

关于Windows NT内存管理器的高层次信息已经够多的了, 所以这里不会再讲什么FLAT模型、虚拟内存之类的东西。这里我们只讲具体的底层的東西。我假定大家都了解>i386的体系结构。

目录

=====

- 00.内核进程线程结构体
- 01.页表
- 02.Hyper Space
- 03.System PTE'S
- 04.Frame data base (MmPfnDatabase)
- 05.Working Set
- 06.向pagefile换页
- 07.page fault的处理
- 08.从内存管理器角度看进程的创建
- 09.上下文切换
- 0a.某些未公开的内存管理器函数
- 0b.结语

附录

- 0c.某些未公开的系统调用
- 0d.附注及代码分析草稿

00.内核进程线程结构体

=====

Windows NT中的每一个进程都是EPROCESS结构体。此结构体中除了进程的属性之外还引用了其它一些与实现进程紧密相关的结构体。例如, 每个进程都有一个或几个线程, 线程在系统中就是ETHREAD结构体。我来简要描述一下存在于这个结构体中的主要的信息, 这些信息都是由对内核函数的研究而得知的。首先, 结构体中有KPROCESS结构体, 这个结构体中又有指向这些进程的内核线程(KTHREAD)链表的指针(分配地址空间), 基优先级, 在内核模式或是用户模式执行进程的线程的时间, 处理器affinity(掩码, 定义了哪个处理器能执行进程的线程), 时间片值。在ETHREAD结构体中还存在着这样的信息: 进程ID、父进程ID、进程映像名、section指针。quota定义了所能使用的分页和非分页池的极限值。VAD(virtual address descriptors)树定义了用户地址空间内存区的状况。关于Working Set的信息定义了在规定时间内有那些物理页是属于进程的。同时还有limit与statistics。ACCESS TOKEN描述了当前进程的安全属性。句柄表描述了进程打开的对象的句柄。该表允许不在每一次访问对象时检查访问权限。在EPROCESS结构体中还有指向PEB的指针。

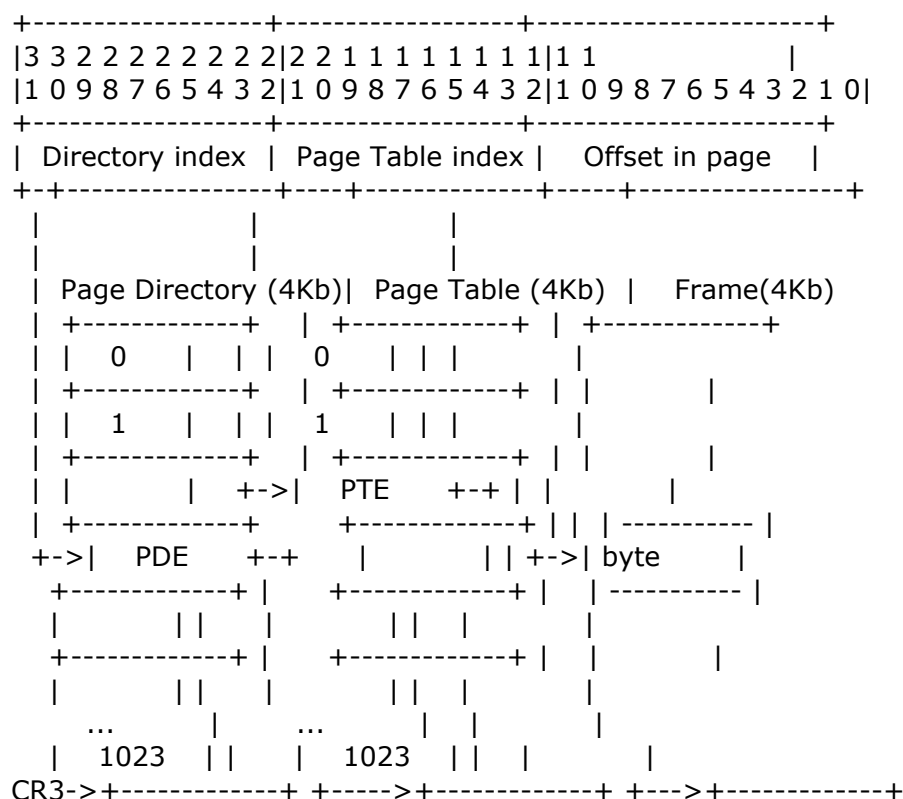
ETHREAD结构体还包含有创建时间和退出时间、进程ID和指向EPROCESS的指针, 启动地址, I/O请求链表和KTHREAD结构体。在KTHREAD中包含有以下信息: 内核模式和用户模式线程的创建时间, 指向内核堆栈基址和顶点的指针、指向服务表的指针、基优先级与当前优先级、指向APC的指针和指向TEB的指针。KTHREAD中包含有许多其它的数据, 通过观察这些数据可以分析出KTHREAD的结构。

01.页表

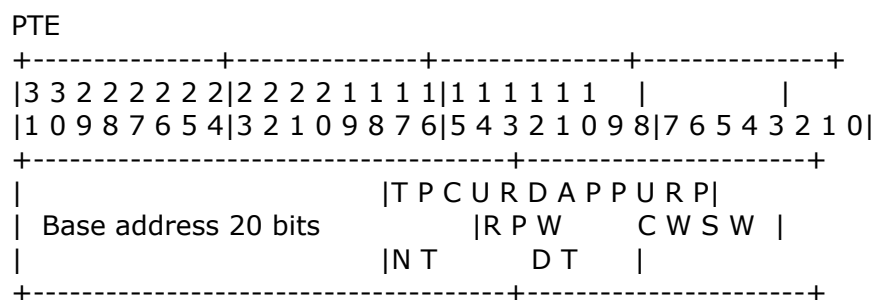
=====

通常操作系统使用页表来进行内存操作。在Windows NT中, 每一个进程都有自己私有的页表(进程的所有线程共享此页表)。相应的, 在进程切换时会发生页表的切换。为了加快对页表的访问, 硬件中有一个translation lookaside buffer(TLB)。在Windows NT中实现了两

级的转换机制。在386+处理器上将虚拟地址转换为物理地址过程(不考虑分段)如下:



注:在Windows NT中,从第4版起,除4Kb的页之外同时还使用了4Mb的页(Pentium及更高)来映射内核代码。但是在Windows NT中没有实际对可变长的页提供支持。



P - 存在位。此位如果未设置, 则在地址转换时会产生异常。一般说来, 在一些情况下 NT 内核会使用未设置此位的 PTE。

RW - 是否能写入

NT使用的为OS设计者分配的空闲位

PPT - proto pte

TRN - transition pte

当P位未设置时, 第5到第9位即派上用场(用于page fault处理)。它们叫做Protection Mask, 样子如下:

* MiCreatePagingFileMap

9 8 7 6 5

```
-----
| | | |
| | | +- Write Copy
| | | +--- Execute
| | +----- Write
| +----- NO CACHE
+----- Guard
```

GUARD | NOCACHE组合就是NO ACCESS

* MmGetPhysicalAddress

函数很短, 但能从中获得很多信息。在虚地址0xc0000000 - 0xc03fffff上映射有进程的页表。并且, 映射的机制非常精巧。在Directory Table(以下称DT)有1100000000b个表项(对应于地址0xc000..-0xc03ff..)指向自己, 也就是说对于这些地址DT用作了页表(Page Table) ! 如果我们使用, 比如说, 地址(为方便起见使用二进制)

1100000000.0000000101.0000001001.00b

0xc0... 页表选择 页表内偏移
页目录

通过页表101b的1001b号, 我们得到了PTE。但这还没完——DT本身映射在地址0xc0300000-0xc0300ffc上。在MmSystemPteBase中有值0xc0300000。为什么这样——看个例子就知道了:

1100000000.1100000000.0000001001.00b

0xc0... 0xc0... 页目录偏移
页目录 页表一
页目录
选择

最后, 在c0300c00包含着用于目录本身的PDE。这个PDE的基地址的值保存在MmSystemPageDirectory中。同时系统为映射物理页MmSystemPageDirectory保留了一个PTE, 这就是MmSystemPagePtes。

这样做能简化寻址操作。例如, 如果有PTE的地址, 则PTE描述的页的地址就等于PTE<<10。反过来: PTE=(Addr>>10)+0xc0000000。

除此之外,在内核中存在着全局变量MmKseg2Frame = 0x20000。该变量指示在从0x80000000开始的哪个地址区域直接映射到了物理内存,也就是说,此时虚拟地址0x80000000 - 0x9ffffffffff映射到了物理地址00000000-1f000000。

还有几个有意思的地方。从c0000000开始有个0x1000*0x200=0x200000=2M的描述地址的表(0-7ffffff)。描述这些页的PDE位于地址c0300000-0xc03007fc。对于i486,在地址c0200000-c027ffff应该是描述80000000到a0000000的512MB的表,但对于Pentium在区域0xc0300800-0xc03009fc是4MB的PDE,其描述了从0到1fc00000的步长为00400000的4M的物理页,也就是说选择了4M的页。对应于这些PDE的虚地址为80000000, 9ffffff。

这样我们就得到了页表的分布:

范围 c0000000 - c01ffffc 用于00000000-7ffffff的页表
范围 c0200000 - c027ffff "吃掉" 4M地址页的地址
范围 c0280000 - c02ffffc 包含用于a0000000 - bffffff的页
范围 c0300000 - c0300ffc PD 本身(描述范围c0000000 - c03ffff)
范围 c0301000 - c03013fc c0400000 - c04ffff HyperSpace (更准确的说,是1/4的hyper space)
范围 c0301400 - c03ffff 包含用于c050000 - fffffff的页

注:在0xc0301000-0xc0301ffc包含有描述hyper space的页表。这是内核的地址空间,且对于不同的进程映射的内容是不同的(另一方面,内核空间又总是在每个用户进程的上下文中)。这是进程私有的区域。例如,working set就位于hyper space中。页表的前256个PTE(hyper space的前1/4)为内核保留,而且在需要快速向frame中映射虚拟地址时使用。

我给出一个向区域0xc0200000-0xc027f000中一个地址进行映射的例子。

1100000000.1000000000.000000000000 = 0xc0200000

- 1) 解析出 PDE #1100000000 (4k 页) 并选出 PageDirectory
- 2) 在 Directory 中选出 PTE #1000000000 (c0300800)
这是个 4MB 的 PDE - 但这里忽略位长度,
因为 PDE 用作了 PTE. 结果 c0200000 - c0200fff 被映射为
80000000-80000fff

c0201000 映射到下面的 - 80400000- 80400fff.
等等直到 c027f000 - 9fc00000

PTE, 位于c0200000到c027ffff - 描述了80000000 - 9ffffc00 (512m)

02.Hyper Space

=====

HyperSpace是内核空间中的一块区域(4mb),不同的进程映射内容不同。对于转换,4MB足够放下页表完整的一页。这个表位于地址0xc0301000 - 0xc0301ffc(PDE的第0个表项位于0xc0300c04)。在内部,为向HyperSpace区域中映射物理页(当需要快速为某个frame组织虚拟地址时)要使用函数:

DWORD MiMapPageInHyperSpace(DWORD BaseAddr,OUT PDWORD Irql);

它返回HyperSpace中的虚拟地址,这个虚拟地址被映射到所要的物理页上。这个函数是如何工作的,工作的时候用到了什么?

