

目 录

1	分页机制	2
1.1	分页机制的实现	2
1.1.1	启动分页机制	2
1.1.2	利用分页机制节约内存	3
1.2	感受分页机制	7

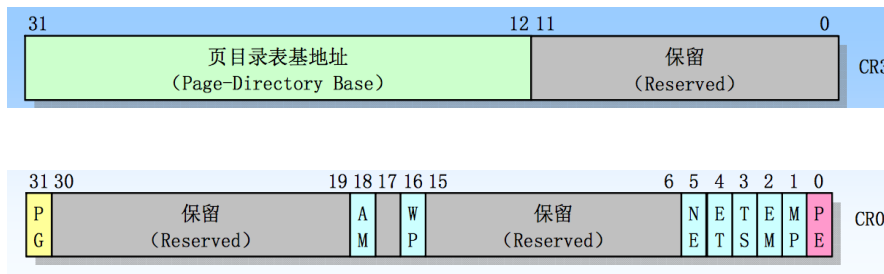
1 分页机制

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。

1.1 分页机制的实现

1.1.1 启动分页机制

代码中使用两级页表机制，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
27     .1:
28     ; 初始化页目录
29     ; 将页表的基地址存入页目录中
30     ; stosd的功能是将eax的内容移入到es所指的地址中
31     stosd ; 每次edi会自动加一
32     add eax, 4096 ; 一个页表的大小为4KB
33     loop .1 ; 当计数寄存器为0时, 结束循环
34
35     ; 初始化所有页表
36     mov ax, SelectorPageTbl
37     mov es, ax
38     mov ecx, 1024 * 1024
39     xor edi, edi
40     xor eax, eax
41     mov eax, PG_P | PG_USU | PG_RWW
42
43     .2:
44     stosd
45     add eax, 4096 ; 每个表项指向一个4K的帧
46     loop .2
47     ; 将页目录基地址存入cr3中
48     mov eax, PageDirBase
49     mov cr3, eax
50     ; 将cr0的最高位PG标志置一
51     mov eax, cr0
52     or eax, 80000000h
53     mov cr0, eax
54     jmp short .3
55
56     .3:
57     nop
58     ret

```

1.1.2 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 <code>AX=E801h</code> and <code>AH=88h</code> 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`。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：

```

1 ; 定义数据段
2 [SECTION .data1]
3 ALIGN 32
4 [BITS 32]
5 LABEL_DATA:
6 _szRAMSize db "RAM size:", 0
7 _dwMCRNumber: dd 0
8 _dwMemSize: dd 0
9 ; 定义一个ARDS数据结构, 大小为20字节
10 _ARDSStruct:
11     _dwBaseAddrLow: dd 0
12     _dwBaseAddrHigh: dd 0
13     _dwLengthLow: dd 0
14     _dwLengthHigh: dd 0
15     _dwType: dd 0
16
17 szRAMSize equ _szRAMSize - $$
18 dwMemSize equ _dwMemSize - $$
19 dwMCRNumber equ _dwMCRNumber - $$
20 ARDSStruct equ _ARDSStruct - $$
21     dwBaseAddrLow equ _dwBaseAddrLow - $$
22     dwBaseAddrHigh equ _dwBaseAddrHigh - $$
23     dwLengthLow equ _dwLengthLow - $$
24     dwLengthHigh equ _dwBaseAddrHigh - $$
25     dwType equ _dwType - $$
26
27 ; 在32位代码中添加 CalMemSize 函数
28 [SECTION .s32]
29 [BITS 32]
30 DispMemSize:
31     mov ax, SelectorData
32     mov ds, ax
33     mov es, ax
34     mov esi, MemChkBuf
35     ; mov [ds:dwMCRNumber] to ecx
36     mov ecx, [dwMCRNumber]
37 .loop:
38
39     ; 开始一个循环, 将缓冲区中20字节读入ARDS结构中
40     mov edx, 5
41     ; 将es:edi指向_ARDSStruct
42     mov edi, ARDSStruct
43 .l1:
44     mov eax, esi
45     ; 将eax移入[es:edi], 并将edi加4
46     stosd
47     add esi, 4
48     dec edx
49     cmp edx, 0
50     jnz .l1
51
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

```

1.2 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，ProcPagingDemo 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。

