RELSOFT TECHNOLOGIES

Introduction to NT Internals

Part 1: Processes, Threads, Fibers and Jobs

Part 1: Processes, Threads, Fibers and Jobs

© 2004 Alex Ionescu
Relsoft Technologies
http://www.relsoft.net

Table of Contents

1. Introduction

This part of the series will deal on how exactly your executable runs, how it gets mapped into memory, what structures describe it, and how NT lets your code run on the CPU. Some knowledge about user/kernel mode and Native API will be helpful, so be sure to read my previous article first.

The structures apply to the latest version of Client Windows, which at the time of writing is Windows XP Service Pack 2 will soon be coming out, but it remains beta for now, and some features might be changed. For example, the current beta does not have support for FLS (Fiber Local Storage). However, since those values are at the end of the structure that should contain them, this won't matter much.

It is however very important to keep in mind that these structures are, for the most part, **undocumented and unsupported**. Everything written in this article **should be valid** for Windows XP but you might notice certain differences with older versions. In some cases, the **offsets themselves might be different.** In this case, **you will need two structure declarations**. As such, **compatibility is not guaranteed with pre-XP versions of NT.** Also, **some offsets are different in Windows 2003, but not Windows Longhorn.**

All the information contained herein is shown **strictly for educational purposes.** It does not constitute any breach of a Microsoft Non-Disclosure Agreement or violate DMCA Copyrights or Intellectual Property. **HOWEVER, YOU SHOULD NOT CREATE ANY COMMERCIAL WORK BASED ON THIS DOCUMENT BECAUSE THIS VIOLATES MICROSOFT CODING STANDARDS AND WILL RENDER YOUR PROGRAM DANGEROUS FOR THE USER AND UNCOMPATIBLE.**

2. Structures and Terminology

Before exploring the details of the structures, it is important to have a basic understanding of the different words and structures that describe executable code on an NT OS. You've probably heard the term *Process* a lot to talk about a program, and also the term *Thread*. It is important to distinguish the two. Furthermore, the term Image, which will be used at many places along this text refers to an executable file, not a picture.

The main thing to understand about a *Process* is that it is **not executable code**. It is simply a **container of *Threads***. That is to say that is gives structural information to the in-memory copy of your executable program, such as which memory is currently allocated, which program is running, how much memory it is using, etc. The *Process* however, does not contain any code. It simply allows the OS (and the user) to know to which executable program a certain *Thread* belongs to. It also contains all the handles and security rights and privileges that *Threads* create.

Therefore, code actually runs in *Threads*. This means that even a non-multithreaded application has a *Thread*. It is not because you don't call the *CreateThread* API function that your code is somehow running in the *Process*; it isn't, and the PE Loader has taken the care to load all your code into a single *Thread*. So what is a *Thread* exactly? As said before, it is the piece of binary code that is running on a CPU. It is defined by a *context*, that is to say the current state of the CPU while running this code. Most importantly, *context* includes all the CPU registers (eax, ebx, etc) and their values. This is of course because Windows constantly switches between threads; a CPU cannot execute two threads in the same time, and only has one group of registers. Therefore, when switching back and forth between threads, the *context* is changed.

Apart from the general *Process* information, which all *Threads* share, there is a lot of data unique to a *Thread*. Threads, for example, can each have their own Structured Error Handling, their own DLL Error value, or perhaps more importantly, their own priority. The Kernel also keeps more internal information about threads, including the Priority (how much CPU time it can take) and the Affinity (which CPUs can it use), as well as the System Call Table to use, and graphical information for WIN32K.SYS.

We've mentioned that a Process is a container of Threads, but it is a little known fact that there can actually be a container of Processes. This is called a Job. Jobs are completely managed by the OS, and do not exist for a CPU, in the sense that a CPU will be aware of Threads and Processes through the OS (since it must manage Process Memory Space and Thread Scheduling), but not of Jobs. Jobs give the advantage that they can define a sandbox environment for one or multiple processes. For example, you can create a Job Object that doesn't have access to Window Handles (hWnds) that are outside of the Job. You can also specify Memory Limits for Jobs, or even Execution Time Limits. Thanks to the power or Jobs, you have a limited, although useful way to secure the system against one, or a group of Processes. Although Job Limits can be read and set from APIs, the Executive Job structure that will be shown is an easier way to get a complete look on the whole Job Object.

Thanks to a new feature in Windows 2000 and up, Threads can also become containers of code called Fibers. Fibers, unlike Threads, are not managed by the Kernel nor have any direct relationship with the CPU. They are also different from Jobs because not even the Executive OS manages them. Fibers are actually Threads that are managed by the application that created them (by the thread that created them). As such, all fiber switching, creation, and deletion is managed by the Thread. The OS provides some simple APIs to change Contexts (thereby changing fibers), but all the APIs are User-Mode and could almost be implemented by the software. Fibers created by the same Thread all share the same information, except for stack (variables) and registers. Note also that the Fiber is not an Object, but is included in this article for completeness.

