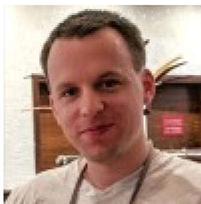
Dissecting Windows Section Objects

linkedin.com/pulse/dissecting-windows-section-objects-artem-baranov

Artem Baranov 

Artem Baranov

Security Researcher at Kaspersky

Published Sep 16, 2022

+ Follow

Instead of introduction

We can't imagine Windows without section objects (or file mapping objects in terms of Windows API) and hardly can we find a Windows kernel subsystem that doesn't address it. The great idea behind section objects is that instead of calling Windows File APIs to work with a file, you can read virtual memory to get file data and write virtual memory to write file data. But this simple concept doesn't have simple things under the hood. To simplify the understanding of this difficult topic, we take Windows x86 edition with 32-bit pointers.

Don't worry if you can't understand all the things, even skilled Windows Internals readers may have difficulties with this topic. I would recommend to read the corresponding chapter from the Windows Internals book, because this blog post includes a lot of technical stuff and

describes some kind of low level things.

The basic terms

So if you're ready, let's get started. First, we need to take a quick look at some technical terms, because without understanding any of them, we can't get the full picture. Next we'll focus on each of them in detail.

- **Section object** - a kernel object described by the _SECTION structure. In the terms of Windows API it's called file mapping object. There're two types of section objects: *pagefile-backed section* and *file-backed section*. The first one is used when processes want to share a region of virtual memory. The file backed section reflects the contents of an actual file on disk.
- **Virtual Memory Manager (VMM)** - a set of *Mm* functions in ntoskrnl that are responsible for all operations related to virtual and physical memory. The VMM also creates, maintains and deletes section objects as well as their substructures (see below).
- **I/O manager** - in the context of our topic, these are *Io* functions in ntoskrnl that are used by the VMM to perform I/O operations with the mapped file data. This subsystem just initiates I/O operations, which are actually performed by file system drivers and disk drivers on device stacks.
- **PTE (Page Table Entry)** - a structure that is used by the CPU and VMM to translate virtual addresses to physical ones.
- **Proto-PTE (Prototype PTE, PPTE)** - a special type of invalid PTEs that is used only by the VMM (not CPU) to work with section objects and serves as an intermediate level for the translation virtual addresses to the mapped section pages (file data). PPTE points to a subsection and helps the VMM to find file data that should be located in the corresponding virtual memory pages.
- **PTE pointing to PPTE** - a special type of hardware invalid PTEs /with zeroed valid (V) flag/ that is designed to find the corresponding PPTE in the Segment structure (PPT).
- **Prototype page table (PPT)** - an array of PPTEs that is a part of Segment structure. Once the process maps a section, the VMM fills the hardware PTEs of the virtual pages with pointers to the elements of this array. When the process unmaps a section, the VMM removes pointers to PPTEs from hardware PTEs.
- **Segment** - a data structure that provides the section object with the necessary information to calculate pointers to subsections, it also contains a PPT.
- **Segment Control Area (or just Control Area, CA)** - a structure containing information required for performing I/O operations with file data in or from the mapped file. It's stored in the non-paged pool. With the help of CA the VMM can address the same file as binary and as executable.

- **Subsection** - a data structure containing the necessary information to calculate offsets relative to the beginning of the mapped file using PPTEs. There is normally only one subsection if the file was mapped as binary. In case if it was mapped as executable, the number of subsections is the same as the number of sections in the mapped executable.
- **Page fault (#PF) for section** - a situation (an exception) when a thread tries to access a virtual page mapped to the section, but its PTE is marked as not valid.
- **Modified page writer** - system threads that are responsible for synchronizing modified file data in virtual memory with a disk file.
- **Page Frame, Page Frame Number, PFN database** - terms describing physical memory: physical memory page, its number, numbers database. The latter includes information about all physical memory pages (page frames) and is designed to track status of each physical page (page frame).

Diving deeper into the Section kernel objects

Section is a kernel object that is created and maintained by the VMM. The *MmCreateSection* function creates the kernel object, allocating memory for it from the paged pool, initializes its fields, creates Control Area and Segment structures if needed (see *MiCreateImageFileMap*, *MiCreateDataFileMap*). To create an object, the caller of *MmCreateSection* must provide a pointer to a FileObject that describes the file to be mapped. Using the FileObject, the functions mentioned above initialize Control Area and Segment structures.

```
1   typedef struct _SECTION
2   {
3       MMADDRESS_NODE Address;
4       PSEGMENT Segment;
5       LARGE_INTEGER SizeOfSection;
6       union {
7           ULONG LongFlags;
8           MMSECTION_FLAGS Flags;
9       } u;
10      MM_PROTECTION_MASK InitialPageProtection;
11  } SECTION, *PSECTION;
12
13
14
```

MmCreateSection is responsible not only for initializing a Section object, but also for initializing and maintaining important PSECTION_OBJECT_POINTERS FILE_OBJECT->SectionObjectPointer structure. You can see its definition below.

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

- .DataSectionObject points to the Control Area structure if a file to be mapped as binary;

- .ImageSectionObject points to the Control Area structure if a file to be mapped as executable;
- .SharedCacheMap points to the shared cache map (see Inside the Windows Cache manager). This field is used by the Cache Manager to cache file data.

As you can see all these three fields point to the structures needed to perform a certain type of file operations. The SECTION_OBJECT_POINTERS structure is created by the FSD when it gets a request to create (open) a file. The Cache Manager deals with .SharedCacheMap. Even if there are no sections for the file object (i e .DataSectionObject and .ImageSectionObject are NULL), .SharedCacheMap is almost always initialized (for disk files), because the Cache Manager caches parts of the file to provide quick access to its data. To create .DataSectionObject and .ImageSectionObject the VMM uses functions *MiCreateDataFileMap* and *MiCreateImageFileMap*.

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

Description of these arguments matches those ones from *NtCreateSection*.

Take a look at the Control Area structure

Segment control area (or just Control Area, CA) is a structure containing the information necessary to perform I/O operations with a section. It's stored in the nonpaged pool and is described by the following structure.

```
1   typedef struct _CONTROL_AREA
2   {
3       PSEGMENT Segment; //ptr to Segment
4       LIST_ENTRY DereferenceList;
5       ULONG NumberOfSectionReferences;
6       ULONG NumberOfPfnReferences;
7       ULONG NumberOfMappedViews;
8       ULONG NumberOfSystemCacheViews;
9       ULONG NumberOfUserReferences;
10      union {
11          ULONG LongFlags;
12          MMSECTION_FLAGS Flags;
13      } u;
14      PFILE_OBJECT FilePointer; //ptr to FileObject
15      PEVENT_COUNTER WaitingForDeletion;
16      USHORT ModifiedWriteCount;
17      USHORT FlushInProgressCount;
18      ULONG WritableUserReferences;
19  } CONTROL_AREA, *PCONTROL_AREA;
20
```

Control Area contains all the necessary data to perform I/O operations with the section.

- Pointer to a Segment containing information from the PE file header and a PPTE array.
- Pointer to a File Object describing mapped file that will be used for I/O operations.
- An array of subsections, which is located after the CA structure in virtual memory, containing the necessary data to calculate file offsets.

The Control Area structure contains the flags that indicate what kind of data is addressed by the section. When the VMM creates a CA object for an executable file using *MiCreateImageFileMap*, its size is equal to the size of the CA structure, plus the size of one Subsection structure multiplied by the number of subsections (i e number of PE sections + 1 for PE header). It's important to note that all _SUBSECTION structures are located immediately after the Control Area and their number is stored in the NumberOfSubsections field. The subsections of one section (Control Area) are linked in the list via .NextSubsection. The !ca comment of WinDbg prints information about Control Area.

```
0: kd> !ca 81b85248

ControlArea  @ 81b85248
  Segment      e16c6440  Flink      00000000  Blink        00000000
  Section Ref         0  Pfn Ref           e  Mapped Views        2
  User Ref            2  WaitForDel        0  Flush Count         0
  File Object  81b8ac70  ModWriteCount     0  System Views        0

  Flags (91000a0) Image File DebugSymbolsLoaded HadUserReference Accessed

      File: \WINDOWS\system32\regapi.dll -> described file

Segment @ e16c6440
  ControlArea     81b85248  BasedAddress  76bc0000
  Total Ptes             f
  WriteUserRef           0  SizeOfSegment     f000 -> size of the _SEGMENT structure
  Committed              0  PTE Template  86042c3e
  Based Addr      76bc0000  Image Base           0
  Image Commit           1  Image Info    e16c64b8
  ProtoPtes       e16c6478 -> ptr to PPTE table (Segment->PrototypePte = &Segment->ThePtes[0])

SizeOfSegment = sizeof(SEGMENT) + (sizeof(MMPTE) * ((ULONG)NumberOfPtes - 1)) + sizeof(SECTION_IMAGE_INFORMATION)


Subsection 1 @ 81b85278
  ControlArea  81b85248  Starting Sector        0  Number Of Sectors    2
  Base Pte     e16c6478  Ptes In Subsect        1  Unused Ptes          0
  Flags              11  Sector Offset          0  Protection           1

Subsection 2 @ 81b85298
  ControlArea  81b85248  Starting Sector        2  Number Of Sectors   58
  Base Pte     e16c647c  Ptes In Subsect        b  Unused Ptes          0
  Flags              31  Sector Offset          0  Protection           3

Subsection 3 @ 81b852b8
  ControlArea  81b85248  Starting Sector       5a  Number Of Sectors    1
  Base Pte     e16c64a8  Ptes In Subsect        1  Unused Ptes          0
  Flags              51  Sector Offset          0  Protection           5

Subsection 4 @ 81b852d8
  ControlArea  81b85248  Starting Sector       5b  Number Of Sectors    2
  Base Pte     e16c64ac  Ptes In Subsect        1  Unused Ptes          0
  Flags              11  Sector Offset          0  Protection           1

Subsection 5 @ 81b852f8
  ControlArea  81b85248  Starting Sector       5d  Number Of Sectors    4
  Base Pte     e16c64b0  Ptes In Subsect        1  Unused Ptes          0
  Flags              11  Sector Offset          0  Protection           1
```