在内核中有这样的变量:

```
MmFirstReservedMappingPte=0xc0301000
MmLastReservedMappingPte=0xc03013fc
```

这两个变量描述了255个pte, 这些pte描述了区域:

0xc0400000-0xc04fffff (1/4 HyperSpace)

在MmFirstReservedMappingPte处是一个pte, 其中的基址扮演了计数器的角色(从0到255)(当然, pte是无效的, p位无效)。为所需地址添加pte时要依赖计数器当前的值.....并且计数器使用了下开口堆栈的原理, 从ff开始。一般来说, 页表中的pte用作信息上的目的并不是唯一的情况。

03.System PTE'S

=====

在内核中有一块这样的内存——系统pte。什么是系统pte, 以及内核如何使用系统pte?

*见函数 MiReserveSystemPtes(...)

系统为空闲PTE维护了某些结构体。首先为了快速满足密集请求(当内核需要pte映射某些物理页时)系统中有个Sytem Ptes Pool。而且pool中有pte blocks(blocks表示请求是以block为单位来满足的, 一个block中有一些pte, 1、2、4、8和16个pte)。

系统中有以下这些表:

```
BYTE MmSysPteTables[16]={0,0,1,2,2,3,3,3,3,4,4,4,4,4,4,4};
DWORD MmSysPteIndex[5]={1,2,4,8,16};
DWORD MmFreeSysPteListBySize[5];
PPTTE MmLastSysPteListBySize[5];
DWORD MmSysPteListBySizeCount[5];
DWORD MmSysPteMinimumFree[5]={100,50,30,20,20}
PVOID MmSystemPteBase;// 0xc0200000
```

在pool中的空闲PTE被组织成了链表(当然, pte是位于页表中, 也就是说链表结构体位于页表中, 这是真的)。链表的元素:

```
typedef struct _FREE_SYSTEM_PTES_BLOCK{
/*pte0*/ SYSPTE_REF NextRef;           // 指向后面的block
/*pte1*/ DWORD FlushUnkn;             // 在Flush时使用
/*pte2*/ DWORD ArrayOfNulls[ANY_SIZE_ARRAY]; // 空闲 PTE
}FREE_SYSTEM_PTES_BLOCK PFREE_SYSTEM_PTES_BLOCK;
```

用作指向后面元素指针的PTE的地址可如此获得:VA=(NextRef>>10)+MmSystemPteBase (低10位永远为0, 相应的p位也为0)。链表最后一个元素NextRef域的值0xfffff000 (-1)。相应的, 链表有5个(block大小分别为1,2,4,8和16个pte)。

*见函数 MiReserveSystemPtes2(...) / MiInitializeSystemPtes

除pool外还有一个undocumented的空闲系统pte链表。

```
PPTTE MmSystemPtesStart[2];
PPTTE MmSystemPtesEnd[2];
SYSPTE_REF MmFirstFreeSystemPte[2];
```

```
DWORD MmTotalFreeSystemPtes[2];
```

在两个链表中有两个引用。链表的元素：

```
typedef struct _FREE_SYSTEM_PTES{
    SYSPTTE_REF Next; // #define ONLY_ONE_PTE_FLAG 2, last = 0xfffff000
    DWORD NumOfFreePtes;
}FREE_SYSTEM_PTES PFREE_SYSTEM_PTES;
```

而且，1号链表原则上没有组织。0号链表(MiReleaseSystemPtes)用于释放的pte。pte有可能进入System Ptes Pool。若在请求MiReserveSystemPtes(...)时pte的数目大于16，则同时pte从0号链表分配。也就是说，0号链表与pool有关联，而1号则没有。

为了使工作的结果不与TLB相矛盾，系统要么使用重载cr3，要么使用命令invlpg。“高级”函数

```
MiFlushPteList(PTE_LIST* PteList, BOOLEAN bFlushCounter, DWORD PteValue);
```

进行以下工作：

初始化PTE并调用invlpg(汇编指令)。

```
typedef struct PTE_LIST{
    DWORD Counter; // max equ 15
    PVOID PtePointersInTable[15];
    PVOID PteMappingAddresses[15];
};
```

如果Counter大于15，则调用KeFlushCurrentTb(只是重载CR3)，并且如果设置了bFlushCounter，则向MmFlushCounter加0x1000。

04. Page Frame Number Data Base (MmPfnDatabase)

=====

内核将有关物理页的信息保存在pfn数据库中(MmPfnDatabase)。本质上讲，这只是个0x18字节长的结构体块。每一个结构体对应一个物理页(顺序排列，所以元素常被称为Pfn - page frame number)。结构体的数量对应于系统中4KB页的数量(或者说是内核可见的页的数量，需要的话可以在boot.ini中使用相应的选项来为NT内核做出这块“坏”页区)。通常，结构体形式如下：

```
typedef struct _PfnDatabaseEntry
{
    union {
        DWORD NextRef; // 0x0 如果frame在链表中，则这个就是frame的号
        // 最后的一个为 -1
        DWORD Misc; // 同时另外一项信息，依赖于上下文
        // 见伪代码(通常 TmpPfn->0...)
        // 通常这里有 *KTHREAD, *KPROCESS,
        // *PAGESUPPORT_BLOCK...
    };
    PPTE PtePpte; // 0x4 指向 pte 或 ppte
    union { // 0x8
        DWORD PrevRef; // 前面的frame或 (-1, 第一个)
        DWORD ShareCounter; // Share 计数器
    };
};
```

```

};
WORD Flags;    // 0xc 见下面
WORD RefCounter; // 0xe 引用计数
DWORD Trans;   // 0x10 ?? 见下面. 用于 pagefile
DWORD ContFrame; // ContainingFrame; // 14
}PfnDatabaseEntry;

```

/*
Flags (名字取自windbg !pfn的结果)

掩码 位 名字 值

```

-----
0001 0    M    Modifyied
0002 1    R    Read In Progress
0004 2    W    WriteInProgress
0008 3    P    Shared
0070 [4:6] Color Color (In fact Always null for x86)
0080 7    X    Parity Error
0700 [8:10] State 0- Zeroed
        /List 1- Free
                2- StandBy
                3- Modified
                4- ModifiedNoWrite
                5- BadPage
                6- Active
                7- Trans
0800 11   E    InPageError

```

Trans域的值用在frame的内容位于PageFile中的时候或是frame的内容位于与这个Page File PTE对应的其它映象文件中的时候。

我给出未设置P位的PTE的例子(这种PTE不由平台体系结构确定, 而由OS确定)。

* 取自 @MiReleasePageFileSpace (Trans)

Page File PTE

```

+-----+-----+-----+-----+
|3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1|1|1|0 0 0 0 0|0 0 0 0|0|
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2|1|0|9 8 7 6 5|4 3 2 1|0|
+-----+-----+-----+-----+
| offset                                |T|P|Protect. |page |0|
|                                |R|P|mask   |file | |
|                                |N|T|      |Num   | |
+-----+-----+-----+-----+

```

Transition PTE

```

+-----+-----+-----+-----+
|3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1|1|1|0 0 0 0 0|0 0 0 0|0|
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2|1|0|9 8 7 6 5|4 3 2 1|0|
+-----+-----+-----+-----+
| PFN                                |T|P|Protect. |C W O W|0|
|                                |R|P|mask   |D T   | |
|                                |N|T|      |      | |
+-----+-----+-----+-----+

```

W - write

O - owner
 WT - write through
 CD - cache disable

可能所有这些现在还不很易懂,但是看完下面就能明白了。当然,这个结构体是未公开的。显然,结构体能够组织成链表。**frame**由以下结构体支持:

```
struct _MmPageLocationList{
PPfnListHeader ZeroedPageListhead;    //&MmZeroedPageListhead
PPfnListHeader FreePageListHead;      //&MmFreePageListHead
PPfnListHeader StandbyPageListHead;    //&MmStandbyPageListHead
PPfnListHeader ModifiedPageListHead;   //&MmModifiedPageListHead
PPfnListHeader ModifiedNoWritePageListHead;//&MmModifiedNoWritePageListHead
PPfnListHeader BadPageListHead;        //&MmBadPageListHead
}MmPageLocationList;
```

这其中包含了6个链表。各域的名字很好的说明了它们的用处。**frame**的状态与这些链表密切相关。下面列举了**frame**的状态:

状态	描述	链表
Zero	清零的可用空闲页	0
Free	可用空闲页	1
Standby	不可用但可轻易恢复的页	2
Modified	要换出的dirty页	3
ModifiedNoWrite	不换出的dirty页	4
Bad	不可用的页(有错误)	5
Active	活动页,至少映射一个虚拟地址	-

frame可能处在6个链表中的某一个,也可能不在这些链表中(状态为**Active**)。如果页属于某个进程,则这个页就被记录在**Working Set**中(见后面)。同时,如果**frame**由内存管理器自己使用,则一般可以不考虑这些**frame**的位置。

每个链表的表头都是下面这个样子:

```
typedef struct _PfnListHeader{
  DWORD Counter; // 链表中frame的数目
  DWORD LogNum;  // 链表号.0 - zeroed, 1- Free etc...
  DWORD FirstFn; // MmPfnDatabase中的第一个frame号
  DWORD LastFn;  // --/-- 最后一个.
}PfnListHeader PPfnListHeader;
```

除此之外,可以用“color”(就是**cache**)来寻址空闲**frame**(**zeroed**或是**free**)。如果看一下附录中的伪代码就容易理解了。我给出两个结构体:

```
struct {
  ColorHashItem* Zeroed; //(-1) нет
  ColorHashItem* Free;
}MmFreePagesByColor;

typedef struct _ColorHashItem{
  DWORD FrameNum;
```

```
PfnDatabaseEntry* Pfn;
} ColorHashItem;
```

