# 内核：内存管理

## 驱动开发的细节

定义局部变量最好使用`ExAllocatePool`来分配内存

关于内存的类型：

1. 分页内存（虚拟内存）：内存不足置换到磁盘上
2. 非分页内存：内存不足也不会被置换到磁盘上

尽量使用分页内存（包括函数），可以使用

```
#pragma alloc_text("PAGE", xxx)
```

指明将xxx函数放入分页内存当中

## 内存管理

逻辑地址 --> 分段 --> 线性地址 --> 分页 --> 物理地址

大部分操作系统直接使用分页

### 分段

#### 段描述符

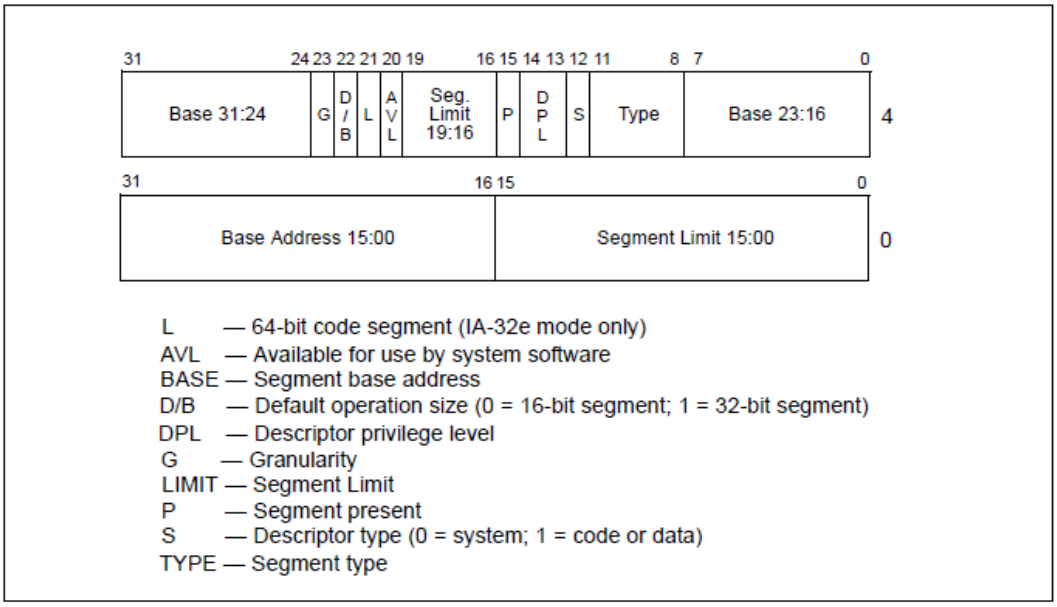


Figure 3-8. Segment Descriptor

一共8字节，分为两部分

- 字段说明：

```
G = 粒度位，为1是4K，为0是字节
base = 段基址，32位
limit = 段界限，20位，
    当 G = 0时，limit * 0 + 0xfff
    当 G = 1时，limit * 4K + 0xfff
D/B = 为0则是16位段，为1则是32位段
L = 64位段标志
AVL = 软件可利用位，由操作系统决定作用
P = 存在位，表示段描述符是否有效
S = 为0则是系统段，为1则是存储段（代码段或数据段）
type = 内存属性（最高位为1是代码段，否则是数据段）
DPL = 特权级（0 ~ 3）
```

