目 录

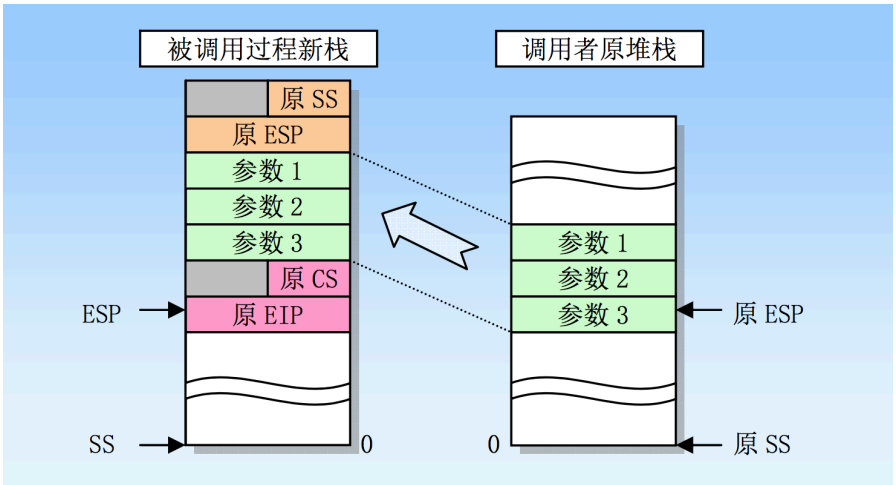
1 实现特权级转移

1.1 理论知识

特权级转移需要借助堆栈切换。当调用门用于把程序控制转移到一个更高级别的非一致性代码段时，处理器会自动切换到目的代码段特权级的堆栈。此时处理器会按照以下步骤切换堆栈：

- 当前任务的 TSS 段存放着特权级 0、1 和 2 的堆栈的初始指针值。处理器会将目的代码段的 DPL 作为新任务的 CPL，并从 TSS 中选择新栈的 SS 和 ESP。
- 将 SS 和 ESP 寄存器的当前值压入新栈，并将新栈的段选择符和栈指针加载到 SS 和 ESP。
- 将调用门描述符中指定的参数从当前栈压入新栈。参数数目由调用门描述符中的 PARAM COUNT 字段决定。
- 将 CS 和 EIP 寄存器的当前值压入新栈，并将目的代码段选择符加载到 CS，将调用门选择符中的偏移值加载到 EIP 中。

调用过程如下图所示：



当调用过程结束后，处理器使用 RET 执行远返回到一个调用过程。此时 CPU 会执行以下步骤：

- 检查保存的 CS 寄存器中的 RPL 字段值，以确定返回时特权级是否需要改变。
- 弹出新栈中的 CS 和 EIP 值，并且检查代码段描述符的 DPL 和代码段选择符的 RPL。

- 如果返回过程会改变特权级，而且此时 RET 指令包含一个参数个数操作数，那么就需要在弹出 CS 和 EIP 之后，将参数个数值加载到 ESP 寄存器中，用于丢弃新栈中的参数。
- 弹出 SS 和 ESP，从而切换回调用者的堆栈。
- 检查 DS、ES、FS 和 GS，如果其中的段选择符指向的段描述符的 DPL 小于新 CPL(仅适用于一致代码段)，处理器将用空选择符来加载这个段寄存器。

综上，使用调用门实现不同特权级之间的调用可以分为两个过程，一个是通过调用门和 call 指令实现从低特权级转移到高特权级，另一个是通过 ret 指令实现从高特权级到低特权级。

