目 录

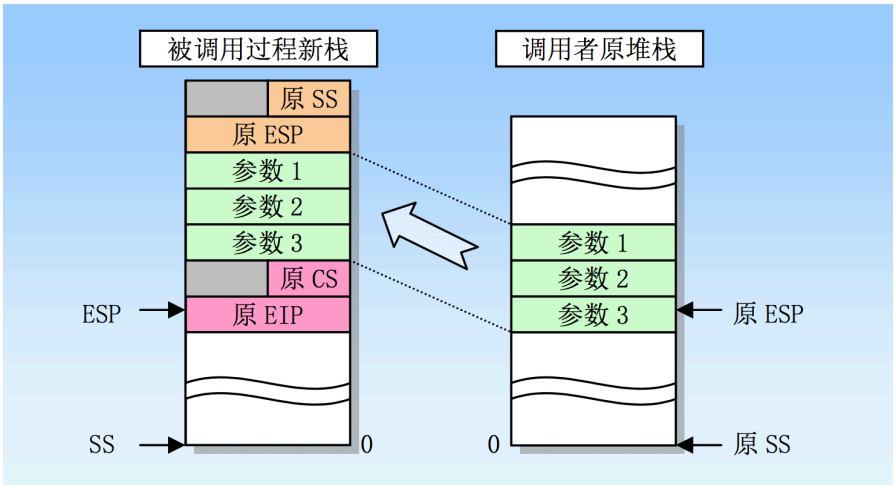
1 实现特权级转移

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：

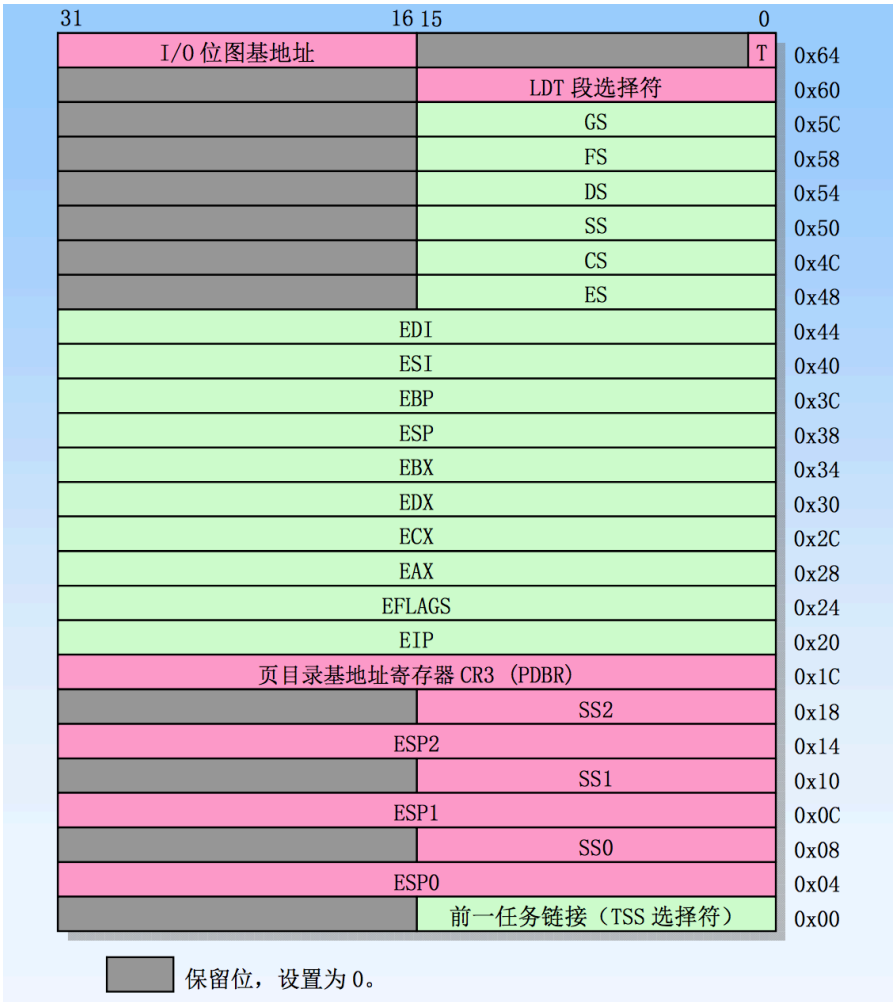
首先添加一个特权级为 3 的代码段，为了实现代码段转移，我们需要添加一个代码段和相应的堆栈段。先是在 GDT 表中添加该代码段和堆栈段的描述符，然后定义堆栈段 ring3 和代码段 ring3。随后，我们在 32 位代码段中通过执行 retf 指令跳转到 ring3 代码段中。

```
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:




```
[SECTION .gdt]
LABEL_DESC_TSS: Descriptor 0, TSSLen-1, DA_386TSS
; ...
SelectorTSS equ LABEL_DESC_TSS-LABEL_GDT
; ...
[SECTION .tss]
ALIGN
[BITS 32]
LABEL_TSS:
    DD 0 ; 前一任务链接
    DD TopOfStack    ; 0级堆栈段基址
    DD SelectorStack ; 0级堆栈选择符
    DD 0 ; 1级堆栈段基址
    DD 0 ; 1级堆栈选择符
    DD 0 ; 2级堆栈段基址
    DD 0 ; 2级堆栈选择符
    DD 0 ; CR3
    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
```

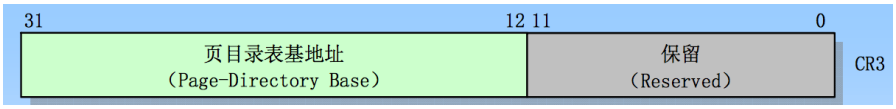
2 分页机制

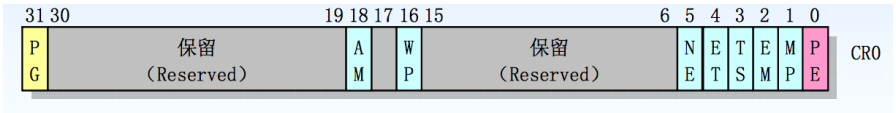
分页机制的理论已经在第二次报告中提到了，这里就直接写代码实现吧。

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。

15h

AH	AL	Description
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，指向一个地址范围描述符结构 ARDS。

- ecx。BIOS 会对 ARDS 进行信息填写，ecx 中存放一个数值，这个数值代表了 BIOS 填写了 ARDS 多少字节。
- ebx。ebx 放置着下一个地址描述符所需要的后续值。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。

下图是地址范围描述符 ARDS 的数据结构：

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（字节）的低 32 位
12	LengthHigh	长度（字节）的高 32 位
16	Type	这个地址范围的地址类型

其中，Type 的取值有三种情况：

- 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。
- 其他数，表示这个内存段被保留，留作以后使用，不可以被 OS 使用。

只能在实模式下使用 int 15h，所以在 16 位代码段中添加相应地代码。将得到的内存空间信息写入缓冲区 _MemChkBuf 中。下面是实现代码：

```
1   _MemChkBuf: times 256 db 0
2   _dwMCRNumber: dd 0
3   ; ...
4   [SECTION .s16]
5   [BITS 16]
6       mov ebx, 0 ; 将ebx的值置为一
7       mov di, _MemChkBuf ; 将内存信息写入缓冲区
8   .loop:
9       mov eax, 0E820h
10      mov ecx, 20 ; 写入缓冲区的字节数
11      mov edx, 0534D4150h
12      int 15h
13      jc LABEL_MEM_CHK_FAIL ; 检查CF标志
14      add di, 20 ; 指向下一个内存信息写入地址
15      inc dword [_dwMCRNumber]
16      cmp ebx, 0 ; 检查是否为最后一个地址范围描述符
17      jne .loop
18      jmp LABEL_MEM_CHK_OK
```

现在内存信息都保存在缓冲区 _MemChkBuf 中，需要有相应的代码计算其中的内存大小。为了实现这一功能，我定义了一个数据段。并且在 32 位代码中添加了 CalMemSize 函数用于计算内存大小。下面是实现代码：

