 12
18    .NoAction:
19        add esi, 020h
20        dec ecx
21        jnz .Begin
```

```
22
23      ret
```

3 扩充内核

3.1 重新放置堆栈和 GDT

现在 GDT 表和堆栈还存放在 loader 中，我们接下来想把它们放进内核中。

切换堆栈和 GDT 的代码如下：

```
1   SELECTOR_KERNEL_CS equ 8
2
3   extern cstart
4
5   extern gdt_ptr
6
7   [SECTION .bss]
8   StackSpace resb 2*1024
9   StackTop:
10
11  [section .text]
12  global _start
13
14  _start:
15      ; 切换堆栈
16      mov esp, StackTop
17
18      ; 更换GDT
19      sgdt [gdt_ptr]
20      call cstart
21      lgdt [gdt_ptr]
22
23      jmp SELECTOR_KERNEL_CS:csinit
24
25  csinit:
26      push 0
27      popfd
28
29      hlt
```

切换堆栈的语句是：

```
1   ; StackTop定义在.bss段中
2   ; 堆栈大小为2KB
3   mov esp, StackTop
```

更换 GDT 的语句是：

```
1   sgdt [gdt_ptr] ; 将GDT寄存器的内容存到gdt_ptr内存单元中
2   call cstart
```

```
3     lgdt [gdt_ptr] ; 将gdt_ptr内存单元中的内容加载到GDT寄存器中
```

cstart 函数将位于 Loader 中的原 GDT 全部复制给新的 GDT，然后将 gdt_ptr 中的内容改为新的 GDT 的基地址和界限。其中 gdt_ptr 和 cstart 分别是一个全局变量和全局函数，它们在 start.c 中定义：

```
1     #include "type.h"
2     #include "const.h"
3     #include "protect.h"
4
5     PUBLIC void * memcpy(void* pDst, void* pSrc, int iSize);
6
7     PUBLIC u8 gdt_ptr[6];
8     PUBLIC DESCRIPTOR gdt[GDT_SIZE];
9
10
11    PUBLIC void cstart()
12    {
13        memcpy(&gdt, (void*)(*((u32*)(&gdt_ptr[2]))), *((u16*)(&gdt_ptr[0]))+1);
14
15        u16* p_gdt_limit = (u16*)(&gdt_ptr[0]);
16        u32* p_gdt_base = (u32*)(&gdt_ptr[2]);
17        *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
18        *p_gdt_base = (u32)&gdt;
19    }
```

在上述代码中，我们可以看到“type.h”、“const.h”和“protect.h”，它们是用于方便而创建的一些头文件，如下所示：

```
1     // type.h文件内容
2     #ifndef _ORANGES_TYPE_H
3     #define _ORANGES_TYPE_H
4
5     typedef unsigned int u32;
6     typedef unsigned short u16;
7     typedef unsigned char u8;
8
9     #endif
10
11    // const.h文件内容
12    #ifndef _ORANGES_CONST_H
13    #define _ORANGES_CONST_H
14
15    #define PUBLIC
16    #define PRIVATE static
17
18    #define GDT_SIZE
19
20    #endif
21
22    // protect.h文件内容
23    #ifndef _ORANGES_PROTECT_H
24    #define _ORANGES_PROTECT_H
25
```

```
26  typedef struct s_descriptor
27  {
28      u16 limit_low;
29      u16 base_low;
30      u8 base_mid;
31      u8 attr1;
32      u8 limit_high_attr2;
33      u8 base_high;
34  }DESCRIPTOR;
35
36  #endif
```

3.2 打印字符的函数

为了让我们的操作系统在运行的时候可以显示一些信息，还需要声明一个打印字符或字符串的函数。

代码如下：

```
1   [SECTION .data]
2   disp_pos dd 0
3
4   [SECTION .text]
5
6   global disp_str
7
8   ; 这个函数使用堆栈进行参数的传递
9   disp_str:
10      push ebp
11      mov ebp, esp
12
13      mov esi, [ebp + 8]
14      mov edi, [disp_pos]
15      mov ah, 0Fh
16
17  .1:
18      lodsb
19      test al, al
20      jz .2
21      cmp al, 0Ah
22      jnz .3
23      push eax
24      mov eax, edi
25      mov bl, 160
26      div bl
27      and eax, 0FFh
28      inc eax
29      mov bl, 160
30      mul bl
31      mov edi, eax
32      pop eax
33      jmp .1
34
```

