

# 1 内存管理

Windows的内存管理是很复杂的，我们没必要精确到每一行代码的学习，所以我们以**线性地址的管理**作为突破口，重点学习线性地址是如何管理的，物理地址、物理页是如何管理的，以及了解我们平时所听到的缺页解决了什么问题，堆、栈内存申请等内容。这样，我们才能对Windows的内存管理有一个清晰的认识。

## 1.1 线性地址的管理

### 1.1.1 进程空间的地址划分

每一个进程都有自己的4GB线性地址空间，这是我们之前的学习中反复提到的，这4GB的线性地址空间的划分大致如下图所示。虽然每个线程名义上有4GB的内存空间，但实际上只有低2G的内存空间可以使用，高2G的内存空间是共用的。除此之外，我们可以根据下图知道，**4GB空间中只有用户模式区和内核区可以访问使用，而空指针赋值区和64KB禁入区是不允许的**。无论是用户空间还是内核空间，肯定都是要记录线性地址的分配情况的，这样才不会出现重复分配的情况。

分区	x86 32位Windows
空指针赋值区	0x00000000 - 0x0000FFFF
用户模式区	0x00010000 - 0x7FFEFFFF
64KB禁入区	0x7FFF0000 - 0x7FFFFFFF
内核	0x80000000 - 0xFFFFFFFF

在内核中，它会通过一个链表将线性地址链起来，这样要想再分配内核空间的线性地址，直接通过这个链表查询线性地址的属性即可找到未分配的线性地址进行申请，这里不过多赘述，可以自行学习。**我们现在主要还是了解用户空间的线性地址的管理，也就是进程的低2G内存空间，这是因为不同进程的高2G地址往往是相同的，因此高2G的地址变化较少**，使用链表的方式足够满足需求，而用户空间的地址管理就较为复杂。

### 1.1.2 用户空间线性地址的管理

在Windows下的用户空间线性地址是通过“搜索二叉树”的方式进行管理。在这里我们使用Windbg来看一下具体是如何管理的。

我们可以通过指令：`!process 0 0`，随便找一个进程，查看它的进程结构体\_EPROCESS，在该结构体的0x11c偏移成员VadRoot（**Vad, Virtual Address Descriptor, 虚拟地址描述符**），这个成员记录当前进程线性地址空间的搜索二叉树（**我们可以称之为Vad树**），其对应的地址是二叉树根节点的地址。（它里面的每一个节点都记录了一块被占用的线性地址空间。）

```

kd> dt _EPROCESS 894f3020
nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x06c ProcessLock : _EX_PUSH_LOCK
+0x070 CreateTime : _LARGE_INTEGER 0x01d9e231`136bd850
+0x078 ExitTime : _LARGE_INTEGER 0x0
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : 0x000006ac Void
+0x088 ActiveProcessLinks : _LIST_ENTRY [ 0x805637b8 - 0x894fbe28 ]
+0x090 QuotaUsage : [3] 0x1688
+0x09c QuotaPeak : [3] 0x1788
+0x0a8 CommitCharge : 0x131
+0x0ac PeakVirtualSize : 0x233e000
+0x0b0 VirtualSize : 0x233e000
+0x0b4 SessionProcessLinks : _LIST_ENTRY [ 0xbadd0014 - 0x894fbe54 ]
+0x0bc DebugPort : (null)
+0x0c0 ExceptionPort : 0xe1372030 Void
+0x0c4 ObjectTable : 0xe1903d68 _HANDLE_TABLE
+0x0c8 Token : _EX_FAST_REF
+0x0cc WorkingSetLock : _FAST_MUTEX
+0x0ec WorkingSetPage : 0x196e1
+0x0f0 AddressCreationLock : _FAST_MUTEX
+0x110 HyperSpaceLock : 0
+0x114 ForkInProgress : (null)
+0x118 HardwareTrigger : 0
+0x11c VadRoot : 0x89cc74b0 Void

```

## \_MMVAD

VadRoot对应的类型实际上就是\_MMVAD结构体，也就表示该结构体为搜索二叉树结点的数据类型，我们可以通过该数据类型对这个节点所对应的线性地址区域有个整体上的认识：

```

kd> dt _MMVAD 0x89cc74b0
nt!_MMVAD
+0x000 StartingVpn : 0x190
+0x004 EndingVpn : 0x19f
+0x008 Parent : (null)
+0x00c LeftChild : 0x89aef698 _MMVAD
+0x010 RightChild : 0x894f4908 _MMVAD
+0x014 u : __unnamed
+0x018 ControlArea : 0x1a070004 _CONTROL_AREA
+0x01c FirstPrototypePte : 0x89ad8498 _MMPTE
+0x020 LastContiguousPte : 0x00300012 _MMPTE
+0x024 u2 : __unnamed

```

该结构体中每个成员的含义如下：

1. **StartingVpn**：当前节点对应的内存的线性地址起始位置（以页为单位，即4KB），因此本例中实际上对应的起始位置为0x190000。
2. **EndingVpn**：当前节点对应的内存的线程地址结束位置（以页为单位，即4KB），因此本例中实际上对应的结束位置为0x19f000。
3. **Parent**：父节点地址，本例中为根节点，没有父节点，所以为空。
4. **LeftChild**：左子树地址。
5. **RightChild**：右子树地址。

6. **u**：用于表示内存属性。
7. **ControlArea**：控制区域。

在知道左子树或右子树地址后，就可以通过指令：dt \_MMVAD 地址，来一层一层找到所有的节点：

```
kd> dt _MMVAD 0x89cc74b0
nt!_MMVAD
+0x000 StartingVpn      : 0x190
+0x004 EndingVpn       : 0x19f
+0x008 Parent          : (null)
+0x00c LeftChild       : 0x89aef698 _MMVAD
+0x010 RightChild      : 0x894f4908 _MMVAD
+0x014 u               : __unnamed
+0x018 ControlArea     : 0x1a070004 _CONTROL_AREA
+0x01c FirstPrototypePte : 0x89ad8498 _MMPTE
+0x020 LastContiguousPte : 0x00300012 _MMPTE
+0x024 u2              : __unnamed

kd> dt _MMVAD 0x89aef698
nt!_MMVAD
+0x000 StartingVpn      : 0x20
+0x004 EndingVpn       : 0x20
+0x008 Parent          : 0x89cc74b0 _MMVAD
+0x00c LeftChild       : 0x89aee718 _MMVAD
+0x010 RightChild      : 0x89b17bd0 _MMVAD
+0x014 u               : __unnamed
+0x018 ControlArea     : 0x0a060004 _CONTROL_AREA
+0x01c FirstPrototypePte : 0xee657645 _MMPTE
+0x020 LastContiguousPte : 0x00000002 _MMPTE
+0x024 u2              : __unnamed
```

根据列出的二叉树结构体，我们知道成员StartingVpn、EndingVpn用于表达线性地址区间，因此我们可以通过遍历所有子树的这两个成员，只要不在线性地址区间内的线性地址就可以被我们使用的。

Windbg提供了一个更快捷的指令：!vad 根节点地址，该指令可以列出所在进程内线性地址的记录情况（包含了结构体地址、节点层级、线性地址区间、内存类型、内存属性等等）：

```
kd> !vad 0x89cc74b0
VAD   Level   Start      End Commit
89aee718 2       10        10      1 Private  READWRITE
89aef698 1       20        20      1 Private  READWRITE
89d6d208 4       30        3f      9 Private  READWRITE
89be14b8 3       40        7f      4 Private  READWRITE
89b17bd0 2       80        82      0 Mapped   READONLY   Pagefile section, shared commit 0x3
89a1bd20 3       90        18f     36 Private  READWRITE
89cc74b0 0      190        19f      6 Private  READWRITE
894f4ef8 4      1a0        1af      0 Mapped   READWRITE   Pagefile section, shared commit 0x3
89b0dd58 3      1b0        1c5      0 Mapped   READONLY    \WINDOWS\system32\unicode.nls
894f4580 4      1d0        20c      0 Mapped   READONLY    \WINDOWS\system32\locale.nls
894f3d60 2      210        250      0 Mapped   READONLY    \WINDOWS\system32\sortkey.nls
894f3d30 5      260        265      0 Mapped   READONLY    \WINDOWS\system32\sorttbls.nls
894f3d00 4      270        2b0      0 Mapped   READONLY    Pagefile section, shared commit 0x41
894f3a60 5      2c0        387      0 Mapped   EXECUTE_READ Pagefile section, shared commit 0x4
89ae31c8 6      390        39f      8 Private  READWRITE
894f4f58 7      3a0        3a0      1 Private  READWRITE
```

## ControlArea

那么我们要了解线性地址到底是被谁占用的，就可以来看一下成员ControlArea，它同样也是一个结构体：\_CONTROL\_AREA。

我们需要关注的是FilePointer成员，当这个值为空的时候，这块内存就是Private类型，也就是进程自己申请的内存。

```
kd> dt _MMVAD 0x81f79a30
nt!_MMVAD
+0x000 StartingVpn      : 0xab0
+0x004 EndingVpn       : 0xab0
+0x008 Parent          : (null)
+0x00c LeftChild       : 0x820de628 _MMVAD
+0x010 RightChild      : 0x820e10d8 _MMVAD
+0x014 u               : unnamed
+0x018 ControlArea     : 0x81fb9b00 _CONTROL_AREA
+0x01c FirstPrototypePte : 0xe1844358 _MMPTE
+0x020 LastContiguousPte : 0xe1844358 _MMPTE
+0x024 u2              : unnamed

kd> dt _CONTROL_AREA 0x81fb9b00
nt!_CONTROL_AREA
+0x000 Segment         : 0xe1844318 _SEGMENT
+0x004 DereferenceList : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x00c NumberOfSectionReferences : 1
+0x010 NumberOfPfnReferences : 0
+0x014 NumberOfMappedViews : 1
+0x018 NumberOfSubsections : 1
+0x01a FlushInProgressCount : 0
+0x01c NumberOfUserReferences : 2
+0x020 u               : unnamed
+0x024 FilePointer     : (null)
+0x028 WaitingForDeletion : (null)
+0x02c ModifiedWriteCount : 0
+0x02e NumberOfSystemCacheViews : 0
```

