

目 录

1	进入内核前的准备	2
1.1	跳入保护模式	2
1.2	开启分页机制	3
1.3	重新放置内核	6
1.3.1	program header table	7
1.4	重新放置内核	8
1.5	进入内核	11
2	内核雏形	12
2.1	定义相关的数据结构和数据类型	12
2.2	C 与汇编程序的相互调用	13
2.2.1	在汇编程序中调用 C 函数	13
2.2.2	在 C 程序中调用汇编函数	14
2.3	移动 GDT 到内核	15
2.3.1	memcpy 函数	15
2.3.2	移动 GDT	16
2.4	显示字符串的函数 disp_str()	17

1 进入内核前的准备

1.1 跳入保护模式

一个操作系统从开机到开始运行，需要经历“引导，加载内核进入内存，跳入保护模式，开始执行内核”。这句话我在上一次的学习报告的开头说过，上一次学习报告完成了前两个部分：引导，加载内核进入内存。我在前几次的学习报告中已经讨论过保护模式的进入，接下来的代码更多的是走一下流程，感受一下如何在 Loader 中跳入保护模式。

我们定义了两个段描述符，一个是可执行段，一个是可读写段。这两个段的大小都是 4GB，基址为 0，所以之后主要靠段内的偏移来进行程序的转移。在感受代码前，我们应该意识到，Loader 模块的物理基址是 $\text{BaseOfLoader} * 10h + \text{OffsetOfLoader}$ 。所以在 Loader 文件中的标号的实际物理地址都是该物理基址加上标号在文件中的偏移。

```

1   BaseOfLoader equ 09000h
2   OffsetOfLoader equ 0100h
3   ; BaseOfLoader是段基址，所以实际物理地址需要左移4位
4   BaseOfLoaderPhyAddr equ BaseOfLoader * 10h + OffsetOfLoader
5
6   LABEL_GDT: Descriptor 0, 0, 0
7   LABEL_DESC_FLAT_C: Descriptor 0, 0ffffh, DA_CR | DA_32 | DA_LIMIT_4K
8   LABEL_DESC_FLAT_RW: Descriptor 0, 0ffffh, DA_DRW | DA_32 | DA_LIMIT_4K
9
10  GdtLen equ $ - LABEL_GDT
11  GdtPtr dw GdtLen - 1
12         dd BaseOfLoaderPhyAddr + LABEL_GDT
13
14  SelectorFlatC equ LABEL_DESC_FLAT_C - LABEL_GDT
15  SelectorFlatRW equ LABEL_DESC_FLAT_RW - LABEL_GDT
16  ; ...
17  LABEL_FILE_LOADED:
18  ; ...
19      lgdt [GdtPtr]
20      cli
21
22      in al, 92h
23      or al, 00000010b
24      out 92h, al
25
26      mov eax, cr0
27      or eax, 1
28      mov cr0, eax
29
30      jmp dword SelectorFlatC : (BaseOfLoaderPhyAddr + LABEL_PM_START)
31
32  [SECTION .s32]
33  ALIGN 32
34  [BITS 32]
35  LABEL_PM_START:
36      $

```

1.2 开启分页机制

既然我们进入了保护模式，就能开启分页机制了。这里开启分页机制的思路和前几次学习报告中的思路是一样的，所以在代码中不会有很多注释。

在此重温一下 ARDS 数据结构：

偏移	名称	意义
0	BaseAddrLow	基地址的低 32 位
4	BaseAddrHigh	基地址的高 32 位
8	LengthLow	长度的低 32 位
12	LengthHigh	长度的高 32 位
16	Type	这个地址范围的地址类型