```
    .3:
        mov [gs:edi], ax
        add edi, 2
        jmp .1

    .2:
        mov [disp_pos], edi

        pop ebp
        ret
```

在 C 语言中就可以对这个函数进行正常的调用:

```
disp_str("\n\n\n\n\n\n\n\n\n\n\n\n\n\n————\"cstart\" begins————\n");
```

3.3 使用 makefile

3.3.1 makefile 快速入门

我在 github 网站上有上传关于 makefile 的笔记，这里再重温一下。
简单的 makefile 如下所示:

```
ASM = nasm
ASMFLAGS = -I include/

TARGET = boot.bin loader.bin

.PHONY: everything clean all

everything: $(TARGET)

clean:
    rm -f $(TARGET)

all: clean everything

boot.bin: boot.asm include/load.inc include/fat12hdr.inc
    $(ASM) $(ASMFLAGS) -o $@ $<

loader.bin: loader.asm include/load.inc include/fat12hdr.inc include/pm.inc
    $(ASM) $(ASMFLAGS) -o $@ $<
```

夸张的说，明白以下两点，makefile 的内容就明白了一半:

1. “=” 用来定义变量。
2. ASM 和 ASMFLAGS 就是两个变量，使用 $(ASM) 和 $(ASMFLAGS) 调用这两个变量。

makefile 最重要的语法如下:

```
target: prerequisites
    command
```

这个语法的意思是：

1. 要想得到 target，需要执行命令 command。
2. target 依赖 prerequisites，当 prerequisites 中至少有一个文件比 target 文件新时，command 才会被执行。

再来看下面的语句：

```
$(ASM) $(ASMFLAGS) -o $@ $<
```

其中，$@ 代表 target，$< 代表 prerequisites 中的第一个文件，这条语句相当于：

```
nasm -I include/ -o loader.bin loader.asm
```

再来看 everything、clean 和 all，它们 3 个不是文件，只是一个动作名称。比如“make clean”将会执行“rm -f $(TARGET)”。关键字.PHONY 用来声明这些动作名称。

all 后面跟着 clean 和 everything，这意味着如果执行“make all”，那么 clean 和 everything 下面的动作也会被执行。“make all”执行的结果如下：

```
rm -f boot.bin loader.bin
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
```

3.3.2 编译当前代码的 makefile

用于编译当前代码的 makefile 如下：

```
ENTRYPOINT = 0x30400
ENTRYOFFSET = 0x400

ASM = nasm
DASM = ndisam
CC = gcc
LD = ld
ASMBFLAGS = -I boot/include/
ASMKFLAGS = -I include/ -f elf
CFLAGS = -I include/ -c -fno-builtin
LDFLAGS = -s -Text $(ENTRYPOINT)
DASMFLAGS = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)

ORANGESBOOT = boot/boot.bin boot/loader.bin
ORANGESKERNEL = kernel.bin
OBJS = kernel/kernel.o kernel/start.o lib/kliba.o lib/string.o
DASMOUTPUT = kernel.bin.asm
```