- 数据结构

```
/*
 * 段描述符，共8字节
 *      base = 段基址，32位
 *      limit = 段界限，20位
 *          当 G = 0时，limit * 0 + 0xfff
 *          当 G = 1时，limit * 4K + 0xfff
 */
struct SegmentDescriptor {
    union {
        unsigned int low32;                     // 低32位字段
        struct {
            unsigned int limit_0_15 : 16;       // 第0-15位的段界限
            unsigned int base_0_15 : 16;        // 第0-15位的段基址
        };
    } SegDesLow;                                // 段描述符低32位字段

    union {
        unsigned int high32;                    // 高32位字段
        struct {
            unsigned int base_16_23 : 8;        // 第16-23位的段基址
            unsigned int type : 4;              // 内存属性（最高位为1是代码段，否
则是数据段）
            unsigned int s : 1;                 // 为0则是系统段，为1则是存储段
（代码段或数据段）
            unsigned int dpl : 2;               // 特权级（0 ~ 3）
            unsigned int p : 1;                 // 存在位，表示段描述符是否有效
            unsigned int limit_16_19 : 4;       // 第16-19位的段界限
            unsigned int avl : 1;               // 软件可利用位，由操作系统决定作
用
            unsigned int l : 1;                 // 64位段标志
            unsigned int d_b : 1;               // 为0则是16位段，为1则是32位段
            unsigned int g : 1;                 // 粒度位，为1单位是4K，为0是单位
是字节
            unsigned int base_24_31 : 8;        // 第24-31位的段基址
        };
    } SegDesHigh;                               // 段描述符高32位字段
};
```

## GDTR全局描述符表寄存器

一共48位，高32位表示地址（段描述符表地址），低16位表示界限（段描述符表大小）

在**windbg**中**显示**为两个，分别是：32位的`gdtr`和16位的`gdtl`

- 相关指令
  - `lgdt m16&32`
    - 将源操作数中的值加载到全局描述符表寄存器(GDTR) 中
    - 此条指令为特权指令
  - `sgdt m`
    - 将全局描述符表格寄存器 (GDTR) 中的内容存储到目标操作数

对于多核处理器来说，每一个核心都有一个全局描述符表（GDT），也即每个核心获取的GDTR不一样

- 在R3程序中，可使用`SetProcessAffinityMask`来指定程序在某些核心上运行

```
BOOL SetProcessAffinityMask(
  HANDLE    hProcess,                   // 进程句柄
  DWORD_PTR dwProcessAffinityMask       // 关联掩码，可取值为0~2^31(32位）和
0~2^63（64位），每一位代表一个核心
);
```

- 在R0程序中，可使用`KeSetSystemAffinityThread`来指定程序运行在哪些核心上

  `KeGetCurrentProcessorNumber`获取当前CPU的核心

  `KeQueryActiveProcessorCount`获取CPU核心数量

```
typedef ULONG_PTR  KAFFINITY;

void KeSetSystemAffinityThread(
  KAFFINITY Affinity    // 关联掩码，每一位代表一个核心
);
```

## LDTR局部描述符表寄存器

一共16位，高13位存放LDT在GDT中的索引值

- 相关指令
  - `lldt r/m16`
    - 将源操作数加载到局部描述符表寄存器 (LDTR) 的段选择子字段
  - `sldt r/m16/m32`
    - 将局部描述符表寄存器 (LDTR) 中的段选择器存储到目标操作数