其中，Type 的取值有三种类型：

1	表示这个内存段是一段可以被 OS 使用的 RAM
2	表示这个地址段正在被使用或者被系统保留，所以一定不可以被 OS 使用。
其他	表示这个内存段被保留，留作以后使用，不可以被 OS 使用。

```

1 LABEL_START:
2     mov ax, cs
3     mov ds, ax
4     mov es, ax
5     mov ss, ax
6     mov sp, BaseOfStack
7
8     ; int 15h, 将ARDS结构体读入es:di所指的位置
9     ; ebx用于寻找下一个地址范围描述符结构ARDS。首次调用int 15h, ebx的值应该设为
10    0
11    mov ebx, 0
12    mov di, _MemChkBuf
13    .MemChkLoop:
14    mov eax, 0E820h
15    ; ARDS共20字节, 所以通过ecx指定int 15h向es:di读入20字节的内容
16    mov ecx, 20
17    mov edx, 0534D4150h
18    int 15h
19    ; 当CF=1时, 表示发生错误。
20    jc .MemChkFail
21    ; 让es:di指向缓冲区的下一个20字节
22    add di, 20
23    inc dword [_dwMCRNumber]
24    ; 如果ebx的值为0, 代表着当前的ARDS是最后一个地址范围描述符。

```

```

24     cmp ebx, 0
25     jne .MemChkLoop
26     jmp .MemChkOk
27 .MemChkFail:
28     mov dword [_dwMCRNumber], 0
29 .MemChkOk:
30     ; ...
31
32 [SECTION .s32]
33 ALIGN
34 [BITS 32]
35 LABEL_PM_START:
36     mov ax, SelectorFlatRW
37     mov ds, ax
38     mov es, ax
39     mov fs, ax
40     mov ss, ax
41     mov esp, TopOfStack
42
43     call DispMemInfo
44     call SetupPaging
45
46     jmp $
47
48 DispMemInfo:
49     push esi
50     push edi
51     push ecx
52     ; ds:si 指向存放ARDS的数据缓冲区
53     mov esi, MemChkBuf
54     ; ecx 存放着ARDS的数目
55     mov ecx, [dwMCRNumber]
56 .loop:
57     ; ARDSStruct 用于存放数据缓冲区当前指向的ARDS结构
58     ; edi 指向ARDS结构体的成员
59     mov edx, 5
60     mov edi, ARDSStruct
61 .l1:
62     mov eax, dword [esi]
63     ; 将eax中存放的数据放入es:edi中, edi自动加4
64     stosd
65     ; 用于指向数据缓冲区的下一个ARDS结构
66     add esi, 4
67     dec edx
68     cmp edx, 0
69     jnz .l1
70     cmp dword [dwType], 1
71     jne .l2
72     ; 这里只需要使用低32位的地址和长度, 因为高32位没有用到
73     ; 地址加长度的最大值就是内存空间的大小
74     mov eax, [dwBaseAddrLow]
75     add eax, [dwLengthLow]
76     cmp eax, [dwMemSize]
77     jb .l2
78     mov [dwMemSize], eax
79 .l2:

```

```

80 ; 使用loop指令会检查ecx是否为0，并将ecx减一
81 loop .loop
82
83 pop ecx
84 pop edi
85 pop esi
86 ret
87
88 SetupPaging:
89     xor edx, edx
90     mov eax, [dwMemSize]
91     ; 一个页表对应4k*1024的内存大小
92     mov ebx, 400000h
93     div ebx
94     mov ecx, eax
95     ; 检查edx是否为0。test指令将执行and操作，不过不会保存执行的结果
96     test edx, edx
97     jz .no_remainder
98     ; 余数不为0的话，页表数目加一
99     inc ecx
100 .no_remainder:
101     push ecx
102
103     mov ax, SelectorFlatRW
104     mov es, ax
105     mov edi, PageDirBase
106     xor eax, eax
107     mov eax, PageTblBase | PG_P | PG_USU | PG_RWW
108 .1:
109     ; 将eax中的内容存入es:eax中
110     stosd
111     add eax, 4096
112     loop .1
113
114     pop eax
115     mov ebx, 1024
116     mul ebx
117     ; 需要初始化的页帧数目为1024*页表数
118     mov ecx, eax
119     mov edi, PageTblBase
120     xor eax, eax
121     mov eax, PG_P | PG_USU | PG_RWW
122 .2:
123     stosd
124     add eax, 4096
125     loop .2
126
127     ; 将目录基址存入cr3中
128     mov eax, PageDirBase
129     mov cr3, eax
130     ; 将cr0的PG位置一
131     mov eax, cr0
132     or eax, 80000000h
133     mov cr0, eax
134     jmp short .3
135 .3:

```

```

136     nop
137     ret
138
139     [SECTION .data1]
140     ALIGN 32
141
142     LABEL_DATA:
143     _szMemChkTitle: db "BaseAddrL BaseAddrH LengthLow LengthHigh Type", 0Ah, 0
144     _szRAMSize: db "RAM size:", 0
145     _szReturn: db 0Ah, 0
146
147     ; 用于存放ARDS结构的数量
148     _dwMCRNumber: dd 0
149     ; 用于存放可用内存的大小
150     _dwMemSize: dd 0
151     ; 代表了ARDS结构
152     _ARDSStruct:
153         _dwBaseAddrLow: dd 0
154         _dwBaseAddrHigh: dd 0
155         _dwLengthLow: dd 0
156         _dwLengthHigh: dd 0
157         _dwType: dd 0
158     _MemChkBuf: times 256 db 0
159
160     szMemChkTitle equ BaseOfLoaderPhyAddr + _szMemChkTitle
161     szRAMSize equ BaseOfLoaderPhyAddr + _szRAMSize
162     szReturn equ BaseOfLoaderPhyAddr + _szReturn
163     ; 代表了_dwMCRNumber的地址
164     dwMCRNumber equ BaseOfLoaderPhyAddr + _dwMCRNumber
165     ; 代表了_dwMemSize的地址
166     dwMemSize equ BaseOfLoaderPhyAddr + _dwMemSize
167     ; 代表了ARDS结构各成员的地址
168     ARDSStruct equ BaseOfLoaderPhyAddr + _ARDSStruct
169         dwBaseAddrLow equ BaseOfLoaderPhyAddr + _dwBaseAddrLow
170         dwBaseAddrHigh equ BaseOfLoaderPhyAddr + _dwBaseAddrHigh
171         dwLengthLow equ BaseOfLoaderPhyAddr + _dwLengthLow
172         dwLengthHigh equ BaseOfLoaderPhyAddr + _dwLengthHigh
173         dwType equ BaseOfLoaderPhyAddr + _dwType
174     MemChkBuf equ BaseOfLoaderPhyAddr + _MemChkBuf
175
176     PageDirBase equ 100000h
177     PageTblBase equ 101000h
178
179     StackSpace: times 1024 db 0
180     TopOfStack equ BaseOfLoaderPhyAddr + $

```

1.3 重新放置内核

我们在此需要根据内核的 program header table 的信息进行内存复制。上一次学习报告中 ELF 文件格式的内容虽然很多，但是太晦涩难懂了。在此现在复习一遍 program header table。

1.3.1 program header table

program header 的定义如下：

```

1  typedef struct
2  {
3      Elf32_Word p_type;
4      Elf32_Off p_offset;
5      Elf32_Addr p_vaddr;
6      Elf32_Addr p_paddr;
7      Elf32_Word p_filesz;
8      Elf32_Word p_memsz;
9      Elf32_Word p_flags;
10     Elf32_Word p_align;
11 } Elf32_Phdr;