```
18
19  .PHONY: everything final image clean realclean disasm all buildimg
20
21  everything: $(ORANGESBOOT) $(ORANGESKERNEL)
22
23  all: realclean everything
24
25  final: all clean
26
27  image: final buildimg
28
29  clean:
30      rm -f $(OBJS)
31
32  realclean:
33      rm -f $(OBJS) $(ORANGESBOOT) $(ORANGESKERNEL)
34
35  disasm:
36      $(DASM) $(DASMFLAGS) $(ORANGESKERNEL) > $(DASMOUTPUT)
37
38  buildimg:
39      dd if=boot/boot.bin of=a.img bs=512 count=1 conv=notrunc
40      sudo mount -o loop a.img /mnt/floppy/
41      sudo cp -fv boot/loader.bin /mnt/floppy/
42      sudo cp -fv kernel.bin /mnt/floppy/
43      sudo umount /mnt/floppy
44
45  boot/boot.bin: boot/boot.asm boot/include/load.inc boot/include/fat12hdr.inc
46      $(ASM) $(ASMBFLAGS) -o $@ $<
47
48  boot/loader.bin: boot/loader.asm boot/include/load.inc boot/include/
        fat12hdr.inc boot/include/pm.inc
49      $(ASM) $(ASMBFLAGS) -o $(ORANGESKERNEL) $(OBJS)
50
51  kernel/kernel.o: kernel/kernel.asm
52      $(ASM) $(ASMKFLAGS) -o $@ $<
53
54  kernel/start.o: kernel/start.c include/type.h include/const.h include/protect.h
55      $(CC) $(CFLAGS) -o $@ $<
56
57  lib/kliba.o: lib/kliba.asm
58      $(ASM) $(ASMKFLAGS) -o $@ $<
59
60  lib/string.o: lib/string.asm
61      $(ASM) $(ASMKFLAGS) -o $@ $<
```

3.4 添加中断处理

我们现在已经身处内核中，讲道理，可以开始添加进程。
但是为了让操作系统可以控制进程，我们还要添加中断处理。
首先是要设置 8259A 和建立 IDT。

3.4.1 初始化 8259A

代码如下：

```
PUBLIC void init_8259A()
{
    out_byte(INT_M_CTL, 0x11);
    out_byte(INT_S_CTL, 0x11);
    out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0);
    out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ8);
    out_byte(INT_M_CTLMASK, 0x4);
    out_byte(INT_S_CTLMASK, 0x2);
    out_byte(INT_M_CTLMASK, 0x1);
    out_byte(INT_S_CTLMASK, 0x1);
    out_byte(INT_M_CTLMASK, 0xFF);
    out_byte(INT_S_CTLMASK, 0xFF);
}
```

8259A 的端口定义在“const.h”文件中：

```
#define INT_M_CTL 0x20
#define INT_M_CTLMASK 0x21
#define INT_S_CTL 0xA0
#define INT_S_CTLMASK 0xA1
```

中断向量定义在“protect.h”文件中：

```
#define INT_VECTOR_IRQ0 0x20
#define INT_VECTOR_IRQ8 0x28
```

out_byte 函数和 in_byte 函数定义在 kliba.asm 文件中：

```
; void out_byte(u16 port, u8 value);
out_byte:
    mov edx, [esp + 4]
    mov al, [esp + 4 + 4]
    out dx, al
    nop
    nop
    ret

; void in_byte(u16 port);
in_byte:
    mov edx, [esp + 4]
    xor eax, eax
    in al, dx
    nop
    nop
    ret
```

为了更好的管理函数，我们建立一个函数声明文件：

```
PUBLIC void out_byte(u16 port, u8 value);
```

```
    PUBLIC u8 in_byte(u16 port);
    PUBLIC void disp_str(char* info);
```

3.4.2 初始化 IDT

初始化 IDT 的代码如下：

```
    #include"global.h"
        // 初始化IDT
        u16* p_idt_limit = (u16*)(&idt_ptr[0]);
        u32* p_idt_base = (u32*)(&idt_ptr[2]);
        *p_idt_limit = IDT_SIZE * sizeof(GATE) - 1;
        *p_idt_base = (u32)(&idt);
```

其中 IDT_SIZE 变量定义在“const.h”文件中：

```
    #define IDT_SIZE 256
```

GDTE 数据结构定义在“protect.h”文件中：

```
    typedef struct s_gate
    {
        u16 offset_low;
        u16 selector;
        u8 dcount;
        u8 attr;
        u16 offset_high;
    }GATE;
```

为了更好的管理这些全局变量，我们创建了一个文件“global.h”：

```
    #ifdef GLOBAL_VARIABLES_HERE
    #undef EXTERN
    #define EXTERN
    #endif

    EXTERN int disp_pos;
    EXTERN u8 gdt_ptr[6];
    EXTERN DESCRIPTOR gdt[GDT_SIZE];
    EXTERN u8 idt_ptr[6];
    EXTERN GATE idt[IDT_SIZE];
```

这里 EXTERN 这么设计，是为了让 EXTERN 在”global.h“中是空值，而在其他文件中是“extern”：