当该值不为空的情况下，则表示当前内存为Mapped类型，即映射了其他类型的文件（DLL、EXE等）在内存中，我们就可以跟进对应的结构体\_FILE\_OBJECT，通过其成员FileName知道映射文件的描述信息。

```

kd> dt _CONTROL_AREA 0x81f967a0
nt!_CONTROL_AREA
+0x000 Segment : 0xe16c52f8 _SEGMENT
+0x004 DereferenceList : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x00c NumberOfSectionReferences : 1
+0x010 NumberOfPfnReferences : 0x78
+0x014 NumberOfMappedViews : 0x15
+0x018 NumberOfSubsections : 5
+0x01a FlushInProgressCount : 0
+0x01c NumberOfUserReferences : 0x16
+0x020 u : unnamed
+0x024 FilePointer : 0x820b03a8 _FILE_OBJECT
+0x028 WaitingForDeletion : (null)
+0x02c ModifiedWriteCount : 0
+0x02e NumberOfSystemCacheViews : 0
kd> dt _FILE_OBJECT 0x820b03a8
nt!_FILE_OBJECT
+0x000 Type : 0n5
+0x002 Size : 0n112
+0x004 DeviceObject : 0x81e222f8 _DEVICE_OBJECT
+0x008 Vpb : 0x82077fa8 _VPB
+0x00c FsContext : 0xe158ad90 Void
+0x010 FsContext2 : 0xe144e498 Void
+0x014 SectionObjectPointer : 0x820fb2c4 _SECTION_OBJECT_POINTERS
+0x018 PrivateCacheMap : (null)
+0x01c FinalStatus : 0n0
+0x020 RelatedFileObject : (null)
+0x024 LockOperation : 0 ''
+0x025 DeletePending : 0 ''
+0x026 ReadAccess : 0x1 ''
+0x027 WriteAccess : 0 ''
+0x028 DeleteAccess : 0 ''
+0x029 SharedRead : 0x1 ''
+0x02a SharedWrite : 0x1 ''
+0x02b SharedDelete : 0x1 ''
+0x02c Flags : 0x44040
+0x030 FileName : _UNICODE_STRING "\WINDOWS\system32\kernel32.dll"
+0x038 CurrentByteOffset : _LARGE_INTEGER 0x0
+0x040 Waiters : 0
+0x044 Busy : 0
+0x048 LastLock : (null)
+0x04c Lock : _KEVENT
+0x05c Event : _KEVENT
+0x06c CompletionContext : (null)

```

## \_MMVAD\_FLAGS

在\_MMVAD结构体中有一个成员u，其指向的是如下所示的一个union共同体结构：

```

1 union {
2     ULONG_PTR LongFlags;
3     MMVAD_FLAGS VadFlags;
4 } u;

```

虽然有两个成员，但是一般情况下只使用VadFlags成员，该成员也是一个结构体\_MMVAD\_FLAGS：

```
kd> dt _MMVAD_FLAGS 0x898de410+0x14
nt!_MMVAD_FLAGS
+0x000 CommitCharge      : 0y0000000000000000100 (0x4)
+0x000 PhysicalMapping   : 0y0
+0x000 ImageMap          : 0y1
+0x000 UserPhysicalPages : 0y0
+0x000 NoChange          : 0y0
+0x000 WriteWatch       : 0y0
+0x000 Protection       : 0y00111 (0x7)
+0x000 LargePages       : 0y0
+0x000 MemCommit        : 0y0
+0x000 PrivateMemory    : 0y0
```

这里我们主要了解以下几个重要的成员：

1. **CommitCharge**：最大可提供物理页的数目。
2. **ImageMap**：表示当前是否映射了镜像文件（通常是可执行文件），值为1，则说明映射了，为0，则不是。
  - a. 在指令!vad所展示表中，ImageMap为1的话都会有一个Exe字符串的标识：
 

898de410	1	400	426	4	Mapped	Exe	EXECUTE_WRITECOPY	\Documents and Settings\Administrator\
8988ace0	5	560	65f	8	Private		READWRITE	
899bd3a8	4	76d70	76d91	1	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\apphelp.dll
89ae23e8	5	77bd0	77bd7	1	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\version.dll
89ae2418	6	77da0	77e48	5	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\advapi32.dll
89c37ec0	7	77e50	77ee0	1	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\rpcrt4.dll
898913f0	3	7c800	7c91b	5	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\kernel32.dll
898de3e0	2	7c920	7c9b3	5	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\ntdll.dll
  - b.
1. **Protection**：表示当前对应内存块的属性，取值如下：
  - 1：READONLY：只读
  - 2：EXECUTE：可执行
  - 3：EXECUTE\_READ：可执行、读
  - 4：READWRITE：读、写
  - 5：WRITECOPY：写拷贝
  - 6：EXECUTE\_READWRITE：可执行、读、写
  - 7：EXECUTE\_WRITECOPY：可执行、写拷贝
2. **PrivateMemory**：表示当前是否是Private类型内存，值为1，则说明是，为0，则说明是Mapped类型。

### 1.1.3 Private Memory

从线性地址角度看，内存分为Private和Mapped类型：

1. 通过VirtualAlloc/VirtualAllocEx申请的就是Private类型内存。（独享物理页）
2. 通过CreateFileMapping映射的就是Mapped类型内存。（会出现与其他进程共享物理页）

### VirtualAlloc

我们来实际使用一下VirtualAlloc函数，便于我们更加清晰的了解私有内存：

```
1 LPVOID VirtualAlloc{
2     LPVOID lpAddress,
3     DWORD dwSize,
4     DWORD flAllocationType,
5     DWORD flProtect
6 };
```



VirtualAlloc函数一共有4个参数，其每个参数的含义和作用如下：

1. **lpAddress**：想要申请的起始内存地址，需要结合第二个参数dwSize。如果申请的内存被占用，则无法申请。如果我们手动去填写该值还需要去遍历Vad树才知道什么地址区间可以申请，过程比较麻烦，所以我们可以将这个值填为NULL，系统就会自动去分配一块没被占用的内存空间。
2. **dwSize**：想要申请的内存空间大小，如果lpAddress的值非NULL，那么lpAddress就是地址的起点，lpAddress+dwSize-1就是地址的终点。该值必须是0x1000（4KB，即一个物理页）的整数倍。
3. **flAllocationType**：分配的类型，有两种类型：
  - a. MEM\_COMMIT：创建节点并分配物理页
  - b. MEM\_RESERVE：只创建节点，不分配物理页
4. **flProtect**：保护属性，例如PAGE\_READWRITE、PAGE\_READONLY。

了解完每个参数的意义之后我们可以来编写一段简单的代码，来使用VirtualAlloc：

```

1  #include <windows.h>
2  #include <stdio.h>
3
4  LPVOID lpAddress;
5
6  void main()
7  {
8      printf("Start ...\n");
9      getchar();
10     printf("VirtualAlloc ...\n");
11     lpAddress = ::VirtualAlloc(NULL, 0x1000*2, MEM_COMMIT,
12     PAGE_READWRITE);
13     printf("lpAddress: %x \n", lpAddress);
14     getchar();
15 }
```

将代码进行编译执行，由于使用了getchar函数所以程序会先停止，通过Windbg来看一下Vad树，然后再回到程序，回车一下，再来看Vad树：

```

kd> !vad 0x898b9900
VAD Level Start End Commit
89a2b4d8 1 10 10 1 Private READWRITE
89c16498 2 20 20 1 Private READWRITE
899f8618 3 30 12f 3 Private READWRITE
89891ae8 4 130 132 0 Mapped READONLY Pagefile section, shared commit 0x3
898db770 5 140 23f 3 Private READWRITE
89d82fa8 6 240 24f 6 Private READWRITE
898db7d8 7 250 25f 0 Mapped READWRITE Pagefile section, shared commit 0x3
89b010d8 8 260 275 0 Mapped READONLY \WINDOWS\system32\unicode.nls
89bc6ed8 9 280 2bc 0 Mapped READONLY \WINDOWS\system32\locale.nls
89bd6070 10 2c0 300 0 Mapped READONLY \WINDOWS\system32\sortkey.nls
89c2c6a0 11 310 315 0 Mapped READONLY \WINDOWS\system32\sorttbls.nls
89be23c0 12 320 360 0 Mapped READONLY Pagefile section, shared commit 0x41
89c688b8 13 370 37f 8 Private READWRITE
89c89348 14 380 382 0 Mapped READONLY \WINDOWS\system32\ctype.nls
898b9900 0 400 426 6 Mapped Exe EXECUTE_WRITECOPY \Program Files\Microsoft Visual Studio\MyProjects\VATest\Debug\VATest.exe
899d5c90 3 430 52f 8 Private READWRITE
898d0d60 2 7c800 7c91b 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\kernel32.dll
89d32ba0 1 7c920 7c9b3 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\ntdll.dll
898cc4f8 3 7f6f0 7f7ef 0 Mapped EXECUTE_READ Pagefile section, shared commit 0x7
899e5288 2 7ffa0 7ffd2 0 Mapped READONLY Pagefile section, shared commit 0x33
89d39708 3 7ffd8 7ffd8 1 Private READWRITE
89af50b8 4 7ffdf 7ffdf 1 Private READWRITE

Total VADs: 22, average level: 6, maximum depth: 14
```

在执行完VirtualAlloc后，新分配了一个大小为0x2000的内存空间（0x390000-0x391000），内存属性也对应了VirtualAlloc的参数。

```

kd> !vad 0x898b9900
VAD Level Start End Commit
89891ae8 2 130 132 0 Mapped READONLY Pagefile section, shared commit 0x3
898db770 1 140 23f 3 Private READWRITE
89d82fa8 3 240 24f 6 Private READWRITE
898db7d8 2 250 25f 0 Mapped READWRITE Pagefile section, shared commit 0x3
89b010d8 4 260 275 0 Mapped READONLY \WINDOWS\system32\unicode.nls
89bc6ed8 3 280 2bc 0 Mapped READONLY \WINDOWS\system32\locale.nls
89bd6070 5 2c0 300 0 Mapped READONLY \WINDOWS\system32\sortkey.nls
89c2c6a0 4 310 315 0 Mapped READONLY \WINDOWS\system32\sorttbls.nls
89be23c0 6 320 360 0 Mapped READONLY Pagefile section, shared commit 0x41
89c688b8 5 370 37f 8 Private READWRITE
89c89348 6 380 382 0 Mapped READONLY \WINDOWS\system32\ctype.nls
89c67ac0 7 390 391 2 Private READWRITE
898b9900 0 400 426 6 Mapped Exe EXECUTE_WRITECOPY \Program Files\Microsoft Visual Studio\MyProjects\VATest\Debug\VATest.exe
899d5c90 3 430 52f 8 Private READWRITE
898d0d60 2 7c800 7c91b 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\kernel32.dll
89d32ba0 1 7c920 7c9b3 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\ntdll.dll
898cc4f8 3 7f6f0 7f7ef 0 Mapped EXECUTE_READ Pagefile section, shared commit 0x7
899e5288 2 7ffa0 7ffd2 0 Mapped READONLY Pagefile section, shared commit 0x33
89d39708 3 7ffd8 7ffd8 1 Private READWRITE
89af50b8 4 7ffdf 7ffdf 1 Private READWRITE

Total VADs: 20, average level: 4, maximum depth: 7