有一套函数使用color来处理frame(处理cache)。例如, MiRemovePageByColor(FrameNum, Color); 看一下这些函数及其参数返回值的名称和函数的反汇编代码, 很容易猜到相应的内容, 所以这里就不描述了, 在说一句, 这些函数都是未导出的。在使用color的时候, 要考虑color掩码, 最后选择color。

Windows NT符合C2安全等级, 所以应该在为进程分配页的时候应将页清零。我们来看一下将frame清零的系统进程的线程。最后, 在Phase1Initialization()中所作的是调用MmZeroPageThread。不难猜到——线程将空闲页清零并将其移动到zeroed页的链表中。

```
MmZeroPageThread
{
//
//.... 没意思的东西我们略过 ;)
//
while(1)
{
KeWaitForSingleObject(MmZeroingPageEvent,8,0,0,0); // 等待事件
while(!KeTryToAcquireSpinLock(MmPfnLock,&OldIrql)); // 获取 PfnDatabase
while(MmFreePageListHead.Count){
    MiRemoveAnyPage(MmFreePageListHead.FirstFn&MmSecondaryColorMask);
    // 从空闲链表中取出页
    Va=MiMapPageToZeroInHyperSpace(MmFreePageListHead.FirstFn);
    KeLowerIrql(OldIrql);

    memset(Va,0,0x1000); // clear page

    while(!KeTryToAcquireSpinLock(MmPfnLock,&OldIrql);
    MiInsertPageInList(&MmZeroedpageListHead,FrameNum);
        // 将已清零的页插入Zero链表
    }
    MmZeroingPageThreadActive=0; // 清标志
    KeLowerIrql(OldIrql);
}
// 永不退出
}

// 函数只是将frame映射到定义的地址上
// 以使其可被清零
DWORD MiMapPageToZeroInHyperSpace(FrameNum)
{
if(FrameNum<MmKseg2Frame)return ((FrameNum+0x80000)<<12); // 落入直接映射区域

TmpPte=0xc0301404;
TmpVa=0xc0501000;
*TmpPte=0;
invlpg((void*)TmpVa); // asm instruction in fact
*TmpPte=FrameNum<<12|ValidPtePte;
return TmpVa; // always 0xc0501000;
}
```

在何时MmZeroingPageEvent被激活? 这发生在向空闲页链表中添加frame的时候:

```

MiInsertPageInList()
{
.....
if(MmFreePageListHead.Count>=MmMinimumFreePagesToZero&&
    !MmZeroingPageThreadActive)
{
    MmZeroingPageThreadActive=1;
    KeSetEvent(&MmZeroingPageEvent,0,0);
}
....
}

```

注:内核并不总是依赖这个线程,有时会遇到这样的代码,它获取一个空闲页,用过后自己将其清零。

05.Working Set

=====

Working Set——工作集,是属于当前进程的物理页集。内存管理器使用一定的机制跟踪进程的工作集。**working set**有两个限额:**maximum working set**和**minimum working set**。这是工作集的最大值和最小值。内存管理器以这两个值为依据来维护进程的工作集(工作集大小不小于最小值,不大于最大值)。在定义条件的时候,工作集被裁减,这时工作集的**frame**落入空闲链表。内核工作集是结构体的总和。

在进程结构体的偏移0xc8(NT4.0)有以下结构体。

```

typedef struct _VM{
/* C8*/ LARGE_INTEGER UpdateTime;          //0
/* D0*/ DWORD Pages;                      //8 called so, by S-Ice authors
/* D4*/ DWORD PageFaultCount              //0c faults;
//          in fact number of MiLocateAndReserveWsle calls
/* D8*/ DWORD PeakWorkingSetSize;         //10 all
/* DC*/ DWORD WorkingSetSize;             //14 in
/* E0*/ DWORD MinimumWorkingSet;         //18 pages, not in
/* E4*/ DWORD MaximumWorkingSet;         //1c bytes
/* E8*/ PWS_LIST WorkingSetList;          //20 data table
/* EC*/ LIST_ENTRY WorkingSetExpansion;    //24 expansion
/* F4*/ BYTE fl0; // Operation???          //2c
    BYTE fl1; // always 2???               //2d
    BYTE fl2; // reserved??? always 0      //2e
    BYTE fl3; //                           //2f
}VM *PVM;

```

WinDbg !procfields的扩展命令用到VM。这里重要的是,跟踪page fault的数量(PageFaultCount), MaximumWorkingSet和MinimumWorkingSet, 管理器以它们为基础来支持工作集。

注:实际上, PageFaultCount并非是严格的计数。这个计数在MiLocateAndReserveWsle函数中被扩大,因为这个函数不只在page fault时被调用,在某些其它情况下也会被调用(真的,很少见)。

下面这个结构体描述了包含工作集页的表。

```

typedef struct _WS_LIST{
DWORD    Quota;                          //0 ??? i'm not shure....
DWORD    FirstFreeWsle;                  // 4 start of indexed list of free items
DWORD    FirstDynamic;                   // 8 Num of working set wsle entries in the sta

```

```

rt
// FirstDynamic
DWORD    LastWsleIndex;    // c above - only empty items
DWORD    NextSlot;        // 10 in fact always == FirstDynamic
// NextSlot
PWSLE    Wsle;            // 14 pointer to table with Wsle
DWORD    Reserved1        // 18 ???
DWORD    NumOfWsleItems;   // 1c Num of items in Wsle table
// (last initialized)
DWORD    NumOfWsleInserted; // 20 of Wsle items inserted (WsleInsert/
//                               WsleRemove)
PWSHASH_ITEM HashPtr;      // 24 pinter to hash, now we can get index of
// Wsle item by address. Present only if
// NumOfWsleItems>0x180
DWORD    HashSize;        // 28 hash size
DWORD    Reserved2;       // 2c ???
}WS_LIST *PWS_LIST;

typedef struct _WSLE{ // 工作集表的元素
    DWORD PageAddress;
}WSLE *PWSLE;

// PageAddress 本身是工作集页的虚地址
// 低12位用作页属性(虚地址总是4K的倍数)

#define WSLE_DONOTPUTINHASH 0x400 // 不放在cache中
#define WSLE_PRESENT 0x1 // 非空元素
#define WSLE_INTERNALUSE 0x2 // 被内存管理器使用的frame

// 未设置WSLE_PRESENT的空闲WSLE本身是下一个空闲WSLE的索引。这样，空闲的WSLE就组织
// 成了链表。最后一个空闲WSLE表示为-1。

#define EMPTY_WSLE (next_emty_wsle_index) (next_emty_wsle_index<<4)
#define LAST_EMPTY_WSLE 0xffffffff0

typedef struct _WSHASH_ITEM{
    DWORD PageAddress; //Value
    DWORD WsleIndex; //index in Wsle table
}WSHASH_ITEM *PWSHASH_ITEM;

//cache函数很简单。内部函数的伪代码:
//MiLookupWsleHashIndex(Value,WorkingSetList)
//{
//Val=value&0xfffff000;
//TmpPtr=WorkingSetList->HashPtr;
//Mod=(Val>>0xa)%(WorkingSetList->HashSize-1);
//if(*(TmpPtr+Mod*8)==Val)return Mod;
//while(*(TmpPtr+Mod*8)!=Val)){
// Mod++;
// if(WorkingSetList->HashSize>Mod)continue;
// Mod=0;
// if(fl)KeBugCheckEx(0x1a,0x41884,Val,Value,WorkingSetList);
// fl=1;
// }
//return Mod;

```

```
//}
```

我们来看一下典型的进程working set。WorkingSetList位于地址MmWorkingSetList (0xc0502000)。这是hyper space的区域, 所以在进程切换时, 要更新这些虚地址, 这样, 每个进程都有自己的工作集结构体。在地址MmWsle (0xc0502690)上是Wsle动态表的起始地址。表的结尾的地址总是0x1000的倍数, 也就是说表可以结束在地址0xc0503000、0xc0504000等等上(这是为了简化对Wsle表大小的操作)。Cache(如果有)位于一个偏移上, Wsle不会向这个偏移增长。我们来详细看一下这个表:

```
// WsList-0xc0502000---
// ....
// -----0xc0502030----
// pde 00 fault counter
// pde 01 fault counter
// pde 02 fault counter
//
// +-Wsle==0xc0502690---      +---Pde/pte   +-----Pfn[0]-----
// |0 c0300000|403 Page Directory   |c0300c00 pde |pProcess
// |4 c0301000|403 Hyper Space      |c0300c04 pte |1
// |8 MmWorkingSetList(c0502000)|403 |c0301408 pte |2
// |c MmWorkingSetList+0x1000 | 403 |.           |3
// |10 MmWorkingSetList+0x2000 | 403 |.           .
// |      ....
// |FirstDynamic*4 FrameN
// |....           |.           .
// |
// |LastWsleIndex*4 FrameM
// +-----+      +-----+      +-----+
// | free items
// ....
// | 0xffffffff0
// +-----+

// Cache
// ....
```

这里有个有意思的地方, 在表的起始部分有FirstDynamic的页, 用于建立Wsle, WorkingSet List和cache。同时这里还有页目录frame, HyperSpace和某些其它的页, 这些页是内存管理器所需要的, 不能从工作集中移出(标志WSLE_INTERNALUSE)。之后, 我们还能看到两种对Pfn frame域偏移0使用的变体。对于页目录frame, 这是指向进程的指针, 对于通常的属于工作集的页, 这是在表内的索引。

在WorkingSetList和Wsle表的起始地址之间还有不大的0x660字节的空闲空间。关于如何分配这些空间的信息是没有的, 但是很快在WorkingSetList开始有用于用户空间(通常为低2GB)的page fault counter, 也就是说如果, 譬如说, 索引0x100的元素有值3, 则表示从3开始(如果不考虑可能的溢出)page fault用于范围[0x40000000-0x403fffff]的页。

工作集的限额在内核模式下可以通过导出的未公开函数来修改:

```
NTOSKRNL MmAdjustWorkingSetSize(
    DWORD MinimumWorkingSet OPTIONAL, // if both == -1
    DWORD MaximumWorkingSet OPTIONAL, // empty working set
    PVM Vm OPTIONAL);
```