## 段选择子

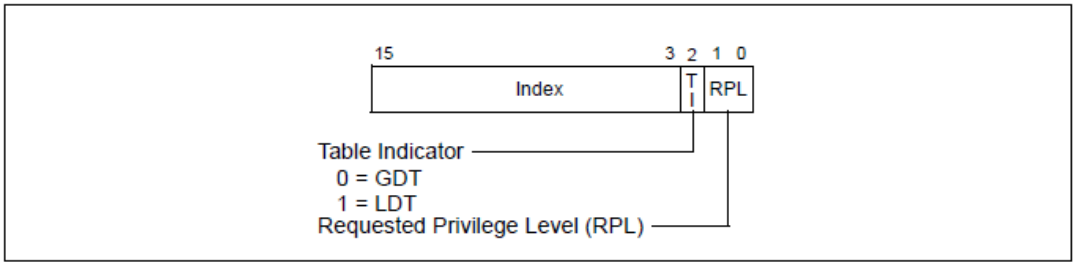


Figure 3-6. Segment Selector

将原先的段寄存器用于充当段选择子，共16位，其中：

- 13位的索引号Index
- 1位的T1标志
  - 当为0时，指向GDT
  - 当为1时，指向LDT
- 2位的特权级RPL（0 ~ 3）

# 分页

分页机制开启标志为：CR0寄存器的最高位`PG = 1`，分页大小以4k最为常见

以下均以4k分页为例

## 页目录表、页表

页目录表存储着页目录表项（PDE），在4k分页下，总共有1024项

页表存储着页表项（PTE），在4k分页下，总共有1024项

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 | 11 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Address of page directory[1] | Ignored | | | | | PCD | PWT | Ignored | | | CR3 |
| Bits 31:22 of address of 4MB page frame / Reserved (must be 0) / Bits 39:32 of address[2] / PAT | Ignored | G | 1 | D | A | PCD | PWT | U/S | R/W | 1 | PDE: 4MB page |
| Address of page table | Ignored | | 0 | Ign | A | PCD | PWT | U/S | R/W | 1 | PDE: page table |
| Ignored | | | | | | | | | | 0 | PDE: not present |
| Address of 4KB page frame | Ignored | G | PAT | D | A | PCD | PWT | U/S | R/W | 1 | PTE: 4KB page |
| Ignored | | | | | | | | | | 0 | PTE: not present |

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

不管是页目录还是页表，每个表项占4个字节，其表项结构基本相同，如上图

通过页目录表的目录项可以找到页表，页表中的表项保存物理基地址

注意：不管是页目录还是页表，高20位记录的地址均为物理地址，且此20位为记录的物理地址的高20位（即此使用此地址需要在末尾填12位的0）

**格式**

P = 存在位

RW = 读写位，为0则是可读可执行，为1则是可读可写可执行

US = 特权级，为0则是系统级页，为1则是用户级页

A = 标明是否访问过

D = 标明是否写入（磁盘和内存的交换情况）

AVL = 软件可利用位

## CR3寄存器

32位的寄存器，存储页目录地址（物理地址）

- 访问CR3寄存器
  - `mov eax, cr3`，属于特权指令
- 访问物理地址
  - `MmMapIoSpace`映射物理地址到虚拟地址

```
PVOID MmMapIoSpace(
  PHYSICAL_ADDRESS     PhysicalAddress,  // 起始物理地址
  SIZE_T               NumberOfBytes,    // 映射的字节数
  MEMORY_CACHING_TYPE CacheType          // 用于映射物理地址范围的缓存属性
);
```