Here is a short table discussing all the structuring of Code on Windows:

Name	***Function***	***Contained By***	***Managed By***	***Object***
Job	Provide a secure environment for a group of Processes	OS Executive	OS Executive	EJOB
Process	Provide a Memory Space and access to Executive Objects for the Threads it contains.	OS Executive *or* Job	OS Kernel *and* OS Executive	EPROCESS *and* KPROCESS
Thread	Execute the code it contains.	Process	OS Kernel	KTHREAD *and* ETHREAD
Fiber	Execute the code it contains	Thread	OS User-Mode *and* Thread	- *

* Fiber information is contained in a User-Mode Structure called FIBER_CONTEXT

3. Image File Execution (Process Creation)

Before NT wants to know anything about your file or the code inside it, it must first be loaded into memory. This is the job of the *PE Loader*, where PE stands for *Portable Executable*, which is the EXE File Format used by Windows NT. Delving into the PE format is beyond the scope of this article (there are many references online), but it suffices to say that all the API Imports are located in a special table. (A slight hiccup here: VB applications, because they use a runtime, only import APIs from the runtime. The APIs that you declare are actually saved in a special VB structure that the runtime reads when you call them, not the OS. This is NOT the case when using Type Libraries (TLB) to declare your APIs, which is why they offer such a huge improvement).

The PE Loader must therefore first load the Import Address Table (IAT), so that your program knows the entrypoints of the API call. Let's say you are calling the *Beep* API. This API is located in kernel32.dll, so the first thing that the PE Loader will do is call *LoadLibrary* with kernel32.dll as a parameter (this is purely a theoretical example, since kernel32.dll is ALWAYS mapped into a process, namely because *LoadLibrary* itself needs to be used by the PE Loader). This will return a handle to the library, which is actually the address in memory where it was loaded. Then, the PE loader will call *GetProcAddress,* and give it the name of the *Beep* function. The entrypoint will be returned, and it is added to the base address of the library to give the final pointer to this API call (the equivalent of AddressOf). This value is now saved in the IAT, at the position where the compiled EXE expects to find this pointer (the compiler makes all the offsets when compiling). This process continues for each DLL that you are importing, and the PE Loader will also perform the same for DLL files, since they are also part of the PE Format.

Now that the PE Loader has loaded the file in memory, it is ready to allow run the code. But what actually happens before the PE Loader even comes into play? A variety of Native API is used to create the process and setup the environment and all the structures. If you feel up for it, you can read about it in Chapter 10 and 11. One should know however the main steps that are done. Basically, after the program is in memory (not in exact order):

- The KPROCESS structure is created.
- The EPROCESS structure is created.
- The first thread (along with KTHREAD and ETHREAD) is created.
- The Initial CPU registers and context is created.
- The K/EPROCESS/THREAD structures are filled out with current CPU state and threading settings.
- The PEB and TEB are created, with specific data about the user-mode process and initial main thread.
- The Environment Settings are created and read from registry or the command-line (information such as the Windows Path, the arguments, etc).
- The initial thread is attached to a thread launcher stub.
- The initial thread is resumed.

The structures are constantly updated with new data when it changes. Any new threads created by the process will also generate the creation of new K/ETHREAD structures and TEBs. I've mentioned a lot of structures and acronyms that you are probably not aware of. The next chapters will document each one of these and explain their use.