```
    // global.c的内容
    #define GLOBAL_VARIABLES_HERE

    #include "type.h"
    #include "const.h"
    #include "protect.h"
```

```
#include "proto.h"
#include "global.h"

// const.h中与EXTERN有关的内容
#define EXTERN extern
```

3.4.3 添加中断处理

当中断或异常发生时，eflags、cs 和 eip 将会被压栈，如果有错误码，那么错误码也会被压栈。

总的来说，如果有错误码，就直接把向量号压栈，然后执行一个中断处理函数。如果没有错误码，就向栈中压入一个 0xFFFFFFFF，再把向量号压栈并执行中断处理函数。

中断处理函数的声明如下：

```
void exception_handler(int vec_no, int err_code, int eip, int cs, int eflags);
```

中断和异常处理的函数如下：

```
extern idt_ptr

global _start
global divide_error
global single_step_exception
global nmi
global breakpoint_exception
global overflow
global bounds_check
global inval_opcode
global copr_not_available
global double_fault
global copr_seg_overrun
global inval_tss
global segment_not_present
global stack_exception
global general_protection
global page_fault
global copr_error

    lidt [idt_ptr] ; 加载idt_ptr内存单元中的值到IDT寄存器中

divide_error:
    push 0xFFFFFFFF
    push 0
    jmp exception

single_step_exception:
    push 0xFFFFFFFF
    push 1
    jmp exception

nmi:
```

```
34          push 0xFFFFFFFF
35          push 2
36          jmp exception
37
38      breakpoint_exception:
39          push 0xFFFFFFFF
40          push 3
41          jmp exception
42
43      overflow:
44          push 0xFFFFFFFF
45          push 4
46          jmp exception
47
48      bounds_check:
49          push 0xFFFFFFFF
50          push 5
51          jmp exception
52
53      inval_opcode:
54          push 0xFFFFFFFF
55          push 6
56          jmp exception
57
58      copr_not_available:
59          push 0xFFFFFFFF
60          push 7
61          jmp exception
62
63      double_fault:
64          push 8
65          jmp exception
66
67      copr_seg_overrun:
68          push 0xFFFFFFFF
69          push 9
70          jmp exception
71
72      inval_tss:
73          push 10
74          jmp exception
75
76      segment_not_present:
77          push 11
78          jmp exception
79
80      stack_exception:
81          push 12
82          jmp exception
83
84      general_protection:
85          push 13
86          jmp exception
87
88      page_fault:
89          push 14
```

```
90          jmp exception
91
92      copr_error:
93          push 0xFFFFFFFF
94          push 16
95          jmp exception
96
97      exception:
98          call exception_handler
99          add esp, 4*2
100         hlt
```

异常处理函数实现如下:

```
1       PUBLIC void exception_handler(int vec_no,int err_code,int eip,int cs,int eflags
            )
2       {
3           int i;
4           int text_color = 0x74; /* 灰底红字 */
5
6           char * err_msg[] = {"#DE Divide Error",
7                       "#DB RESERVED",
8                       "—— NMI Interrupt",
9                       "#BP Breakpoint",
10                      "#OF Overflow",
11                      "#BR BOUND Range Exceeded",
12                      "#UD Invalid Opcode (Undefined Opcode)",
13                      "#NM Device Not Available (No Math Coprocessor)",
14                      "#DF Double Fault",
15                      "    Coprocessor Segment Overrun (reserved)",
16                      "#TS Invalid TSS",
17                      "#NP Segment Not Present",
18                      "#SS Stack-Segment Fault",
19                      "#GP General Protection",
20                      "#PF Page Fault",
21                      "——  (Intel reserved. Do not use.)",
22                      "#MF x87 FPU Floating-Point Error (Math Fault)",
23                      "#AC Alignment Check",
24                      "#MC Machine Check",
25                      "#XF SIMD Floating-Point Exception"
26          };
27
28          /* 通过打印空格的方式清空屏幕的前五行，并把 disp_pos 清零 */
29          disp_pos = 0;
30          for(i=0;i<80*5;i++){
31              disp_str(" ");
32          }
33          disp_pos = 0;
34
35          disp_color_str("Exception! --> ", text_color);
36          disp_color_str(err_msg[vec_no], text_color);
37          disp_color_str("\n\n", text_color);
38          disp_color_str("EFLAGS:", text_color);
39          disp_int(eflags);
40          disp_color_str("CS:", text_color);
41          disp_int(cs);
```