为处理WorkingSet, 管理器使用了许多内部函数, 了解了这些函数就能明白其工作的原理。

06.向pagefile换页

=====

frame可以是空闲的——当RefCounter等于0且位于一个链表中时。frame可以属于工作集。在缺少空闲frame时或是在达到treshhold时, 就会发生frame的换出。这方面的高层次函数是有的。这里的任务是用伪代码来证实。

在NT中有最多16个pagefile。pagefile的创建发生于模块SMSS.EXE。这时打开文件及其句柄向PsInitialSystemProcess进程的句柄表拷贝。我给出创建pagefile的未公开系统函数的原型(如果不从核心调用的话就必须有创建这种文件的权限)。

```
NTSTATUS NTAPI NtCreatePagingFile(
    PUNICODE_STRING FileName,
    PLARGE_INTEGER MinLen, // 高位双字应为0
    PLARGE_INTEGER MaxLen, // minlen应大于1M
    DWORD Reserved // 忽略
);
```

每个pagefile都有一个PAGING_FILE结构体。

```
typedef struct _PAGING_FILE{
    DWORD MinPagesNumber;    //0
    DWORD MaxPagesNumber;    //4
    DWORD MaxPagesForFlushing; //8 (换出页的最大值)
    DWORD FreePages;         //c(Free pages in PageFile)
    DWORD UsedPages;         //10 忙着的页
    DWORD MaxUsedPages;      //14
    DWORD CurFlushingPosition; //18 -???
    DWORD Reserved1;         //1c
    PPAGEFILE_MDL Mdl1; // 20 0x61 - empty ???
    PPAGEFILE_MDL Mdl2; // 24 0x61 - empty ???
    PRTL_BITMAP PagefileMap; // 28 0 - 空闲, 1 - 包含换出页
    PFILE_OBJECT FileObject; //2c
    DWORD NumberOfPageFile; //30
    UNICODE_STRING FileName; //34
    DWORD Lock;              //3d
}PAGING_FILE *PPAGING_FILE;
```

```
DWORD MmNumberOfActiveMdlEntries;
DWORD MmNumberOfPagingFiles;
```

```
#define MAX_NUM_OF_PAGE_FILES 16
PPAGING_FILE MmPagingFile[MAX_NUM_OF_PAGE_FILES];
```

在内存子系统启动时(MmInitSystem(...))会启动线程MiModifiedPageWriter, 该线程进行以下工作: 初始化MiPaging和 MiMappedFileHeader, 在非换出域中创建并初始化MmMappedFileMdl, 建立优先级LOW_REALTIME_PRIORITY+1, 等待KEVENT, 初始化MmMappedPageWriterEvent和MmMappedPageWriterList链表, 启动MiMappedPageWriter线程, 启动函数MiModifiedPageWriterWorker。

在任务MiModifiedPageWriterWorker中会等待事件MmModifiedPageWriterEvent, 处理链表M

mModifiedNoWritePageList和MmModifiedPageList并准备实现向映象文件或pagefile的页换出(调用MiGatherMappedPages或是MiGatherPagefilePages)。

在MiGatherPagefilePages中使用IoAsynchronousPageWrite()函数进行frame的换出。而且不是一个frame, 而是一簇(页数目总和为MmModifiedWriteClusterSize)。向pagefile换出页是由PAGING_FILE结构体中的PagefileMap来跟踪的。

研究函数的伪代码在appendix.txt中。这里描述伪代码没有什么意义——都很简单。

07.page fault的处理

=====

对于转向对pagefault的研究, 我们现在有了所有必须的信息了。转换线性地址时, 当线性地址(分页机制打开)的所用的PDE/PTE的P(present)位无效或是违反了保护规则, 在+i386处理器里会产生异常14。这时, 在堆栈中有错误代号, 包含有以下信息: 用户/内核错误位(异常发生在ring3还是ring0?), 读写错误位(试图读还是写?), 页存在位。除此之外, 在CR2寄存器中存有产生异常的32位线性地址。内核中处理14号中断的是_KiTrap0E。

当要转换的页没有相应的物理页时, 内存管理器执行确定好的工作来“修正”。这些是由异常处理函数调用高层函数MmAccessFault (Wr,Addr,P);来完成的。在对伪代码的进行分析之前, 想一下在什么样的情况下会发生page fault是很有用的。

最显然的就是访问错误, 这时ring3的代码试图写入PTE/PDE中未设置U位的页或是写入了只读的页(PTE/PDE中未设置W位)。再有, 页可以被换出到页面文件中, 对应于这些页的PTE中未设置P位, 但有信息指示在哪个页面文件中寻找frame, 以及frame的偏移。还有一个类似的情况——frame属于映象文件。除此之外, 所转换的页可能只属于已分配的内存区(使用NtAllocateMemory), 也可能转换的是原先没转换过的页, 这中情况下, VMM分配清零过的frame(这是C2的要求)。最后, 异常还可能由写copy on write页和转换共享内存引发。以上只列出了主要的情况。

处理的结果通常是向当前进程的Working Set中添加相应的frame。

异常的每一种情况都相应有一个内部的结构体与之相关联, VMM就处理这些结构体。这些结构体十分复杂, 要对它们进行完整的描述的话, 需要反汇编大量的函数。目前还没有大部分结构体的完整信息, 但对于理解异常处理程序来说并不要求知道这些。我来大致描述一下VAD和PPTE的概念, 研究异常处理程序的伪代码要用到。

VAD

操作虚拟地址需要用到VAD (Virtual Address Descriptor)。我们熟知的(有一个几乎与之同名的Win32函数调用这个函数)未公开函数NtAllocateVirtualMemory (ring0下是ZwAllocateVirtualMemory)操作这些结构体。

每一个VAD都描述了虚地址空间中的区域, 实际上, 除了区域的起止地址外还有保护信息(见ZwAllocateVirtualMemory函数的参数)。而同时还有其它一些特殊的信息(目前除了首部之外还没有VAD的完整信息)。VAD结构体只对用户地址(低2GB)有意义, 使用这些结构体VMM可以捕获到发生异常的区域。VAD的结构是一个平衡二叉树(有内部函数负责修整此树), 这是为查找而进行的优化。在VAD中有两个指向后面元素——左右子树——的指针。树的根位于EPROCESS结构体的VadRoot域(NT 4.0下是偏移0x170)。当然, 每一个进程都有自己的VAD树。VAD的首部形式如下:

```
typedef struct vad_header {
```

PPTe

PTE points to PPTE

*MmAccessFault

NTSTATUS MmAccessFault (BOOL Wr, DWORD Addr, BOOL P)

轮到MiDispatchFault了，它主要是基于一些更低级的函数：MiResolveTransitionFault、MiResolveDemandZeroFault、MiResolveDemandZeroFault、MiResolveProtoPteFault和MiResolvePageFileFault。从这些函数的名字可以明显看出，这个函数用于确定更为具体的情况：

状态为'transition' (可能会很快回收入Working Set)的页应该是空白的frame, PTE指向PTE并且frame换出到相应的页面文件中。在与页面文件有关的和某些与PPTE有关的情况下, 接着可能需要从文件中读取frame, 此时函数返回值为0xc0033333, 表示必须从文件中读取页。这在MiDispatchFault中是靠IoPageRead进行的。我们来更仔细的研究一下所提到的函数。我们从MiResolveDemandZeroFault开始。

如果看一下这个函数的伪代码, 则可以轻易的明白它的工作逻辑。请求zero frame并且进程得到这个frame。这时执行函数MiRemoveZeroPage或是MiRemoveAnyPage。第一个函数从zero页的链表中取一页。如果未能成功, 则通过第二个函数选择任何一页。这样的话, 该页就由MiZeroPhysicalPage来清零。最终, 在MiAddValidPageToWorkingSet中, 该清零的页被添加到工作集中(恰好, 这个事实证明在分配内存时进程不能取得对未处理页的访问)。现在我们来研究一下更为复杂的情况——一页位于页面文件中。

前面的伪代码需要一个结构体。在准备从文件中读取页的时候, 会填充PAGE_SUPPORT_BLOCK结构体。之后, 对所有即将参与到操作中的PFN进行以下操作: 设置read in progress标志并在Misc域中写入PAGE_SUPPORT_BLOCK的地址(函数MiInitializeReadInProgressPfn)。最后, 函数返回magic number 0xc0033333, 表示随后要在IoPageRead调用中使用此结构体(恰巧, IoPageRead被导出了, 但是未公开的。从其伪码中可以很容易地得到其原型)。

```
typedef struct _PAGE_SUPPORT_BLOCK{    // size: 0x98
    DISPATCHER_HEADER DispHeader;    // 0 FastMutex
    IO_STATUS_BLOCK IoStatusBlock;    // 0x10
    LARGE_INTEGER AddrInPageFile;    // 0x18 (file offset)
    DWORD RefCounter;                // 0x20 (0|1) ???
    KTHREAD Thread;                  // 0x24
    PFILE_OBJECT FileObject;         // 0x28
    DWORD AddrPte;                   // 0x2c
    PPFN pPfn;                       // 0x30
    MDL Mdl;                         // 0x34
    DWORD MdlFrameBuffer[0x10];     // 0x50
    LIST_ENTRY PageSupportList;      // 0x90 与MmInPageSupportList有关的链表
}PAGE_SUPPORT_BLOCK *PAGE_SUPPORT_BLOCK;

struct _MmInPageSupportList{
    LIST_ENTRY PageSupportList;
    DWORD Count;
}MmInPageSupportList;
```

函数MiResolvePageFileFault本身非常简单, 除了填充相应的结构体并返回0xc0033333之外什么也不干。剩下的就是执行MiDispatchFault。这很合乎情理, 如果还记得复用代码的原则的话。

还有一个不太复杂的函数MiResolveTransitionFault。对于状态为transition的frame还需要再多说几句。从这个状态中frame可以很快地返回到进程的Working Set中。

于是, 留下了最后一种情况——PROTO PTE。这种情况的处理函数也不太复杂, 而且支撑其的基础我们已经讲过了。实际上还有一个函数与这种情况有关, 这就是MiCompleteProtoPteFault, 从MiDispatchFault中调用。要想理解这些函数的工作就去看一下伪代码。

07. section 对象

=====

NT 中的section对象就是一块内存, 这块内存由一个进程独有或几个进程共享。在Win32子系

统中section就是文件映射 (file mapping object)。我们来看一下section对象到底是什么。

section是NT下非常常用的对象, 执行系统使用section来将可执行映象加载到内存中并用其来管理cache。section同时也用在向进程地址空间中映射文件。这时访问文件就像访问内存。section对象, 就像其它的对象一样, 是由对象管理器创建的。高层次的信息告诉我们, 对象的body中包含着以下类型的信息: section的最大值, 保护属性, 其它属性。什么是section的最大可访问值, 这不说也知道。保护属性是用于section页的属性。其它section属性有表示是文件section还是为空值(映射入页面文件)的标志, 以及section是否是base的。base的section以相同的虚拟地址映射入所有进程的地址空间。