```
    ; 定义数据段
    [SECTION .data1]
    ALIGN 32
    [BITS 32]
    LABEL_DATA:
    _szRAMSize db "RAM size:", 0
    _dwMCRNumber: dd 0
    _dwMemSize: dd 0
    ; 定义一个ARDS数据结构，大小为20字节
    _ARDSStruct:
        _dwBaseAddrLow: dd 0
        _dwBaseAddrHigh: dd 0
        _dwLengthLow: dd 0
        _dwLengthHigh: dd 0
        _dwType: dd 0

    szRAMSize equ _szRAMSize - $$
    dwMemSize equ _dwMemSize - $$
    dwMCRNumber equ _dwMCRNumber - $$
    ARDSStruct equ _ARDSStruct - $$
        dwBaseAddrLow equ _dwBaseAddrLow - $$
        dwBaseAddrHigh equ _dwBaseAddrHigh - $$
        dwLengthLow equ _dwLengthLow - $$
        dwLengthHigh equ _dwBaseAddrHigh - $$
        dwType equ _dwType - $$

    ; 在32位代码中添加CalMemSize函数
    [SECTION .s32]
    [BITS 32]
    DispMemSize:
        mov ax, SelectorData
        mov ds, ax
        mov es, ax
        mov esi, MemChkBuf
        ; mov [ds:dwMCRNumber] to ecx
        mov ecx, [dwMCRNumber]
    .loop:

        ; 开始一个循环，将缓冲区中20字节读入ARDS结构中
        mov edx, 5
        ; 将es:edi指向_ARDSStruct
        mov edi, ARDSStruct
    .1:
        mov eax, esi
        ; 将eax移入[es:edi]，并将edi加4
        stosd
        add esi, 4
        dec edx
        cmp edx, 0
        jnz .1

```

```
52          cmp dword [dwType], 1 ; 查看内存段的类型
53          jne .2 ; 如果内存段可用，则跳转到.2
54          mov eax, [dwBaseAddrLow]
55          add eax, [dwLengthLow]
56          cmp eax, [dwMemSize]
57          jb .2
58          ; mov eax to [es:dwMemSize]
59          mov [dwMemSize], eax
60      .2:
61          loop .loop
```

这里就能得到内存大小了，存储在 dwMemSize 中。随后根据内存大小计算应该初始化多少页目录项和页表项。下面是实现的代码：

```
1       SetupPaging:
2           xor edx, edx
3           ; eax存放着内存空间总大小
4           mov eax, [dwMemSize]
5           ; ebx存放着一个页表对应的大小
6           mov ebx, 40000h
7           ; div指令，eax除以ebx，商存储在eax中，余数存储在edx中
8           div ebx
9           ; ecx中存放着商
10          mov ecx, eax
11          ; 检查余数是否为0
12          test edx, edx
13          jz .no_remainder
14          ; 如果余数不为零，需要初始化的页表数加一
15          inc ecx
16      .no_remainder:
17          push ecx
18          ; 虽然根据ecx中值的大小初始化页目录和页表
```

2.2 感受分页机制

由于分页机制的存在，程序使用的都是线性地址空间，而不再是物理地址。这样操作系统就为应用程序提供了一个不依赖于物理内存的平台，应用程序也不必关心实际上有多少物理内存，也不必关心正在使用的是哪一段内存，甚至不必关心某一个地址是在物理内存里面还是在硬盘中。

首先说一下我想实现的功能。我定义了两个函数 ProcPagingDemo 和 LinearAddrDemo，ProPagingDemo 函数实现了向 LinearAddrDemo 这个线性地址的转移。一开始，我让 LinearAddrDemo 映射到物理地址空间中的 ProcFoo 处。虽然我切换页目录表和页表，让 LinearAddrDemo 映射到 ProcBar 这个物理地址上。

程序一开始的时候，分页机制是默认物理地址等于线性地址的。这样我让 LinearAddrDemo 和 ProcFoo 的值相等，这样调用 LinearAddrDemo 时，就相当于调用 ProcFoo 处的代码。虽然分页机制将 LinearAddrDemo 映射到 ProcBar，这样调用 LinearAddrDemo 时，就相当于调用 ProcBar 处的代码。