## 线性地址转换为物理地址

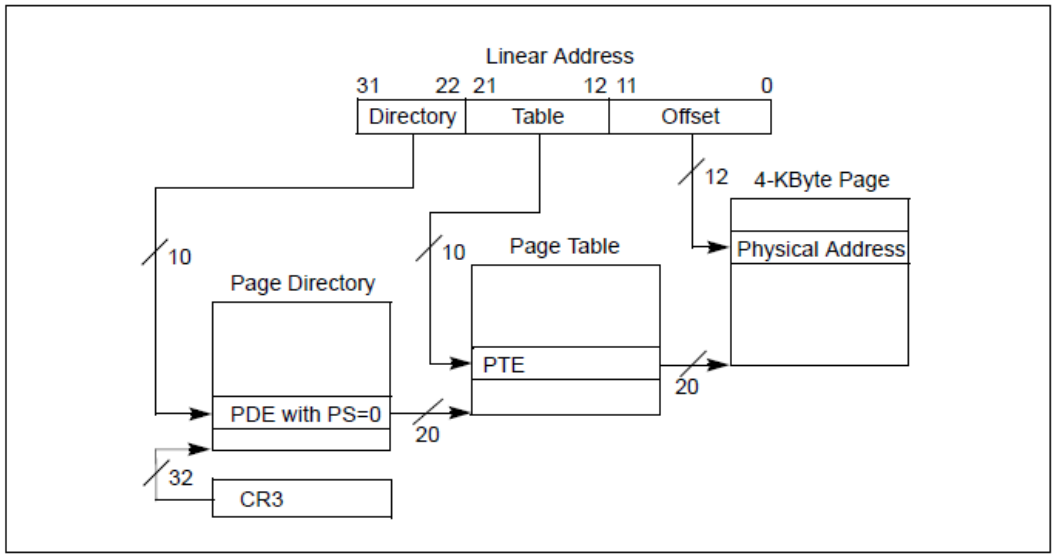


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

将32位的线性地址由低到高分为三部分：

1. 第0 ~ 11位：12位的偏移`Offset`
2. 第12 ~ 21位：10位的**页表索引**
3. 第22 ~ 31位：10位的**页目录表索引**

过程：

1. 通过**页目录表索引**在页目录中找到**页目录表项**，表项的**高20位**为页表的地址（物理地址，在地址末尾填12位的0）
2. 通过**页表索引**在页表中找到**页表项**，表项的**高20位**为映射的物理基址（在地址末尾填12位的0）
3. 最后`物理基址+Offset`就为物理地址

## TLB高速缓存

只要CR3寄存器修改了，CPU就会刷新TLB

```
mov eax, cr3
mov cr3, eax    ; 此时就会刷新TLB
```

## PAE

物理地址扩展，在32位上开启PAE，可将物理地址升到36位，在64位上开启，物理地址最多有52位（64位太大也没必要）

CR4寄存器中的第5位PAE Flag控制PAE是否开启

**32位的PAE**

在开启了PAE之后，地址转换如下图（每个表项均为8字节）：

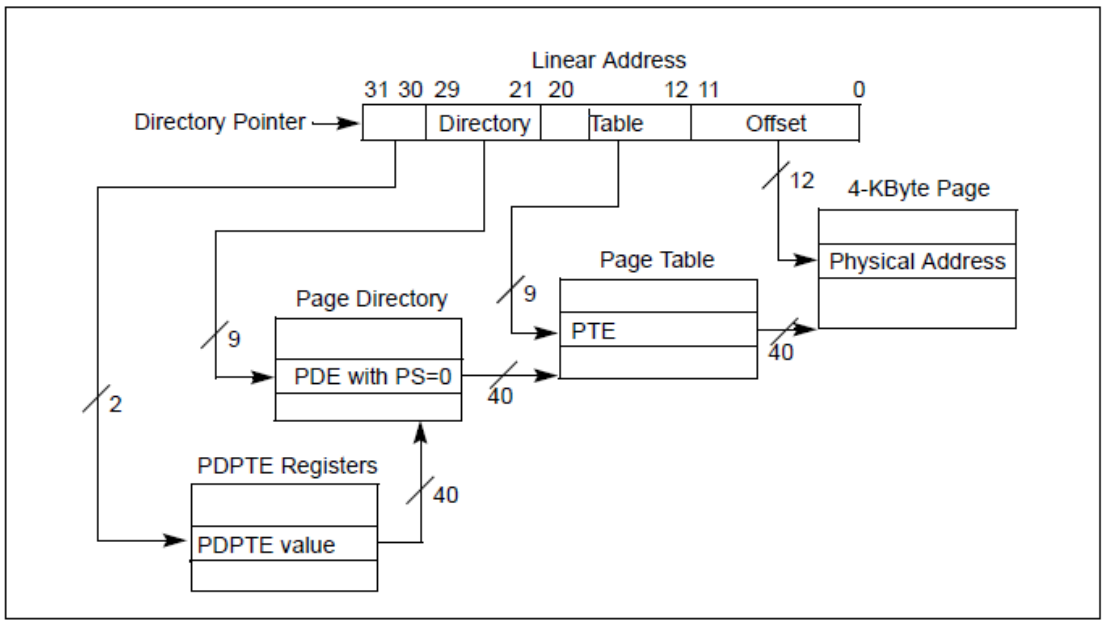


Figure 4-5. Linear-Address Translation to a 4-KByte Page using PAE Paging

其中：

- PDPTE寄存器依然是CR3寄存器

Table 4-7. Use of CR3 with PAE Paging

| Bit Position(s) | Contents |
|---|---|
| 4:0 | Ignored |
| 31:5 | Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation |
| 63:32 | Ignored (these bits exist only on processors supporting the Intel-64 architecture) |