4. User-Mode Process Structures

4.1 Process Environment Block (PEB)

The PEB is the Process Environment Block. It is a high-level user-mode structure that contains some important information about the current process:

```
Public Type PEB
   InheritedAddressSpace          As Byte
   ReadImageFileExecOptions       As Byte
   BeingDebugged                  As Byte
   Spare                          As Byte
   Mutant                         As Long
   SectionBaseAddress             As Long
   ProcessModuleInfo              As Long ' // PEB_LDR_DATA
   ProcessParameters              As Long ' // RTL_USER_PROCESS_PARAMETERS
   SubSystemData                  As Long
   ProcessHeap                    As Long
   FastPebLock                    As Long ' // CRITICAL_SECTION
   AcquireFastPebLockRoutine      As Long
   ReleaseFastPebLockRoutine      As Long
   EnvironmentUpdateCount         As Long
   KernelCallBackTable            As Long ' // WIN32K_CALLBACK
   EventLogSection                As Long
   EventLog                       As Long
   ExecuteOptions                 As Long
   FreeList                       As Long ' // PEB_FREE_BLOCK
   TlsBitMapSize                  As Long
   TlsBitMap                      As Long ' // RTL_BITMAP
   TlsBitMapData                  As LARGE_INTEGER
   ReadOnlySharedMemoryBase       As Long
   ReadOnlySharedMemoryHeap       As Long
   ReadOnlyStaticServerData       As Long
   InitAnsiCodePageData           As Long
   InitOemCodePageData            As Long
   InitUnicodeCaseTableData       As Long
   NumberOfProcessors             As Long
   NtGlobalFlag                   As Long ' // GLOBAL_FLAGS
   Padding                        As Long
   CriticalSectionTimeout         As LARGE_INTEGER
   HeapSegmentReserve             As Long
   HeapSegmentCommit              As Long
   HeapDeCommitTotalFreeThreshold As Long
   HeapDeCommitFreeBlockThreshold As Long
   NumberOfHeaps                  As Long
   MaxNumberOfHeaps               As Long
   ProcessHeapsList               As Long
   GdiSharedHandleTable           As Long ' // GDI_HANDLE_TABLE
```

```
    ProcessStarterHelper          As Long
    GdiInitialBatchLimit          As Long
    LoaderLock                    As Long ` // CRITICAL_SECTION
    NtMajorVersion                As Long
    NtMinorVersion                As Long
    NtBuildNumber                 As Integer
    NtCSDVersion                  As Integer
    PlatformId                    As Long
    Subsystem                     As Long
    MajorSubsystemVersion         As Long
    MinorSubsystemVersion         As Long
    AffinityMask                  As Long ` // KAFFINITY
    GdiHandleBuffer(33)           As Long
    PostProcessInitRoutine        As Long
    TlsExpansionBitmap            As Long
    TlsExpansionBitmapBits(127)   As Byte
    SessionId                     As Long
    AppCompatFlags                As LARGE_INTEGER
    AppCompatFlagsUser            As LARGE_INTEGER
    ShimData                      As Long
    AppCompatInfo                 As Long
    CSDVersion                    As UNICODE_STRING
    ActivationContextData         As Long ` // ACTIVATIONCONTEXT_DATA
    ProcessAssemblyStorageMap     As Long ` // ASSEMBLY_STORAGE_MAP
    SystemDefaultActivationData   As Long ` // ACTIVATIONCONTEXT_DATA
    SystemAssemblyStorageMap      As Long ` // ASSEMBLY_STORAGE_MAP
    MinimumStackCommit            As Long
    FlsCallBack                   As Long
    FlsListHead                   As LIST_ENTRY
    FlsBitmap                     As Long ` // RTL_BITMAP
    FlsBitmapBits(3)              As Long
    FlsHighIndex                  As Long
End Type
```

This is quite a lengthy structure, and few books or other pieces of information actually describe what these fields really mean. We are now going to take a look at what every of these fields represent in detail. This information is mostly based on reverse engineering, so some might be guesses.

InheritedAddressSpace

This flag indicates if the process is being forked.

ReadImageFileExecOptions

This Boolean field seems to specify whether special Image Characteristics were read and applied.

BeingDebugged

This Boolean value indicates if the process is currently being debugged.

Mutant

This field is a Handle to a Mutex Object related to the creation of the process.

SectionBaseAddress

This field contains the Base Address of the process.

ProcessModuleInfo

This field is a pointer to the PEB_LDR_DATA Structure which will be shown later.

ProcessParameters

This field is a pointer to the RTL_USER_PROCESS_PARAMETERS Structure which will be shown later.

SubSystemData

This field contains a pointer to variable data that some subsystems might need. WIN32 files don't seem to use this.

ProcessHeap

This field is a pointer to the Process' Heap

FastPebLock

This field points to a Kernel Critical Section used when modifying the PEB with FastPEB routines

AcquireFastPebLockRoutine

This field has the pointer to the function used to acquire the Critical Section above.

ReleaseFastPebLockRoutine

This field has the pointer to the function used to release the Critical Section above.

EnvironmentUpdateCount

This field counts the number of times that Environment Settings have changed.

KernelCallBackTable

This field is used by WIN32K.SYS (the Win32 Subsystem) to be able to call user functions from kernel mode. Specifically, it is used to call the window procedure of a GUI from the driver itself. The field is a pointer to a table that KeUserCallback will read by index and pointer.

EventLogSection

This field has Event Log information if specified.

EventLog

This field has Event Log information if specified.

ExecuteOptions

This field is used to hold certain execution options for the image file, notably the ones located in the respective "Image File Execution Options" registry key for this image, if applicable.

FreeList

This field is a pointer to PEB_FREE_BLOCK (shown later) that describes which parts of the PEB are currently empty.

TlsBitMapSize

This field holds the size of the TLS (Thread Local Storage) bitmap size if the process uses TLS.

TlsBitMap

This field is a pointer to an RTL_BITMAP Structure (shown later) which describes the TLS Bitmap if the process uses TLS.

TlsBitMapData

This field holds the TLS Bitmap Data if the process uses TLS.

ReadOnlySharedMemoryBase

This field has a pointer to a system-wide shared memory location (read-only). It is usually 0x7F6F0000

ReadOnlySharedMemoryHeap

This field has a pointer to a system-wide shared memory location (read-only). It is usually 0x7F6F0000

ReadOnlyStaticServerData

This field has a pointer **to a pointer** to a system-wide shared memory location (read-only). It is usually empty.