```

如上所述，我们就能理清VirtualAlloc函数申请内存的过程了，它会在进程低2G还没有使用的内存空间中，分配一个指定大小的私有内存空间，然后将其对应的\_MMVAD结构体添加到对应进程的Vad树中。

## 堆与栈

我们在之前的初级篇中了解到C语言可以使用malloc、C++语言可以使用new的方式来“申请内存”，为什么在这里我们不以它两为例呢。

我们可以来看一下这两个方法的调用链，也就表示无论是malloc和new，其实都是HeapAlloc：

1	malloc -> _nh_malloc_dbg -> _heap_alloc_dbg -> _heap_alloc_base -> HeapAlloc
2	new -> _nh_malloc -> _nh_malloc_dbg -> _heap_alloc_dbg -> _heap_alloc_base -> HeapAlloc

## HeapAlloc

HeapAlloc是一个在堆中分配内存的函数，但它并不直接申请内存，因为HeapAlloc函数并没有进入到Ring 0，所以它没有权限直接向操作系统申请内存。那么HeapAlloc是如何在堆中分配内存呢？这涉及到对堆的理解。堆实际上是操作系统通过调用VirtualAlloc函数预先分配的一大块内存区域。HeapAlloc的作用是从这个预分配的内存区域中划分出一小部分来使用。

可以做一个比喻来理解，将VirtualAlloc看作是一个批发市场，需要一次性从操作系统那里批量购买内存，而且必须是4KB的整数倍。而HeapAlloc则类似于零售商，从已经批发到批发市场（VirtualAlloc）的货物中购买一小部分。换句话说，HeapAlloc并不是直接向操作系统申请内存，而是在已经预分配的堆内存中分配小块内存供程序使用。这种方式可以提高内存分配的效率，减少频繁调用操作系统的开销。

## 栈

栈其实和堆一样，也是预先分配好的内存，但是栈不需要HeapAlloc这种API来进行分配，可以直接使用，例如我们声明一个局部变量就会使用到栈。

## 实验

我们可以写一段代码，结合Vad树来证实一下我们上面的内容。



```
1 #include <windows.h>
2 #include <stdio.h>
3
4 int a = 0x4321;
5
6 void main()
7 {
8     getchar();
9
10    int x = 0x12345678;
11    int* y = (int*)malloc(sizeof(int)*128);
12
13    printf("Global: %x \n", &a);
14    printf("Stack: %x \n", &x);
15    printf("Heap: %x \n", y);
16
17    getchar();
18 }
```

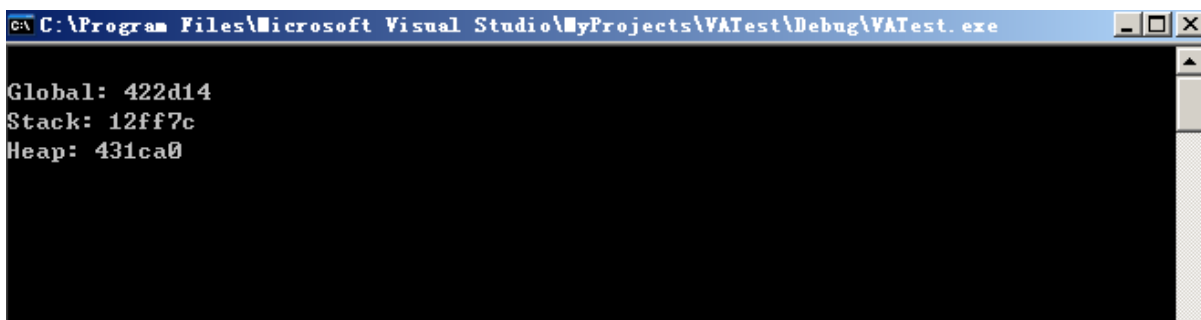
运行程序，在第一个getchar停留，看一下Vad树，接着继续执行：

```

kds lvsd 0x899596d0
VAD Level Start End Commit
89d52ce0 1 10 10 1 Private READWRITE
898d2a68 2 20 20 1 Private READWRITE
8988a050 3 30 12f 3 Private READWRITE
89b18dc0 4 130 132 0 Mapped READONLY Pagefile section, shared commit 0x3
8993e0f0 5 140 23f 3 Private READWRITE
898b6240 6 240 24f 6 Private READWRITE
8990b0f0 7 250 25f 0 Mapped READWRITE Pagefile section, shared commit 0x3
89c81f40 8 260 275 0 Mapped READONLY \WINDOWS\system32\unicode.nls
8986c160 9 280 2bc 0 Mapped READONLY \WINDOWS\system32\locale.nls
89c37d98 10 2c0 300 0 Mapped READONLY \WINDOWS\system32\sortkey.nls
89bc3aa0 11 310 315 0 Mapped READONLY \WINDOWS\system32\sorttbls.nls
8989aac8 12 320 360 0 Mapped READONLY Pagefile section, shared commit 0x41
89d6ae78 13 370 37f 6 Private READWRITE
89bde5a0 14 380 382 0 Mapped READONLY \WINDOWS\system32\ctype.nls
899596d0 0 400 426 6 Mapped Exe EXECUTE_WRITECOPY \Program Files\Microsoft Visual Studio\MyProjects\VATest\Debug\VATest.exe
89d4bce0 3 430 52f 8 Private READWRITE
899b55f8 2 7c800 7c91b 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\kernel32.dll
89cf0e98 1 7c920 7c9b3 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\ntdll.dll
89b01538 3 7f6f0 7ff7ef 0 Mapped EXECUTE_READ Pagefile section, shared commit 0x7
89cf0ec8 2 7ffa0 7fffd2 0 Mapped READONLY Pagefile section, shared commit 0x33
89ad8168 3 7ffdf5 7ffdf5 1 Private READWRITE
89bc1aa8 4 7ffdf5 7ffdf5 1 Private READWRITE

Total VADS: 22, average level: 6, maximum depth: 14

```



执行完成在第二个getchar停留，再来看一下Vad树，我们会发现没有多出子节点，并且无论是全局变量、局部变量、堆，都是在已使用的内存地址区间去申请使用的，全局变量是编译时就会打包在文件里的，所以它的地址就在Mapped类型内存地址区间中（即当前进程的VATest.exe文件）：

```

kd> !vad 0x899596d0
VAD      Level      Start      End Commit
89d52ce0  1          10         10      1 Private  READWRITE
89d2a68  2          20         20      1 Private  READWRITE
8988a050  3          30         12f     3 Private  READWRITE
89b18dc0  4         130         132     0 Mapped   READONLY   Pagefile section, shared commit 0x3
8993e0f0  5         140         23f     3 Private  READWRITE
898b6240  6         240         24f     6 Private  READWRITE
8990b0f0  7         250         25f     0 Mapped   READWRITE   Pagefile section, shared commit 0x3
89c81f40  8         260         275     0 Mapped   READONLY   \WINDOWS\system32\unicode.nls
8986c160  9         280         2bc     0 Mapped   READONLY   \WINDOWS\system32\locale.nls
89c37d98 10         2c0         300     0 Mapped   READONLY   \WINDOWS\system32\sortkey.nls
89bc3aa0 11         310         315     0 Mapped   READONLY   \WINDOWS\system32\sorttbls.nls
8989aac8 12         320         360     0 Mapped   READONLY   Pagefile section, shared commit 0x41
89d6ae78 13         370         37f     8 Private  READWRITE
89bde5a0 14         380         382     0 Mapped   READONLY   \WINDOWS\system32\ctype.nls
899596d0  0         400         426     6 Mapped   Exe EXECUTE_WRITECOPY \Program Files\Microsoft Visual Studio\MyProjects\VATest\Debug\VATest.exe
89d4bce0  3         430         52f     8 Private  READWRITE
899b55f8  2        7c800       7c91b   5 Mapped   Exe EXECUTE_WRITECOPY \WINDOWS\system32\kernel32.dll
89cf0e98  1        7c920       7c9b3   5 Mapped   Exe EXECUTE_WRITECOPY \WINDOWS\system32\ntdll.dll
89b01538  3        7f6f0       7f7ef   0 Mapped   EXECUTE_READ   Pagefile section, shared commit 0x7
89cf0ec8  2        7ffa0       7ffd2   0 Mapped   READONLY   Pagefile section, shared commit 0x33
89ad8168  3        7ffd5       7ffd5   1 Private  READWRITE
89bc1aa8  4        7ffdf       7ffdf   1 Private  READWRITE

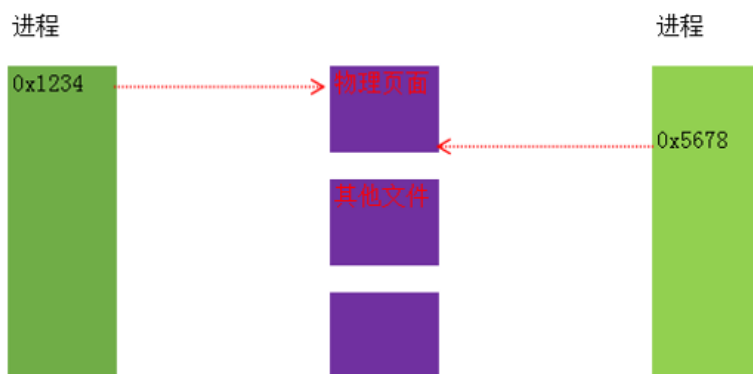
```

Total VADs: 22, average level: 6, maximum depth: 14

综上所述，我们可以知道**VirtualAlloc/VirtualAllocEx**是申请私有内存的唯一方式。

### 1.1.4 Mapped Memory

我们再来看一下Mapped类型内存，其实我们通过查看Vad树就会发现一个进程的Mapped类型内存是多于私有内存的，Mapped类型内存会出现与其他进程共享物理页的情况。