```
42          disp_color_str("EIP:", text_color);
43          disp_int(eip);
44
45          if(err_code != 0xFFFFFFFF){
46              disp_color_str("Error code:", text_color);
47              disp_int(err_code);
48          }
49      }
```

其中 disp_color_str() 函数如下：

```
1       disp_color_str:
2           push ebp
3           mov ebp, esp
4
5           mov esi, [ebp + 8]
6           mov edi, [disp_pos]
7           mov ah, [ebp + 12]
8
9       .1:
10          lodsb
11          test al, al
12          jz .2
13          cmp al, 0Ah
14          jnz .3
15          push eax
16          mov eax, edi
17          mov bl, 160
18          div bl
19          and eax, 0FFh
20          inc eax
21          mov bl, 160
22          mul bl
23          mov edi, eax
24          pop eax
25          jmp .1
26
27      .3:
28          mov [gs:edi], ax
29          add edi, 2
30          jmp .1
31
32      .2:
33          mov [disp_pos],edi
34
35          pop ebp
36          ret
```

除了显示字符串，还编写了显示整数的函数：

```
1       PUBLIC char* itoa(char* str, int num)
2       {
3           char* p = str;
4           char ch;
5           int i;
```

```
6         int flag=0;
7
8         *p++ = '0';
9         *p++ = 'x';
10
11        if(num == 0)
12            *p++ = '0';
13        else
14        {
15            for(i = 28; i>=0; i-=4)
16            {
17                ch = (num>>1) & 0xF;
18                if(flag || (ch > 0))
19                {
20                    flag = 1;
21                    ch += '0';
22                    if(ch > '9')
23                        ch += 7;
24                    *p++ = ch;
25                }
26            }
27        }
28        *p = 0;
29        return str;
30    }
31
32    PUBLIC void disp_int(int input)
33    {
34        char output[16];
35        itoa(output, input);
36        disp_str(output);
37    }
```

3.4.4 设置 IDT

虽然添加了中断处理函数，但是还需要设置 IDT 才能实现中断机制。

首先写一个设置门描述符的函数，代码如下：

```
1     // 初始化门描述符
2     PRIVATE void init_idt_desc(unsigned char vector, u8 desc_type, int_handler
          handler, unsigned char privilege)
3     {
4         GATE* p_gate = &idt[vector];
5         u32 base = (u32)handler;
6         p_gate->offset_low = base & 0xFFFF;
7         p_gate->selector = SELECTOR_KERNEL_CS;
8         p_gate->dcount = 0;
9         p_gate->attr = desc_type | (privilege << 5);
10        p_gate->offset_high = (base >> 16) & 0xFFFF;
11    }
```

其中 int_handler 是一个函数指针，在“type.h”中定义：

```
typedef void (*int_handler)();
```

所有异常处理函数的声明需要和这个一样，如下所示：

```
void divide_error();
void single_step_exception();
void nmi();
void breakpoint_exception();
void overflow();
void bounds_check();
void inval_opcode();
void copr_not_available();
void double_fault();
void copr_seg_overrun();
void inval_tss();
void segment_not_present();
void stack_exception();
void general_protection();
void page_fault();
void copr_error();
```

设置 IDT 表的代码如下：

```
PUBLIC void init_prot()
{
    init_8259A();

    // 全部初始化成中断门(没有陷阱门)
    init_idt_desc(INT_VECTOR_DIVIDE,    DA_386IGate,
            divide_error,           PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_DEBUG,     DA_386IGate,
            single_step_exception,  PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_NMI,       DA_386IGate,
            nmi,                    PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_BREAKPOINT,    DA_386IGate,
            breakpoint_exception,   PRIVILEGE_USER);

    init_idt_desc(INT_VECTOR_OVERFLOW,  DA_386IGate,
            overflow,               PRIVILEGE_USER);

    init_idt_desc(INT_VECTOR_BOUNDS,    DA_386IGate,
            bounds_check,           PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_INVAL_OP,  DA_386IGate,
            inval_opcode,           PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_COPROC_NOT,    DA_386IGate,
            copr_not_available, PRIVILEGE_KRNL);

    init_idt_desc(INT_VECTOR_DOUBLE_FAULT,  DA_386IGate,
            double_fault,           PRIVILEGE_KRNL);
```