InitAnsiCodePageData

This field has a pointer to a system-wide shared memory location that contains an ANSI Codepage Table.

InitOemCodePageData

This field has a pointer to a system-wide shared memory location that contains an OEM Codepage Table.

InitUnicodeCaseTableData

This field has a pointer to a system-wide shared memory location that contains an Unicode Codepage Case Translation Table.

NumberOfProcessors

This field indicates how many processors the process should run on.

NtGlobalFlag

This field contains the NT Global Flag (shown later)

CriticalSectionTimeout

This field indicates how much time must pass before a Kernel Critical Section times out if it is not released.

HeapSegmentReserve

This field indicates how much Heap Memory to reserve.

HeapSegmentCommit

This field indicates how much Heap Memory to commit.

HeapDeCommitTotalFreeThreshold

This field indicates when the Heap can/should (?) be decommited.

HeapDeCommitFreeBlockThreshold

This field indicates when the Heap can't/shouldn't (?) be decommited.

NumberOfHeaps

This field contains the number of Heaps that this process has.

MaxNumberOfHeaps

This field contains the maximum number of Heaps this process can have.

ProcessHeapsList

This field contains a pointer **to a pointer** that lists all the Heaps this process has.

GdiSharedHandleTable

This field contains a pointer to a GDI_HANDLE Structure (described later) which has information about every single GDI Object created by the process.

ProcessStarterHelper

This field contains a pointer to the Function that started the process.

GdiInitialBatchLimit

This field contains the initial maximum GDI batches that the process can have.

LoaderLock

This field contains a pointer to the Critical Section that the PE Loader used when loading the process.

NtMajorVersion

This field contains NT Version Information.

NtMinorVersion

This field contains NT Version Information.

NtBuildNumber

This field contains NT Version Information.

NtCSDVersion

This field contains NT Version Information. (SP Number)

PlatformId

This field contains NT Version Information. (Platform ID; Server, Workstation, etc)

Subsystem

This field contains the subsystem that this process uses (Win32, POSIX, OS/2, etc)

MajorSubsystemVersion

This field contains subsystem version information.

MinorSubsystemVersion

This field contains subsystem version information.

AffinityMask

This field contains the Affinity Flags (KAFFINITY) which are described later.

GdiHandleBuffer(33)

This field contains a buffer that seems to be used by GDI to store frequently used Handles instead of reading from the table.

PostProcessInitRoutine

This field contains the function that cleaned up process initialization.

TlsExpansionBitmap

This field contains TLS data if the Process uses TLS.

TlsExpansionBitmapBits(127)

This field contains TLS data if the Process uses TLS.

SessionId

This field has the Terminal Services Session ID if the Process is being run under TS.

AppCompatFlags

This field contains the Application Compatibility flags loaded from the registry entry for this image file.

AppCompatFlagsUser

This field contains the same data as above, but user-specific instead of system specific.

ShimData

This field contains information used by .NET Shims.

AppCompatInfo

This field contains more Application Compatibility Information

CSDVersion

This field contains the Service Pack in string format

ActivationContextData

This field points to an Activation Context structure (unknown). Activation contexts are data structures in memory containing information that the system can use to redirect an application to load a particular DLL version, COM object instance, or custom window version.

ProcessAssemblyStorageMap

This field contains .NET information used by the .NET Framework.

SystemDefaultActivationData

This field contains the Default System Activation Context.

SystemAssemblyStorageMap

This field contains .NET information used by the .NET Framework.

MinimumStackCommit

This field indicates the minimum stack size to load for this process.

FlsCallBack

This field contains a pointer **to a pointer** to an FLS (Fiber Local Storage) callback function, if the process uses FLS.

FlsListHead

This field contains an unknown List Entry structure probably pointing to the different Fibers.

FlsBitmap

This field has a pointer to an RTL_BITMAP Structure containing the FLS Bitmap.

FlsBitmapBits

This field probably indicates flags or mask settings for the FLS Bitmap structure (such as which bits are in use)

FlsHighIndex

This field indicates the highest FLS Index in the process.

This completes all the information on the PEB Main Structure, but as you've seen, the PEB comprises other important structures that we should look at. The ones you'll use most often are RTL_USER_PROCESS_PARAMETERS and PEB_LDR_DATA, shown below.