如下图所示，是一个进程的Vad树，有很多Mapped类型内存，有具体的文件时我们可以称之为**共享文件**，没有具体文件则称之为**共享物理页**（即Pagefile section, shared commit xxx部分）。

```

kd> !vad 0x899596d0
VAD Level Start End Commit
89d52ce0 1 10 10 1 Private READWRITE
89d2a68 2 20 20 1 Private READWRITE
8988a050 3 30 12f 3 Private READWRITE
89b18dc0 4 130 132 0 Mapped READONLY Pagefile section, shared commit 0x3
8993e0f0 5 140 23f 3 Private READWRITE
898b6240 6 240 24f 6 Private READWRITE
8990b0f0 7 250 25f 0 Mapped READWRITE Pagefile section, shared commit 0x3
89c81f40 8 260 275 0 Mapped READONLY \WINDOWS\system32\unicode.nls
8986c160 9 280 2bc 0 Mapped READONLY \WINDOWS\system32\locale.nls
89c37d98 10 2c0 300 0 Mapped READONLY \WINDOWS\system32\sortkey.nls
89bc3aa0 11 310 315 0 Mapped READONLY \WINDOWS\system32\sorttbls.nls
8989aac8 12 320 360 0 Mapped READONLY Pagefile section, shared commit 0x41
89d6ae78 13 370 37f 8 Private READWRITE
89bde5a0 14 380 382 0 Mapped READONLY \WINDOWS\system32\ctype.nls
899596d0 0 400 426 6 Mapped Exe EXECUTE_WRITECOPY \Program Files\Microsoft Visual Studio\MyProjects\VA Test\Debug\VA Test.exe
89d4bce0 3 430 52f 8 Private READWRITE
899b55f8 2 7c800 7c91b 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\kernel32.dll
89cf0e98 1 7c920 7c9b3 5 Mapped Exe EXECUTE_WRITECOPY \WINDOWS\system32\ntdll.dll
89b01538 3 7f6f0 7f7ef 0 Mapped EXECUTE_READ Pagefile section, shared commit 0x7
89cf0ec8 2 7ffa0 7ffd2 0 Mapped READONLY Pagefile section, shared commit 0x33
89ad8168 3 7ffd5 7ffd5 1 Private READWRITE
89bc1aa8 4 7ffdf 7ffdf 1 Private READWRITE

Total VADs: 22, average level: 6, maximum depth: 14

```

## 共享内存函数

我们首先了解一下共享内存所需的三个函数：CreateFileMapping、OpenFileMapping、MapViewOfFile。

### CreateFileMapping

CreateFileMapping的作用是**创建一个共享的物理页/文件，执行成功返回映射对象句柄**，语法格式：

```

1 HANDLE CreateFileMapping(
2     HANDLE hFile,
3     LPSECURITY_ATTRIBUTES lpAttributes,
4     DWORD flProtect,
5     DWORD dwMaximumSizeHigh,
6     DWORD dwMaximumSizeLow,
7     LPCTSTR lpName
8 );

```

参数含义如下：

1. **hFile**：指定要映射的文件句柄（共享文件）或者是INVALID\_HANDLE\_VALUE（共享物理页）。
2. **lpAttributes**：指定安全属性，通常设为NULL。
3. **flProtect**：指定保护属性，确定其他进程对共享内存的访问权限，常见的属性有PAGE\_READONLY、PAGE\_READWRITE等。
4. **dwMaximumSizeHigh**和**dwMaximumSizeLow**：两个参数合起来指定了文件映射对象的最大大小。这两个参数一起构成了一个64位的整数，表示以字节为单位的最大大小。**dwMaximumSizeLow**通常可以设为BUFSIZ。
5. **lpName**：指定映射对象的名称。没有名称，可以设置为NULL。

### OpenFileMapping

OpenFileMapping的作用是**打开一个已存在的文件映射对象，并返回对应的句柄**，语法格式：

```

1 HANDLE OpenFileMapping(
2     DWORD dwDesiredAccess,
3     BOOL bInheritHandle,

```

```

4     LPCSTR lpName
5 );

```

参数含义如下：

1. **dwDesiredAccess**：指定映射对象的文件数据的访问方式，要与CreateFileMapping中设置的flProtect参数相匹配。
  - a. FILE\_MAP\_ALL\_ACCESS 对应 PAGE\_READWRITE
  - b. FILE\_MAP\_COPY 对应 PAGE\_WRITECOPY
  - c. FILE\_MAP\_EXECUTE 对应 PAGE\_EXECUTE\_READ 或 PAGE\_EXECUTE\_READWRITE
  - d. FILE\_MAP\_READ 对应 PAGE\_READONLY 或 PAGE\_READWRITE
  - e. FILE\_MAP\_WRITE 对应 PAGE\_READWRITE
2. **blInheritHandle**：指定句柄是否可以被子进程继承。如果为TRUE，则允许继承；如果为FALSE，则不允许继承。
3. **lpName**：指定已存在的映射对象的名称。

### MapViewOfFile

MapViewOfFile的作用是将创建的映射对象映射到调用进程的地址空间（物理页与线性地址进行关联），执行成功返回共享映射对象的首地址指针，语法格式：

```

1 LPVOID WINAPI MapViewOfFile(
2     HANDLE hFileMappingObject,
3     DWORD dwDesiredAccess,
4     DWORD dwFileOffsetHigh,
5     DWORD dwFileOffsetLow,
6     SIZE_T dwNumberOfBytesToMap
7 );

```

参数含义如下：

1. **hFileMappingObject**：指定要映射的映射对象句柄，CreateFileMapping或OpenFileMapping函数返回的映射对象句柄。
2. **dwDesiredAccess**：与OpenFileMapping的参数意义一样。
3. **dwFileOffsetHigh**和**dwFileOffsetLow**：指定要映射的文件的偏移量。这两个参数结合起来表示一个64位的偏移量。可以使用0表示从文件的开头进行映射。
4. **dwNumberOfBytesToMap**：指定要映射的字节数。可以与CreateFileMapping要映射的大小对应，即映射全部。

## 共享内存实验

### 共享物理页

我们已经了解了基本的函数，接下来我们写两段代码。

首先第一段代码用来创建映射对象并共享，如下所示：

```

1 #include <windows.h>
2 #include <stdio.h>
3
4 #define M_NAME "MappedMemoryA"

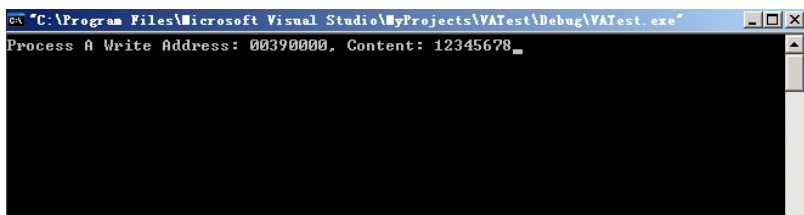
```

```

5
6 void main()
7 {
8     HANDLE hMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
9     PAGE_READWRITE, 0, BUFSIZ, M_NAME);
10    LPTSTR lpBuff = (LPTSTR)MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0,
11    BUFSIZ);
12    *(PDWORD)lpBuff = 0x12345678;
13    printf("Process A Write Address: %p, Content: %x", lpBuff,
14    *(PDWORD)lpBuff);
15    getchar();
16 }

```

我们可以运行该程序，然后去查看Vad树，成功创建并映射了我们要共享的数据。



89cc6250	6	2c0	300	0	Mapped	READONLY	\WINDOWS\system32\sortkey.nls
898873f8	5	310	315	0	Mapped	READONLY	\WINDOWS\system32\sorttbls.nls
89bc3aa0	7	320	360	0	Mapped	READONLY	Pagefile section, shared commit 0x41
899bb418	6	370	37f	8	Private	READWRITE	
89a2f0c8	7	380	382	0	Mapped	READONLY	\WINDOWS\system32\ctype.nls
89943dd8	8	390	390	0	Mapped	READWRITE	Pagefile section, shared commit 0x1
89a184a0	1	400	426	4	Mapped	Execute	\Program Files\Microsoft Visual Studio\MyProjects\VA Test\Debug\VA Test.exe
898b6240	4	430	52f	8	Private	READWRITE	

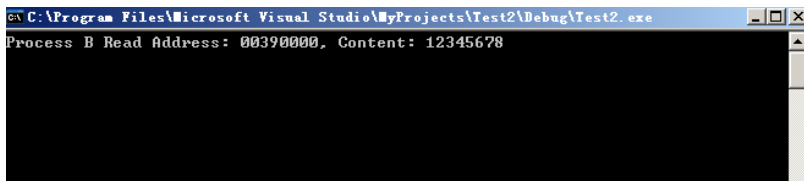
接着第二段代码用于读取共享的数据，我们需要在原有进程不关闭的情况下另起一个进程：

```

1  #include <windows.h>
2  #include <stdio.h>
3
4  #define M_NAME "MappedMemoryA"
5
6  void main()
7  {
8      HANDLE hMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, M_NAME);
9      LPTSTR lpBuff = (LPTSTR)MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0,
10      BUFSIZ);
11      printf("Process B Read Address: %p, Content: %x", lpBuff,
12      *(PDWORD)lpBuff);
13      getchar();
14  }

```

同样运行该程序，B进程成功的通过共享映射对象的名称获取到了A进程共享的数据，并且在B进程的Vad树中也多了一个节点。



Process B Read Address: 00390000, Content: 12345678

89ce6b88	6	370	37f	8 Private	READWRITE	
89ce6a60	7	380	382	0 Mapped	READONLY	\WINDOWS\system32\ctype.nls
899c4220	8	390	390	0 Mapped	READWRITE	Pagefile section, shared commit 0x1
898b63b0	1	400	426	5 Mapped Exe	EXECUTE_WRITECOPY	\Program Files\Microsoft Visual Studio\MyProjects\Test2\Debug\Test2.exe
89511178	4	430	52f	8 Private	READWRITE	

因此我们得出一个结论，一个进程在底层准备了一个或多个物理页，这些物理页在准备阶段不可被使用。但是，任何进程只要获得了该物理页的句柄，就可以将其映射到自己的内存空间中，从而能够使用这些内存。通过这种方式，进程可以存储数据或读取数据（根据创建物理页时设置的属性），实现了资源共享的目的。