```
32
33          init_idt_desc(INT_VECTOR_COPROC_SEG,    DA_386IGate,
34                  copr_seg_overrun,       PRIVILEGE_KRNL);
35
36          init_idt_desc(INT_VECTOR_INVAL_TSS, DA_386IGate,
37                  inval_tss,      PRIVILEGE_KRNL);
38
39          init_idt_desc(INT_VECTOR_SEG_NOT,   DA_386IGate,
40                  segment_not_present,    PRIVILEGE_KRNL);
41
42          init_idt_desc(INT_VECTOR_STACK_FAULT,   DA_386IGate,
43                  stack_exception,        PRIVILEGE_KRNL);
44
45          init_idt_desc(INT_VECTOR_PROTECTION,    DA_386IGate,
46                  general_protection, PRIVILEGE_KRNL);
47
48          init_idt_desc(INT_VECTOR_PAGE_FAULT,    DA_386IGate,
49                  page_fault,     PRIVILEGE_KRNL);
50
51          init_idt_desc(INT_VECTOR_COPROC_ERR,    DA_386IGate,
52                  copr_error,     PRIVILEGE_KRNL);
53      }
```

3.4.5 设置外部中断程序

虽然我们初始化了 8259A，但是我们并没有设置相应的外部中断程序。

两片级联的 8259A 可以挂接 15 个不同的外部设备，也就有 15 个中断处理程序。和之前设置 IDT 表的中断处理程序的过程相似，首先设置中断例程：

```
1   extern  spurious_irq
2
3   global  hwint00
4   global  hwint01
5   global  hwint02
6   global  hwint03
7   global  hwint04
8   global  hwint05
9   global  hwint06
10  global  hwint07
11  global  hwint08
12  global  hwint09
13  global  hwint10
14  global  hwint11
15  global  hwint12
16  global  hwint13
17  global  hwint14
18  global  hwint15
19
20  %macro  hwint_master    1
21      push    %1
22      call    spurious_irq
23      add     esp, 4
24      hlt
```

```
25      %endmacro
26      ; ---------------------------------
27
28      ALIGN   16
29      hwint00:                ; Interrupt routine for irq 0 (the clock).
30              hwint_master    0
31
32      ALIGN   16
33      hwint01:                ; Interrupt routine for irq 1 (keyboard)
34              hwint_master    1
35
36      ALIGN   16
37      hwint02:                ; Interrupt routine for irq 2 (cascade!)
38              hwint_master    2
39
40      ALIGN   16
41      hwint03:                ; Interrupt routine for irq 3 (second serial)
42              hwint_master    3
43
44      ALIGN   16
45      hwint04:                ; Interrupt routine for irq 4 (first serial)
46              hwint_master    4
47
48      ALIGN   16
49      hwint05:                ; Interrupt routine for irq 5 (XT winchester)
50              hwint_master    5
51
52      ALIGN   16
53      hwint06:                ; Interrupt routine for irq 6 (floppy)
54              hwint_master    6
55
56      ALIGN   16
57      hwint07:                ; Interrupt routine for irq 7 (printer)
58              hwint_master    7
59
60      ; ---------------------------------
61      %macro  hwint_slave     1
62              push    %1
63              call    spurious_irq
64              add     esp, 4
65              hlt
66      %endmacro
67      ; ---------------------------------
68
69      ALIGN   16
70      hwint08:                ; Interrupt routine for irq 8 (realtime clock).
71              hwint_slave     8
72
73      ALIGN   16
74      hwint09:                ; Interrupt routine for irq 9 (irq 2 redirected)
75              hwint_slave     9
76
77      ALIGN   16
78      hwint10:                ; Interrupt routine for irq 10
79              hwint_slave     10
80
```