```
LinearAddrDemo equ 00401000h
ProcFoo equ 00401000h
ProcBar equ 00501000h
ProcPagingDemo equ 00301000h
```

为了将代码放置在 ProcFoo 和 ProcBar 这两个地方，还需要写两个函数，在程序运行时将这两个函数的执行代码复制过去。所以需要写一个复制函数 MemCpy。实现代码如下：

```
; void* MemCpy(void* es:pDest, void* ds:pSrc, int iSize);
MemCpy:
    push ebp
    mov ebp, esp

    push esi
    push edi
    push ecx

    mov edi, [ebp + 8] ; pDest
    mov esi, [ebp + 12] ; pSrc
    mov ecx, [ebp + 16] ; iSize
.1:
    cmp ecx, 0
    jz .2

    mov al, [ds:esi]
    inc esi

    mov byte [es:edi], al
    inc edi

    dec ecx
    jmp .1
.2:
    mov eax, [ebp + 8]

    pop ecx
    pop edi
    pop esi
    mov esp, ebp
    pop ebp

    ret
```

这个代码是逐字节复制的，比较简单，就不详细说明了。需要注意的是，这个函数假设源数据存放在 ds 段中，目的地址在 es 段中，所以在调用这个函数之前，需要给 ds 和 es 赋值。而且这个函数需要三个参数，根据实现代码，需要将长度、源代码地址和目的地址压入栈中。需要注意的是，因为在调用函数之前压入了 3 个参数，一个参数的大小是 4 字节。为了在调用函数结束后跳过这三个参数，需要将 esp 指针的值加 12。

下面是相应的初始化代码：

```
PagingDemo:
    mov ax, cs
    mov ds, ax
    mov ax, SelectorFlatRW
    mov es, ax

    push LenFoo
    push OffsetFoo
    push ProcFoo
    call MemCpy
    add esp, 12

    push LenBar
    push OffsetBar
    push ProcBar
    call MemCpy
    add esp, 12

    push LenPagingDemoAll
    push OffsetPagingDemoProc
    push ProcPagingDemo
    call MemCpy
    add esp, 12

PagingDemoProc:
OffsetPagingDemoProc equ PagingDemoProc - $$
    mov eax, LinearAddrDemo
    call eax
    retf
LenPagingDemoAll equ $ - PagingDemoProc

foo:
OffsetFoo equ foo - $$
    mov ah, 0Ch
    mov al, 'F'
    mov [gs:((80 * 17 + 0) * 2)], ax
    mov al, 'o'
    mov [gs:((80 * 17 + 1) * 2)], ax
    mov [gs:((80 * 17 + 2) * 2)], ax
    ret
LenFoo equ $ - foo

bar:
OffsetBar equ bar - $$
    mov ah, 0Ch
    mov al, 'B'
    mov [gs:((80 * 18 + 0) * 2)], ax
    mov al, 'o'
    mov [gs:((80 * 18 + 1) * 2)], ax
    mov [gs:((80 * 18 + 2) * 2)], ax
    ret
LenBar equ $ - bar
```

所有初始化工作做好之后，就可以在分页机制下，通过调用 ProcPagingDemo 从而调用 LinearAddrDemo 线性地址处的代码。注意，LinearAddrDemo 的值是 00401000h。而分

页机制中直接让线性地址等于物理地址。所以页表中偏移 00401000h 的位置指向的就是 00401000h，也就是 SelectorFlatRW:ProcFoo 函数的地址。这样相当于调用 ProcFoo 处的代码。调用代码如下：

```
    ; 通过分段机制得到相应的线性地址LinearAddrDemo
    ; LinearAddrDemo为00401000h
    ; 高10位为0000000001b，这样对应页目录中的第二个页表
    ; 中间10位为000000000001b，这样对应页表中的第二个页
    ; 一个页的大小是4k，一个页表就能指向4M的地址
    ; 第二个页表的第二个页就是4M+4k，所以物理地址是00401000h
    ; 所以LinearAddrDemo线性地址对应的代码是ProcFoo
    call SelectorFlatC:ProcPagingDemo
```

之后变换页表和页目录，让页表中偏移 00401000h 的位置指向的是 00501000h，也就是 ProcBar 函数的地址。下面是切换页目录和页表的实现代码：