## 共享文件

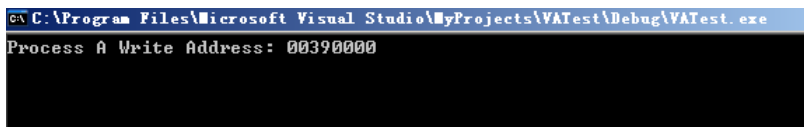
我们依旧需要写两端代码，第一段用于创建并映射共享文件：

```

1  #include <windows.h>
2  #include <stdio.h>
3
4  #define M_NAME "MappedMemoryA"
5
6  void main()
7  {
8      DWORD dwBytesWritten;
9
10     HANDLE hFile = CreateFile("C:\\\\abc.txt", GENERIC_READ | GENERIC_WRITE,
11                               FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
12     WriteFile(hFile, M_NAME, strlen(M_NAME), &dwBytesWritten, NULL);
13
14     HANDLE hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0,
15                                     BUFSIZ, M_NAME);
16     LPTSTR lpBuff = (LPTSTR)MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0,
17                                             BUFSIZ);
18     printf("Process A Write Address: %p", lpBuff);
19     getchar();
20 }

```

运行如上代码，会在C盘创建一个abc.txt然后写入MappedMemoryA，接着将该文件的句柄映射到Vad树中实现共享。



Process A Write Address: 00390000

89ce3360	5	310	315	0 Mapped	READONLY	\WINDOWS\system32\sorttbls.nls
898b11b8	7	320	360	0 Mapped	READONLY	Pagefile section, shared commit 0x41
89944ac0	6	370	37f	8 Private	READWRITE	
89ce6b48	7	380	382	0 Mapped	READONLY	\WINDOWS\system32\ctype.nls
89c37d98	8	390	390	0 Mapped	READWRITE	\\abc.txt
89509e38	1	400	426	6 Mapped Exe	EXECUTE_WRITECOPY	\Program Files\Microsoft Visual Studio\MyProjects\VAst\Debug\VAst.exe
89c0dc48	4	430	52f	8 Private	READWRITE	

第二段用于读取共享文件：

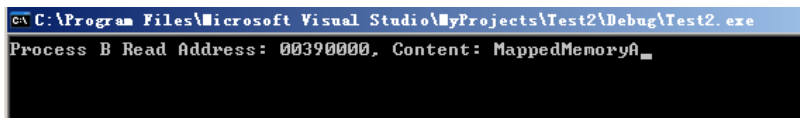


```

1  #include <windows.h>
2  #include <stdio.h>
3
4  #define M_NAME "MappedMemoryA"
5
6  void main()
7  {
8      HANDLE hMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, M_NAME);
9      LPTSTR lpBuff = (LPTSTR)MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0,
10     BUFSIZ);
11     printf("Process B Read Address: %p, Content: %s", lpBuff, lpBuff);
12     getchar();
13 }

```

运行如上代码，我们可以成功读取共享文件的内容，共享文件的主要好处是能有效地处理大文件，减少重复的内存加载和拉伸操作，从而节省资源开销。需要注意的是，如果一个进程修改了共享文件，那么所有使用该文件的进程都会受到影响：



```

C:\Program Files\Microsoft Visual Studio\MyProjects\Test2\Debug\Test2.exe
Process B Read Address: 00390000, Content: MappedMemoryA_

```

## 写拷贝

而我们会发现我们的共享文件在Vad树中并没有“Exe”的字符串标识，这是因为我们的文件并不是用来执行的，如果是用来执行的文件则会有该字符串标识，例如我们使用LoadLibrary来加载文件在Vad树中就会有所体现，并且在这里内存属性为可执行、写拷贝：

```

void main()
{
    LoadLibrary("C:\\Windows\\System32\\cmd.exe");

    getchar();
}

```



```

C:\Program Files\Microsoft Visual Studio\MyProjects\Test2\Debug\Test2.exe

```

894d2670	0	400	426	5	Mapped	Exe	EXECUTE_WRITECOPY	\Program Files\Microsoft Visual Studio\MyProjects\Test2\Debug\Test2.exe
89874a98	3	430	52f	8	Private		READWRITE	
8952b508	4	4ad00	4ad74	29	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\cmd.exe
899383a8	2	7c800	7c91b	5	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\kernel32.dll
895211c0	1	7c920	7c9b3	5	Mapped	Exe	EXECUTE_WRITECOPY	\WINDOWS\system32\ntdll.dll
89a99bc8	3	7f6f0	7f7ef	0	Mapped		EXECUTE_READ	Pagefile section, shared commit 0x7
894e0410	2	7ffa0	7ffd2	0	Mapped		READONLY	Pagefile section, shared commit 0x33

因此我们知道LoadLibrary函数底层实现利用了映射文件的机制来实现共享文件，被LoadLibrary加载的映射文件具备EXECUTE\_WRITECOPY内存保护属性，该属性用于防止其他进程修改映射文件。

当一个进程试图修改映射文件时，如果该文件的内存保护属性是EXECUTE\_WRITECOPY，操作系统会使该进程指向一个新的物理页，该物理页保存着映射文件的副本，这样进程对映射文件的修改不会影响真正的映射文件，这种保护机制可用于防止对系统函数进行Hook等操作。

## 1.2 物理内存的管理

在保护模式篇的学习中我们实际上已经有了一些物理内存的概念，结合之前的学习我们来重新的了解一下物理内存的管理。

### 1.2.1 认识物理内存

#### 最大物理内存

1. 10-10-12分页：最多识别物理内存为4GB；
2. 2-9-9-12分页：最多识别物理内存为64GB。

#### 操作系统限制

在Windows XP 32位操作系统中，即使是2-9-9-12分页模式，也仍然无法超越4GB，这是因为在内核函数中（Ntoskrnl.exe模块），存在一个名为MmAddPhysicalMemoryEx的函数。该函数调用了ExVerifySuite函数，其限制了操作系统无法识别超过4GB的情况。关于ExVerifySuite函数的详细分析超出了当前学习的范围，请自行进行进一步研究。

```

PAGELK:80188380 loc_80188380:                                ; CODE XREF: MmAddPhysicalMemoryEx(x,x,x)+A7↑j
PAGELK:80188380      cmp     edi, ebx
PAGELK:80188382      jnb     short loc_80188405
PAGELK:80188384      push    7                ; SuiteType
PAGELK:80188386      call    _ExVerifySuite@4 ; ExVerifySuite(x)
PAGELK:80188388      cmp     al, 1
PAGELK:8018838D      jnz     short loc_801883C6
PAGELK:8018838F      mov     eax, 1000000h
PAGELK:801883C4      jmp     short loc_801883E7
PAGELK:801883C6 ; -----
PAGELK:801883C6 loc_801883C6:                                ; CODE XREF: MmAddPhysicalMemoryEx(x,x,x)+C3↑j
PAGELK:801883C6      cmp     _MmProductType, 690057h
0018B3B6 8018B3B6: MmAddPhysicalMemoryEx(x,x,x)+BC (Synchronized with Hex View-1)

```

#### 实际物理内存

我们可以通过任务管理器来查看实际的物理内存：

总数		物理内存 (K)	
句柄数	6705	总数	2096552
线程数	352	可用数	1744576
进程数	31	系统缓存	267808

也可以通过Windbg来查看，即指令dd MmNumberOfPhysicalPages，这个值是物理页总个数，我们只要乘以4KB即可算出物理内存的KB大小：

```

kd> dd MmNumberOfPhysicalPages
80562108  0007ff6a 00000040 00000000 7ffff000
80562118  80000000 7ffeffff 00000000 00000001
80562128  00000000 00000000 00000000 00000001

```



### 1.2.2 空闲页的管理

#### 全局数组及成员

在操作系统中，存在一个全局数组，用于记录所有物理页的信息。

- 1 数组指针：\_MMPFN\* MmPfnDatabase
- 2 数组长度：MmNumberOfPhysicalPages

每一个物理页都是该全局数组的一个成员，对应结构体\_MMPFN，该结构体有很多union类型成员，也就表示当前结构体的意义很多：

```

1  kd> dt _MMPFN
2  nt!_MMPFN
3      +0x000 u1                : __unnamed
4      +0x004 PteAddress        : Ptr32 _MMPTE
5      +0x008 u2                : __unnamed
6      +0x00c u3                : __unnamed
7      +0x010 OriginalPte       : _MMPTE
8      +0x018 u4                : __unnamed
9
10 typedef struct _MMPFN
11 {
12     union
13     {
14         PFN_NUMBER Flink;
15         ULONG WsIndex;          // 该页面在进程工作集链表中的索引
16         PKEVENT Event;
17         NTSTATUS ReadStatus;

```

```

18     SINGLE_LIST_ENTRY NextStackPfn;
19     SWAPENTRY SwapEntry;
20 } u1;
21 PMMPTE PteAddress;           // 执行此页面的PTE的虚拟地址
22 union
23 {
24     PFN_NUMBER Blink;
25     ULONG_PTR ShareCount;     // 指向该页面的PTE数量
26 } u2;
27 union
28 {
29     struct
30     {
31         USHORT ReferenceCount; // 代表这个页面必须要保留在内存中的引用计
数
32         MMPFNENTRY e1;
33     };
34     struct
35     {
36         USHORT ReferenceCount;
37         USHORT ShortFlags;
38     } e2;
39 } u3;
40 union
41 {
42     MMPTTE OriginalPte;       // 包含了指向此页面的PTE的原始内容
43     LONG AweReferenceCount;
44     PMM_RMAP_ENTRY RmapListHead;
45 };
46 union
47 {
48     ULONG_PTR EntireFrame;
49     struct
50     {
51         ULONG_PTR PteFrame:25;
52         ULONG_PTR InPageError:1;
53         ULONG_PTR VerifierAllocation:1;
54         ULONG_PTR AweAllocation:1;
55         ULONG_PTR Priority:3;
56         ULONG_PTR MustBeCached:1;
57     };
58 } u4;           // 指向该页面的PTE所在的页表页面的物理页帧编号，以及一些
标志位

```

如下图所示，我们可以在Windbg中查看该全局数组指针（0x81086000），当前这个地址就对应了一个\_MMPFN结构体（用于描述物理页的信息），该结构体大小为0x1C（操作系统版本决定），该结构体本身并没有记录描述信息的归属。

```
kd> dd MmPfnDatabase
805620a8 81086000 0000ff00 00000006 0000003f
805620b8 0000f4ec 0006d36d 001a3087 00003eef
805620c8 0005f464 00000000 00135f28 0000c700
805620d8 0000263c 00000000 00000000 000085f0
805620e8 0000085f 00001c81 00000227 0000001e
805620f8 000000fa 0006ab9a 0007ffff 0007ffff
80562108 0007ff6a 00000040 00000000 7fff0000
80562118 80000000 7ffeffff 00000000 00000001
```

微软使用了一种很巧妙的方式来找到结构体的归属物理页：MmNumberOfPhysicalPages存储的值表示物理页的总数，每个\_MMPFN结构体用来描述一个物理页，因此就让每个物理页与全局数组中每个\_MMPFN结构体按顺序对应。

算式为：**MmPfnDatabase + 结构体大小 \* 数组索引**，举例说明：

1. 第一个物理页（0x0000 - 0x0FFF，**去掉后三位就是索引0x0000**）对应的\_MMPFN结构体地址：  
0x81086000 + 0x1C \* 0；
2. 第二个物理页（0x1000 - 0x1FFF，**去掉后三位就是索引0x1000**）对应的\_MMPFN结构体地址：  
0x81086000 + 0x1C \* 1；
3. 以此类推...