We can also explore these structures manually for the first three subsections.

```
1  0: kd> dt _CONTROL_AREA 81b85248
2  nt!_CONTROL_AREA
3     +0x000 Segment            : 0xe16c6440 _SEGMENT
4     +0x004 DereferenceList    : _LIST_ENTRY [ 0x0 - 0x0 ]
5     ...
6     +0x024 FilePointer        : 0x81b8ac70 _FILE_OBJECT
7     +0x028 WaitingForDeletion : (null)
8     +0x02c ModifiedWriteCount : 0
9     +0x02e NumberOfSystemCacheViews : 0
10
11 0: kd> dt _SUBSECTION 81b85248+30
12 nt!_SUBSECTION
13    +0x000 ControlArea        : 0x81b85248 _CONTROL_AREA
14    +0x004 u                  : __unnamed
15    +0x008 StartingSector     : 0
16    +0x00c NumberOfFullSectors : 2
17    +0x010 SubsectionBase     : 0xe16c6478 _MMPTE
18    +0x014 UnusedPtes         : 0
19    +0x018 PtesInSubsection : 1
20    +0x01c NextSubsection     : 0x81b85298 _SUBSECTION ->next
21
22 0: kd> dt _SUBSECTION 81b85248+30+20 //0x81b85298
23 nt!_SUBSECTION
24    +0x000 ControlArea        : 0x81b85248 _CONTROL_AREA
25    +0x004 u                  : __unnamed
26    +0x008 StartingSector     : 2
27    +0x00c NumberOfFullSectors : 0x58
28    +0x010 SubsectionBase     : 0xe16c647c _MMPTE
29    +0x014 UnusedPtes         : 0
30    +0x018 PtesInSubsection : 0xb
31    +0x01c NextSubsection     : 0x81b852b8 _SUBSECTION
32
33 0: kd> dt _SUBSECTION 81b85248+30+20+20 //0x81b852b8
34 nt!_SUBSECTION
35    +0x000 ControlArea        : 0x81b85248 _CONTROL_AREA
36    +0x004 u                  : __unnamed
37    +0x008 StartingSector     : 0x5a
38    +0x00c NumberOfFullSectors : 1
39    +0x010 SubsectionBase     : 0xe16c64a8 _MMPTE
40    +0x014 UnusedPtes         : 0
41    +0x018 PtesInSubsection : 1
42    +0x01c NextSubsection     : 0x81b852d8 _SUBSECTION
```

Further, we'll discuss this output in more detail

As it was mentioned earlier, the FILE_OBJECT structure has a very important structure called _SECTION_OBJECT_POINTERS. This structure addresses two CAs, one for a binary mapping type and second if the file is mapped as executable (the same file can be mapped as

both binary and executable). These CAs point to different Segments with their own PPTE tables. This structure is maintained by the FSD.

Subsections are allocated in virtual memory strongly after the CA structure. For example, if the Control Area describes executable view, then *ControlArea = ExAllocatePoolWithTag (NonPagedPool, sizeof(CONTROL_AREA) + (sizeof(SUBSECTION) * SubsectionsAllocated), 'iCmM').*

A few words about Subsections

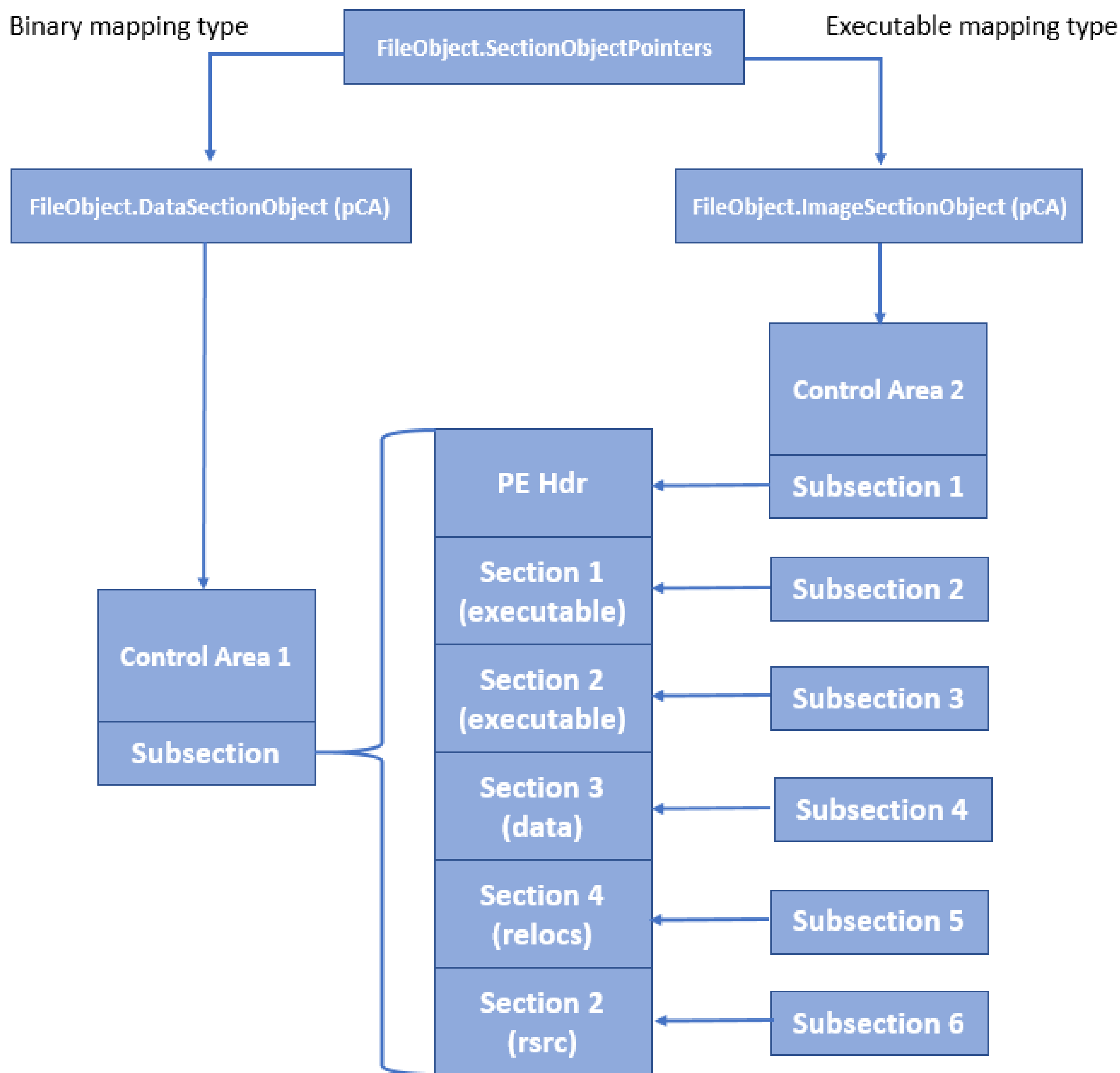
Subsection (_SUBSECTION) is a data structure containing the necessary information to calculate file offsets for the mapped file using the PPTEs. In case of a binary mapping type, there's only one subsection, but if the file is mapped as executable, then there're as many sections as there are in the executable. Since all the PTEs describing this subsection will have the same page protection bits (copy-on-write, read only, etc), it would be logically to maintain one data structure for all these PTEs. This data structure is called Subsection. All PPTEs point to the same corresponding subsection for both binary and executable mapping types. Moreover, the subsections contain the starting sector of the beginning of the PE's section. It's taken from the PE header as Raw_section_offset/SECTOR_SIZE. Also the subsection stores a pointer to the first PPTE in the segment's PPTE table and number of PTEs for this subsection (i e the number of virtual pages for this PE section, its VirtualSize rounded to a multiple of PAGE_SIZE). Having the address of the structure (executable mapping type), we can easily calculate the offset in the PE file, which this PPTE describes (as a distance between the base and current PTEs). If Pte is a pointer to PPTE, then the formula is.