```

program header 描述的是系统准备程序运行所需的一个段的信息。结构体中各成员的意义如下：

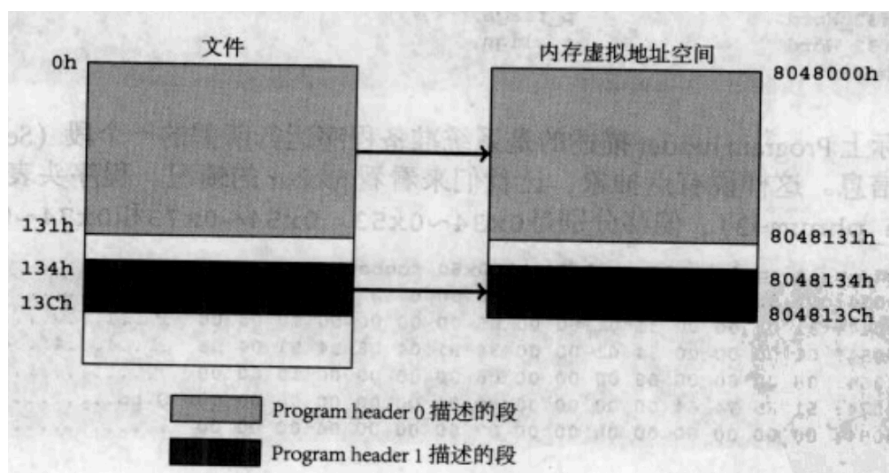
p_type	当前 program header 所描述的段的类型
p_offset	段的第一个字节在文件中的偏移
p_vaddr	段的第一个字节在内存中的虚拟地址
p_paddr	在物理地址定位相关的系统中，此项是为物理地址保留的
p_filesz	段在文件中的长度
p_memsz	段在内存中的长度
p_flags	与段相关的标志
p_align	根据此项值来确定段在文件以及内存中如何对齐

program header 描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。

举一个例子，假设一个程序的 object 文件中有 2 个 program header，取值如下：

名称	program header 0	program header 1
p_type	0x1	0x1
p_offset	0x0	0x134
p_vaddr	0x8048000	0x8049134
p_paddr	0x8048000	0x8049134
p_filesz	0x131	0x8
p_memsz	0x131	0x8
p_flags	0x5	0x6
p_align	0x1000	0x1000

那么这个程序在文件中的位置、大小以及它被放进内存后所在的位置和大小如下：



1.4 重新放置内核

由 ld 生成的可执行文件中 p_vaddr 的值总是一个类似于 0x8048XXX 的值。我们不能让编译器来决定内核加载到什么地方，有两种解决的方法：

- 一是通过修改页表让 0x8048XXX 映射到较低的地址。
- 另一种方法是通过修改 ld 的选项让它生成的可执行代码中 p_vaddr 的值变小。

第二种方法如下所示：

```
1 nasm -f elf -o kernel.o kernel.asm
2 ld -s -Ttext 0x30400 -o kernel.bin kernel.o
```

这样子就将程序的入口地址变为 0x30400，ELF header 的信息会位于 0x30400 之前。ELF header 的内容如下所示：

e_ident	...	
e_type	2H	可执行文件
e_machine	3H	80386
e_version	1H	
e_entry	30400H	入口地址
e_phoff	34H	program header table 在文件中的偏移量
e_shoff	448H	section header table 在文件中的偏移量
e_flags	0H	
e_ehsize	34H	ELF header 大小

e_phentsize	20H	每一个 program header 大小 20H 字节
e_phnum	1H	program header table 中只有一个条目
e_shentsize	28H	每一个 section header 大小 28H 字节
e_shnum	4H	section header table 有 4 个条目
e_shstndx	3H	包含节名称的字符串表是第 3 个节

实际上，我们需要根据 program header table 的信息对内核在内存中的位置进行整理。上面通过 ld 调整程序的入口地址，只是为了让可执行代码中的 p_vaddr 的值变小。在实际操作过程中，我们通过 program header 得知段在文件中的偏移 p_offset，得知段的大小 p_filesz，然后将它移动到段在内存的虚拟地址 p_vaddr。如下所示：

```
1 memcpy(p_vaddr, BaseOfLoaderPhyAddr + p_offset, p_filesz);
```

如果 program header 的内容如下所示：

p_type	1H	PT_LOAD
p_offset	0H	段的第一个字节在文件中的偏移
p_vaddr	30000H	段的第一个字节在内存中的虚拟地址
p_paddr	30000H	
p_filesz	40DH	段在文件中的长度
p_memsz	40DH	段在内存中的长度
p_flags	5H	
p_align	1000H	