## 物理页状态

物理页有很多种状态，在\_MMPFN.u3.e1处的结构是一个位段（\_MMPFNENTRY），其结构如下，其中成员PageLocation表示了当前物理页的状态：

```
1 kd> dt _MMPFNENTRY
2 nt!_MMPFNENTRY
3 +0x000 Modified : Pos 0, 1 Bit
4 +0x000 ReadInProgress : Pos 1, 1 Bit
5 +0x000 WriteInProgress : Pos 2, 1 Bit
6 +0x000 PrototypePte : Pos 3, 1 Bit
7 +0x000 PageColor : Pos 4, 3 Bits
8 +0x000 ParityError : Pos 7, 1 Bit
9 +0x000 PageLocation : Pos 8, 3 Bits
10 +0x000 RemovalRequested : Pos 11, 1 Bit
11 +0x000 CacheAttribute : Pos 12, 2 Bits
12 +0x000 Rom : Pos 14, 1 Bit
13 +0x000 LockCharged : Pos 15, 1 Bit
14 +0x000 DontUse : Pos 16, 16 Bits
```

每一种状态对应一个链表，该链表串着所有当前状态的物理页，成员PageLocation的值和链表、状态、含义对应如下（都可以表示为空闲的物理页）：

取值	对应的全局链表	状态类型	含义
0	MmZeroedPageListHead	零化	表示页面是空闲的，不属于任何工作集。零化链表中的每个物理页的内容已经被全部清零，即所有数据被初始化为零。
1	MmFreePageListHead	空闲	表示页面是空闲的，不属于任何工作集。物理页是周转使用的，页中的数据可能包含不确定的内容。当系统处于空闲状态时，有专门的线程从该链表中获取物理页，并将其清零后放入零化链表。
2	MmStandbyPageListHead	备用	<p>这种页面原本属于某个进程或系统工作集，但现在已从工作集中移除。这些页面包含的数据仍然对原来的工作集有效，原来工作集中的页表项PTE仍指向该页面，但已被标记为无效的正在转移的PTE。<b>这种状态是导致缺页异常的主要原因之一。</b></p> <p><b>简而言之：</b>当系统内存不够的时候，操作系统会把物理内存中的数据交换到硬盘上，此时页面不是直接挂到空闲链表上去，而是挂到备用链表上，虽然释放了，但里边的内容还是有意义的。</p>
3	MmModifiedPageListHead	修改	类似于备用状态，页面已从原来的工作集中移除，但页面的内容已被修改。原来工作集中的PTE仍然指向该物理页面，但已被标记为无效的正在转移的PTE。如果系统要回收此类页面以供其他用途，则必须将页面的内容写入磁盘。
4	MmModifiedNoWritePageListHead	已修改但不写出	类似于已修改状态，但区别在于内存管理器不会将页面的内容写入磁盘。这种状态通常用于临时修改的数据，系统不需要将其持久化到磁盘中，以提高性能。
5	MmBadPageListHead	损坏	表示页面发生了硬件错误，系统不再使用该页面。这种页面通常被标记为无效页，不再分配给任何进程或工作集。



每个链表的结构如下，可以看见都是第3、4个成员为LIST\_HEAD，它们都是链表头的索引，不同的是前者是从前往后，后者是从后往前：

```

1  MMPFNLIST MmZeroedPageListHead = {0, ZeroedPageList, LIST_HEAD,
   LIST_HEAD};
2  MMPFNLIST MmFreePageListHead = {0, FreePageList, LIST_HEAD, LIST_HEAD};
3  MMPFNLIST MmStandbyPageListHead = {0, StandbyPageList, LIST_HEAD,
   LIST_HEAD};
4  MMPFNLIST MmModifiedPageListHead = {0, ModifiedPageList, LIST_HEAD,
   LIST_HEAD};
5  MMPFNLIST MmModifiedNoWritePageListHead = {0, ModifiedNoWritePageList,
   LIST_HEAD, LIST_HEAD};
6  MMPFNLIST MmBadPageListHead = {0, BadPageList, LIST_HEAD, LIST_HEAD};
7  MMPFNLIST MmRomPageListHead = {0, StandbyPageList, LIST_HEAD, LIST_HEAD};

```

## 查询链表实验

我们了解到所有物理页都记录在一个全局数组中，每个物理页对应一个\_MMPFN物理页描述结构，空闲的物理页有6种状态，并分别对应一个全局链表。现在我们将把这些知识点联系在一起，并通过一个实验来巩固所学。我们以零化链表为例，来查询该链表上的所有物理页。

直接在Windbg中通过指令：`dd MmZeroedPageListHead`，来找到链表头索引：

```

kd> dd MmZeroedPageListHead
805518e8  0005b43a 00000000 0001a7a8 0001ce85
805518f8  00000000 00000001 ffffffff ffffffff
80551908  0000f760 00000002 00070dab 0006f3a8
80551918  000000c7 00000003 ffffffff ffffffff
80551928  00000000 00000004 ffffffff ffffffff
80551938  00000000 00000005 ffffffff ffffffff
80551948  00000000 00000002 ffffffff ffffffff
80551958  805518e8 805518f8 80551908 80551918

```

接着我们只要通过指令：`dd 0x81086000 + 0x1c * 0x0001a7a8`，即之前所说的公式，找到\_MMPFN结构体，再根据PageLocation（0xC偏移位结构体成员的成员）确认这是一个零化物理页：

```

kd> dt _MMPFNENTRY 81086000 + 1c * 0001a7a8 + 0xc
nt!_MMPFNENTRY
+0x000 Modified          : 0y0
+0x000 ReadInProgress    : 0y0
+0x000 WriteInProgress   : 0y0
+0x000 PrototypePte     : 0y0
+0x000 PageColor         : 0y000
+0x000 ParityError       : 0y0
+0x000 PageLocation      : 0y000
+0x000 RemovalRequested  : 0y0
+0x000 CacheAttribute    : 0y11
+0x000 Rom               : 0y0
+0x000 LockCharged       : 0y0
+0x000 DontUse           : 0y0000000000000000 (0)

```

而在这里我们找到了一个零化物理页，想要找到第二个该怎么办？其实很简单，我们再回到\_MMPFN结构体上，它的成员u1可以表示为Flink，u2可以表示为Blink：

```
kd> dt _MMPFN 81086000 + 1c * 0001a7a8
nt!_MMPFN
+0x000 u1 : __unnamed
+0x004 PteAddress : 0xc0023101 _MMPTE
+0x008 u2 : __unnamed
+0x00c u3 : __unnamed
+0x010 OriginalPte : _MMPTE
+0x018 u4 : __unnamed
kd> dx -id 0,0,ffffffff8055c0a0 -r1 (*((ntkrpamp! __unnamed *)0xffffffff8136b668))
*((ntkrpamp! __unnamed *)0xffffffff8136b668) [Type: __unnamed]
[+0x000] Flink : 0xffffffff [Type: unsigned long]
[+0x000] WsIndex : 0xffffffff [Type: unsigned long]
[+0x000] Event : 0xffffffff [Type: _KEVENT *]
[+0x000] ReadStatus : -1 [Type: long]
[+0x000] NextStackPfn [Type: _SINGLE_LIST_ENTRY]
kd> dx -id 0,0,ffffffff8055c0a0 -r1 (*((ntkrpamp! __unnamed *)0xffffffff8136b660))
*((ntkrpamp! __unnamed *)0xffffffff8136b660) [Type: __unnamed]
[+0x000] Flink : 0x7c7a8 [Type: unsigned long]
[+0x000] WsIndex : 0x7c7a8 [Type: unsigned long]
[+0x000] Event : 0x7c7a8 [Type: _KEVENT *]
[+0x000] ReadStatus : 509864 [Type: long]
[+0x000] NextStackPfn [Type: _SINGLE_LIST_ENTRY]
```

因此我们可以来看下当前结构体，u1是有值的，u2是没有值，那么我们就可以通过u1来找到下一个零化物理页对应的结构体：

```
kd> dd 0x81086000 + 0x1c * 0x0001a7a8
8136b660 0007c7a8 c0023101 ffffffff 00003000
8136b670 0007c7a8 00000000 0001a2c7 0001a0ae
8136b680 c0713291 00017b68 00003000 0001b2a9
8136b690 00000000 00015cc9 0001b42b c0712a09
8136b6a0 00019b69 00003000 0001b2aa 00000000
8136b6b0 000158f6 00015792 c0010f01 0001450d
8136b6c0 00003000 0001a4eb 00000000 0004f153
8136b6d0 0000139d c00003c8 00000001 00011601
```

