

最近在看老潘的《Windows 内核原理与实现》，在讲 windows 页目录自映射的时候发现很有意思。

BOOLEAN

```
MmInitSystem (  
    IN ULONG Phase,  
    IN PLOADER_PARAMETER_BLOCK LoaderBlock  
)  
{  
    ...  
    //  
    // 初始化一些与体系结构相关的数据结构  
    //  
  
    MmInitMachineDependent (LoaderBlock);  
    ...  
}
```

VOID

```
MmInitMachineDependent (  
    IN PLOADER_PARAMETER_BLOCK LoaderBlock  
)  
{  
    ...  
    // 这里开始建立一些页表用来映射系统 pte, 扩展的非换页内存池  
  
    StartPde = MiGetPdeAddress (MmNonPagedSystemStart);  
    EndPde = MiGetPdeAddress ((PVOID)((PCHAR)MmNonPagedPoolEnd - 1));  
  
    while (StartPde <= EndPde) {  
  
        ASSERT (StartPde->u.Hard.Valid == 0);  
  
        //  
        // Map in a page table page, using the  
        // slush descriptor if one exists.  
        //  
  
        TempPde.u.Hard.PageFrameNumber = MxGetNextPage (1, TRUE);  
        *StartPde = TempPde;  
        PointerPte = MiGetVirtualAddressMappedByPte (StartPde);  
        RtlZeroMemory (PointerPte, PAGE_SIZE);  
        StartPde += 1;  
    }  
}
```

```
...  
}
```

MmNonPagedSystemStart 和 MmNonPagedPoolEnd 分别代表了非换页内存的开始也结束地址，上面代码的作用就是给这段连续的虚拟地址建立页表，就是给页表分配物理内存，然后清 0。

MiGetPdeAddress 定义在 Mi386.h 中：

```
#define MiGetPdeAddress(va) ((PMMPTe)(((ULONG)(va)) >> 22) << 2) + PDE_BASE)
```

作用是通过一个虚拟地址 va，得到其对应的 pde 地址。先右移 22 位得到 pde 便宜，在乘以 4，得到实际的偏移，最后加上页目录表的开始地址，得到 pde 的地址。所以 StartPde 是一个 pde 的地址，注意不是其对应的内容。下面开始循环，一个一个的建立页表。

MxGetNextPage 得到下一个空闲的物理页面的页帧号，然后赋值给

TempPde.u.Hard.PageFrameNumber，MiGetPdeAddress 将计算出的 pte 转化成了 MMPTE 类型的指针：

```
typedef struct _MMPTE {  
    union {  
        ULONG Long;  
        HARDWARE_PTE Flush;  
        MMPTE_HARDWARE Hard;  
        MMPTE_PROTOTYPE Proto;  
        MMPTE_SOFTWARE Soft;  
        MMPTE_TRANSITION Trans;  
        MMPTE_SUBSECTION Subsect;  
        MMPTE_LIST List;  
    } u;  
} MMPTE;
```

一个 union 结构，在看 MMPTE_HARDWARE 类型：

```
typedef struct _MMPTE_HARDWARE {  
    ULONG Valid : 1;  
#if defined(NT_UP)  
    ULONG Write : 1;        // UP version  
#else  
    ULONG Writable : 1;      // changed for MP version  
#endif  
    ULONG Owner : 1;  
    ULONG WriteThrough : 1;  
    ULONG CacheDisable : 1;  
    ULONG Accessed : 1;  
    ULONG Dirty : 1;  
    ULONG LargePage : 1;  
    ULONG Global : 1;  
    ULONG CopyOnWrite : 1; // software field  
    ULONG Prototype : 1;   // software field  
#if defined(NT_UP)  
    ULONG reserved : 1;    // software field  
#else
```

```

        ULONG Write : 1;          // software field - MP change
#endif

        ULONG PageFrameNumber : 20;
} MMPTTE_HARDWARE, *PMMPTE_HARDWARE;

```

MMPTTE_HARDWARE 是一个 32 位字节大小的结构体，高 12 位代表了一个页表或页表项的结构。x86 体系中页表和页表项的结构几乎一样，windows 不做区别。最后 20 位保存了这个物理页表对应的页帧号。下面继续看：

*StartPde = TempPde; 将这个 pte 进行了赋值，也就是说 pde 指向了刚才分配的页表。然后就要对这个物理页表进行初始化清 0。要注意，现在 windows 已经开启了页映射机制，CPU 只认虚拟地址，但是现在系统的页表还没有建立起来，内核不能直接去访问刚才新分配的那个物理地址。如果要访问就需要知道这个物理地址对应的虚拟地址。按照传统逆向映射，现在只知道物理地址，最多能确定这个虚拟地址的中间 10 位。windows 做了一个很巧妙的设计，能让在知道一个 pde 地址的情况下，很容易算出 pte 指向的物理地址所对应的虚拟地址，知道了虚拟地址，就能访问页表的物理地址了。windows 是这样设计的：

页目录地址设为 0xc0300000，这个地址不是随便设计的，它的特点是高 10 位和中间 10 位都是 1100000000：

```

c    0    3    0    0    0    0    0
1100 0000 0011 0000 0000 0000 0000 0000
+---10-----+---10-----+---12-----

```

也就是说一个 pde 左移 10 位后，其对应的 pde 不变！而且页目录表的 0x300，也就是 768 项指向了页目录表本身。这 2 个巧妙的设计能让 windows 访问页目录表和页表本身所在页面的时候非常方便，而且确定了 pde 和 pte 的特殊关系，即 pde 左移 10 即是 pte 的地址，当然这种情况只是发生在访问页目录表和页表本身的时候才行的通。这 2 个设计恰恰满足了上面代码的需求。这种机制即称为 windows 的页目录自映射方案。

```

页目录表 +-----+<---
          |-----|  |
          |-----|  |
0xc00    |-----|----第 768 项指向了页目录表开始的地址
          |-----|
          |-----|
StartPde |-----|
          |-----|
          |.....|
          |-----|

```

pde 的 2 次查表过程如下：

- 1、通过高 10 位算出偏移为 0xc00，然后 0xc00 处有指向了 0xc0300000 处。
- 2、通过中间 10 位算出偏移仍为 0xc00，就是说对应的页表还是指向页目录的地址，其实页存放页目录也需要一个页，这个页的地址就是 0xc0300000。
- 3、在通过低 12 位偏移找到对应的 pde。

pte 的 2 次查表过程如下：

- 1、通过高 10 位算出偏移为 0xc00，然后 0xc00 处有指向了 0xc0300000 处。
- 2、通过中间 10 位算出偏移，内容还是 pde 的内容。

3、pde 指向了一个页表范围（1024 个）的开始地址， 加上偏移， 得到最后的页表地址。

接着上面的代码分析：

```
PointerPte = MiGetVirtualAddressMappedByPte (StartPde);
```

```
#define MiGetVirtualAddressMappedByPte(PTE) ((PVOID)((ULONG)(PTE) << 10))
```

它的作用就是通过 pde 的地址， 得到 pte 对应的物理地址所对应的虚拟地址。 原理在上面已经分析过了。

现在虚拟地址已经得到了， 那么 kernel 就可以给刚才分配的物理内存清 0 了：

```
RtlZeroMemory (PointerPte, PAGE_SIZE);
```