```

1  LinearAddrDemo equ 00401000h
2  ProcFoo equ 00401000h
3  ProcBar equ 00501000h

```

```
4 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：

```
1 ; void* MemCpy(void* es:pDest, void* ds:pSrc, int iSize);
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] ; pDest
11    mov esi, [ebp + 12] ; pSrc
12    mov ecx, [ebp + 16] ; iSize
13
14    .1:
15        cmp ecx, 0
16        jz .2
17
18        mov al, [ds:esi]
19        inc esi
20
21        mov byte [es:edi], al
22        inc edi
23
24        dec ecx
25        jmp .1
26
27    .2:
28        mov eax, [ebp + 8]
29
30        pop ecx
31        pop edi
32        pop esi
33        mov esp, ebp
34        pop ebp
35
36        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：

```
1 PagingDemo:
2     mov ax, cs
3     mov ds, ax
4     mov ax, SelectorFlatRW
```



```

5      mov es, ax
6
7      push LenFoo
8      push OffsetFoo
9      push ProcFoo
10     call MemCpy
11     add esp, 12
12
13     push LenBar
14     push OffsetBar
15     push ProcBar
16     call MemCpy
17     add esp, 12
18
19     push LenPagingDemoAll
20     push OffsetPagingDemoProc
21     push ProcPagingDemo
22     call MemCpy
23     add esp, 12
24
25     PagingDemoProc:
26     OffsetPagingDemoProc equ PagingDemoProc - $$
27     mov eax, LinearAddrDemo
28     call eax
29     retf
30     LenPagingDemoAll equ $ - PagingDemoProc
31
32     foo:
33     OffsetFoo equ foo - $$
34     mov ah, 0Ch
35     mov al, 'F'
36     mov [gs:((80 * 17 + 0) * 2)], ax
37     mov al, 'o'
38     mov [gs:((80 * 17 + 1) * 2)], ax
39     mov [gs:((80 * 17 + 2) * 2)], ax
40     ret
41     LenFoo equ $ - foo
42
43     bar:
44     OffsetBar equ bar - $$
45     mov ah, 0Ch
46     mov al, 'B'
47     mov [gs:((80 * 18 + 0) * 2)], ax
48     mov al, 'o'
49     mov [gs:((80 * 18 + 1) * 2)], ax
50     mov [gs:((80 * 18 + 2) * 2)], ax
51     ret
52     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：

```

1 ; 通过分段机制得到相应的线性地址 LinearAddrDemo
2 ; LinearAddrDemo 为 00401000h
3 ; 高10位为 000000001b, 这样对应页目录中的第二个页表
4 ; 中间10位为 00000000001b, 这样对应页表中的第二个页
5 ; 一个页的大小是 4k, 一个页表就能指向 4M 的地址
6 ; 第二个页表的第二个页就是 4M+4k, 所以物理地址是 00401000h
7 ; 所以 LinearAddrDemo 线性地址对应的代码是 ProcFoo
8 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:

```

1 PSwitch:
2     mov ax, SelectorFlatRW
3     mov es, ax
4     mov edi, PageDirBase1
5     xor eax, eax
6     mov eax, PageTblBase1 | PG_P | PG_USU | PG_RWW
7     mov ecx, [PageTableNumber]
8
9     .1:
10    stosd
11    add eax, 4096
12    loop .1
13
14    mov eax, [PageTableNumber]
15    mov ebx, 1024
16    mul ebx
17    mov ecx, ecx
18    mov edi, PageTblBase1
19    xor eax, eax
20    mov eax, PG_P | PG_USU | PG_RWW
21
22    .2:
23    stosd
24    add eax, 4096
25    loop .2
26
27    ; 改变了 LinearAddrDemo 对应的物理地址
28    ; 处理 LinearAddrDemo 的高10位, 也就是页目录的偏移量
29    mov eax, LinearAddrDemo
30    shr eax, 22
31    mov ebx, 4096
32    mul ebx
33    ; ecx 此时存放着相应页表的偏移字节
34    mov ecx, eax
35
36    ; 处理 LinearAddrDemo 的中间10位, 也就是页表的偏移量
37    mov eax, LinearAddrDemo
38    shr eax, 12
39    and eax, 03FFh
40    mov ebx, 4
41    mul ebx ; eax 此时存放着页表的偏移字节
42    ; 加上 ecx 后, eax 此时存放着相应页的偏移字节
43    add eax, ecx
44    ; 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 SelectorFlatC:ProcPagingDemo，就会调用 ProcBar 函数。