4.2 Process Parameters Block (PPB)

This structure is responsible for holding the most common parameters that are usually requested from a Process, such as Windowing data

```
Public Type RTL_USER_PROCESS_PARAMETERS
      MaximumLength                  As Long
      Length                         As Long
      Flags                          As Long
      DebugFlags                     As Long
      ConsoleHandle                  As Long
      ConsoleFlags                   As Long
      StdInputHandle                 As Long
      StdOutputHandle                As Long
      StdErrorHandle                 As Long
      CurrentDirectoryPath           As UNICODE_STRING
      CurrentDirectoryHandle         As Long
      DllPath                        As UNICODE_STRING
      ImagePathName                  As UNICODE_STRING
      CommandLine                    As UNICODE_STRING
      Environment                    As Long
      StartingPositionLeft           As Long
      StartingPositionTop            As Long
      Width                          As Long
      Height                         As Long
      CharWidth                      As Long
      CharHeight                     As Long
      ConsoleTextAttributes          As Long
      WindowFlags                    As Long
      ShowWindowFlags                As Long
      WindowTitle                    As UNICODE_STRING
      DesktopName                    As UNICODE_STRING
      ShellInfo                      As UNICODE_STRING
      RuntimeData                    As UNICODE_STRING
      DLCurrentDirectory(31)         As RTL_DRIVE_LETTER_CURDIR
End Type
```

Once again, let's take a look at what these values mean.

MaximumLength

This field indicates the maximum length this structure can expand to.

Length

This field indicates the length of the structure.

Flags

This field indicates if the structure is normalized or not

DebugFlags

This field contains unknown debug flags.

ConsoleHandle

This field has a hWnd to the Console used by this process (if applicable)

ConsoleFlags

This field contains Console flags, if applicable (unknown).

StdInputHandle

This field contains the Console Input Handle, if applicable.

StdOutputHandle

This field contains the Console Output Handle, if applicable.

StdErrorHandle

This field contains the Console Error Handle, if applicable.

CurrentDirectoryPath

This field has the current path in DOS format ("C:\WINDOWS")

CurrentDirectoryHandle

This field contains the File Handle to the current directory.

DllPath

This field contains DOS paths, separated by a semicolon, on where the process should look for DLLs.

ImagePathName

This field contains the DOS Path of the image file.

CommandLine

This field contains the command line of the process.

Environment

This field points to the Process Environment, where the Environment Settings are located (SYSTEMPATH, WINVER, etc)

StartingPositionLeft

This field holds the starting position of the process's window, if applicable.

StartingPositionTop

This field holds the starting position of the process's window, if applicable.

Width

This field holds the width of the process's window, if applicable.

Height

This field holds the height of the process's window, if applicable.

CharWidth

This field holds the width of a console character, if applicable.

CharHeight

This field holds the width of a console character, if applicable.

ConsoleTextAttributes

This field holds flags on how the text should fill the console.

WindowFlags

This field holds window flags that describe the Window.

ShowWindowFlags

This field holds the flags to use when showing the main process window, if applicable (minimized, maximized, etc)

WindowTitle

This field contains the name of the Window Title of the process, if applicable.

DesktopName

This field contains the name of the Desktop of the process.

ShellInfo

This field contains Windows Shall information for the process.

RuntimeData

This field contains strings that the process might need, if applicable.

DLLCurrentDirectory

This field contains the DLL Paths that will be needed, in an array for up to 32 paths. The structure is described below:

```
Public Type _RTL_DRIVER_LETTER_CURDIR
      Flags                   As Integer
      Length                  As Integer
      TimeStamp               As Long
      DosPath                 As UNICODE_STRING
End Type
```

4.3 Loader Data (LDRD)

The second useful structure you'll need is the Loader (LDR) data, which will tell you all the DLLs that have been loaded by the process. Basically, you won't need PSAPI ever again.

```
Public Type _PEB_LDR_DATA
      Length                  As Integer
      Initialized             As Long
      SsHandle                As Long
      InLoadOrderModuleList   As LIST_ENTRY
      InMemoryOrderModuleList As LIST_ENTRY
      InInitOrderModuleList   As LIST_ENTRY
      EntryInProgress         As Long
End Type
```

It is not necessary to talk in length about these fields. All you will want to read is any of the three Module Lists, which describe the DLLs loaded either by their location in memory, by their initialization order, or by the defined load order. These lists are organized in what Microsoft calls List Entries, which are defined as:

```
Public Type LIST_ENTRY
      Flink                   As LIST_ENTRY
      Blink                   As LIST_ENTRY
End Type
```

4.4 Loaded Module (LDR_LM)

This seems to create confusion...where are the DLLs? Actually, List Entries are only headers to a certain data that is being "listed". These headers simply point to the next entry, either in the forward direction (F-

Link) or backward (B-Link). In this case, the information that follows each entry is called organized according to the LDR_MODULE structure.

```
Public Type _LDR_MODULE
      InLoadOrderModuleList   As LIST_ENTRY
      InMemoryOrderModuleList As LIST_ENTRY
      InInitOrderModuleList   As LIST_ENTRY
      BaseAddress             As Long
      EntryPoint              As Long
      SizeOfImage             As Long
      FullDllName             As UNICODE_STRING
      BaseDllName             As UNICODE_STRING
      Flags                   As Long
      LoadCount               As Integer
      TlsIndex                As Integer
      HashTableEntry          As LIST_ENTRY
      TimeDateStamp           As Long
      LoadedImports           As Long
      EntryActivationContext  As Long ' // ACTIVATION_CONTEXT
      PatchInformation        As Long
End Type
```

Notice that the pointers to the List Entries repeat themselves for each Module. Most of the elements in the structure are self-explanatory. The TLS Index refers to Thread Local Storage, if it is used. The Hash Table Entry is a pointer to a new List Entry. Hash Tables are the mechanism that the PE Loader uses when loading DLLs and finding APIs. Their structure is unknown.