为了得到此对象结构的真实信息, 我反汇编了一些用于section的内存管理器函数。下面的信息可是在别的地方见不到的。我们先来看结构体。

系统中的每一个文件都是对象 (NTDDK.H中有描述) FILE_OBJECT。在这个结构体中有SectionObjectPointer。NTDDK.H中同样有它的结构。

```
//
:
PSECTION_OBJECT_POINTERS SectionObjectPointer;
:
//

typedef struct _SECTION_OBJECT_POINTERS {
    PVOID DataSectionObject;
    PVOID SharedCacheMap;
    PVOID ImageSectionObject;
} SECTION_OBJECT_POINTERS;
```

在结构体中有两个指针——DataSectionObject 和 ImageSectionObject。NTDDK.H把它们写成了PVOID, 因为它们引用的是未公开的结构体。DataSectionObject用在将文件作为数据打开的时候。ImageSectionObject——此时当作映象。这些指针的类型全都一样, 且可以称之为PCONTROL_AREA。所有下面这些结构体都是Windows 2K的, 较之NT 4.0的有些变化。

```
typedef struct _CONTROL_AREA { // for NT 5.0, size = 0x38
    PSEGMENT pSegment; //00
    PCONTROL_AREA Flink; //04
    PCONTROL_AREA Blink; //08
    DWORD SectionRef; //0c
    DWORD PfnRef; //10
    DWORD MappedViews; //14
    WORD Subsections; //18
    WORD FlushCount; //1a
    DWORD UserRef; //1c
    DWORD Flags; //20
    PFILE_OBJECT FileObject; //24
    DWORD Unknown; //28
    WORD ModWriteCount; //2c
    WORD SystemViews; //2e
    DWORD PagedPoolUsage; //30
    DWORD NonPagedPoolUsage; //34
} CONTROL_AREA, *PCONTROL_AREA;
```

我们可以看到, CONTROL_AREA形成了一个链表, 结构体中包含着统计值和标志。为了理解标志所代表的信息, 我给出它们的值(用于NT5.0

```

/***** nt5.0 *****/
#define BeingDeleted          0x1
#define BeingCreated          0x2
#define BeingPurged           0x4
#define NoModifiedWriting     0x8
#define FailAllIo             0x10
#define Image                 0x20
#define Based                 0x40
#define File                  0x80
#define Networked             0x100
#define NoCache               0x200
#define PhysicalMemory        0x400
#define CopyOnWrite           0x800
#define Reserve               0x1000
#define Commit                0x2000
#define FloppyMedia           0x4000
#define WasPurged             0x8000
#define UserReference         0x10000
#define GlobalMemory          0x20000
#define DeleteOnClose         0x40000
#define FilePointerNull       0x80000
#define DebugSymbolsLoaded    0x100000
#define SetMappedFileIoComplete 0x200000
#define CollidedFlush         0x400000
#define NoChange              0x800000
#define HadUserReference      0x1000000
#define ImageMappedInSystemSpace 0x2000000

```

紧随CONTROL_AREA之后的是Subsection的数目Subsections。每一个Subsection都描述了关于具体的文件映射section的信息。例如，read-only, read-write, copy-on-write等等的section。NT5.0的SUBSECTION结构体：

```

typedef struct _SUBSECTION { // size=0x20 nt5.0
    // +0x10 if GlobalOnlyPerSession
    PCONTROL_AREA ControlArea; //38, 00
    DWORD Flags; //3c, 04
    DWORD StartingSector; //40, 08
    DWORD NumberOfSectors; //44, 0c
    PVOID BasePte; //48, 10 pointer to start pte
    DWORD UnusedPtes; //4c, 14
    DWORD PtesInSubsect; //50, 18
    PSUBSECTION pNext; //54, 1c
} SUBSECTION, *PSUBSECTION;

```

在subsection中有指向CONTROL_AREA的指针，标志，指向base Proto PTE的指针，Proto PTE的数目。StartingSector是4K block的编号，文件中的section起始于此。在标志中还有额外的信息：

```

#define SS_PROTECTION_MASK          0x1f0
#define SS_SECTOR_OFFSET_MASK      0xfff00000 // (low 12 bits)
#define SS_STARTING_SECTOR_HIGH_MASK 0x000ffc00 // (nt5 only) (in pages)

//other 5 bit(s)

```

```
#define ReadOnly    1
#define ReadWrite   2
#define CopyOnWrite 4
#define GlobalMemory 8
#define LargePages 0x200
```

我们来看剩下的最后一个结构体SEGMENT，它描述了所有的映射和用于映射section的Proto PTE。SEGMENT的内存是从paged pool中分配的。我给出SEGMENT结构体(NT 5.0)

```
typedef struct _SEGMENT {
    PCONTROL_AREA ControlArea; //00
    DWORD BaseAddr; //04
    DWORD TotalPtes; //08
    DWORD NonExtendedPtes; //0c
    LARGE_INTEGER SizeOfsegemnt; //10
    DWORD ImageCommit; //18
    DWORD ImageInfo; //1c
    DWORD ImageBase; //20
    DWORD Committed; //24
    PTE PteTemplate; //28 or 64 bits if pae enabled
    DWORD BasedAddr; //2c
    DWORD BaseAddrPae; //30 if PAE enabled
    DWORD ProtoPtes; //34
    DWORD ProtoPtesPae; //38 if PAE enabled
}SEGMENT,*PSEGMENT;
```

正如我所料，结构体包含着对CONTROL_AREA的引用，指向Proto PTE的pool的指针和所有section的信息。有个东西需要解释一下。结构体的样子依赖于是否支持PAE。PAE就是Physical Address Extension。从第5版开始，Windows NT包含了支持PAE的内核Ntkrnlpa.exe。总的来讲，支持PAE就意味着在NT里可以使用的虚拟地址不是4GB而是64GB。在使用PAE时的地址转换又多了一级——所有的虚地址空间被分为4部分。在打开PAE时PTE和PDE的大小不是4B而是8B，这我们可以从SEGMENT结构体中看出。现在还不需要进一步详细的讲PAE，毕竟很少用到，所以我们就此打住。

描述section的所有结构体都介绍过了，而section对象结构体本身还没有提到。从直观上可以想到，它应该会引用到SEGMENT或是CONTROL_AREA，因为有了这两个结构体后就可以得到保存的所有信息。通过反汇编得到的section对象的body为以下形式：

```
typedef struct _SECTION_OBJECT { // size 0x28
    VAD_HEADER VadHeader; // 0
    PSEGMENT pSegment; //0x14 Segment
    LARGE_INTEGER SectionSize; //0x18
    DWORD ControlFlags; //0x20
    DWORD PgProtection; //0x24
} SECTION_OBJECT, *SECTION_OBJECT;
```

```
#define PageFile    0x10000
#define MappingFile 0x8000000
#define Based       0x40
#define Unknown     0x800000 // not sure, in fact it's AllocAttrib&0x400000
```

我们看到，所得的结构体完全符合现有的高层信息的描述。唯一可能有疑问的就是VAD_HEADER。它描述了base section在地址空间中的位置。VAD_HEADER位于顶点为_MmSectionBasedRoot的VAD树中。我们再次体会到，要理解操作系统的工作原理，就要理解其内部的结构。为了有一个总体上的把握，下面给出了描述section的结构体间互相联系的一个图。

```
SECTION_OBJECT->SEGMENT<->CONTROL_AREA->FILE_OBJECT-
>SECTION_OBJECT_POINTERS+
      ^                               |
      +-----+-----+-----+-----+
```

08.从内存管理器角度看进程的创建

前面我们从Win32角度介绍过进程的创建，也讲过内存管理器和对象管理器的工作原理，以及section对象结构体。现在最有意思的当然就是在进程创建中将内存管理器也考虑进来。

进程是用未公开的系统调用NtCreateProcess()创建的。下面给出其伪代码：

```

/*****
/* -- Here it is, just wrapper -- */
NtCreateProcess(
    OUT Handle,
    IN ACCESS_MASK Access,
    IN POBJECT_ATTRIBUTES ObjectAttrib,
    IN HANDLE Parent,
    IN BOOLEAN InheritHandles,
    IN HANDLE SectionHandle,
    IN HANDLE DebugPort,
    IN HANDLE ExceptionPort
)
{
if(Parent)
{
    ret=PspCreateProcess(Handle,
        Access,
        ObjectAttrib,
        Parent,
        InheritHandles,
        SectionHandle,
        DebugPort,
        ExceptionPort);
}
else ret=STATUS_INVALID_PARAMETER;
return ret;
}

```

我们看到，NtCreateProcess是对另一个内部函数PspCreateProcess的封装。NtCreateProcess进行的唯一工作就是检查Parent(父进程句柄)。但是接下来我们看到，对于NT来说这并没有什么意义，因为总的来说，进程的继承性本身没有特别的意义。现在我们来查看PspCreateProcess()。

```

PspCreateProcess(
    OUT PHANDLE Handle,
    IN ACCESS_MASK Access,
    IN POBJECT_ATTRIBUTES ObjectAttrib,
    IN HANDLE Parent,
    IN BOOLEAN InheritHandles,
    IN HANDLE SectionHandle,

```

```

    IN HANDLE DebugPort,
    IN HANDLE ExceptionPort
);

```

我很快注意到，函数中的Parent参数可以接受值0，这就表明在NtCreateProcess中检验此参数是为了限制用户模式。函数的参数中有对section、debug port和exception port、父进程的引用。通过调用ObReferenceObjectByHandle，可以得到指向这些对象的指针。实际上父进程句柄通常传递的是-1，这表示是当前进程。如果Parent等于0，则进程的affinity就不从父进程处取得，而是从系统变量中取得。

```

if(Parent)
{
    //Get pointer to father's body
    ObReferenceObjectByHandle(Parent,0x80,PsProcessType,PrevMode,&pFather,0)
;
    AffinityMask=pFather->Affinity; // on witch processors will be executed
    Prior=8;
}
else {
    pFather=0;
    AffinityMask=KeActiveProcessors;
    Prior=8;
}