(((PUCHAR)Pte - (PUCHAR)Subsection->SubsectionBase) / sizeof(PTE)) << PAGE_SHIFT + Subsection->StartingSector * SECTOR_SIZE

or for x86

(((PUCHAR)Pte - (PUCHAR)Subsection->SubsectionBase) / 4) << 12 + Subsection->StartingSector * SECTOR_SIZE

If Subsection is a ptr to the subsection, then the first PTE that describes it is FirstPte = &Subsection->SubsectionBase[0], and it's boundary, LastPte = &Subsection->SubsectionBase[Subsection->PtesInSubsection]. I e if X - the address of a PE file's subsection in virtual memory, then &Subsection->SubsectionBase[0] <= Pte < &Subsection->SubsectionBase[Subsection->PtesInSubsection].



Exploring the Segment structure

Unlike the Control Area structure that is designed to perform I/O operations with a file, the Segment stores information about a PE file that was taken from its PE header. In case of a binary file, this data isn't used. According to its purpose, a Segment also stores the Proto-PTE table (array) that addresses the offsets from the beginning of the mapped file through the Subsection structures. For example, if the VMM needs to load file data from the mapped file into virtual memory, it locates the corresponding Proto-PTE entry in the Segment table via not valid hardware PTE, which caused a page fault, from the page table. Next, using the Control Area structure and the calculated file offset, the VMM reads data from the file into virtual memory.

MmCreateSection creates segments using the following functions. It happens only if the file is mapped for the first time, otherwise the function gets a pointer to it via FileObject. Note that no matter how many sections have been created for the file object, there's always only one segment structure per type of mapping (binary, executable) for all of them. The same applies to Control Area structures, there's only one Control Area per type of mapping regardless of the number of created sections.

NTSTATUS MiCreateImageFileMap (IN PFILE_OBJECT File, OUT PSEGMENT Segment)

*NTSTATUS MiCreateDataFileMap (IN PFILE_OBJECT File, OUT PSEGMENT *Segment, IN PUINT64 MaximumSize, IN ULONG SectionPageProtection, IN ULONG AllocationAttributes, IN ULONG IgnoreFileSizing)*

As you can see *MiCreateImageFileMap* accepts fewer arguments, because it reads all the necessary information from the PE header of the executable file to be mapped. Description of other arguments you can find in *NtCreateSection*.

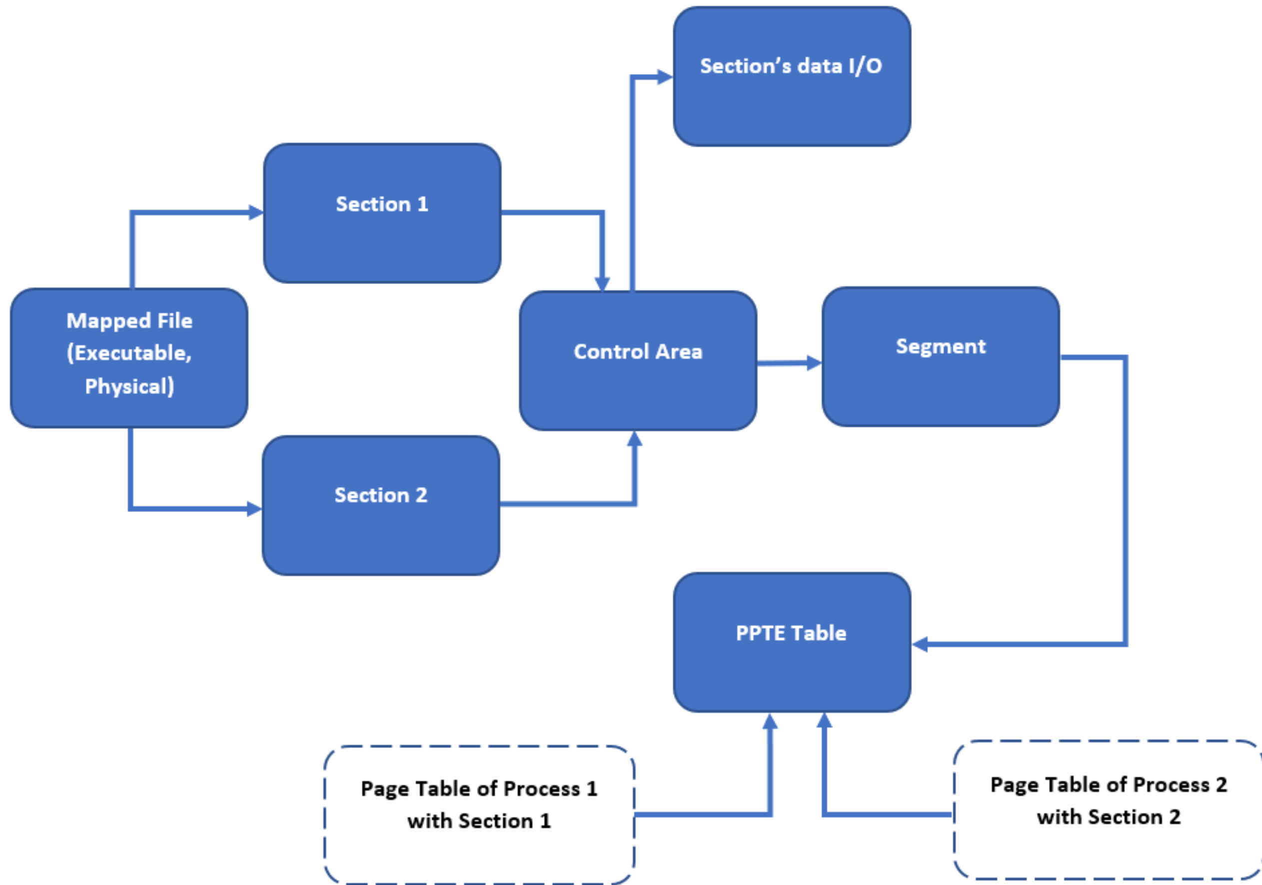
The following structure describes Segment.

```
1  typedef struct _SEGMENT
2  {
3      struct _CONTROL_AREA *ControlArea;
4      ULONG TotalNumberOfPtes;
5      ULONG NonExtendedPtes;
6      ULONG Spare0;
7
8      UINT64 SizeOfSegment;
9      MMPTE SegmentPteTemplate;
10     ...
11     union
12     {
13         SIZE_T ImageCommitment;     // for image-backed sections only
14         PEPROCESS CreatingProcess;  // for pagefile-backed sections only
15     } u1;
16
17     union
18     {
19         PSECTION_IMAGE_INFORMATION ImageInformation;    // for images only
20         PVOID FirstMappedVa;        // for pagefile-backed sections only
21     } u2;
22
23     PMMPTE PrototypePte;
24     MMPTE ThePtes[MM_PROTO_PTE_ALIGNMENT / PAGE_SIZE];
25 } SEGMENT, *PSEGMENT;
26
```

- **ControlArea** - pointer to the corresponding CA.
- **TotalNumberOfPtes** - roughly mapped_file_size/PAGE_SIZE.
- **SizeOfSegment** - size of the structure in bytes. *MiCreateImageFileMap* calculates it as SizeOfSegment = sizeof(SEGMENT) + (sizeof(MMPTE) * ((ULONG)TotalNumberOfPtes - 1)) + sizeof(SECTION_IMAGE_INFORMATION).

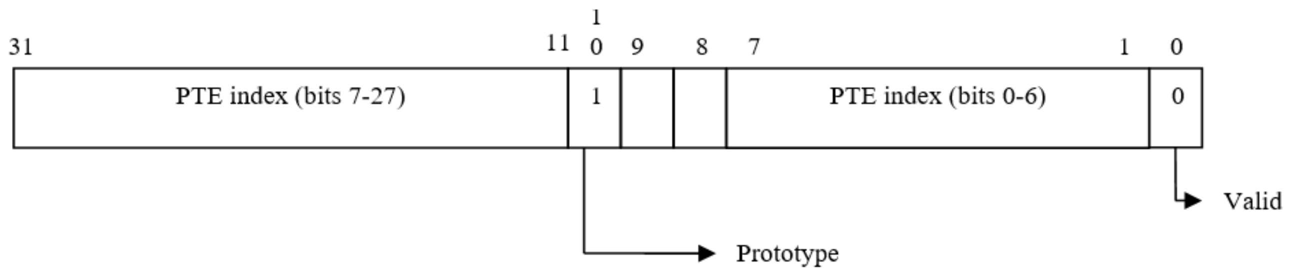
- **PrototypePte** - pointer to an array of PPTE. In fact, it's NewSegment->PrototypePte = &NewSegment->ThePtes[0].
- **ThePtes** - an array of PPTE, PPTE page table.