1.2 代码实现

1.2.1 高特权级到低特权级

根据上一节可知，处理器通过 ret 指令实现从高特权级到低特权级。在 ret 指令执行之前，堆栈中应该已经有了 ss、esp、cs 和 eip。如下图所示：

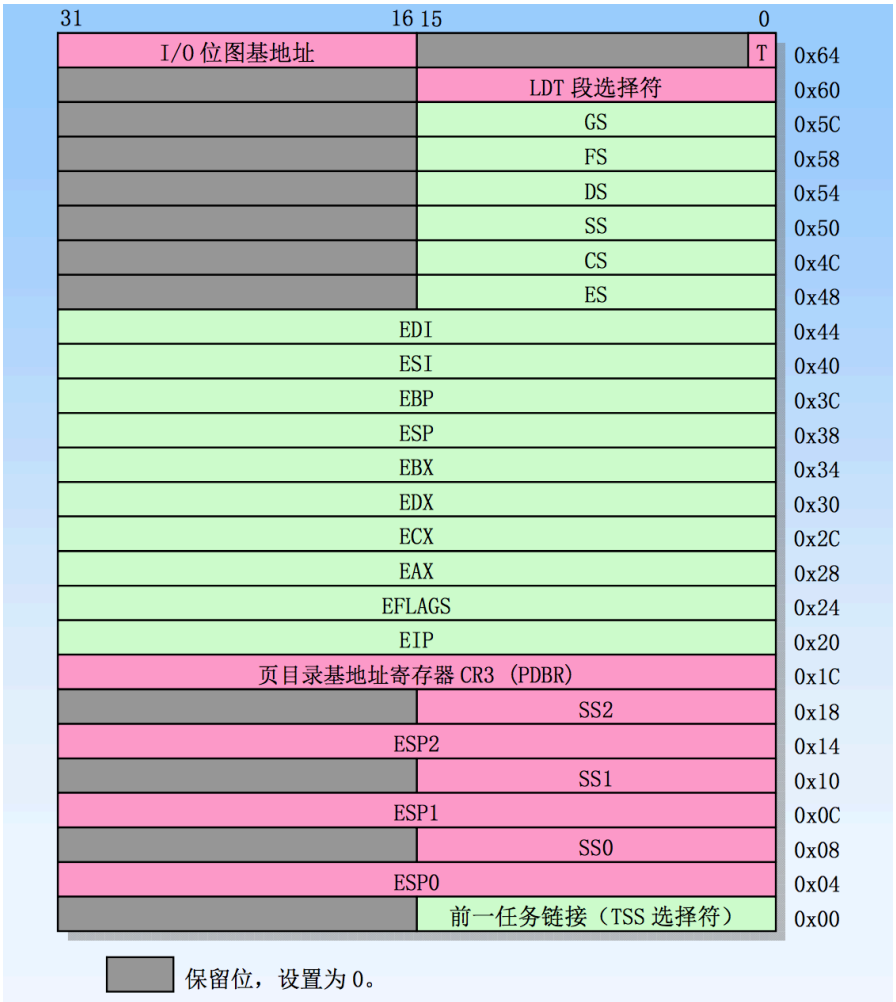
首先添加一个特权级为 3 的代码段，为了实现代码段转移，我们需要添加一个代码段和相应的堆栈段。先是在 GDT 表中添加该代码段和堆栈段的描述符，然后定义堆栈段 ring3 和代码段 ring3。随后，我们在 32 位代码段中通过执行 retf 指令跳转到 ring3 代码段中。

```
1    ; 在GDT中添加相应的代码段和堆栈段
2    [SECTION .gdt]
3    ; ...
4    ; 特权级为3
```

```
5     LABEL_DESC_CODE_RING3: Descriptor 0, SegCodeRing3Len-1, DA_C+DA_32+DA_DPL3
6     LABEL_DESC_STACK3: Descriptor 0, TopOfStack3, DA_DRWA+DA_32+DA_DPL3
7     ; ...
8     ; 请求特权级为3
9     SelectorCodeRing3 equ LABEL_DESC_CODE_RING3-LABEL_GDT+SA_RPL3
10    SelectorStack3 equ LABEL_DESC_STACK3-LABEL_GDT+SA_RPL3
11    ; ...
12    ; 定义堆栈段ring3
13    [SECTION .s3]
14    ALIGN 32
15    [BITS 32]
16    LABEL_STACK3:
17        ; 该堆栈段有512个字节大小
18        times 512 db 0
19    TopOfStack3 equ $-LABEL_STACK3-1
20    ; ...
21    ; 定义代码段ring3
22    [SECTION .ring3]
23    ALIGN 32
24    [BITS 32]
25    LABEL_CODE_RING3:
26        mov ax, SelectorVideo
27        mov gs, ax
28        mov edi, (80*14 + 0) * 2
29        mov ah, 0Ch
30        mov al, '3'
31        mov [gs:edi], ax
32        jmp $
33    SegCodeRing3Len equ $-LABEL_CODE_RING3
34    ; ...
35    [SECTION .s32]
36    [BITS 32]
37    LABEL_SEG_CODE32:
38        ; ...
39        ; 压入ss
40        push SelectorStack3
41        ; 压入esp
42        push TopOfStack3
43        ; 压入cs
44        push SelectorCodeRing3
45        ; 压入eip
46        push 0
47        ; 执行ret指令
48        retf
```