As you've noticed, the PEB also points to a variety of other structures. Although they are not really useful to the average programmer, they are shown and explained below for the sake of completeness.

```
Public Type CRITICAL_SECTION
      DebugInfo               As Long ' // CRITICAL_SECTION_DEBUG
      LockCount               As Long
      RecursionCount          As Long
      OwningThread            As Long
      LockSemaphore           As Long
      Reserved                As Long
End Type
```

Critical Sections are a special form of synchronization objects that the Kernel supports, much like Mutexes (these will be discussed in a future article). They allow a certain resource to be accessed only once, while the thread is executing it under a "Critical Section". The FastPEBLock and LoaderLock values in the PEB point to the Critical Section object that they use, or used, in order to modify the PEB or load the application (in order to ensure that nobody else can touch the PEB while their code is running).

4.5 Various other structures (PEB_FREE_BLOCK, RTL_BITMAP)

FreeList points to the PEB_FREE_BLOCK structure below:

```
Public Type PEB_FREE_BLOCK
      NextBlock                  As PEB_FREE_BLOCK
      Size                       As Long
End Type
```

It basically describes which, if any, parts of the PEB are free to be written to, using an idea similar to List Entries (the structures link between themselves). A similar structure, RTL_BITMAP, is used for Thread Local Storage (TLS) information, shown below:

```
Public Type RTL_BITMAP
      Size                       As Long
      Buffer                     As Long
End Type
```

4.6 Flags (GLOBAL_FLAG, KAFFINITY)

Apart from these structures, there also certain flags which are used by the PEB. They are all documented in the SDK, but I have included them below for reference.

```
Public Enum GLOBAL_FLAGS
      FLG_STOP_ON_EXCEPTION = &H1
      FLG_SHOW_LDR_SNAPS = &H2
      FLG_DEBUG_INITIAL_COMMAND = &H4
      FLG_STOP_ON_HANG_GUI = &H8
      FLG_HEAP_ENABLE_TAIL_CHECK = &H10
      FLG_HEAP_ENABLE_FREE_CHECK = &H20
      FLG_HEAP_VALIDATE_PARAMETERS = &H40
      FLG_HEAP_VALIDATE_ALL = &H80
      FLG_POOL_ENABLE_TAIL_CHECK = &H100
      FLG_POOL_ENABLE_FREE_CHECK = &H200
      FLG_POOL_ENABLE_TAGGING = &H400
      FLG_HEAP_ENABLE_TAGGING = &H800
      FLG_USER_STACK_TRACE_DB = &H1000
      FLG_KERNEL_STACK_TRACE_DB = &H2000
      FLG_MAINTAIN_OBJECT_TYPELIST = &H4000
      FLG_HEAP_ENABLE_TAG_BY_DLL = &H8000
      FLG_IGNORE_DEBUG_PRIV = &H10000
      FLG_ENABLE_CSRDEBUG = &H20000
      FLG_ENABLE_KDEBUG_SYMBOL_LOAD = &H40000
      FLG_DISABLE_PAGE_KERNEL_STACKS = &H80000
      FLG_HEAP_ENABLE_CALL_TRACING = &H100000
      FLG_HEAP_DISABLE_COALESCING = &H200000
      FLG_ENABLE_CLOSE_EXCEPTION = &H400000
      FLG_ENABLE_EXCEPTION_LOGGING = &H800000
      FLG_ENABLE_HANDLE_TYPE_TAGGING = &H1000000
      FLG_HEAP_PAGE_ALLOCS = &H2000000
      FLG_DEBUG_WINLOGON = &H4000000
      FLG_ENABLE_DBGPRINT_BUFFERING = &H8000000
      FLG_EARLY_CRITICAL_SECTION_EVT = &H10000000
      FLG_DISABLE_DLL_VERIFICATION = &H80000000
End Enum
```

These flags, called the NT Debug Flags, specify several debug messages and operations that the OS should do. They are contained in the PEB because each image can specify specific Global Flags based on the registry setting for that file. More information is available on MSDN.

Other flags in the PEB are the Kernel Affinity Flags. While priority defines how much CPU time a Process should take, Affinity describes which CPUs (and in which share) a Process should use on a multiple-CPU system. KAFFINITY should be split up in a binary number.

Each bit set refers to one CPU allowed to be used. For example “31” in decimal, or “11111” in binary, means that the thread can run on CPU 1, 2, 3, 4 and 5. Because these binary numbers can make any pattern (such as “10101010”), which directly translates into a decimal number from 0 to 255, it is not helpful to create an enumeration, since any number has a different meaning.