- 线性地址的高30 - 31位表示了PDPTE最多有4项
- 高21 - 29位的PDE索引和12 - 20位的PTE索引分别表示了PDE和PTE最多有512项
- 最后0 - 11位依然是偏移
- 在各个表中的项目中，拿低12位后的高24位是基址

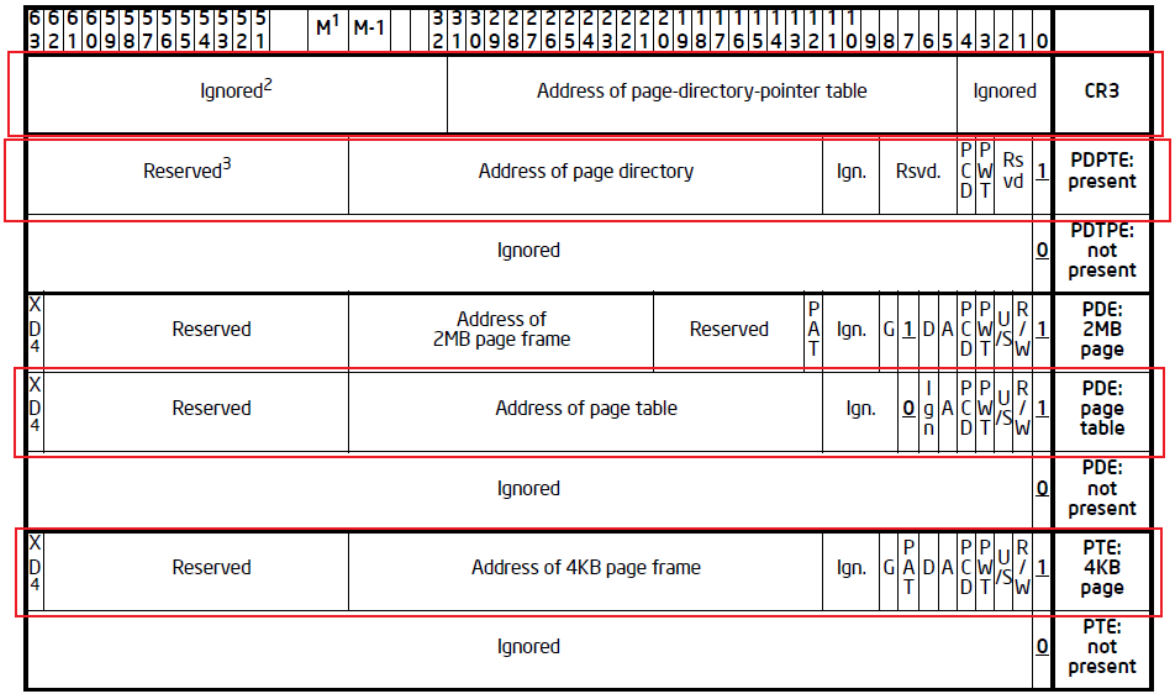

Figure 4-7. Formats of CR3 and Paging-Structure Entries with PAE Paging

**64位的PAE**

线性地址为48位的，CPU只用这48位，而在操作系统中，剩余的高16位是这低48位的符号扩展

例如：

- 在R3中范围为`0000 0000 0000 0000 ~ 0000 7000 0000 0000`
- 在R0中范围为`ffff 8000 0000 0000 ~ ffff ffff ffff ffff`