```
PSwitch:
    mov ax, SelectorFlatRW
    mov es, ax
    mov edi, PageDirBase1
    xor eax, eax
    mov eax, PageTblBase1 | PG_P | PG_USU | PG_RWW
    mov ecx, [PageTableNumber]
.1:
    stosd
    add eax, 4096
    loop .1

    mov eax, [PageTableNumber]
    mov ebx, 1024
    mul ebx
    mov ecx, ecx
    mov edi, PageTblBase1
    xor eax, eax
    mov eax, PG_P | PG_USU | PG_RWW
.2:
    stosd
    add eax, 4096
    loop .2

    ; 改变了LinearAddrDemo对应的物理地址
    ; 处理LinearAddrDemo的高10位，也就是页目录的偏移量
    mov eax, LinearAddrDemo
    shr eax, 22
    mov ebx, 4096
    mul ebx
    ; ecx此时存放着相应页表的偏移字节
    mov ecx, eax

    ; 处理LinearAddrDemo的中间10位，也就是页表的偏移量
    mov eax, LinearAddrDemo
    shr eax, 12
    and eax, 03FFh
    mov ebx, 4
```

```
39          mul ebx ; eax此时存放着页表的偏移字节
40          ; 加上ecx后，eax此时存放着相应页的偏移字节
41          add eax, ecx
42          ; 加上PageTblBase1，得到LinearAddrDemo所指的项
43          add eax, PageTblBase1
44          ; 将该项的内容改为ProcBar的地址
45          mov dword [es:eax], ProcBar | PG_P | PG_USC | PG_RWW
46
47          ; 改变cr3的值来切换页目录
48          mov eax, PageDirBase1
49          mov cr3, eax
50          jmp short .3
51      .3:
52          nop
53          ret
```

改变后，线性地址 LinearAddrDemo 对应的物理地址是 ProcBar。再次调用 Selector-FlatC:ProcPagingDemo，就会调用 ProcBar 函数。

3 学习 Java

因为最近参加了挑战杯，所以需要使用 android 编程。之前我使用的都是 C\C++，所以对 Java 不是很了解。以后在学习报告中，除了操作系统的内容，还会添加 Java 学习的内容。

3.1 第一个 Java 程序

3.1.1 名字管理

Java 为了给一个类库生成不会与其他名字混淆的名字，让程序员反过来使用自己的 Internet 域名，从而保证它们是独一无二的。比如域名为 MindView.net，那么各种应用工具库就被命名为 net.mindview.utility.foibles。反转域名后，句点就用来代表子目录的划分。

3.1.2 static 关键字

现在有两个需求，如下所示：

- 只想为某个特定域分配单一存储空间，而不去考虑究竟要创建多少对象，甚至根本就不创建任何对象。
- 希望某个函数不与包含它的类的任何对象关联在一起。也就是说，即使没有创建对象，也可以调用这个函数。

static 关键字可以满足这两方面的需求。当声明一个事物是 static 的时候，就意味着这个域或这个函数不会与包含它的那个类的任何对象实例关联在一起。

引用 static 变量有两种方法。可以通过一个对象去定位它，也可以通过其类名直接引用。

static 函数的重要用法是在不创建任何对象的前提下就可以调用它。

3.1.3 Hello world

下面是第一个完整的程序，可以打印出“hello world”：

```
1  import java.util.*;
2
3  public class HelloWorld
4  {
5      public static void main(String[] args)
6      {
7          System.out.println("hello world");
8      }
9  }
```

3.1.4 编码风格

代码风格的规定如下：

- 类名的首字母要大些。如果类名由几个单词构成，那么就把它们并在一起，其中每个内部单词的首字母都采用大写形式。
- 其他内容，第一个字母采用小写。如果该内容由几个单词构成，那么就把它们并在一起，其中每个内部单词的首字母都采用大写形式。