1.2.2 低特权级到高特权级

从低特权级到高特权级转移的时候，需要用到 TSS，所以要添加 TSS 段。需要根据 TSS 的结构定义 TSS，TSS 的结构如下图:




```
1   [SECTION .gdt]
2   LABEL_DESC_TSS: Descriptor 0, TSSLen-1, DA_386TSS
3   ; ...
4   SelectorTSS equ LABEL_DESC_TSS-LABEL_GDT
5   ; ...
6   [SECTION .tss]
7   ALIGN
8   [BITS 32]
9   LABEL_TSS:
10      DD 0 ; 前一任务链接
11      DD TopOfStack    ; 0级堆栈段基址
12      DD SelectorStack ; 0级堆栈选择符
13      DD 0 ; 1级堆栈段基址
14      DD 0 ; 1级堆栈选择符
15      DD 0 ; 2级堆栈段基址
16      DD 0 ; 2级堆栈选择符
17      DD 0 ; CR3
18      DD 0 ; EIP
```

```
        DD 0 ; EFLAGS
        DD 0 ; EIP
        DD 0 ; EAX
        DD 0 ; ECX
        DD 0 ; EDX
        DD 0 ; EBX
        DD 0 ; ESP
        DD 0 ; EBP
        DD 0 ; ESI
        DD 0 ; EDI
        DD 0 ; ES
        DD 0 ; CS
        DD 0 ; SS
        DD 0 ; DS
        DD 0 ; FS
        DD 0 ; GS
        DD 0 ; LDT段选择符
        DW 0 ; 调试陷阱T标志位
        DW $-LABEL_TSS+2 ; I/O位图基址
        DB 0ffh ; I/O位图结束标志
    TSSLen equ $-LABEL_TSS
```

接着添加调用门，用于不同特权级的转移。调用门的添加步骤在第四次中已经讲到了，在这里就不再详细论述。添加调用门成功后，就可以在 ring3 代码段中通过调用门转移到特权级为 0 的代码段中。

```
    [SECTION .gdt]
    ; ...
    ; 定义一个特权级为0的代码段
    LABEL_DESC_CODE_TEST: Descriptor 0, SegCodeDestLen-1, DA_C+DA_32
    ; 定义一个能够跳转到0特权级代码段的门描述符
    LABEL_CALL_GATE_TEST: Gate SelectorCodeDest, 0, 0, DA_386Gate+DA_DPL3
    ; ...
    SelectorCallGateTest equ LABEL_CALL_GATE_TEST - LABEL_GDT

    ; 在32位代码段中加载TSS描述符
    ; 需要在特权级变换之前加载TSS描述符
    [SECTION .s32]
    ; ...
    mov ax, SelectorTSS
    ltr ax
    ; ...

    ; 在ring3代码段中通过调用门跳转到特权级为0的代码段中
    [SECTION .ring3]
    ALIGN 32
    [BITS 32]
    LABEL_CODE_RING3:
        mov ax, SelectorVideo
        mov gs, ax
        mov edi, (80 * 14 + 0) * 2
        mov ah, 0Ch
        mov al, '3'
        mov [gs:edi], ax
        call SelectorCallGateTest:0
```

```
30      jmp $
31  SegCodeRing3Len equ $-LABEL_CODE_RING3
```

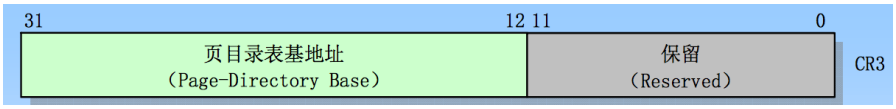
2 分页机制

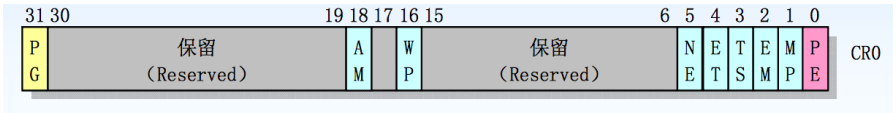
分页机制的理论已经在第二次报告中提到了，这里就直接写代码实现吧。

2.1 分页机制的实现

2.1.1 启动分页机制

代码中使用两极页表机制，第一级为页目录，大小为 4KB，有 1024 个表项，每个表项对应一个第二级页表。第二级页表也有 1024 个表项，每个表项对应一个物理页。首先我会先初始化页目录，然后初始化每一个页表。之后，将页目录基址存放 cr3 中。然后将 cr0 的 PG 位置一，表示开启分页机制。我在第一次报告中提到过 cr0 和 cr3，以下是它们的结构图。