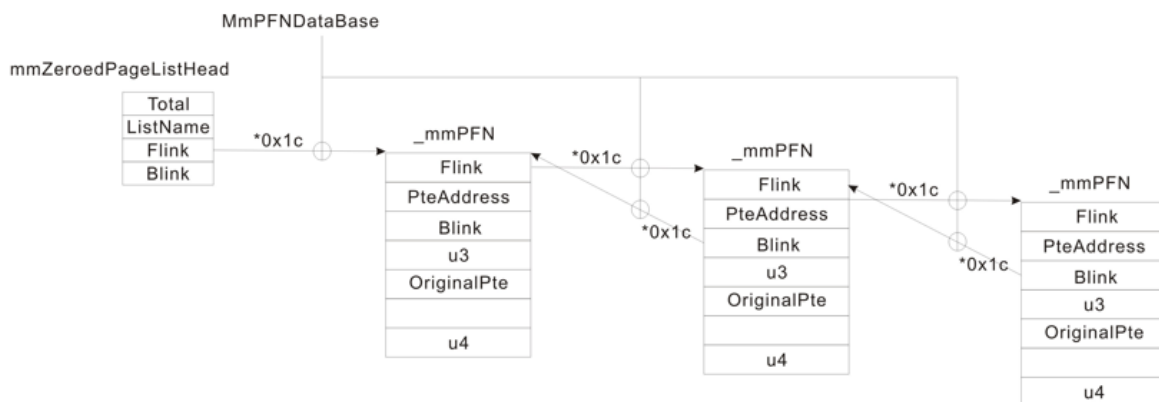
即通过指令：`dd 0x81086000 + 0x1c * 0x0007c7a8 (Flink)`，成功找到了另一个零化物理页\_MMPFN结构体：

```

kd> dt _MMPFN 0x81086000 + 0x1c * 0x0007c7a8
nt!_MMPFN
+0x000 u1 : __unnamed
+0x004 PteAddress : 0xc0023301 _MMPTE
+0x008 u2 : __unnamed
+0x00c u3 : __unnamed
+0x010 OriginalPte : _MMPTE
+0x018 u4 : __unnamed
kd> dt _MMPFNENTRY 0x81086000 + 0x1c * 0x0007c7a8 + 0xc
nt!_MMPFNENTRY
+0x000 Modified : 0y0
+0x000 ReadInProgress : 0y0
+0x000 WriteInProgress : 0y0
+0x000 PrototypePte : 0y0
+0x000 PageColor : 0y000
+0x000 ParityError : 0y0
+0x000 PageLocation : 0y000
+0x000 RemovalRequested : 0y0
+0x000 CacheAttribute : 0y11
+0x000 Rom : 0y0
+0x000 LockCharged : 0y0
+0x000 DontUse : 0y0000000000000000 (0)

```

那么也就表示在当前描述零化物理页的\_MMPFN结构体中，u1、u2成员就分别代表了Flink、Blink，通过这两个成员将所有零化物理页串成一个链表。



不同状态物理页描述结构体\_MMPFN对应的成员意义如下图所示：

```

union {
    WSLE_NUMBER WsIndex;
} u1;
PMMPTE PteAddress;
union {
    ULONG_PTR ShareCount;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
    };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
    };
} u4;

```

(a) 活动状态页面的 PFN 项

```

union {
    PFN_NUMBER Flink;
} u1;
PMMPTE PteAddress;
union {
    PFN_NUMBER Blink;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
    };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
    };
} u4;

```

(c) 零化的或空闲页面的 PFN 项

```

union {
    PFN_NUMBER Flink;
} u1;
PMMPTE PteAddress;
union {
    PFN_NUMBER Blink;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
    };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
    };
} u4;

```

(b) 备用状态或已修改状态的 PFN 项

```

union {
    PKEVENT Event;
    NTSTATUS ReadStatus;
} u1;
PMMPTE PteAddress;
union {
    ULONG_PTR ShareCount;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
    };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
    };
} u4;

```

(d) 转移状态的 PFN 项

### 1.2.3 活动页的管理

物理页分为两大类：空闲页、活动页，我们了解了空闲页再来看一下活动页。

首先任意打开一个进程，定位到进程结构体 0x1f8 偏移位，有一个 Vm 成员，是 \_MMSUPPORT 结构体，该结构体下有一个 VmWorkingSetList 成员，它记录着当前进程相关的工作集信息：

```
kd> dt _MMSUPPORT 898ba020+1f8
nt!_MMSUPPORT
+0x000 LastTrimTime : _LARGE_INTEGER 0x01d9e628`6032b7ef
+0x008 Flags : _MMSUPPORT_FLAGS
+0x00c PageFaultCount : 0xc1
+0x010 PeakWorkingSetSize : 0xc8
+0x014 WorkingSetSize : 0xc8
+0x018 MinimumWorkingSetSize : 0x32
+0x01c MaximumWorkingSetSize : 0x159
+0x020 VmWorkingSetList : 0xc0883000 _MMWSL
+0x024 WorkingSetExpansionLinks : _LIST_ENTRY [ 0x89c83c3c - 0x8989923c ]
+0x02c Claim : 0
+0x030 NextEstimationSlot : 0
+0x034 NextAgingSlot : 0
+0x038 EstimatedAvailable : 0
+0x03c GrowthSinceLastEstimate : 0xc1
kd> dt _MMWSL 0xc0883000
nt!_MMWSL
+0x000 Quota : 0
+0x004 FirstFree : 7
+0x008 FirstDynamic : 7
+0x00c LastEntry : 0x32
+0x010 NextSlot : 7
+0x014 Wsle : 0xc0883cfc MMWSLE
+0x018 LastInitializedWsle : 0xc0
```

VmWorkingSetList对应的结构体\_MMWSL还有一个Wsle成员，用来描述一个有效页面。  
这部分海东老师也是一笔带过，想要具体了解可以阅读《Windows内核原理与实现》。

1.3 缺页异常

1.3.1 什么是缺页异常

我们以32位的PTE为例，属性位P位标识当前页面是否有效，当CPU访问一个地址，其PTE的P位为0，此时就会产生缺页异常。



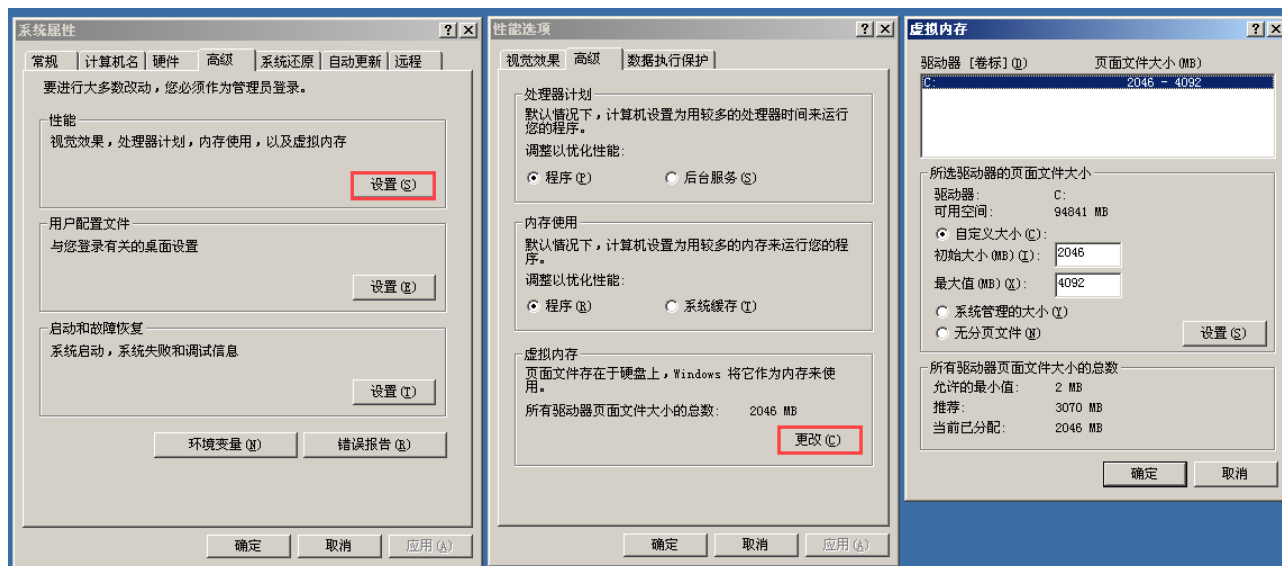
缺页异常并非总是不利的，实际上，在Windows系统中，每秒都会发生缺页异常。正是通过这种异常机制，Windows才能够更有效地利用物理页资源。

1.3.2 内存交换与虚拟内存

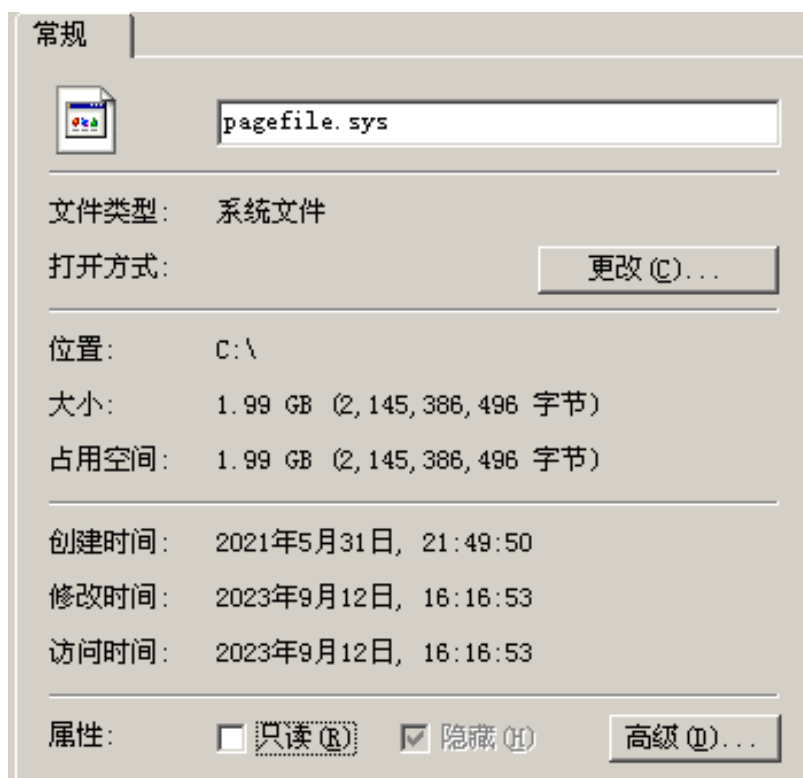
假设当前操作系统只有2M的有效物理内存。如果一个线程使用VirtualAlloc函数申请了某个线性地址对应的物理页，并且一直占用不释放，很快就会占满整个内存。然而线程并非始终处于执行状态，有时会进入休眠，进入等待队列。线程处于休眠状态，却仍然占用着物理页，导致内存利用效率非常低下。  
为了提高物理内存的利用效率，Windows引入了内存交换机制。该机制的核心思想是，只有正在被使用的线性地址才会被分配物理页。如果某个线性地址在一段时间内没有被使用，或者当前的物理页资源接近耗尽，操作