Perhaps the following image gives you a better understanding.



Behind the curtain of Section PTEs

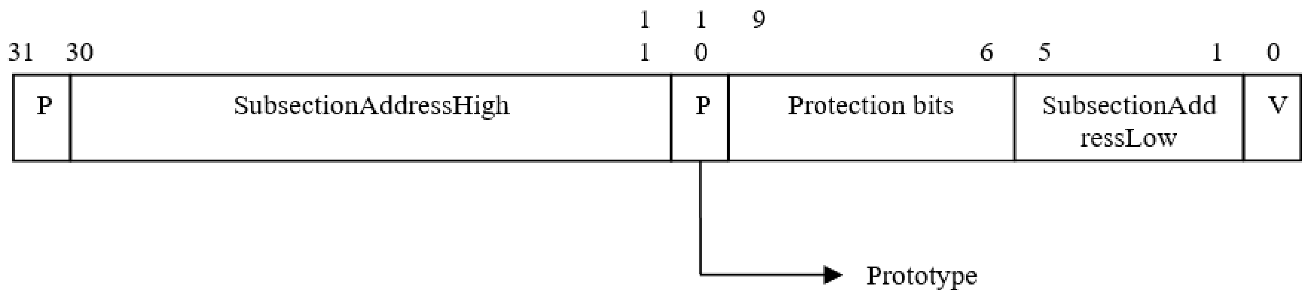
As it was mentioned many times earlier, PPTEs and hardware PTEs pointing to them are key things to understand the virtual addresses translation concept for the mapped sections properly. The difference between them is that the first is stored in the Segment object, while the second in the process's page table (hardware PTE). Both can be in two major states - valid and invalid (P bit in the structure). Zeroed bit means that the mapped page is absent in physical memory and signals the VMM that its content should be read from disk. If the P bit is true, this virtual page is resident in physical memory and no additional actions are required from the VMM. The invalid PTE has a flag signaling that this PTE points to PPTE, i e belongs to the memory mapped file. Once a thread tries to access an invalid memory page, a page fault exception occurs and the VMM exception handler analyzes the PTE to learn what kind of pages it describes. There are several types of invalid PTEs, but we won't discuss this topic here. Also note that in case of a resident virtual page the VMM stores a pointer to PPTE and its value in the PFN database. Let's take a look at the format of these structures. You can the format of the PTE pointing to PPTE in the following pic.




```
1  typedef struct _MMPTE_PROTOTYPE
2  {
3      ULONG Valid : 1;
4      ULONG ProtoAddressLow : 7;
5      ULONG ReadOnly : 1;  // read only access
6      ULONG WhichPool : 1;
7      ULONG Prototype : 1;
8      ULONG ProtoAddressHigh : 21;
9  } MMPTE_PROTOTYPE;
10
```

Once you get the ProtoIndex, you can calculate the PPTE address with this formula: PrototypePteAddress = *MmPagedPoolStart* + PrototypeIndex << 2.

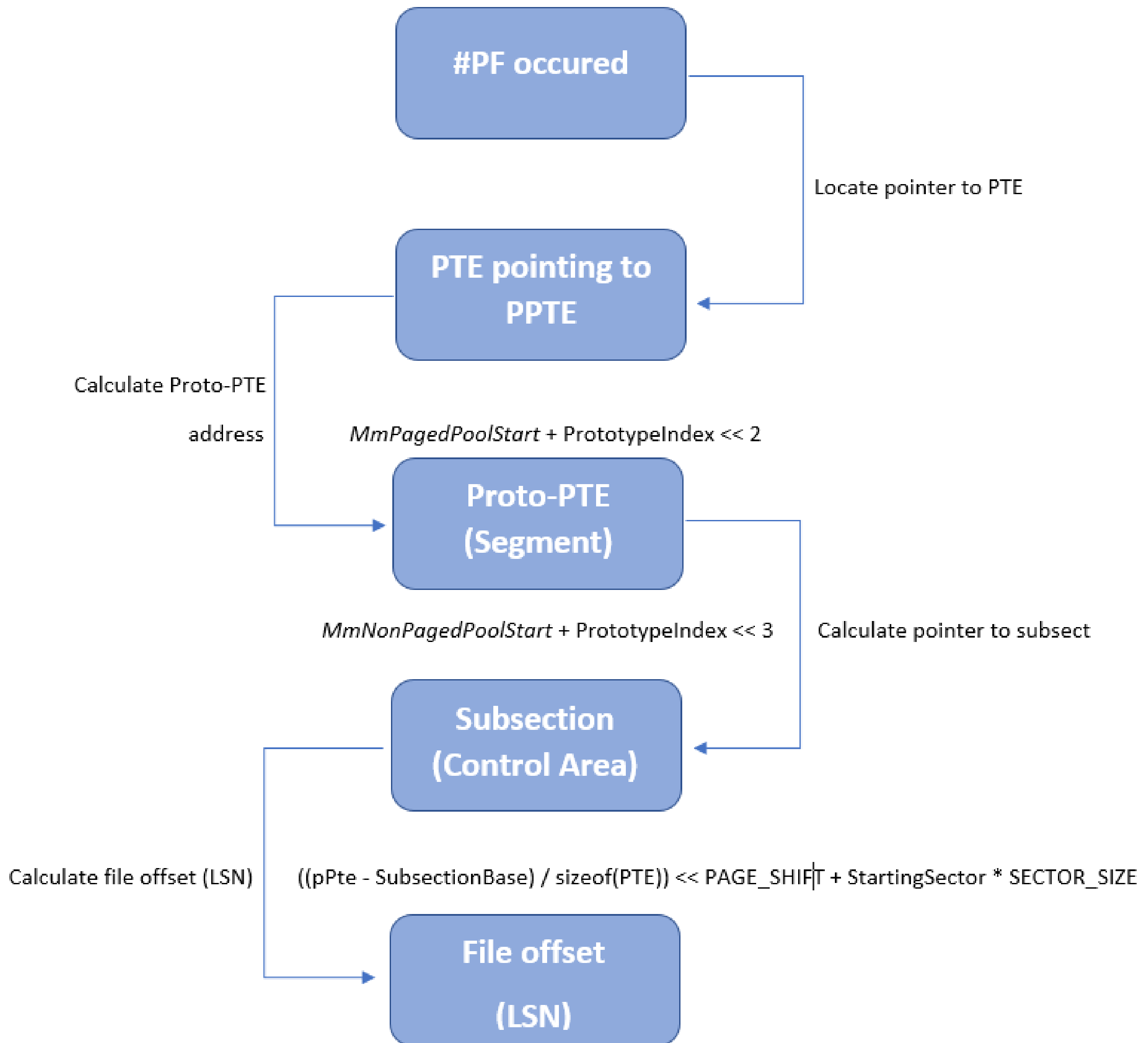
Below you can see PPTE format.




```
1  typedef struct _MMPTE_SUBSECTION {
2      ULONG Valid : 1;
3      ULONG SubsectionAddressLow : 4;
4      ULONG Protection : 5;
5      ULONG Prototype : 1;
6      ULONG SubsectionAddressHigh : 20;
7      ULONG WhichPool : 1;
8  } MMPTE_SUBSECTION;
9
```

SubsectionAddress = *MmSubsectionBase* + PrototypeIndex << 3. *MmSubsectionBase* is usually equal to *MmNonPagedPoolStart*, becausethe WhichPool bit is usually set to 1.

Now, using our knowledge, we can put all the pieces together and make a complete picture of the actions for getting file data when a thread tries to access a virtual page belonging to a mapped file.