```

优先级总是为8。随后，创建进程对象。NT4.0下其大小为504字节。

```

// size of process body - 504 bytes
// creating process object... (type object PsProcessType)
ObCreateObject(PrevMode,PsProcessType,ObjectAttrib,PrevMode,0,504,&pProcess);
// clear body
memset(pProcess,0,504);

```

初始化某些域和Quota Block(见对象管理器的相关介绍)。

```

pProcess->CreateProcessReported=0;
pProcess->DebugPort=pDebugPort;
pProcess->ExceptPort=pExceptPort;

// Inherit Quota Block, if pFather==NULL, PspDefaultQuotaBlock
PspInheritQuota(pProcess,pFather);

if(pFather){
    pProcess->DefaultHardErrorMode=pFather->DefaultHardErrorMode;
    pProcess->InheritedFromUniqueProcessId=pFather->UniqueProcessId;
}
else {
    pProcess->InheritedFromUniqueProcessId=0;
    pProcess->DefaultHardErrorMode=1;
}

```

之后，调用MmCreateProcessAddressSpace，创建地址上下文。参数是函数得到的指向进程的指针、工作集的大小和指向结果结构体的指针。这个结构体形式如下：

```

struct PROCESS_ADDRESS_SPACE_RESULT{
    dword Dt; // dict. table phys. addr.
    dword HypSpace; // hyp space page phys. addr.
}

```

```

dword WorkingSet; // working set page phys. addr.
}CASResult;

```

```

MmCreateProcessAddressSpace(PsMinimumWorkingSet,pProcess,&CASResult);

```

我们看到，函数向我们返回的是页表的物理地址描述符(用于新地址空间的CR3的内容)，Hyper Space的页地址和工作集的页地址。在此之后是初始化进程对象的某些域：

```

pProcess->MinimumWorkingSet=MinWorkingSet;
pProcess->MaximumWorkingSet=MaximumWorkingSet;

```

```

KeInitializeProcess(pProcess,Prior,AffinityMask,&CASResult,pProcess->
DefaultHardErrorProcessing&0x4);

```

```

pProcess->ForegroundQuantum=PspForegroundQuantum;

```

如果有父进程且设置了标志参数，则会继承父进程的句柄表：

```

if(pFather) // if there is father and inherithandle, so, inherit handle db
{
    pFather2=0;
    if(bInheritHandle)pFather2=pFather;
    ObInitProcess(pFather2,pProcess); // see info about ObjectManager
}

```

下面的东西比较有意思，证明了NT执行系统的灵活性，从表面上是看不出来的。如果在参数中有指定的section，则使用这个section来初始化进程的地址空间，否则其工作就会像*UNIX中的fork()。

```

if(pSection)
{
    MmInitializeProcessAddressSpace(pProcess,0,pSection);
    ObDereferenceObject(pSection);
    res=ObInitProcess2(pProcess); //work with unknown byte +0x22 in process
    if(res>=0)PspMapSystemDll(pProcess,0);
    Flag=1; //Created addr space
}
else { // if there is futher, but no section, so, do operation like fork()
    if(pFatherProcess){
        if(PsInitialSystemProcess==pFather){
            MmRes=MmInitializeProcessAddressSpace(pProcess,0,0)
;
        }
        else {
            pProcess->SectionBaseAddress=pFather->SectionBaseAddress;
            MmRes=MmInitializeProcessAddressSpace(pProcess,pFather,0);
            Flag=1; //created addr space
        }
    }
}

```

接下来是使用PsActiveProcessHead将进程插入Active Process链表，创建Peb和做其它辅助性的工作。我们不再赘述。最后，当所有的工作都做完后，进行安全子系统方面的工作。我们过去曾研究过安全子系统(见对象管理器部分)，所以这里只简单的给出其伪代码。只是我注意到，如果父进程是system(句柄值等于PspInitialSystemProcessHandle)，则不对其

安全性进行检验。

```
// finally, security operations
if(pFather&&PspInitialSystemProcessHandle!=Father)
{
    ObGetObjectSecurity(pProcess,&SecurityDescriptor,&MemoryAllocated);
    pToken=PsReferencePrimaryToken(pProcess);
    AccessRes=SeAccessCheck(SecurityDescriptor,&SecurityContext,
        0,0x2000000,
        0,0,&PsProcessToken->GenericMapping,
        PrevMode,pProcess->GrantedAccess,
        &AccessStatus);
    ObDereferenceObject(pToken);
    ObReleaseObjectSecurity(SecurityDescriptor,MemoryAllocated);
    if(!AccessRes)pProcess->GrantedAccess=0;
    pProcess->GrantedAccess|=0x6fb;
}
else{
    pProcess->GrantedAccess=0x1f0fff;
}

if(SeDetailedAuditing)SeAuditProcessCreation(pProcess,pFather);
```

最有意思的是函数KeInitializeProcess和MmCreateProcessAddressSpace。前一个函数除了初始化进程对象的其它成员之外，还要初始化TSS中的IO位图的偏移。

```
pProcess->IopmOffset=0x20ad; // IOMAP BASE!!!
// You can patch kernel here and
// got i/o port control ;)
```

偏移的选取是这样的，它指向I/O位图，这样就能阻止进程直接使用I/O端口。

在函数MmCreateProcessAddressSpace中进行的是进程地址空间的创建。我就不给出所有的伪代码了，只简要的写主要的操作。它为Hyper Space, Working Set和Page Directory选择页。反汇编后的代码证实了，它们是从zero frame链表中选出或是由MiZeroPhysicalPage函数来清零的。之后初始化新创建的Page Directory。

```
pProcess->WorkingSetPage=Frame3; // WorkingSetPage
(MmPfnDatabase+0x18*Frame)->Pte=0xc0300000;
ValidPde_U=ValidPdePde&0xefff^Frame2; // HyperSpace

/*****IMPORTANT!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!/
/* 重要! 这里初始化PD */
/*****/

Va=MiMapPageInHyperSpace(Frame,&LastIrql);
// no we got Va of our new Page Directory
// Fill some fields
*(Va+0xc04)=ValidPde_U; // HyperSpace
ValidPde_U=ValidPde_U&0xffff^PhysAddr; // DT
*(Va+0xc00)=ValidPde_U; // self-pde

// copy from current process, kernel address mapping
memcpy(
    (MmVirtualBias+0x80000000)>>0x14+Va, // it's like that we found,
```

```

// what MmVirtualBias is it ;)
(MmVirtualBias+0x80000000)>>0x14+0xc0300000,
0x80 // 32 pdes -> 4Mb*32=128Mb
);

memcpy( // copy pdes, corresponding to NonPagedArea
    MmNonPagedSystemStart>>0x14+Va,
    MmNonPagedSystemStart>>0x14+0xc0300000,
    (0xc0300ffc-MmNonPagedSystemStart>>0x14+0xc0300000)&0xffffffffc+4);

memcpy(Va+0xc0c, // cache, forgot about it now, it's another story ;)
    0xc0300c0c,
    (MmSystemCacheEnd>>0x14)-0xc0c+4
);

```

也就是将PDE拷贝到内核地址空间中去(其对所有的进程不变, Hyper Space除外), 而且是拷贝到不可换出的区域。同时这个空间是属于系统cache的。

09.上下文切换

=====

知道了ETHREAD、EPROCESS结构体和内存管理器的工作原理, 就不难猜到上下文切换时会发生什么。Windows NT的设计者使用线程, 不关心共享的是谁的地址空间, 也就是说有两种可能: 线程属于当前进程——必需要切换到另一个线程(更新堆栈并更换GDT描述符), 而线程属于另一个进程, 必需切换到那个进程(重新加载CR3)。对此, 为了证实我的推测, 我反汇编了KeAttachProcess函数。这个函数是未公开的, 但所有已知的函数都用其来切换到另一进程的地址空间。通过KeDetachProcess可以返回到当前进程。KeAttachProcess使用下述内部函数:

KiAttachProcess - KeAttachProcess仅仅是对这个函数的封装
 KiSwapProcess - 更换地址空间。(本质上就是重新加载CR3)
 SwapContext - 更换上下文。一般不管地址空间的切换, 只调整线程上下文。
 KiSwapThred - 切换到链表中的下一个线程(SwapContext)调用

下面给出这些内部函数的伪代码。

```

-----
/***** KeAttachProcess *****/
// just wrapper
//
KeAttachProcess(EPROCESS *Process)
{
    KiAttachProcess(Process, KeRaiseIrqlToSynchLevel);
}
/***** KiAttachProcess *****/

KiAttachProcess(EPROCESS *Process, Irql){

    //CurThread=fs:124h
    //CurProcess=CurThread->ApcState.Process;

    if(CurProcess!=Process){
        if(CurProcess->ApcStateIndex || KPCR->DpcRoutineActive)KeBugCheckEx...
    }
}

```

```

//if we already in process's context
if(CurProcess==Process){KiUnlockDispatcherDatabase(Irql);return;}

Process->StackCount++;
KiMoveApcState(&CurThread->ApcState,&CurThread->SavedApcState);

// init lists
CurThread->ApcState.ApcListHead[0].Blink=&CurThread->ApcState.ApcListHead[0];
CurThread->ApcState.ApcListHead[0].Flink=&CurThread->ApcState.ApcListHead[0];
CurThread->ApcState.ApcListHead[1].Blink=&CurThread->ApcState.ApcListHead[1];
CurThread->ApcState.ApcListHead[1].Flink=&CurThread->ApcState.ApcListHead[1];

//fill curtheads's fields
CurThread->ApcState.Process=Process;

CurThread->ApcState.KernelApcInProgress=0;
CurThread->ApcState.KernelApcPending=0;
CurThread->ApcState.UserApcPending=0;

CurThread->ApcState.ApcStatePointer.SavedApcState=&CurThread->SavedApcState;
CurThread->ApcState.ApcStatePointer.ApcState=&CurThread->ApcState;

CurThread->ApcStateIndex=1;

//if process ready, just swap it...
if(!Process->State)//state==0, ready
{
    KiSwapProcess(Process,CurThread->SavedApcState.Process);
    KiUnlockDispatcherDatabase(Irql);
    return;
}

CurThread->State=1; //ready?
CurThread->ProcessReadyQueue=1;

//put Process in Thread's waitlist
CurThread->WaitListEntry.Flink=&Process->ReadyListHead.Flink;
CurThread->WaitListEntry.Blink=Process->ReadyListHead.Blink;

Process->ReadyListHead.Flink->Flink=&CurThread->WaitListEntry.Flink;
Process->ReadyListHead.Blink=&CurThread->WaitListEntry.Flink;

// else, move process to swap list and wait
if(Process->State==1){//idle?
    Process->State=2; //trans
    Process->SwapListEntry.Flink=&KiProcessInSwapListHead.Flink;
    Process->SwapListEntry.Blink=KiProcessInSwapListHead.Blink;
    KiProcessInSwapListHead.Blink=&Process->SwapListEntry.Flink;
    KiSwapEvent.Header.SignalState=1;
    if(KiSwapEvent.Header.WaitListHead.Flink!=&KiSwapEvent.Header.WaitListHead.
Flink)
        KiWaitTest(&KiSwapEvent,0xa); //fastcall
}

CurThread->WaitIrql=Irql;