```
81      ALIGN   16
82      hwint11:                ; Interrupt routine for irq 11
83              hwint_slave     11
84
85      ALIGN   16
86      hwint12:                ; Interrupt routine for irq 12
87              hwint_slave     12
88
89      ALIGN   16
90      hwint13:                ; Interrupt routine for irq 13 (FPU exception)
91              hwint_slave     13
92
93      ALIGN   16
94      hwint14:                ; Interrupt routine for irq 14 (AT winchester)
95              hwint_slave     14
96
97      ALIGN   16
98      hwint15:                ; Interrupt routine for irq 15
99              hwint_slave     15
```

这里的 spurious_irq() 函数的实现代码如下：

```
1   PUBLIC void spurious_irq(int irq)
2   {
3       disp_str("spurious_irq: ");
4       disp_int(irq);
5       disp_str("\n");
6   }
```

然后再在 IDT 表中填入外部中断处理函数，如下所示：

```
1   void    hwint00();
2   void    hwint01();
3   void    hwint02();
4   void    hwint03();
5   void    hwint04();
6   void    hwint05();
7   void    hwint06();
8   void    hwint07();
9   void    hwint08();
10  void    hwint09();
11  void    hwint10();
12  void    hwint11();
13  void    hwint12();
14  void    hwint13();
15  void    hwint14();
16  void    hwint15();
17
18  /*======================================================================*
19                              init_prot
20  *======================================================================*/
21  PUBLIC void init_prot()
22  {
23      // ...
24      init_idt_desc(INT_VECTOR_IRQ0 + 0,      DA_386IGate,
```

```
25                      hwint00,                PRIVILEGE_KRNL);
26
27          init_idt_desc(INT_VECTOR_IRQ0 + 1,      DA_386IGate,
28                      hwint01,                PRIVILEGE_KRNL);
29
30          init_idt_desc(INT_VECTOR_IRQ0 + 2,      DA_386IGate,
31                      hwint02,                PRIVILEGE_KRNL);
32
33          init_idt_desc(INT_VECTOR_IRQ0 + 3,      DA_386IGate,
34                      hwint03,                PRIVILEGE_KRNL);
35
36          init_idt_desc(INT_VECTOR_IRQ0 + 4,      DA_386IGate,
37                      hwint04,                PRIVILEGE_KRNL);
38
39          init_idt_desc(INT_VECTOR_IRQ0 + 5,      DA_386IGate,
40                      hwint05,                PRIVILEGE_KRNL);
41
42          init_idt_desc(INT_VECTOR_IRQ0 + 6,      DA_386IGate,
43                      hwint06,                PRIVILEGE_KRNL);
44
45          init_idt_desc(INT_VECTOR_IRQ0 + 7,      DA_386IGate,
46                      hwint07,                PRIVILEGE_KRNL);
47
48          init_idt_desc(INT_VECTOR_IRQ8 + 0,      DA_386IGate,
49                      hwint08,                PRIVILEGE_KRNL);
50
51          init_idt_desc(INT_VECTOR_IRQ8 + 1,      DA_386IGate,
52                      hwint09,                PRIVILEGE_KRNL);
53
54          init_idt_desc(INT_VECTOR_IRQ8 + 2,      DA_386IGate,
55                      hwint10,                PRIVILEGE_KRNL);
56
57          init_idt_desc(INT_VECTOR_IRQ8 + 3,      DA_386IGate,
58                      hwint11,                PRIVILEGE_KRNL);
59
60          init_idt_desc(INT_VECTOR_IRQ8 + 4,      DA_386IGate,
61                      hwint12,                PRIVILEGE_KRNL);
62
63          init_idt_desc(INT_VECTOR_IRQ8 + 5,      DA_386IGate,
64                      hwint13,                PRIVILEGE_KRNL);
65
66          init_idt_desc(INT_VECTOR_IRQ8 + 6,      DA_386IGate,
67                      hwint14,                PRIVILEGE_KRNL);
68
69          init_idt_desc(INT_VECTOR_IRQ8 + 7,      DA_386IGate,
70                      hwint15,                PRIVILEGE_KRNL);
71      }
```

需要认识到，是因为我们设置了 IDT 表，并且将它放入 IDT 寄存器，所以有中断时，才会有硬件机制响应并触发对应的中断处理函数。