4.7 GDI Structures (HANDLE_TABLE, GDI_OBJECT)

The final structure that will be shown in this section is the GDI Handle Table. The Win32 Graphical Subsystem is an extremely important part of the NT OS, and no reading material, save a book by Feng Yuan even seems to mention it or talk about its structures. The PEB mentions this table in the GdiSharedHandleTable, which is a pointer to the following structure:

```
Public Type GDI_HANDLE_TABLE
      KernelInfo              As Long
      ProcessID               As Integer
      Count                   As Integer
      MaxCount                As Integer
      Type                    As Integer  ' // GDI_OBJECT
      UserInfo                As Long
End Type
```

Basically every graphical object that a Process owns has an associated GDI Handle. This table lists all the GDI Objects created, according to their type, and points to their respective user-mode or kernel-mode structure that describes them more in detail. It is beyond the scope of this article to document all the possible GDI Object Structures, but if enough people request it, I will consider adding it. For now, a basic enumeration of the possible GDI Types should be enough.

```
Public Enum GDI_OBJECT
      DeviceContext = &H1
      Region = &H4
      Bitmap = &H5
      Palette = &H8
```

```
        Font = &HA
        Brush = &H10
        EnhancedMetaFile = &H21
        Pen = &H30
End Enum
```

This concludes the Chapter on the User-Mode Structures. The next chapter will move on with the Kernel-Mode Structures. Since these will not generally available to the VB Programmer (I will soon show a way how), you may skip this section unless you are genuinely interested in the inner workings of Processes.

5. Kernel-Mode Process Structures

Apart from the PEB, the Kernel itself must know critical information about the process, in order for scheduling and other important system tasks. Furthermore, the graphical subsystem must also be made aware of the process's rights in respect to the screen. All this critical information, plus the memory allocations, the object table, the different quotas, etc, are held in a master structure called EPROCESS, which in itself contains and points to a variety of other structures.

5.1 Executive Process (EPROCESS)

The EPROCESS structure is the equivalent of the PEB, and holds all the kernel-mode information needed for the process. Before looking at the structures it points to, let's see how the EPROCESS structure itself looks like in detail.

```
Public Type EPROCESS
   Pcb                      As KPROCESS
   ProcessLock              As EX_PUSH_LOCK
   CreateTime               As FILETIME
   ExitTime                 As FILETIME
   RundownProtect           As EX_RUNDOWN_REF
   UniqueProcessId          As Long
   ActiveProcessLinks       As LIST_ENTRY
   QuotaUsage(2)            As Long
   QuotaPeak(2)             As Long
   CommitCharge             As Long
   PeakVirtualSize          As Long
   VirtualSize              As Long
   SessionProcessLinks      As LIST_ENTRY
   DebugPort                As Long           ' // LPC_PORT_OBJECT
   ExceptionPort            As Long           ' // LPC_PORT_OBJECT
   ObjectTable              As Long           ' // HANDLE_TABLE
   Token                    As EX_FAST_REF    ' // TOKEN
   WorkingSetLock           As FAST_MUTEX
   WorkingSetPage           As PFN_NUMBER
   AddressCreationLock      As FAST_MUTEX
   HyperSpaceLock           As KSPIN_LOCK
   ForkInProgress           As Long           ' // ETHREAD
   HadwareTrigger           As Long
```

```
    VadRoot                     As Long      ' // MM_AVL_TABLE
    VadHint                     As Long      ' // MM_ADDRESS_NODE
    CloneRoot                   As Long      ' // MM_CLONE_DESCRIPTOR
    NumberOfPrivatePages        As PFN_NUMBER
    NumberOfLockedPages         As PFN_NUMBER
    Win32Process                As Long      ' // W32PROCESS
    Job                         As Long      ' // EJOB
    SectionObject               As Long      ' // SECTION_OBJECT
    SectionBaseAddress          As Long
    QuotaBlock                  As Long      ' // EPROCESS_QUOTA_BLOCK
    WorkingSetWatch             As Long      ' // PAGEFAULT_HISTORY
    Win32WindowStation          As HANDLE
    InheritedFromProcessId      As HANDLE
    LdtInformation              As Long      ' // LDT_INFORMATION
    VadFreeHint                 As Long      ' // MM_ADDRESS_NODE
    VdmObjects                  As Long      ' // VDM_OBJECTS
    DeviceMap                   As Long      ' // DEVICE_MAP
    PhysicalVadList             As LIST_ENTRY
    PageDirectoryPte            As HARDWARE_PTE_X86
    Padding2                    As LARGE_INTEGER
    Session                     As Long
    ImageFileName(15)           As Byte
    JobLinks                    As LIST_ENTRY
    LockedPagesList             As LIST_ENTRY
    ThreadListHead              As LIST_ENTRY
    SecurityPort                As Long      ' // LPC_PORT_OBJECT
    PaeTop                      As Long
    ActiveThreads               As Long
    GrantedAccesss              As ACCESS_MASK
    DefaultHardErrorAction      As Long
    LastThreadExitStatus        As NTSTATUS
    Peb                         As Long      ' // PEB
    PrefetchTrace               As EX_FAST_REF
    ReadOperationCount          As LARGE_INTEGER
    WriteOperationCount         As LARGE_INTEGER
    OtherOperationCount         As LARGE_INTEGER
    ReadTransferCount           As LARGE_INTEGER
    WriteTransferCount          As LARGE_INTEGER
    OtherTransferCount          As LARGE_INTEGER
    CommitChargeLimit           As Long
    CommitChargePeak            As Long
    AweInfo                     As Long
    SeAuditProcessCreation      As SE_AUDIT_PROCESS_CREATION_INFO
    Vm                          As MMSUPPORT
    ModifiedPageCount           As Long
    NumberOfVads                As Long
    JobStatus                   As JOB_STATUS_FLAGS
    Flags                       As EPROCESS_FLAGS
    ExitStatus                  As NTSTATUS
    NextPageColor               As Integer
    SubSystemMinorVersion       As Byte
    SubSystemMajorVersion       As Byte
    SubSystemVersion            As Integer
    PriorityClass               As Byte
    WorkingSetIsUnsafe          As Byte
    Cookie                      As Long
End Type
```