```

```
KiSwapThread();
return;
}
```

从这个函数可以得到以下结论。进程可以处于以下状态——0(准备), 1(Idle), 2(Trans——切换)。这证实了高层次的信息。KiAttachProcess使用了另外两个函数KiSwapProcess和KiSwapThread。

```
/****** KiSwapProcess *****/
```

```
KiSwapProcess(EPROCESS* NewProcess, EPROCESS* OldProcess)
{
// just reload cr3 and small work with TSS

// TSS=KPCR->TSS;
// xor eax,eax
// mov gs,ax
TSS->CR3=NewProcess->DirectoryTableBase;//0x1c
// mov cr3,NewProcess->DirectoryTableBase
TSS->IopmOffset=NewProcess->IopmOffset;//0x66
if(WORD(NewProcess->LdtDescriptor)==0){ltdt 0x00; return;}
//GDT=KPCR->GDT;
(QWORD)GDT->0x48=(QWORD)NewProcess->LdtDescriptor;
(QWORD)GDT->0x108=(QWORD)NewProcess->Int21Descriptor;
ltdt 0x48;
return;
}
```

切换进程上下文。正如我所料, 这个函数只是重新加载CR3寄存器, 再加上一点相关的操作。例如, 用IopmOffset域的值建立TSS中的I/O位图的偏移。还必需将选择子的值加载到ldt(只用于VDM session)。

```
/****** SwapContext *****/
```

```
SwapContext(NextThread, CurThread, WaitIrql)
{

NextThread.State=ThreadStateRunning; //2
KPCR.DebugActive=NextThread.DebugActive;

cli();

//Save Stack
CurThread.KernelStack=esp;

//Set stack
KPCR.StackLimit=NextThread.StackLimit;
KPCR.StackBase=NextThread.InitialStack;

tmp=NextThread.InitialStack-0x70;
newcr0=cr0&0xfffffff1|NextThread.NpxState|(tmp+0x6c);
if(newcr0!=cr0)reloadcr0();
if(!*(tmp-0x1c)&0x20000)tmp-=0x10;
TSS=KPCB.TSS;
TSS->ESP0=tmp;
```

```

//set pTeb
KPCB.Self=NextThread.pTeb;
esp=NextThread.KernelStack;
sti();

//correct GDT
GDT=KPCB.GDT;
WORD(GDT->0x3a)=NextThread.pTeb;
BYTE(GDT->0x3c)=NextThread.pTeb>>16;
BYTE(GDT->0x3f)=NextThread.pTeb>>24;

//if we must swap processes, do it (like KiSwapProcess)

if(CurThread.ApcState.Process!=NextThread.ApcState.Process)
{
    //***** like KiSwapProcess
}

NextThread->ContextSwitches++;

KPCB->KeContextSwitches++;

if(!NextThread->ApcState.KernelApcPending)return 0;

//popf;
// jnz HalRequestSoftwareInterrupt// return 0

return 1;
}

```

切换堆栈, 修正GDT, 以使FS寄存器指向TEB。如果线程属于当前进程, 则不进行上下文切换。否则, 进行的操作和KiSwapProcess中的大致差不多。

为了一致, 我给出KeDetachProcess的原型。

```
KeDetachProcess(EPROCESS *Process,Irql);
```

我们看到——这些函数的伪码实际上完全描述出了操作系统的上下文切换。总的说来, 代码分析表明, 理解OS的主要途径就是要知道它的内部结构。

0a.某些未公开的内存管理器函数

=====

SP3的ntoskrnl.exe的内存管理器导出了以下符号:

```

467 1D0 00051080 MmAdjustWorkingSetSize
468 1D1 0001EDFA+MmAllocateContiguousMemory
469 1D2 00051A14+MmAllocateNonCachedMemory
470 1D3 0001EAE8+MmBuildMdlForNonPagedPool
471 1D4 000206BC MmCanFileBeTruncated
472 1D5 0001EF5A+MmCreateMdl
473 1D6 0002095C MmCreateSection
474 1D7 00021224 MmDbgTranslatePhysicalAddress

```

```

475 1D8 000224AC MmDisableModifiedWriteOfSection
476 1D9 000230C8 MmFlushImageSection
477 1DA 0001FA9C MmForceSectionClosed
478 1DB 0001EEA0+MmFreeContiguousMemory
479 1DC 00051AFE+MmFreeNonCachedMemory
480 1DD 0001EEAC+MmGetPhysicalAddress
481 1DE 00024028 MmGrowKernelStack
482 1DF 0004E144 MmHighestUserAddress
483 1E0 0002645A+MmIsAddressValid
484 1E1 00026CD8+MmIsNonPagedSystemAddressValid
485 1E2 0001F5D8 MmIsRecursiveIoFault
486 1E3 00026D56+MmIsThisAnNtAsSystem
487 1E4 000766C8+MmLockPagableDataSection
488 1E5 000766C8 MmLockPagableImageSection
489 1E6 0001F160+MmLockPagableSectionByHandle
490 1E7 0001ED18+MmMapIoSpace
491 1E8 0001EB74+MmMapLockedPages
492 1E9 0001F5F6 MmMapMemoryDumpMdl
493 1EA 00076A14 MmMapVideoDisplay
494 1EB 0005206C MmMapViewInSystemSpace
495 1EC 00079B0E MmMapViewOfSection
496 1ED 0007A7EE+MmPageEntireDriver
497 1EE 0001E758+MmProbeAndLockPages
498 1EF 00026D50+MmQuerySystemSize
499 1F0 00052A8A+MmResetDriverPaging
500 1F1 0004E0A4 MmSectionObjectType
501 1F2 00079D28 MmSecureVirtualMemory
502 1F3 0001EFCE MmSetAddressRangeModified
503 1F4 0007684E MmSetBankedSection
504 1F5 0001EF2C+MmSizeOfMdl
505 1F6 0004E0A0 MmSystemRangeStart
506 1F7 0001F516+MmUnlockPagableImageSection
507 1F8 0001EA16+MmUnlockPages
508 1F9 0007669A+MmUnmapIoSpace
509 1FA 0001ECA8+MmUnmapLockedPages
510 1FB 00076A2E MmUnmapVideoDisplay
511 1FC 00052284 MmUnMapViewInSystemSpace
512 1FD 0007AFE4 MmUnMapViewOfSection
513 1FE 0007A00A MmUnsecureVirtualMemory
514 1FF 0004DDCC MmUserProbeAddress

```

这里的符号 '+' 表示函数在 DDK 中有记载。我这里给出某些未公开函数的原型。

// 调整 working set 的大小.

```

NTOSKRNL NTSTATUS MmAdjustWorkingSetSize(
    DWORD MinimumWorkingSet OPTIONAL, // if both == -1
    DWORD MaximumWorkingSet OPTIONAL, // empty working set
    PVM Vm OPTIONAL);

```

// can file be truncated???

```

NTOSKRNL BOOLEAN MmCanFileBeTruncated(
    PSECTION_OBJECT_POINTERS SectionPointer, // see FILE_OBJECT
    PLARGE_INTEGER NewFileSize
);

```

// create section. NtCreateSection call this function...

```
NTOSKRNL NTSTATUS MmCreateSection (
    OUT PVOID          *SectionObject,
    IN ACCESS_MASK      DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER    MaximumSize,
    IN ULONG             SectionPageProtection,//PAGE_XXXX
    IN ULONG             AllocationAttributes,//SEC_XXX
    IN HANDLE            FileHandle OPTIONAL,
    IN PFILE_OBJECT      File OPTIONAL
);
```

```
typedef enum _MMFLUSH_TYPE {
    MmFlushForDelete,
    MmFlushForWrite
} MMFLUSH_TYPE;
```

```
NTOSKRNL BOOLEAN MmFlushImageSection (
    IN PSECTION_OBJECT_POINTERS SectionObjectPointer,
    IN MMFLUSH_TYPE              FlushType
);
```

NTOSKRNL DWORD MmHighestUserAddress; // 一般为0x7ffeffff

NTOSKRNL BOOLEAN MmIsRecursiveIoFault();

//其代码

```
#define _MmIsRecursiveIoFault() ( \
    (PsGetCurrentThread()->DisablePageFaultClustering) | \
    (PsGetCurrentThread()->ForwardClusterOnly) \
)
```

NTOSKRNL POBJECT_TYPE MmSectionObjectType; //标准的Section对象

NTOSKRNL DWORD MmSystemRangeStart; //一般为0x80000000

NTOSKRNL DWORD MmUserProbeAddress; //一般为0x7fff0000

```
NTOSKRNL PVOID MmMapVideoDisplay( // для i386 wrapper в MmMapIoSpace
    IN PHYSICAL_ADDRESS PhysicalAddress,
    IN ULONG NumberOfBytes,
    IN BOOLEAN CacheEnable
);
```

```
NTOSKRNL VOID MmUnmapVideoDisplay ( // для i386 wrapper в MmUnmapIoSpace
    IN PVOID BaseAddress,
    IN ULONG NumberOfBytes
);
```

// 将frame的范围标记为更改并进行相应的操作

```
NTOSKRNL VOID MmSetAddressRangeModified(
    PVOID StartAddress,
    DWORD Length
);
```

```

);

// 在NtMapViewOfSection中调用
typedef enum _SECTION_INHERIT {
ViewShare=1;
ViewUnmap=2;
}SECTION_INHERIT;

NTOSKRNL NTSTATUS MmMapViewOfSection(
    PVOID pSection,
    PEPROCESS pProcess,
    OUT PVOID *BaseAddress,
    DWORD ZeroBits,
    DWORD CommitSize,
    OUT PLARGE_INTEGER SectionOffset OPTIONAL,
    OUT PDWORD ViewSize,
    SECTION_INHERIT InheritDisposition,
    DWORD AllocationType,
    DWORD ProtectionType
);

NTOSKRNL NTSTATUS MmUnmapViewOfSection(
    PEPROCESS Process,
    PVOID Address
);

PVOID MmLockPagableImageSection(
    PVOID AddressWithinImageSection // same entry as MmLockPagableDataSection
);

// 减少StackLimit(堆栈增长)
NTSTATUS MmGrowKernelStack(
    PVOID CurESP //栈顶的地址
);

```

I talk to the wind
 My words are all carried away
 I talk to the wind
 The wind does not hear
 The wind cannot hear.