A little practice

Let's get to the Proto-PTE table. Take a random process, dump its basic information and go to the table.

```
1  > !process 0 0
2
3  PROCESS ffffca0cabb485c0
4      SessionId: 0  Cid: 07d0    Peb: ef0697b000  ParentCid: 0338
5      DirBase: 13692000  ObjectTable: ffffb981ce2b7380  HandleCount: 149.
6      Image: VSSVC.exe
7
8  > !handle 0 3 ffffca0cabb485c0
9
10 0030: Object: ffffb981c8277a50  GrantedAccess: 00000003 (Inherit) Entry: ffffb981ce3c70c0
11 Object: ffffb981c8277a50  Type: (ffffca0ca8c71c50) Directory
12     ObjectHeader: ffffb981c8277a20 (new version)
13         HandleCount: 43  PointerCount: 1407269
14         Directory Object: ffffb981c7c16b20  Name: KnownDlls
15
16         Hash Address          Type                      Name
17         ---- -------          ----                      ----
18          00  ffffb981c8287c10 Section                   kernel32.dll
19
20 > !object ffffb981c8287c10
21
22 Object: ffffb981c8287c10  Type: (ffffca0ca8d0ada0) Section
23     ObjectHeader: ffffb981c8287be0 (new version)
24     HandleCount: 0  PointerCount: 1
25     Directory Object: ffffb981c8277a50  Name: kernel32.dll
26
27 > dt _SECTION ffffb981c8287c10 -r1
28
29 nt!_SECTION
30    +0x000 SectionNode      : _RTL_BALANCED_NODE
31    ...
32    +0x018 StartingVpn      : 0
33    +0x020 EndingVpn        : 0
34    +0x028 u1               : <unnamed-tag>
35       +0x000 ControlArea      : 0xffffca0c`aa900880 _CONTROL_AREA
36       +0x000 FileObject       : 0xffffca0c`aa900880 _FILE_OBJECT
37       +0x000 RemoteImageFileObject : 0y0
38       +0x000 RemoteDataFileObject : 0y0
39    +0x030 SizeOfSection    : 0xae000
40  ...
```

```
42  > !ca 0xffffca0c`aa900880
43
44 ControlArea  @ ffffca0caa900880
45   Segment      ffffb981c8297cb0  Flink      ffffca0cabb4d230  Blink        ffffca0caab19e00
46   Section Ref                 1  Pfn Ref                   6f  Mapped Views              2a
47   User Ref                   2b  WaitForDel                 0  Flush Count              a88
48   File Object  ffffca0caa900c90  ModWriteCount              0  System Views            348f
49   WritableRefs           c0000b
50   Flags  (a0) Image File
51
52       \Windows\System32\kernel32.dll
53
54 Segment @ ffffb981c8297cb0
55   ControlArea      ffffca0caa900880  BasedAddress  00007ffbcb640000
56   Total Ptes                     ae
57   Segment Size                ae000  Committed                      0
58   Image Commit                    2  Image Info     ffffb981c8297cf8
59   ProtoPtes        ffffb981c7f24a90
60   Flags  (c4820000) ProtectionMask
61
62 > dq ffffb981c7f24a90
63
64 ffffb981`c7f24a90  8a000000`37295121 00000000`2c624860
65 ffffb981`c7f24aa0  0a000000`2c625121 0a000000`2c626121
66 ffffb981`c7f24ab0  0a000000`2c627121 0a000000`2c628121 -> Subsections addresses (offsets)
67
```

We can go a bit deeper and calculate the offsets manually. To explore these structures it's better to take information from the cache slots as in the case of usual user-mode processes, the kernel can delay the creation of the Proto-PTE table until a thread addresses the mapped file data. I got a list of the cache slots on my system and select one describing the registry hive file NTUSER.DAT. Since it's a data file, there's only one subsection for its Control Area.

```
1  Vacb #186    0x81936170 -> 0xc7080000
2      File: 0x81749818
3      Offset: 0x00080000
4  \Documents and Settings\Art\NTUSER.DAT
5
6  We can see that this cache slot maps NTUSER.DAT with offset 0x80000 by address 0xc7080000.
7
8  0: kd> !pte 0xc7080000
9                 VA c7080000
10 PDE at   C0300C70        PTE at C031C200
11 contains 01CF0963       contains 0554A921
12 pfn 1cf0 -G-DA--KWEV    pfn 554a -G--A—KREV
13
14 According to the PTE content, it's valid, this means that we can restore its original PTE
15 pointing to the PPTE from the PFN database.
16
17 0: kd> !pfn 554a
18     PFN 0000554A at address 8107FEF0
19     flink       000018C8  blink / share count 00000001  pteaddress E15B7208
20     reference count 0001   Cached     color 0
21     restore pte 86D204CE  containing page        00496E  Active      P
22       Shared
23
24 We've got PPTE address E15B7208 and its original content 86D204CE. Translate it to the
25 subsection address and further we can get to the Segment and Control Area structures with formula
26 SubsectionAddress = MmNonPagedPoolStart + PrototypeIndex << 3.
27
28 86D204CE = 1 00001101101001000000 1 00110 0111 0
29                                   |                |
30                                   |                |->is ptr to subsection
31                                   |->is mapped file
32
33 000011011010010000000111 = DA407 * 8 + 81181000 = 6D2038 + 81181000 = 81853038
34
```

```
35 Print the subsection.
36
37 0: kd> dt _subsection 81853038
38 nt!_SUBSECTION
39    +0x000 ControlArea      : 0x81853008 _CONTROL_AREA
40    +0x004 u                : __unnamed
41    +0x008 StartingSector   : 0
42    +0x00c NumberOfFullSectors : 0x100
43    +0x010 SubsectionBase   : 0xe15b7008 _MMPTE
44    +0x014 UnusedPtes       : 0
45    +0x018 PtesInSubsection : 0x100
46    +0x01c NextSubsection   : (null)
47
48 And
49
50 +0x004 u                  : __unnamed
51       +0x000 LongFlags         : 0x60
52       +0x000 SubsectionFlags   : _MMSUBSECTION_FLAGS
53          +0x000 ReadOnly          : 0y0
54          +0x000 ReadWrite         : 0y0
55          +0x000 SubsectionStatic : 0y0
56          +0x000 GlobalMemory      : 0y0
57          +0x000 Protection        : 0y00110 (0x6) - MM_EXECUTE_READWRITE
58          +0x000 LargePages        : 0y0
59          +0x000 StartingSector4132 : 0y0000000000 (0)
60          +0x000 SectorEndOffset   : 0y000000000000 (0)
61
62 Print the control area.
63
64 0: kd> dt _control_area 0x81853008
65 nt!_CONTROL_AREA
66    +0x000 Segment          : 0xe1559ba0 _SEGMENT
67    +0x004 DereferenceList  : _LIST_ENTRY [ 0x0 - 0x0 ]
68    +0x00c NumberOfSectionReferences : 1
69    +0x010 NumberOfPfnReferences : 0xe5
70    +0x014 NumberOfMappedViews : 4
71    +0x018 NumberOfSubsections : 1 // The view is described by one subsection (binary file)
72    +0x01a FlushInProgressCount : 0
73    +0x01c NumberOfUserReferences : 0
74    +0x020 u                : __unnamed
75    +0x024 FilePointer      : 0x81749818 _FILE_OBJECT
76    +0x028 WaitingForDeletion : (null)
77    +0x02c ModifiedWriteCount : 0
78    +0x02e NumberOfSystemCacheViews : 4
```

Now we can calculate the file offset starting from which the file is mapped to the cache slot using this formula.

FileOffset_LSN = (((PUCHAR)Pte - (PUCHAR)Subsection->SubsectionBase) / 4) << 12 + Subsection->StartingSector * SECTOR_SIZE

(E15B7208 - E15B7008) / 4 *1000 + 0 = 80000, this value you can see in the VACB structure above (Offset: 0x00080000).

Here's another example.

```
Vacb #221     0x819364b8 -> 0xc8900000
    File: 0x81905d10
    Offset: 0x004c0000
\$Mft

As we can see this VACB structure describes the $Mft file, which was
mapped to the chache slot at 0xc8900000.

0: kd> !pte 0xc8900000
               VA c8900000
PDE at   C0300C88        PTE at C0322400
contains 01CF6963      contains 03AF5921
pfn 1cf6 -G-DA--KWEV    pfn 3af5 -G--A-KREV

0: kd> !pfn 3af5
    PFN 00003AF5 at address 810586F8
    flink       00001717  blink / share count 00000001  pteaddress E1449300
    reference count 0001   Cached     color 0
    restore pte 87CC64C2  containing page        0036F2  Active      P
      Shared

PPTE address is E1449300 и его исходное содержимое - 87CC64C2.

87CC64C2 = 1 00001111100110001100 1 00110 0001 0
000011111001100011000001 = F98C1 * 8 + 81181000 = 8194D608

dt _subsection 8194D608
nt!_SUBSECTION
   +0x000 ControlArea      : 0x8194d5d8 _CONTROL_AREA
   +0x004 u                : __unnamed
   +0x008 StartingSector   : 0
   +0x00c NumberOfFullSectors : 0x1000
   +0x010 SubsectionBase   : 0xe1448000 _MMPTE
   +0x014 UnusedPtes       : 0
   +0x018 PtesInSubsection : 0x1000
   +0x01c NextSubsection   : 0x818d1e18 _SUBSECTION

Calculate the offset with our formula (E1449300 - e1448000) / 4 * 1000 = 0x004C0000.