在开启了PAE之后，地址转换如下图（每个表项均为8字节）：

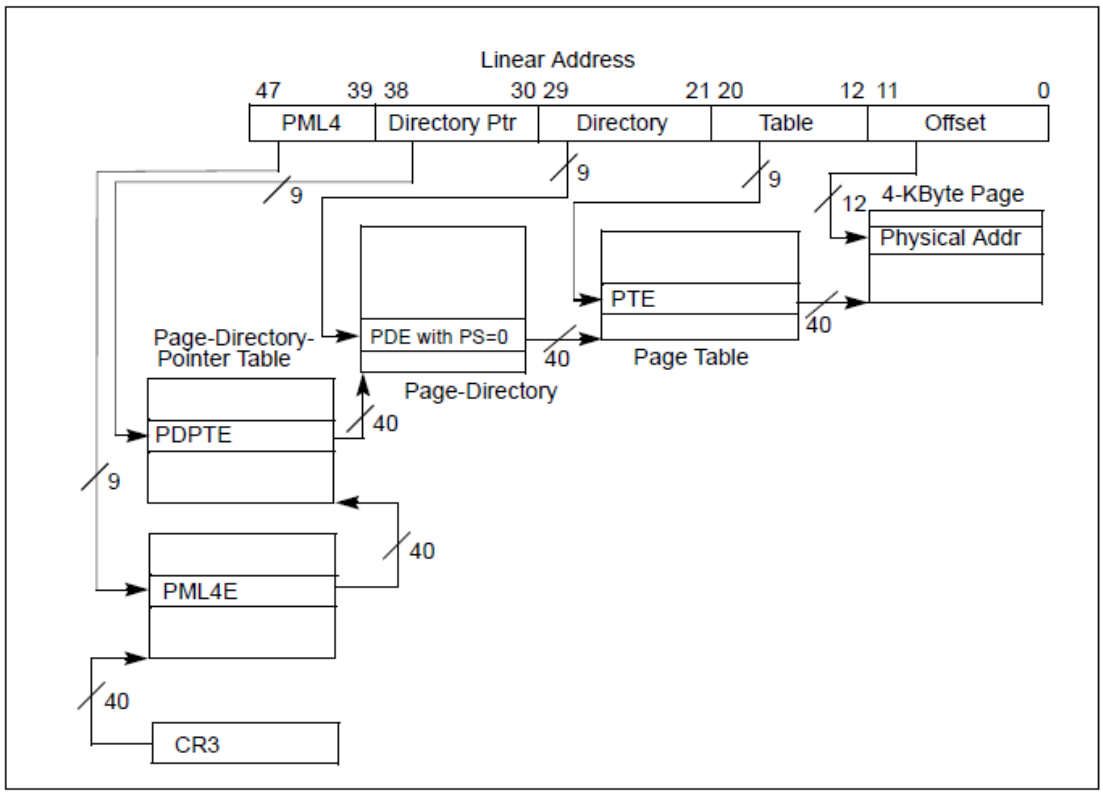


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

其中：

- CR3寄存器存储了PML4E表的地址（与32位的不同）
- PML4E表中的项目指向了PDPTE表
- 线性地址部分字段发送了改变，其余与32位结构大体类似
- 在各个表中的项目中，拿低12位后的高40位是基址