King Crimson'69 -I Talk to the Wind

0b.结语

=====

就到这里吧。如果综合的来看所有这些描述,对内存管理器多少可以得到一些概念。遗憾的是,这些东西还远不能称之为完整。内存管理器,大概是最复杂和最重要的内核组件,对其进行完整的描述,我还得深挖不止十个八个的函数。但是主要的基本的东西我这里都写到了。对于进一步反汇编内核来说,这些应该是很有帮助的吧,谁知道呢... ;)

Best Regards, Peter Kosy aka Gloomy.
 Melancholy Coding '2001.

mailto:gl00my@mail.ru

P.S. 我知道我的“大作”不可避免的会有错误。我将非常高兴的听取批评和建议。

附录

0c.某些未公开的系统调用

=====

这里我描述了一些有用的Zw/Nt函数, 这些函数可以在USER模式下或是驱动程序中调用(Zw类的)。几乎所有这些函数都来自于

Коберниченко的“Недокументированные возможности Windows NT”一书。再加上Working Set结构体的值, 就可以描述用于NtQueryVirtualMemory的MEMORY_WORKING_SET_INFORMATION。

```
NTSYSAPI NTSTATUS NTAPI NtAllocateVirtualMemory(
    HANDLE Process,
    OUT PVOID *BaseAddr,
    DWORD ZeroBits,
    OUT PDWORD RegionSize,
    DWORD AllocationType, // MEM_RESERVE|MEM_COMMIT|MEM_TOP_D
    OWN
    DWORD Protect); // PAGE_XXXX...
```

```
NTSYSAPI NTSTATUS NTAPI NtFreeVirtualMemory(
    HANDLE Process,
    OUT PVOID* BaseAddr,
    OUT PULONG RegionSize,
    DWORD FreeType //MEM_DECOMMIT|MEM_RELEASE
);
```

```
NTSYSAPI NTSTATUS NTAPI NtCreateSection(
    OUT PHANDLE Section,
    ACCESS_MASK DesiredAccess, //SECTION_MAP_XXX...
    POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    PLARGE_INTEGER MaximumSize OPTIONAL,
    DWORD SectionPageProtection, //PAGE_...
    DWORD AllocationAttributes, //SEC_XXX
    HANDLE FileHandle OPTIONAL // NULL - pagefile
);
```

```
typedef enum _SECTION_INHERIT {
    ViewShare=1;
    ViewUnmap=2;
}SECTION_INHERIT;
```

```
NTSYSAPI NTSTATUS NTAPI NtMapViewOfSection(
    HANDLE Section,
    HANDLE Process,
    OUT PVOID *BaseAddress,
    DWORD ZeroBits,
    DWORD CommitSize,
```

```

    OUT PLARGE_INTEGER SectionOffset OPTIONAL,
    OUT PDWORD ViewSize,
    SECTION_INHERIT InheritDisposition,
    DWORD AllocationType,
    //MEM_TOP_DOWN, MEM_LARGE_BAGE, MEM_AUTO_ALIGN=0x400000
    00
    DWORD ProtectionType // PAGE_...
);

```

```

#define UNLOCK_TYPE_NON_PRIVILEGED 0x00000001L
#define UNLOCK_TYPE_PRIVILEGED    0x00000002L

```

```

NTSYSAPI NTSTATUS NTAPI NtLockVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *RegionAddress,
    IN OUT PULONG RegionSize,
    IN ULONG UnlockTypeRequired
);

```

```

NTSYSAPI NTSTATUS NTAPI NtUnlockVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *RegionAddress,
    IN OUT PULONG RegionSize,
    IN ULONG UnlockTypeRequested
);

```

```

NTSYSAPI NTSTATUS NTAPI NtReadVirtualMemory(
    IN HANDLE ProcessHandle,
    IN PVOID StartAddress,
    OUT PVOID Buffer,
    IN ULONG BytesToRead,
    OUT PULONG BytesReaded OPTIONAL
);

```

```

NTSYSAPI NTSTATUS NTAPI NtWriteVirtualMemory(
    IN HANDLE ProcessHandle,
    IN PVOID StartAddress,
    IN PVOID Buffer,
    IN ULONG BytesToWrite,
    OUT PULONG BytesWritten OPTIONAL
);

```

```

NTSYSAPI NTSTATUS NTAPI NtProtectVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *RegionAddress,
    IN OUT PULONG RegionSize,
    IN ULONG DesiredProtection,
    OUT PULONG OldProtection
);

```

```

NTSYSAPI NTSTATUS NTAPI NtFlushVirtualMemory(
    IN HANDLE ProcessHandle,
    IN PVOID* StartAddress,
    IN PULONG BytesToFlush,
    OUT PIO_STATUS_BLOCK StatusBlock
);

```

```
);

typedef enum _MEMORYINFOCLASS {
    MemoryBasicInformation,
    MemoryWorkingSetInformation,

    // 还有class 2 - 这是VAD中的信息, 我目前还不完全了解VAD结构体, 也就不能写出
    // 相应的INFO结构。

```

```
} MEMORYINFOCLASS;
```

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    ULONG AllocationProtect;
    ULONG RegionSize;
    ULONG State;
    ULONG Protect;
    ULONG Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

```
#define WSFRAMEINFO_SHARED_FRAME 0x100
#define WSFRAMEINFO_INTERNAL_USE 0x4
#define WSFRAMEINFO_UNKNOWN 0x3
```

```
typedef struct _MEMORY_WORKING_SET_INFORMATION {
    ULONG SizeOfWorkingSet;
    DWORD WsEntries[ANYSIZE_ARRAY]; // is Page VA | WSFRAMEINFO...
```

```
} MEMORY_ENTRY_INFORMATION, *PMEMORY_ENTRY_INFORMATION;
```

```
NTSYSAPI NTSTATUS NTAPI NtQueryVirtualMemory(
    IN HANDLE ProcessHandle,
    IN PVOID RegionAddress,
    IN MEMORYINFOCLASS MemoryInformationClass,
    IN PVOID VirtualMemoryInfo,
    IN ULONG Length,
    OUT PULONG ActualLength OPTIONAL
);
```

0d.附注及代码分析草稿

```
=====
```

```
**** K MmCreateProcessAddressSpace ... ****
```

```
=====
```

```
__fastcall MiTotalCommitLimit(PVOID pProcess, DWORD NumOfPages); // edx:ecx
有statistic
```

```
dd MmTotalCommitLimit
dd MmTotalCommittedPages
```

如果NumOfPages+MmTotalCommittedPages不超过Limit - 一切OK, 并只是简单的修正statistic.

否则开始线程间的协作。

选择time out值(如果请求 ≥ 10 页, 则为20秒), 否则为-1秒。接着填充某个结构体, 大概是这个样子:

```
typedef struct _REQUEST_FOR_COMMITTED_MEMORY{
    LIST_ENTRY ListEntry;
    DWORD PagesToCommit;
    DWORD Result;
    KSEMAPHORE Semaphore;
} _REQUEST_FOR_COMMITTED_MEMORY;
```

这个结构体(或链表的元素)被插入到全局结构体中的全局链表ListofRequest:

[Pre List Item]<->[Our List Item]<->[ListofRequest]

```
typedef struct _COMMIT_MEMORY_REQUEST_LIST{
    KSEMAPHORE CommitMemorySemaphore;
    LIST_ENTRY ListofRequest;
} COMMIT_MEMORY_REQUEST_LIST;
```

之后对CommitMemorySemaphore使用KeReleaseSemaphore并等待REQUEST_FOR_COMMITTED_MEMORY中带有time out的信号量。

如果未超出time out并因此Result不为0, 则再校验一次Limit并输出OK(如果limit有问题——则所有都重新开始)。如果结果为0, MiCouseOverCommitPopup。如果发生了time out, 分析如下:

如果ListofReques.Flink==&ListofReques.Flink, 也就是说所有的请求都在队列的尾部, 则再一次等待信号量——并且已经没有time out了, 因为不是我们的问题;)

如果ListofReques.Flink==&RequestForCommittedMemory.ListEntry, 就是说队列中的下一个是我们的请求(???)。则从队列中收回请求, 因为是从我们这里来的。

现在来看我们想看的几个页。如果 ≥ 10 则MiCouseOverCommitPopup, 否则MiChargeCommitmentCantExpand, 之后输出。

所有的操作都需要cli sti, 同时使用FastMutex(进程的10ch偏移), 在进程创建时调用这个函数不会进行此操作。

现在, MiCouseOverCommitPopup(PagesNum,CommitTotalLimitDelta);又做些什么呢——如果我们想要页数大于128——则ExRaiseStatus(STATUS_COMMITMENT_LIMIT);如果小于则IoRaiseInformationalHardError(STATUS_COMMITMENT_LIMIT,0,0);(这些函数都是公开的)。如果成功调用最后一个函数——则累加statistic:

```
MiOverCommitCallCount++;
MmTotalCommitLimit+=CommitTotalLimitDelta;
MmExtendedCommit+=CommitTotalLimitDelta;
MmTotalCommittedPages+=PagesNum;
且不修正 MmPeakCommintment;
```

如果不成功但MiOverCommitCallCount==0, 所有都等于statistic, 否则ExRaiseStatus(STATUS_COMMITMENT_LIMIT);

辅助函数:

DWORD NTOSKRNL RtlRandom(PDWORD Seed);

不奇怪, 这个函数没有公开。该函数使用一个128个DWORD的表。在操作之后被此表和Seed被修正。可以看到, 这给出了最大周期。

如果有两个event
MmAvailablePagesEventHigh 和
MmAvailablePagesEventHigh.

MiSectionInitialization:
MmDereferenceSegmentHeader: это структура описанная выше
с добавленным
spinlock сверху.
创建线程MiDereferenceSegmentThread

PsChargePoolQuota(PVOID Process,DWORD Type(NP/P),DWORD Charge);

[TO DO] --> MmInfoCounters!!!! 使用相应的NtQueryInfo...可以获得非常多有用的信息