```

Now look at a more interesting case with PE files, Control Area of which has more than one subsection (one Subsection per one PE subsection). We can simplify our task and skip the first steps, starting with Control Areas. !memusage command can help us.

```
81783448    84      0     0     0     0     0  mapped_file( localspl.dll )
8185bd20     0     96     0     0     0     0  mapped_file( drw50009.vdb )
817ab818   444    128     0   340     0     0  mapped_file( ole32.dll )
81783148    12      0     0     0     0     0  mapped_file( pjlmon.dll )

```

We can see the addresses of the Control Area structures in the first column. Print it for ole32.dll.

```
1  0: kd> !ca 817ab818
2
3  ControlArea  @ 817ab818
4    Segment      e172eaa0  Flink      00000000  Blink        00000000
5    Section Ref         1  Pfn Ref          8f  Mapped Views       13
6    User Ref           14  WaitForDel        0  Flush Count         0
7    File Object  81847da0  ModWriteCount     0  System Views        0
8
9    Flags (90000a0) Image File HadUserReference Accessed
10                   |
11                   |->the file mapped as image
12
13       File: \WINDOWS\system32\ole32.dll
14
15 Segment @ e172eaa0
16   ControlArea       817ab818  BasedAddress   774e0000
17   Total Ptes             13d
18   WriteUserRef             0  SizeOfSegment   13d000
19   Committed                0  PTE Template   862a8c3a
20   Based Addr        774e0000  Image Base            0
21   Image Commit             7  Image Info     e172efd0
22   ProtoPtes         e172ead8
23
24
```

```
25 Subsection 1 @ 817ab848
26   ControlArea  817ab818  Starting Sector        0  Number Of Sectors    2
27   Base Pte     e172ead8  Ptes In Subsect        1  Unused Ptes          0
28   Flags              11  Sector Offset          0  Protection           1
29
30 Subsection 2 @ 817ab868
31   ControlArea  817ab818  Starting Sector        2  Number Of Sectors  8f8
32   Base Pte     e172eadc  Ptes In Subsect      11f  Unused Ptes          0
33   Flags              31  Sector Offset          0  Protection           3
34
35 Subsection 3 @ 817ab888
36   ControlArea  817ab818  Starting Sector      8fa  Number Of Sectors   30
37   Base Pte     e172ef58  Ptes In Subsect        6  Unused Ptes          0
38   Flags              31  Sector Offset          0  Protection           3
39
40 Subsection 4 @ 817ab8a8
41   ControlArea  817ab818  Starting Sector      92a  Number Of Sectors   33
42   Base Pte     e172ef70  Ptes In Subsect        7  Unused Ptes          0
43   Flags              51  Sector Offset          0  Protection           5
44
45 Subsection 5 @ 817ab8c8
46   ControlArea  817ab818  Starting Sector      95d  Number Of Sectors    c
47   Base Pte     e172ef8c  Ptes In Subsect        2  Unused Ptes          0
48   Flags              11  Sector Offset          0  Protection           1
49
50 Subsection 6 @ 817ab8e8
51   ControlArea  817ab818  Starting Sector      969  Number Of Sectors   69
52   Base Pte     e172ef94  Ptes In Subsect        e  Unused Ptes          0
53   Flags              11  Sector Offset          0  Protection           1
54
```

For clarity, copy the results to the table. If we open the PE file in the Cerbero PE Insider tool, we'll see that it has five sections. Note that the first subsection is allocated for the PE header.

	Information from the Subsections				Information from the PE header			
#	Starting sector	Number of sectors	Number of PTEs in subsection	Protection	Raw offset	Raw Size	Virtual Size	Protection
1 Hdr	0	2	1	MM_READONLY	0	-	1000	-
2 .text	2	8f8	11f	MM_EXECUTE_READ	400	11f000	11ef5e	EXECUTE/READ
3 .orpc	8fa	30	6	MM_EXECUTE_READ	11F400	6000	5F0E	EXECUTE/READ
4 .data	92a	33	7	MM_WRITECOPY	125400	6600	69FC	INIT DATA/READ/ WRITE
5 .rsrc	95d	C	2	MM_READONLY	12BA00	1800	17F8	INIT DATA/READ
6 .reloc	969	69	E	MM_READONLY	12D200	D200	D0D4	INIT/DISCARD/ READ

- We can see that the subsections number 2-3 are executable and match the PE sections .text and .orpc. This means that they address the PPTEs with code sections. The 4th subsection describes global data and has the copy-on-write protection. The rest are only available for read access.
- The first subsection describes the file header and starts at offset 0. On disk, the PE header fits in two sectors and occupies one page of virtual memory. Thus, it can be described by one PPTE.
- The second subsection describes the first PE file's code section. It starts from the second sector (0x400 / SECTOR_SIZE == 2). The virtual size of this section is 0x11EF5E, i e rounding it up to a multiple of the page size, 0x11EF5E + 0xA2 = 0x11F000 / PAGE_SIZE = 0x11F. This value matches the number of PTEs in the subsection. We can calculate number of sectors for the section from the header Raw size, 0x11F000 / 200 = 0x8F8 that equals the number of sectors for this section.
- The third section also contains code and starts with sector number 0x11F400 / 0x200 = 0x8FA. The size is 0x6000 bytes (in this case we take the physical size as it larger than virtual), 0x6000/0x1000 = 6 PTEs.
- The 4th section starts from 0x125400 / 0x200 = 0x92A, the size 0x7000 / 0x1000 = 7 PTEs.
- The 5th section starts from 0x12BA00 / 0x200 = 0x95D, размер 0x2000 / 0x1000 = 2 PTEs.
- The 6th, 0x12D200 / 0x200 = 0x969, размер 0xE000 / 0x1000 = 0xE PTEs.

Let's check out the formula mentioned above in practice. Take the third subsection, which describes the ole32.dll section starting at offset 0x8FA.

```
0: kd> dt _subsection 817ab888 SubsectionBase
nt!_SUBSECTION
   +0x010 SubsectionBase : 0xe172ef58 _MMPTE

Get content of the first PPTE that describes this section.

0: kd> dd 0xe172ef58 l1
e172ef58  0c779121

It's valid, then

0: kd> !pfn c779
    PFN 0000C779 at address 8112B358
    flink       000006E7  blink / share count 00000007  pteaddress E172EF58
    reference count 0001   Cached     color 0
    restore pte 862A8C62  containing page        00B8A9  Active      P
      Shared

862A8C62 = 1 00001100010101010001 1 00011 0001 0;
00001100010101010001 = C5511 * 8 + 81181000 = 817AB888, got an address of our subsection.

Now having a pointer to the PPTE, we can get the file offset that it describes.
Use our formula,
(((PUCHAR)Pte - (PUCHAR)Subsection->SubsectionBase) / 4) << 12 + Subsection->StartingSector * SECTOR_SIZE.
(E172EF58 - 0xE172EF58) = 0 + 8fa * 200 = 11F400, this value matches the one located in the section header.

#define MM_ZERO_ACCESS         0  // this value is not used.
#define MM_READONLY            1
#define MM_EXECUTE             2
#define MM_EXECUTE_READ        3
#define MM_READWRITE           4  // bit 2 is set if this is writable.
#define MM_WRITECOPY           5
#define MM_EXECUTE_READWRITE   6
#define MM_EXECUTE_WRITECOPY   7

typedef struct _MMSUBSECTION_FLAGS {
    unsigned ReadOnly : 1;
    unsigned ReadWrite : 1;
    unsigned SubsectionStatic : 1;
    unsigned GlobalMemory: 1;
    unsigned Protection : 5; //MM_* macros
    unsigned Spare : 1;
    unsigned StartingSector4132 : 10;   // 2 ** (42+12) == 4MB*4GB == 16K TB
    unsigned SectorEndOffset : 12;
} MMSUBSECTION_FLAGS;
```

Dispatching #PF exceptions for mapped files

As we know the I/O Manager and VMM minimize the performance overhead by performing most of their operations asynchronously and by demand. Probably Windows developers don't know about this principle, because synchronous operations are default behavior for Windows API while the situation with Native and kernel API is reversed. This principle also applies to section objects. When *MapViewOfFile* Windows API returns control to the caller thread, it doesn't mean that mentioned subsystems copy the file data to virtual memory for RW just as if a thread modified the mapped file data in virtual memory, it doesn't mean that these changes will be immediately flushed to the physical file. Instead, the VMM delays the actual I/O operation until a thread of the process tries to access the file data by reading virtual memory. Once it happened, the **#PF** exception occurs and the VMM initiates an I/O operation to read file data into virtual memory. The common work in this case falls on the shoulders of *MiDispatchFault* function.