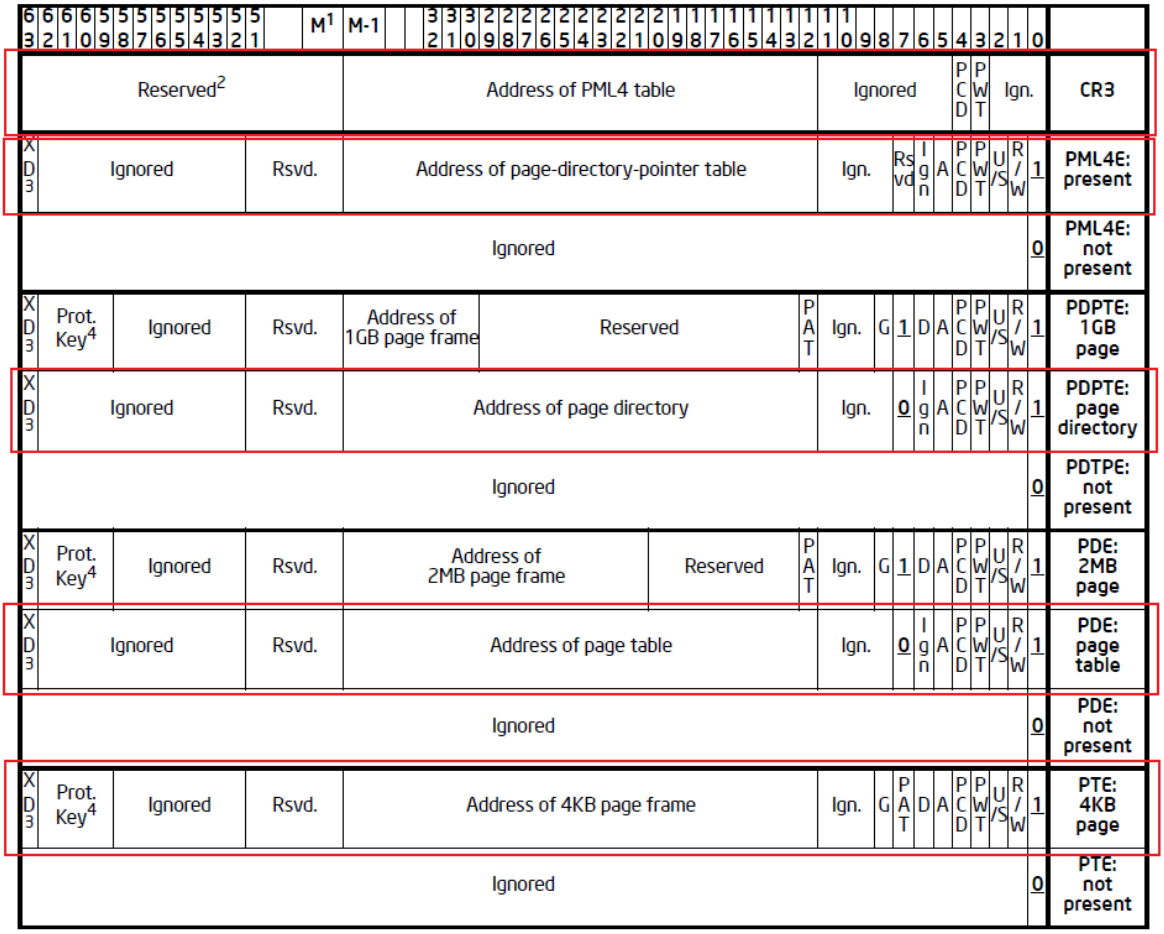


Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging

# 读写指定进程的内存

以win7 x86为例`PsGetCurrentProcess`和`PsGetCurrentThread`实现如下：

```
; 在低版本中fs启用，gs弃用。
; 而在win10中fs弃用，而gs启用
PsGetCurrentProcess:
    mov eax, dowrd ptr fs:[0x124]
    mov eax, dword ptr [eax + 0x50]
    ret

PsGetCurrentThread:
    mov eax, dowrd ptr fs:[0x124]
    ret
```

根据泄露的源码来看，fs指向的是`_kpcr`结构体，这个结构体每个版本都不一样

## 遍历进程

因为`_kpcr`、`EPROCESS`等内核对象结构中，各个成员的偏移可能不一样，故偏移要根据当前系统进行调试确定

遍历进程的原理：通过获取当前进程的`EPROCESS`结构中`ActiveProcessLinks`成员（此成员为一个双向链表节点的指针），遍历双向链表就可以遍历完整个进程列表

```
// 获取_EPROCESS.ActiveProcessLinks的偏移
// 在不同版本的系统上不一样
ULONG get_activeprocesslinks_offset()
{
    NTSTATUS status = STATUS_SUCCESS;

    // 获取系统版本
    ULONG offset = 0;
    RTL_OSVERSIONINFOW os_info = { 0 };
    status = RtlGetVersion(&os_info);
    if(!NT_SUCCESS(status)) {
        return offset;
    }

    // 判断系统版本
    switch(os_info.dwMajorVersion) {
    case 6:
        switch(os_info.dwMinorVersion) {
        case 1:              // win7
#ifdef _WIN64
            offset = 0x188;
#else
            offset = 0xb8;
#endif
            break;
        }
        break;
    case 10:                 // win10
#ifdef _WIN64
        offset = 0x2f0;
#else
        offset = 0xb8;
#endif
        break;
    }
    return offset;
}

// 获取指定进程的EPROCESS结构指针
PEPROCESS find_process_by_id(ULONG pid)
{
    // 获取ActiveProcessLinks字段的偏移
    ULONG offset = get_activeprocesslinks_offset();
    if(offset == 0) {
        return NULL;
    }

    // 遍历进程
    PEPROCESS first_eprocess = NULL, traverse_eprocess = NULL;
    first_eprocess = PsGetCurrentProcess();
    traverse_eprocess = first_eprocess;
    UINT8 found = FALSE;

    do {
        // 从EPROCESS中获取PID
        ULONG process_id = (ULONG)PsGetProcessId(traverse_eprocess);
        if(process_id == pid) {
```

```
            // 找到
            found = TRUE;
            break;
        }

        // 根据偏移计算下一个EPROCESS
        traverse_eprocess = (PEPROCESS)((PUCHAR)(((PLIST_ENTRY)
((PUCHAR)traverse_eprocess + offset))->Flink) - offset);
    } while(traverse_eprocess != first_eprocess);

    if(!found) {
        return NULL;
    }

    return traverse_eprocess;
}
```