这样的话，我们内核放置函数的语句如下：

```
1 memcpy(3000h, 9000h + 0, 40Dh);
```

memcpy 函数在第五次学习报告中有提到过，这里将它的代码实现复习一下：

```
1 ; 该函数使用堆栈读入输入参数，使用 eax 作为返回参数
2 ; 函数有三个参数，分别是 dest、source 和 size
3 MemCpy:
4     push ebp
5     mov ebp, esp
6
7     push esi
8     push edi
9     push ecx
10    ; 目的地址单元
11    mov edi, [ebp + 8]
12    ; 源数据地址单元
13    mov esi, [ebp + 12]
```

```

14      ; 源数据内容大小
15      mov ecx, [ebp + 16]
16      .1:
17          cmp ecx, 0
18          jz .2
19          ; 将源数据的1字节读入al中
20          mov al, [ds:esi]
21          ; 让esi指向下一字节
22          inc esi
23          ; 将al的值读入目的地址单元中
24          mov byte [es:edi], al
25          ; 让esi指向下一字节
26          inc edi
27
28          dec ecx
29          jmp .1
30      .2:
31          ; 让eax存放目的地址
32          mov eax, [ebp + 8]
33
34          pop ecx
35          pop edi
36          pop esi
37          mov esp, ebp
38          pop ebp
39          ret

```

在这里我总结一下整理内核的步骤:

- 根据 ELF header 得到 e_phnum、e_phoff。利用这两个值就可以遍历 program header table 中所有的 program header。
- 根据 program header 得到 p_vaddr、p_offset 和 p_filesz。将这三个参数从后往前压栈，调用 MemCpy 函数。

这样一来，就能把内核的各个段安排到内存中合适的位置，代码如下：

```

1      call InitKernel
2      ; ...
3      InitKernel:
4          xor esi, esi
5          ; 将e_phnum移入cx中
6          mov cx, word [BaseOfKernelFilePhyAddr+2Ch]
7          ; movzx指令将源操作数取出来,然后置于目的操作数,目的操作数其余位用0填充
8          movzx ecx, cx
9          ; 将program header table在文件中的偏移量放到esi中
10         mov esi, [BaseOfKernelFilePhyAddr+1Ch]
11         ; 让esi指向program header table
12         add esi, BaseOfKernelFilePhyAddr
13     .Begin:
14         ; 将program header成员p_type的值移入eax
15         mov eax, [esi + 0]
16         ; 检查p_type是否是PT_NULL
17         cmp eax, 0

```

```

18 ; 如果是，就避开这个program header
19 jz .NoAction
20 ; 将p_filesz的值压栈，也就是将段在文件中的长度压栈
21 ; ，作为函数所需的源数据大小
22 push dword [esi + 010h]
23 ; 将p_offset的值存放eax，也就是将段在文件中的偏移存入eax
24 mov eax, [esi + 04h]
25 ; 加上内核实际物理地址，得到段在文件中的实际物理地址
26 add eax, BaseOfKernelFilePhyAddr
27 ; 将段的实际物理地址压栈，作为函数所需的源数据地址
28 push eax
29 ; 将段在内存的虚拟地址压栈，作为函数所需的目的地地址
30 push dword [esi + 08h]
31 ; 调用MemCpy函数
32 call MemCpy
33 ; 清除堆栈中的参数
34 add esp, 12
35 .NoAction:
36 ; 一个program header有32个字节
37 ; 让esi指向下一个program header
38 add esi, 020h
39 ; 将ecx的值减一
40 dec ecx
41 jnz .Begin
42
43 ret

```

1.5 进入内核

进入内核的代码很简单，相当于开始指向内核的代码，只要向内核跳转即可，代码如下：

```

1 ; 030400h是内核的程序入口
2 jmp SelectorFlatC:030400h