There're several possible situations for the sections describing file-backed data. Note that the section PPTE can be in the states inherent in a hardware PTE.

- The PPTE is invalid and points to a subsection. In this case, the VMM needs to load the corresponding file data from disk into physical memory. The *MiResolveMappedFileFault* function is responsible for this, but an actual I/O operations is initiated by *MiDispatchFault*.
- The PPTE is valid and points to a page frame. The VMM just need to fill the hardware PTE with this frame number.
- The PPTE marked as copy-on-write.

MiDispatchFault calls *MiResolveProtoPteFault* passing it a pointer to PTE and PPTE. *MiResolveProtoPteFault* works with PPTE as well as with usual PTE, because PPTE can be in the same states as hardware PTE. The function starts by validating the PPTE, i e whether it's located in physical memory or not.

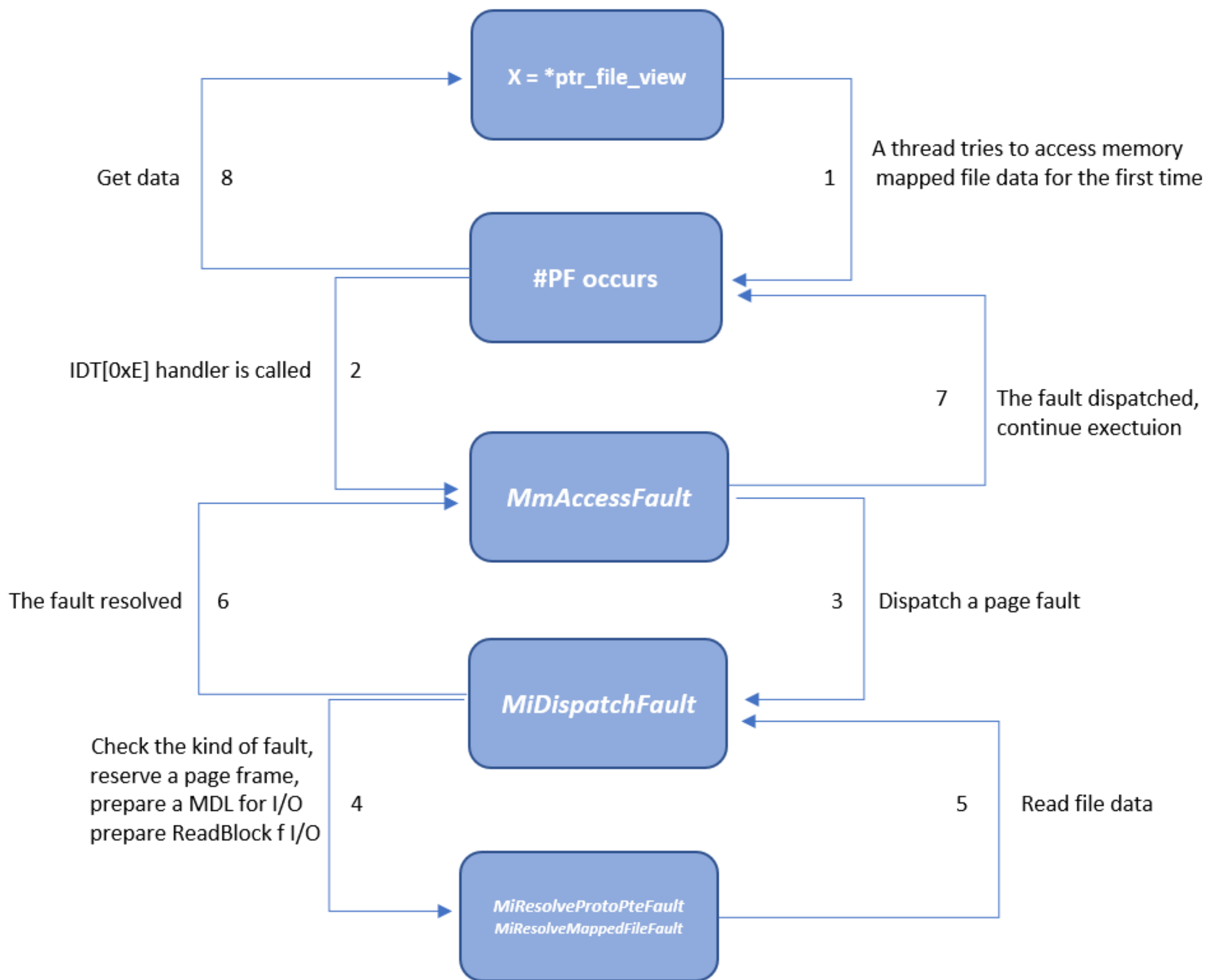
```
1  if (TempPte.u.Hard.Valid)
2  {
3      //get a PFN from the PTE
4          PageFrameIndex = MI_GET_PAGE_FRAME_FROM_PTE (&TempPte);
5      //get a PFN item from the database
6          Pfn1 = MI_PFN_ELEMENT (PageFrameIndex);
7      //increment reference count (share count) of the PTE pointing to the PPTE array
8          Pfn1->u2.ShareCount += 1;
9          return MiCompleteProtoPteFault (StoreInstruction,
10                                         FaultingAddress,
11                                         PointerPte,
12                                         PointerProtoPte,
13                                         OldIrql,
14                                         unk);
15 }
```

After checking the rights access to the page, the function checks the case when the PPTE is marked as Demand Zero and its hardware PTE marked as copy-on-write. In this case, the VMM resolves the fault by calling *MiResolveDemandZeroFault* and passing it a pointer to real PTE. Further, *MiResolveProtoPteFault* makes the PPTE valid, it can be in the following states: Demand Zero, Transition, Page File, Pagefile-backed, File-backed.

```
if (TempPte.u.Soft.Prototype == 1)
{
//File-backed
        status = MiResolveMappedFileFault (PointerProtoPte,
                                           ReadBlock,
                                           Process,
                                           OldIrql);
     }
    else if (TempPte.u.Soft.Transition == 1) {
//Transition
        status = MiResolveTransitionFault (FaultingAddress,
                                           PointerProtoPte,
                                           Process,
                                           ...);
    }
    else if (TempPte.u.Soft.PageFileHigh == 0) {
//Demand Zero
        status = MiResolveDemandZeroFault (FaultingAddress,
                                           PointerProtoPte,
                                           Process,
                                           ...);
    }
    else {
    //Pagefile-backed
        status = MiResolvePageFileFault (FaultingAddress,
                                         PointerProtoPte,
                                         CapturedPteContents,
                                         ...);
}

```

MiResolveProtoPteFault and *MiResolveMappedFileFault* functions perform important steps: reserve a page frame (physical page), initializes the corresponding entry in the PFN database, prepare a MDL structure and a special ReadBlock structure for further disk read operation. You can see the entire process in the following diagram.



Inside the Page Writers

In the last part of this blog post we're gonna discuss the Mapped Page Writer subsystem (thread), which is a part of the Modified Page Writer subsystem (or just thread). At this point we already know how the VMM and I/O manager read mapped file data to process's virtual memory in order to provide access to it. But what about writing file data? As was mentioned above, the VMM minimizes the performance overhead by performing its operations that involve disk I/O by demand. That's why the actual read operation on a memory mapped file only happens when a thread tries to access a file view and not when executing *MapViewOfFile*.

The VMM has two system threads called *MiModifiedPageWriter* and *MiMappedPageWriter*. In fact, *MiModifiedPageWriter* just creates *MiMappedPageWriter* thread and shifts the rest of work to *MiModifiedPageWriterWorker*. These last two functions (*MiModifiedPageWriterWorker* and *MiMappedPageWriter*) are two infinite loops that can be called Modified Page Writer, because they implement all its functionality. The first one is responsible for gathering information about modified pages belonging to the page file

(*MiGatherPagefilePages*) and about modified pages belonging to the mapped files (*MiGatherMappedPages*). It also adjusts the frequency of the flushing operations or how often modified pages will be written to disk. The second thread takes the information prepared by *MiModifiedPageWriterWorker* and performs the actual disk write operation (for mapped files).