## 读写进程内存

首先需要遍历进程，找到指定进程的`KPROCESS`（此结构为`EPROCESS`的第一个成员），取出`KPROCESS`的`DirectoryTableBase`目录表基址

```
0: kd> dt _KPROCESS
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : Uint4B
```

在写进程内存时，需要关闭内存的保护属性，内存保护属性：CR0的第15位

```
/*
 * 读取指定进程的内存
 *      pid: 进程id
 *      mem_addr: 内存地址
 *      read_buf: 缓冲区
 *      buf_size: 缓冲区大小
 *  返回值: IoStatus的Information
 */
ULONG_PTR read_process_memory(ULONG pid, PVOID mem_addr, PVOID read_buf, ULONG
buf_size)
{
    // 获取指定进程的EPROCESS
    PEPROCESS eprocess = find_process_by_id(pid);
    // 找到目录表基址
    PVOID old_cr3 = 0;
    PVOID dir_base = (PVOID)(*(PUINT_PTR)((PUCHAR)eprocess + 0x18));

    __asm {
        // 屏蔽中断，禁止线程切换
        cli

        // 保存原来的cr3
```

```
        mov eax, cr3
        mov old_cr3, eax

        // 修改cr3
        mov eax, dir_base
        mov cr3, eax
    }

    // 检查内存是否有效
    if(MmIsAddressValid(mem_addr)) {
        RtlCopyMemory(read_buf, mem_addr, buf_size);
    }

    __asm {
        // 还原cr3
        mov eax, old_cr3
        mov cr3, eax

        // 恢复中断
        sti
    }

    return buf_size;
}

/*
 * 写入指定进程的内存
 *      pid: 进程id
 *      mem_addr: 内存地址
 *      write_buf: 缓冲区
 *      buf_size: 缓冲区大小
 *  返回值: IoStatus的Information
 */
ULONG_PTR write_process_memory(ULONG pid, PVOID mem_addr, PVOID write_buf, ULONG
buf_size)
{
    // 获取指定进程的EPROCESS
    PEPROCESS eprocess = find_process_by_id(pid);
    // 找到目录表基址
    PVOID old_cr3 = 0;
    PVOID dir_base = (PVOID)(*(PUINT_PTR)((PUCHAR)eprocess + 0x18));

    __asm {
        // 屏蔽中断，禁止线程切换
        cli

        // 关闭内存写保护
        mov eax, cr0
        and eax, not 10000h
        mov cr0, eax

        // 保存原来的cr3
        mov eax, cr3
        mov old_cr3, eax

        // 修改cr3
        mov eax, dir_base
        mov cr3, eax
```

```
    }

    // 检查内存是否有效
    if (MmIsAddressValid(mem_addr)) {
        RtlCopyMemory(mem_addr, write_buf, buf_size);
    }

    __asm {
        // 还原cr3
        mov eax, old_cr3
        mov cr3, eax

        // 还原内存写保护
        mov eax, cr0
        or eax, 10000h
        mov cr0, eax

        // 恢复中断
        sti
    }

    return buf_size;
}
```