```

这样操作系统就真正地开始执行内核了。“引导，加载内核进入内存，跳入保护模式，开始执行内核”这四部分内容到此我们都完成了。

2 内核雏形

2.1 定义相关的数据结构和数据类型

为了让内核的代码更清楚，在本节将定义一些相关的数据结构和数据类型。

我定义了宏 PUBLIC 和 PRIVATE 来区分文件中的普通函数和静态函数，用于更好地区分 static 函数和可以被其他文件引用的函数，

```

1  #ifndef _CONST_H_
2  #define _CONST_H_
3
4  #define PUBLIC
5  #define PRIVATE static
6
7  #define GDT_SIZE 128
8
9  #endif

```

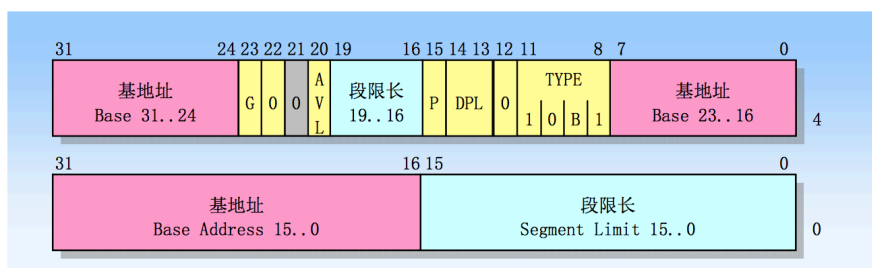
为了在处理段描述符或其他数据结构时，能清楚每个成员类型的长度，我在 type.h 中定义了 u8、u16 和 u32 等类型，分别代表 8 位、16 位和 32 位的数据类型。

```

1  #ifndef _TYPE_H
2  #define _TYPE_H
3
4  typedef unsigned int u32;
5  typedef unsigned short u16;
6  typedef unsigned char u8;
7
8  #endif

```

同时我用 C 语言定义了段描述符结构体。因为段描述符是第三次学习报告的内容，时间隔了比较久，所以在此再复习一下段描述符。先看一下它的格式：



对它的各个位解释如下：

0-15 位	段限长的第一部分，16 位
16-31 位	基地址的第一部分，16 位

32-39 位	基地址的第二部分，8 位
40-47 位	段描述符的属性的第一部分，8 位
48-55 位	8 位，低 4 位是段限长的第二部分， 高 4 位是属性的第二部分
56-63 位	基地址的第三部分，8 位

```
1  #ifndef _PROTECT_H_
2  #define _PROTECT_H_
3
4  typedef struct s_descriptor
5  {
6      u16 limit_low;
7      u16 base_low;
8      u8  base_mid;
9      u8  attr1;
10     u8 limit_high_attr2;
11     u8 base_high;
12 }DESCRIPTOR;
13
14 #endif
```

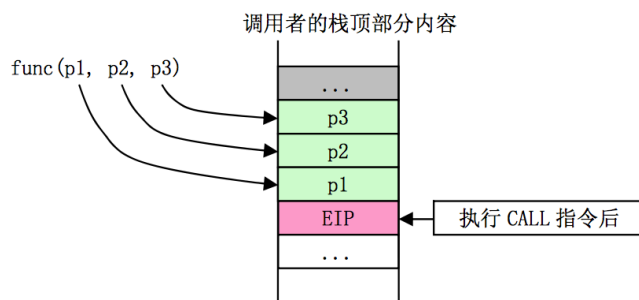
2.2 C 与汇编程序的相互调用

因为之后经常会涉及到 C 语言和汇编程序的相互调用，所以学习这个是很有必要的。

2.2.1 在汇编程序中调用 C 函数

在汇编程序调用一个 C 函数时，程序首先需要按照逆向顺序把函数参数压入栈中，即函数最右的一个参数最先压入栈，函数最左的一个参数最后压入栈，然后执行 call 指令。在调用函数返回后，程序需要将先前压入栈中的函数参数清除掉。