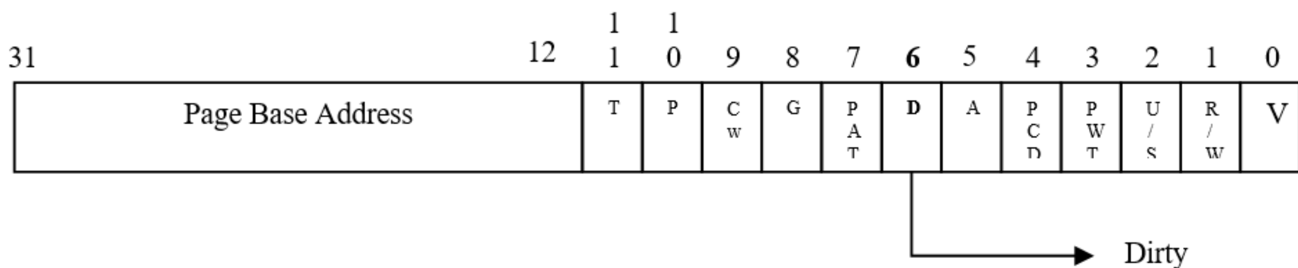
The main part of *MiModifiedPageWriterWorker* is an infinite loop with waiting on the *MiMappedPagesTooOldEvent* event. This event can be set in several circumstances and adjusts the frequency of performing flushing. To provide a fixed time frequency of flushing, the VMM uses a timer object and a DPC object (*MiModifiedPageWriterTimerDpc*), i e the DPC handler *MiModifiedPageWriterTimerDispatch* calls every time a timer expires. Since this handler is executed with high IRQL DPC_DISPATCH (2, the scheduler level), the system ensures its operation in privileged mode. The *MmInitSystem* function initializes this object during the system startup and when *MiModifiedPageWriterWorker* need to gather dirty memory pages for the first time, it sets this timer. The timer is set for 3 seconds.

```
KeInitializeDpc ( <-- MmInitSystem
    &MiModifiedPageWriterTimerDpc,
    MiModifiedPageWriterTimerDispatch,
    NULL
    )

KeSetTimerEx ( <-- MiModifiedPageWriterWorker if MmMappedFileHeader.ListHead is empty
    &MiModifiedPageWriterTimer,
    MiModifiedPageLife,
    0,
    &MiModifiedPageWriterTimerDpc
    )

```

Forgot to mention that physical memory pages (frames) that were modified since the section was mapped are called Dirty. The CPU sets this bit at the first write operation to the page. Once the VMM processes this page in a certain way, it resets this bit. Without it, the VMM wouldn't be able to track the changes made by the thread on the mapped page and synchronize them with the file data on disk.



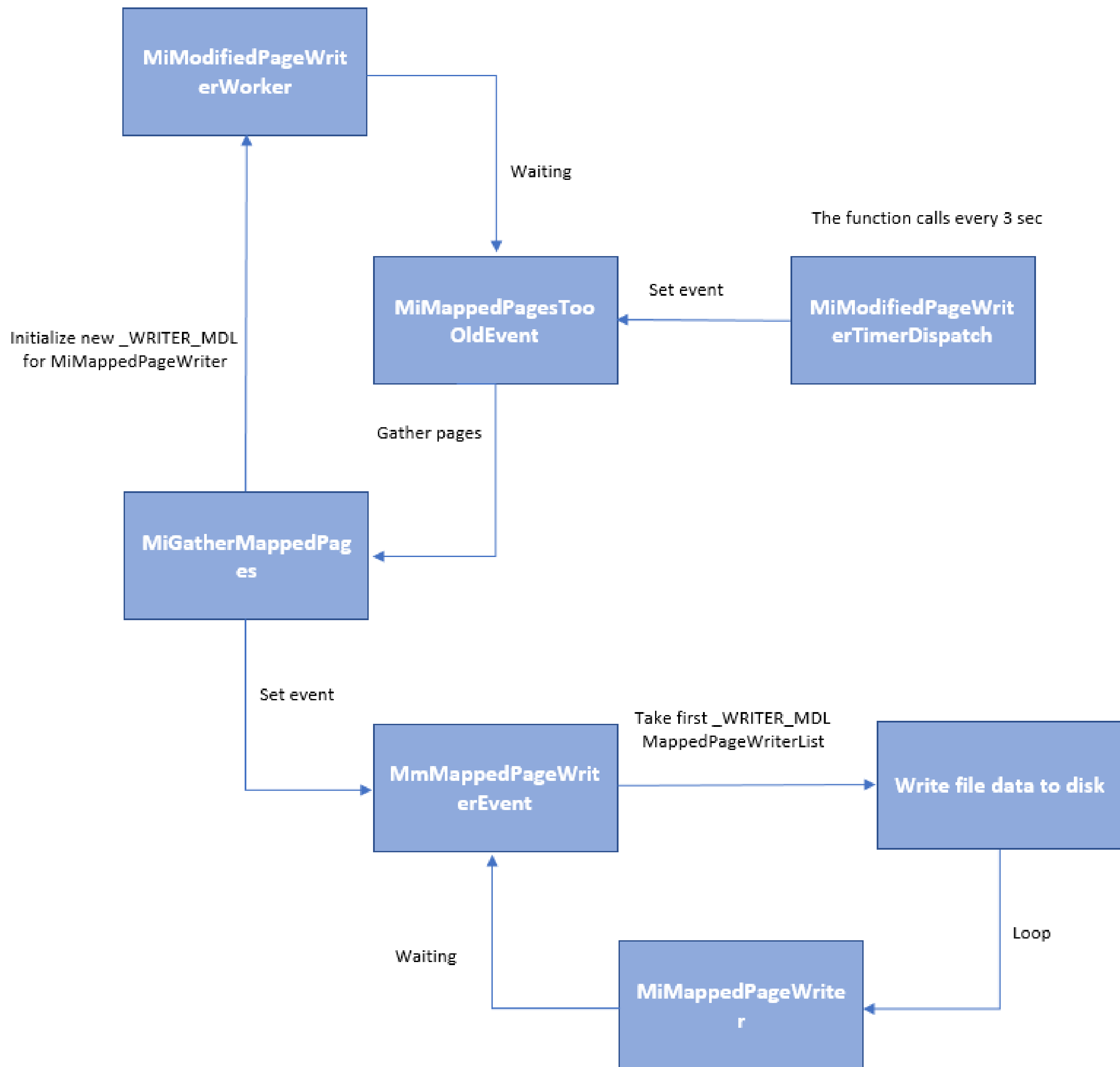
For the convenience of flushing file data and, in order to reduce overhead, *MiMappedPageWriter* doesn't flush every dirty page separately, instead, *MiGatherMappedPages* gathers in the packet (MMMOD_WRITER_MDL_ENTRY) a set of dirty frames that belong to the same section and are adjacent to this dirty frame. The

MMMOD_WRITER_MDL_ENTRY structure describes a set of dirty PFNs that should be written to disk. In fact, the VMM uses two MDL lists, one for a paging file and another for sections. The pool of these MDL items is allocated in the *NtCreatePagingFile* function that is responsible for creating page files. The same for memory mapped files - *MmMappedFileHeader*.

As it was mentioned above, *MiMappedPageWriter* is responsible for initiating disk write operation, here's its pseudocode, in which the details are omitted.

```
1  VOID
2  MiMappedPageWriter (
3      IN PVOID StartContext
4      )
5  {
6      NTSTATUS Status;
7      PETHREAD CurrentThread;
8      IO_PAGING_PRIORITY IrpPriority;
9      PMMMOD_WRITER_MDL_ENTRY ModWriterMdlEntry;
10
11     CurrentThread = PsGetCurrentThread ();
12
13     KeSetPriorityThread (&CurrentThread->Tcb, 17); // LOW_REALTIME_PRIORITY + 1
14
15     for(;;)
16     {
17
18         KeWaitForSingleObject (&MmMappedPageWriterEvent, // This event is set by MiGatherMappedPages
19                                WrVirtualMemory,          // when it initialized the next ModWriterMdlEntry
20                                KernelMode,
21                                FALSE,
22                                NULL);
23
24         if (!IsListEmpty (&MmMappedPageWriterList)) // Make sure that the list isn't empty
25         {
26
27             ModWriterMdlEntry = (PMMMOD_WRITER_MDL_ENTRY)RemoveHeadList (
28                                 &MmMappedPageWriterList);
29
30             //initiate disk write operation (flush section data)
31             Status = IoAsynchronousPageWrite (ModWriterMdlEntry->File, &ModWriterMdlEntry->Mdl,
32                         &ModWriterMdlEntry->WriteOffset, MiWriteComplete, ModWriterMdlEntry,
33                             IrpPriority, &ModWriterMdlEntry->u.IoStatus, &ModWriterMdlEntry->Irp);
34         }
35     }
36 }
37
```

Please take a look at the following diagram for understanding the entire process.



However, a thread can force the VMM to write modified data to disk immediately. Windows API provides applications with the *FlushViewOfFile* function. The VMM's internal function *MiFlushSectionInternal* is responsible for flushing section data and calls *IoAsynchronousPageWrite* to perform disk write operation.

Instead of conclusion

Thank you for your attention and hope you enjoyed the blog post. Windows Sections is quite a difficult topic, especially, for beginners, because you should already have an idea of other Windows kernel subsystems to understand it properly.

If you have any comments or remarks, please let me know and feel free to contact. I'm going to cover several other topics on the Windows VMM internals such as the PFN database, hyperspace and virtual address translation.