在代码中添加 SetupPaging 函数，用于启动分页机制。代码如下：

```
1   ; 页目录为4KB
2   PageDirBase equ 200000h
3   ; 页表开始地址
4   PageTblBase equ 201000h
5   ; ...
6   ; 在GDT表中添加页目录和页表的描述符
7   [SECTION .gdt]
8   LABEL_DESC_PAGE_DIR: Descriptor PageDirBase, 4095, DA_DRW
9   LABEL_DESC_PAGE_TBL: Descriptor PageTblBase, 1023, DA_DRW | DA_LIMIT_4K
10  ; ...
11  SelectorPageDir equ LABEL_DESC_PAGE_DIR - LABEL_GDT
12  SelectorPageTbl equ LABEL_DESC_PAGE_TBL - LABEL_GDT
13  ; ...
14  ; 在32位代码段中添加SetupPaging函数，用于启动分页机制
15  [SECTION .s32]
16  [BITS 32]
17  LABEL_SEG_CODE32:
```

```
18          call SetupPaging
19      SetupPaging:
20          mov ax, SelectorPageDir
21          mov es, ax
22          mov ecx, 1024
23          xor edi, edi
24          xor eax, eax
25          mov eax, PageTblBase | PG_P | PG_USU | PG_RWW
26      .1:
27          ; 初始化页目录
28          ; 将页表的基地址存入页目录中
29          ; stosd的功能是将eax的内容移入到es所指的地址中
30          stosd ; 每次edi会自动加一
31          add eax, 4096 ; 一个页表的大小为4KB
32          loop .1 ; 当计数寄存器为0时，结束循环
33
34          ; 初始化所有页表
35          mov ax, SelectorPageTbl
36          mov es, ax
37          mov ecx, 1024 * 1024
38          xor edi, edi
39          xor eax, eax
40          mov eax, PG_P | PG_USU | PG_RWW
41      .2:
42          stosd
43          add eax, 4096 ; 每个表项指向一个4K的帧
44          loop .2
45          ; 将页目录基地址存入cr3中
46          mov eax, PageDirBase
47          mov cr3, eax
48          ; 将cr0的最高位PG标志置一
49          mov eax, cr0
50          or eax, 80000000h
51          mov cr0, eax
52          jmp short .3
53      .3:
54          nop
55          ret
```

2.1.2 利用分页机制节约内存

在上一节的代码中，虽然实现了代码，但是也暴露了两个问题：

- 分页机制确实是实现了，但是它没有带来实质性的好处。
- 页表占用的内存太大了。

在代码实现中，我用了 4MB 的空间用于存放页表，这些页表可以映射 4GB 的内存空间。假设现在的内存空间只有 16MB 大小，那么页表数根本不需要那么多，只需要 4 个就够了。所以，操作系统有必要知道内存的容量，从而进行内存管理。

可以通过执行指令 int 15h，来获得机器内存空间的大小。首先介绍一下 int 15h 指令。

15h 是中断向量号，进行软中断后，系统会根据 eax 寄存器的值执行相应的系统调用。下面相应的功能表，来源来自维基百科。

	AH	AL	Description
15h	00h		Turn on cassette drive motor
	01h		Turn off cassette drive motor
	02h		Read data blocks from cassette
	03h		Write data blocks to cassette
	4Fh		Keyboard Intercept
	83h		Event Wait
	84h		Read Joystick
	85h		Sysreq Key Callout
	86h		Wait
	87h		Move Block
	88h		Get Extended Memory Size
	89h		Switch to Protected Mode
	C0h		Get System Parameters
	C1h		Get Extended BIOS Data Area Segment
	C2h		Pointing Device Functions
	C3h		Watchdog Timer Functions - PS/2 systems only
	C4h		Programmable Option Select - MCA bus PS/2 systems only
	D8h		EISA System Functions - EISA bus systems only
	E8h	01h	Get Extended Memory Size (Newer function, since 1994). Gives results for memory size above 64 Mb.
	E8h	20h	Query System Address Map. The information returned from E820 supersedes what is returned from the older AX=E801h and AH=88h interfaces.

该中断处理函数需要五个输入参数，如下：

- eax。根据上述的功能表可知，当 eax = 0E820h 时，中断函数将返回机器内存大小。
- ebx。ebx 放置着”continuation value”，用于寻找下一个地址范围描述符结构 ARDS。首次调用 int 15h 时，将 ebx 置为 0。
- es:di。这个指向一个地址范围描述符结构 ARDS。
- ecx。用于表示 ARDS 的大小，以字节为单位。
- edx。签名‘SMAP’，需要将 edx 设为 0534D4150h，BIOS 使用该签名对调用者将要请求的系统映像信息进行校验。

函数也有五个输出值，如下：

- CF。CF=0，表示没有发生错误。
- eax。存放着签名’SMAP’，0534D4150h。
- es:di。和输入值相同，只想一个地址范围描述符结构 ARDS。