系统将该线性地址对应的物理页上的数据保存到硬盘，并将对应的页表项（PTE）中的P位设置为0。这样，该物理页就被释放出来供其他线性地址使用。

我们可以在Windows XP中找到虚拟内存：系统属性->高级->性能->设置->高级->虚拟内存->更改。

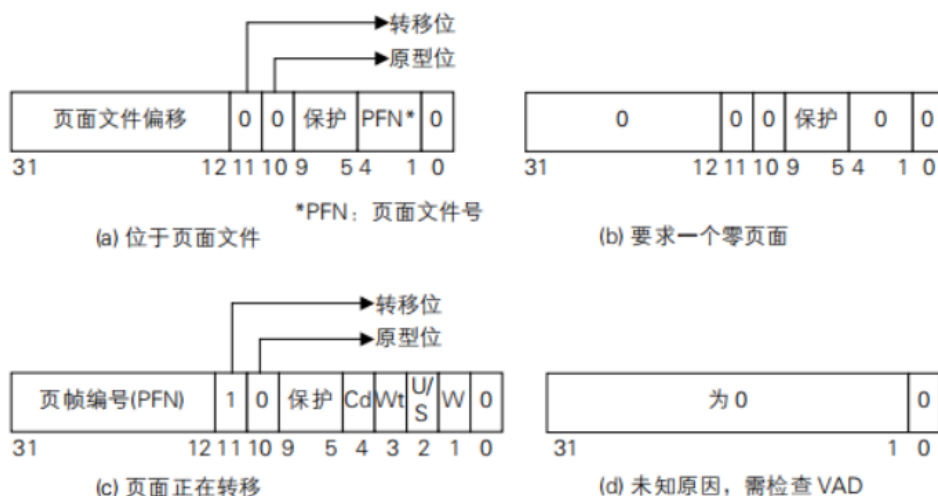


而这个虚拟内存实际上存放在C盘下的pagefile.sys，其对应的大小就是上图中的初始化的大小，内存交换到硬盘上的数据就会存在这个文件中。



当操作系统将物理页上的数据保存到硬盘上时，也将该线性地址对应物理页的PTE的属性位P位设为了0。一旦该线性地址再次被访问，由于P位为0，则会触发缺页异常。那么操作系统是如何处理的呢？当P位为0时，此时的PTE被称作无效PTE，有以下四种情形：





在不同的情况下, 处理缺页异常的方式也不同。其中, 由于内存交换引起的缺页异常属于第一种情况, 即页面文件中的情况。在这种情况下, 页表项 (PTE) 的1-4位、5-9位和12-31位都有值, 表示线性地址是有效的, 只是对应的数据位于硬盘上。

当发生这种缺页异常时, 异常处理程序 (例如e号中断) 会根据PTE的描述 (即12-31位的页面文件偏移), 从pagefile.sys中获取数据内容, 并将其加载到一个新的物理页上。然后将PTE的12-31位设置为新的物理页地址, 并将P位设置为1。这样, 缺页异常的处理就完成了。这种类型的缺页异常非常常见, 几乎时刻都在发生, 但对于用户来说, 是察觉不到任何异常的, 程序仍然正常执行。

### 1.3.3 保留与提交的误区

回顾之前介绍的VirtualAlloc函数, 其第三个参数flAllocationType可以有两个值, 含义如下:

1. MEM\_RESERVE: 申请内存时, 仅保留线性地址, 不分配物理页。
2. MEM\_COMMIT: 可以有物理页, 但不是立即有或者一直有。

我们一直认为当参数值为MEM\_COMMIT时, 申请内存就会给我们物理页, 但事实并非如此, 我们可以写一段代码来论证:

```

1  #include <windows.h>
2  #include <stdio.h>
3
4  void main()
5  {
6      LPVOID lpAddress = ::VirtualAlloc(NULL, 0x1000*2, MEM_COMMIT,
7      PAGE_READWRITE);
8      getchar();
9      *(PDWORD)lpAddress = 0x12345678;
10     getchar();
11 }

```

执行代码, 在第一个getchar停留, 来到Windbg查看进程的Cr3, 使用指令: !vtop Cr3 线性地址, 会帮我们自动将线性地址转换为物理地址, 但是在转换过程中PTE为空, 所以最后提示PAE zero PTE, 也就表示当前并没有物理页与线性地址进行关联。

```
C:\Program Files\Microsoft Visual Studio\MyProjects\VATest\Debug\VATest.exe
Address: 00390000
```

Failed to get VadRoot

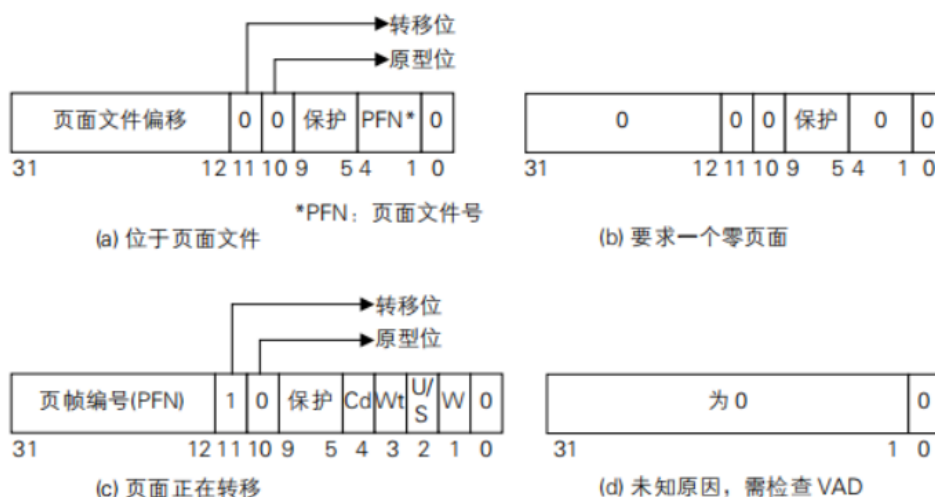
```
PROCESS 8989b020 SessionId: 0 Cid: 099c Peb: 7ffdc000 ParentCid: 00ac
DirBase: 0a3c0420 ObjectTable: e1138b98 HandleCount: 15.
Image: VATest.exe
```

```
kd> !vtop 0x0a3c0420 0x390000
X86VtoP: Virt 0000000000390000, pagedir 000000000a3c0420
X86VtoP: PAE PDPE 000000000a3c0420 - 0000000040591801
X86VtoP: PAE PDE 0000000040591008 - 000000003f0f8867
X86VtoP: PAE PTE 000000003f0f8c80 - 0000000000000000
X86VtoP: PAE zero PTE
Virtual address 390000 translation fails, error 0xD0000147.
```

接着我们回到程序按回车键，在第二个getchar停留，此时已经向申请的线性地址写入了0x12345678，再使用上述指令在Windbg中查看，此时PTE有值，且成功算出了物理地址：

```
kd> !vtop 0x0a3c0420 0x390000
X86VtoP: Virt 0000000000390000, pagedir 000000000a3c0420
X86VtoP: PAE PDPE 000000000a3c0420 - 0000000040591801
X86VtoP: PAE PDE 0000000040591008 - 000000003f0f8867
X86VtoP: PAE PTE 000000003f0f8c80 - 8000000040c19867
X86VtoP: PAE Mapped phys 0000000040c19000
Virtual address 390000 translates to physical address 40c19000.
```

其实这里也运用了缺页异常机制，当CPU访问线性地址时，如果发现PTE的值为0，就会触发无效PTE的第四种情况（原因未知，需要检查Vad）。在这种情况下，操作系统会检查当前进程的VAD树。



如果线性地址存在于Vad树中，操作系统将为该线性地址分配一个物理页，并填写PTE的12-31位和1-9位，将P位设置为1。这样，线性地址与物理页之间建立了映射关系。

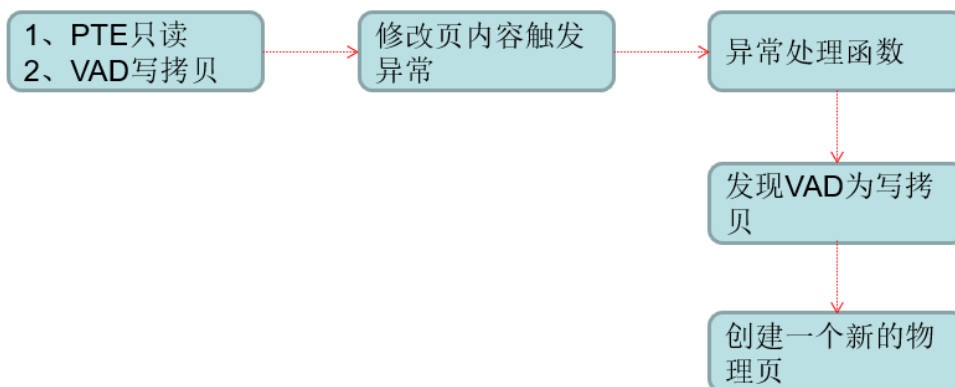
如果线性地址在Vad树中不存在，操作系统将报告0xC0000005错误，表示访问违规。通过利用缺页异常的机制，操作系统能够动态地管理线性地址和物理页之间的映射关系，以提供有效的内存访问和保护机制。

### 1.3.4 写拷贝原理

最后我们来看一下写拷贝的原理，PTE是与物理内存相关的，而Vad树则与线性地址有关，写拷贝实现就利用了这两者的属性。

下面是具体的步骤：

1. 当一个进程试图对受到写拷贝保护的文件进行写操作时，操作系统会检查PTE的R/W属性。
2. 受写拷贝保护的文件的物理页所在的PTE的R/W属性被设置为0（只读）。
3. 当操作系统检测到进程尝试向只读的物理页写入数据时，会触发异常并跳转到异常处理函数。
4. 异常处理函数会查找进程的Vad树，并发现该文件的内存保护属性为写拷贝。
5. 操作系统会创建一个新的物理页，并将源文件的内容拷贝到新的物理页中。
6. 让试图修改文件的进程中的映射指向这个新的物理页。



通过这种方式，写拷贝实现了对受保护文件的写保护，确保进程对文件的修改只会影响到副本，而不会修改源文件本身。（Bypass写拷贝也很简单，直接将PTE的R/W属性设1）