汇编中调用 C 函数的时候没有对参数的压栈有严格的约束。首先见下图这个例子：



如果我们没有专门为调用函数 func() 压入参数就直接调用它的话, 那么 func() 函数仍然会把存放 EIP 位置以上的栈中其他内容作为自己的参数使用。

2.2.2 在 C 程序中调用汇编函数

在 C 程序中调用汇编函数时, 类似于汇编程序调用 C 函数, 程序也是将函数的最右参数最先压入栈, 将函数的最左参数最后压入栈。

下面是一个例子:

```
1  # 导出_myadd函数
2  .global _myadd
3
4  .text
5  # int myadd(int a, int b, int* res);
6  # eax存放着返回值
7  _myadd:
8      pushl %ebp
9      movl %esp, %ebp
10     # 4(%ebp)地址单元存放着EIP
11     # 8(%ebp)地址单元存放着第一个参数
12     # 12(%ebp)地址单元存放着第二个参数
13     # 16(%ebp)地址单元存放着第三个参数
14     movl 8(%ebp), %eax
15     movl 12(%ebp), %edx
16     xorl %ecx, %ecx
17     addl %eax, %edx
18     jo 1f
19     movl 16(%ebp), %eax
20     movl %edx, (%eax)
21     incl %ecx
22 1:
23     movl %ecx, %eax
24     movl %ebp, %esp
25     popl %ebp
26     ret
```

```
1  int myadd(int a, int b, int* res);
2
3  int main()
4  {
5      int a = 5;
6      int b = 10;
7      int c;
8      myadd(a, b, &c);
9      return 0;
10 }
```

2.3 移动 GDT 到内核

之前我们在 Loader 模块内定义了 GDT 表，随后就开始执行内核。为了继续利用 Loader 中 GDT 表的信息，应该将 GDT 表复制到内核中。

目前我们 gdt 寄存器中存放着 Loader 模块中 GDT 表的基地址以及 GDT 表的长度。而我们在内核中重新定义一个 GDT 表，也是有一个地址，于是就可以使用 memcpy 函数将 Loader 中的 GDT 表的信息复制到内核中的 GDT 表中。随后，我们应该更新 gdt 寄存器中的信息。

2.3.1 memcpy 函数

在学习 Loader 模块或分页机制的时候，我们都有接触过 MemCpy 函数。这里在 linux 汇编下再来感受一下，并且体验如何在 C 程序中调用这个函数。代码如下：

```

1  [SECTION .text]
2  global memcpy
3  ; void* memcpy(void* es:dest, void* ds:src, int iSize);
4  ; eax 存放着返回值
5  memcpy:
6      push ebp
7      ; 让 ebp 指向栈顶
8      mov ebp, esp
9      push esi
10     push edi
11     push ecx
12
13     ; ebp+4 存放着 EIP
14     ; ebp+8 存放着 dest
15     ; ebp+12 存放着 src
16     ; ebp+16 存放着 iSize
17     mov edi, [ebp + 8]
18     mov esi, [ebp + 12]
19     mov ecx, [ebp + 16]
20
21     .1:
22     cmp ecx, 0
23     jz .2
24
25     mov al, [ds:esi]
26     inc esi
27
28     mov byte [es:edi], al
29     inc edi
30
31     dec ecx
32     jmp .1
33
34     .2:
35     ; 让 eax 存放着目的地址，作为返回值
36     mov eax, [ebp + 8]
37
38     pop ecx
39     pop edi

```

```

38     pop esi
39     mov esp, ebp
40     pop ebp
41
42     ret

```

在 C 程序中调用 memcpy 函数如下所示：

```

1     void* memcpy(void* dest, void* src, int iSize);
2
3     int main()
4     {
5         int dest[10];
6         int src[10] = {1,2,3,4,5,6,7,8,9,10};
7         memcpy(dest, src, 10);
8         return 0;
9     }

```

2.3.2 移动 GDT

有了前面的铺垫和思路的介绍，将 Loader 模块中 GDT 的内容复制到内核中新的 GDT 表中的代码就清晰了很多。代码如下：

```

1     SELECTOR_KERNEL_CS equ 8
2     ; 引入函数
3     extern cstart
4     ; 引入全局变量
5     extern gdt_ptr
6
7     ; bss段用于放置未初始化的变量
8     [section .bss]
9     LABEL_STACK:
10    StackSpace resb 2*1024
11    StackTop:
12
13    ; text段用于放置代码
14    [section .text]
15
16    ; 导出 _start
17    global _start
18    _start:
19        ; 让 esp 指向 StackTop
20        mov esp, StackTop
21        ; 将全局描述符表寄存器的内容复制给 gdt_ptr 地址单元
22        sgdt [gdt_ptr]
23        ; 调用 cstart 函数
24        call cstart
25        ; 将 gdt_ptr 地址单元中的内容加载到全局描述符寄存器中
26        lgdt [gdt_ptr]
27
28        jmp SELECTOR_KERNEL_CS: csinit
29
30    csinit:

```



```

31     push 0
32     popfd
33     hlt

```

```

1     #include "type.h"
2     #include "const.h"
3     #include "protect.h"
4
5     PUBLIC void* memcpy(void* pDst, void* pSrc, int iSize);
6     // gdt_ptr是48位, 所以这里数组大小为6
7     PUBLIC u8 gdt_ptr[6];
8     // 在内核中新的GDT表可以存放128个段描述符
9     PUBLIC DESCRIPTOR gdt[GDT_SIZE];
10
11     PUBLIC void cstart()
12     {
13         memcpy((void*)gdt, (void*)((u32*)&gdt_ptr[2]), *((u16*)&gdt_ptr[0])
14             +1);
15         // 指向gdt_ptr的低16位的地址
16         u16* p_gdt_limit = (u16*)&gdt_ptr[0];
17         // 指向gdt_ptr的高32位的地址
18         u32* p_gdt_base = (u32*)&gdt_ptr[2];
19         // 更新gdt_ptr中的GDT限长
20         *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
21         // 更新gdt_ptr中GDT表的基地址
22         *p_gdt_base = (u32)gdt;
23     }

```

2.4 显示字符串的函数 disp_str()

```

1     [SECTION .data]
2     disp_pos dd 0
3
4     [SECTION .text]
5
6     ; 导出 disp_str() 函数
7     global disp_str
8
9     ; void disp_str(char* info);
10    disp_str:
11        push ebp
12        mov ebp, esp
13
14        ; 让 esi 存放字符串的地址
15        mov esi, [ebp + 8]
16        mov edi, [disp_pos]
17        mov ah, 0Fh
18    .L1:
19        ; 将 ds:esi 指向的地址的一个字节读入 al
20        lodsb
21        ; 判断 ds:edi 是否指向字符串的尽头
22        test al, al

```

```

23      ; 如果是，就结束程序
24      jz .2
25      ; '\n'的ascii码是0Ah
26      ; 判断是否读入回车键
27      cmp al, 0Ah
28      ; 如果不是，就跳到.3直接输出当前字符串
29      jnz .3
30      ; 如果是，就显示出回车键的效果
31      ; 将eax的值压栈，保存eax的原值
32      push eax
33      ; edi指向要显示的下一个字符的位置，低8位为行号，8~15位为列号
34      ; 修改edi的值，先让eax保存edi的值
35      mov eax, edi
36      ; 每行有160个字节
37      mov bl, 160
38      ; 得到当前的行号，存放在al中
39      div bl
40      ; 将列号清零
41      and eax, 0FFh
42      ; eax中的值加一，行号加一
43      inc eax
44      mov bl, 160
45      ; 让eax指向下一行的头一个字节
46      mul bl
47      ; 修改edi的值结束，将eax中的值赋给edi
48      mov edi, eax
49      ; 恢复eax的原值
50      pop eax
51      jmp .1
52      .3:
53      mov [gs:edi], ax
54      add edi, 2
55      jmp .1
56      .2:
57      mov [disp_pos], edi
58
59      pop ebp
60      ret

```

C 程序调用 disp_str 函数的例子如下：

```

1      PUBLIC void disp_str(char* info);
2
3      int main()
4      {
5          disp_str("hello world\n");
6          return 0;
7      }

```