Pcb

This field contains the Process Control Block, which is the Kernel Process (KPROCESS) Structure. It contains data the kernel needs about the process.

ProcessLock

This field contains a structure defining the lock to use when modifying fields in EPROCESS, to avoid any race conditions or similar.

CreateTime

This field contains the time when the process was created.

ExitTime

This field contains the time when the process was exited.

RundownProtect

This field contains a Rundown Protection structure, which avoids the kernel prematurely killing the process while it's being created.

UniqueProcessId

This field has the PID of the Process.

ActiveProcessLinks

This field is a List Entry pointing to other EPROCESS Structures of running processes.

QuotaUsage

This field contains information about the process usage of the set quotas.

QuotaPeak

This field contains information about the peak process usage of the set quotas.

CommitCharge

This field holds the physical memory usage of the process.

PeakVirtualSize

This field holds the maximum memory usage of the process.

VirtualSize

This field holds the current memory usage of the process.

ActiveProcessLinks

This field is a List Entry pointing to other EPROCESS Structures of running processes, but only in the current Terminal Services Session.

DebugPort

This field contains the LPC Port used when debugging the process.

ExceptionPort

This field contains the LPC Port used when the process generates exceptions (errors).

Object Table

This field contains a pointer to the process' Object Table, which will be described in detail later.

Token

This field contains a pointer to the security token of the process. It is under a Fast Reference Structure, so the last byte will change each time you read it. Ignore it and set it to 0.

WorkingSetLock

This field contains a lock to be used when modifying the process' working set (memory areas)

WorkingSetPage

This field points to the Page Number that contains the process' working set.

AddressCreationLock

This field contains a lock to be used when creating addresses for the process.

HyperSpaceLock

This field contains a lock to be used when accessing Hyperspace memory for the process.

ForkInProgress

This field points to an Executive Thread (ETHREAD) Structure of a thread if the process is being forked.

HardwareTrigger

Unknown.

VadRoot

This field points to a VAD Root Structure, which defines the Virtual Addresses used by the process. This will be described later.

VadHint

This field caches the last VAD entry.

CloneRoot

This field points to VAD information for a clone Process.

NumberOfPrivatePages

This field holds the number of private memory pages that the process is using.

NumberOfLockedPages

This field holds the number of locked memory pages that the process is using.

Win32Process

This field points to the W32PROCESS Structure used by GDI. It is currently unknown.

Job

This field points to an Executive Job (EJOB) Structure if the process is part of a Job (more on Jobs later).

SectionObject

This field points to a SECTION_OBJECT structure that describes a Memory Section.

SectionBaseAddress

This field points to the base address of the Process Section, usually the Image Base of the process.

QuotaBlock

This field points to a EPROCESS_QUOTA_BLOCK structure which contains different quotas for the process. Explained later.

WorkingSetWatch

This field points to a PAGEFAULT_HISTORY structure that saves page faults generated by the process.

Win32WindowStation

This field points to the Window Station ID Number in which the Process is running in.

InheritedFromProcessId

This field holds the parent PID who crated this process, if applicable.

LdtInformation

This field contains information about the Local Descriptor Table (LDT) if used by the process, pointer to an LDT_INFORMATION structure.

VadFreeHint

This field indicates some sort of "hint" to find free Virtual Addresses.

VdmObjects

This field points to an unknown structure/memory area containing VDM Objects (used for 16-bit programs).

DeviceMap

This field points to a DEVICE_MAP structure holding the DOS Devices the process can use.

PhysicalVadList

This field points to a structure that has the physical location of the VAD entries.

PageDirectoryPte

This field holds the PTE Flags for the Page Directory of the process.

Session

This field contains the Terminal Services Session ID of the Process.

ImageFileName

This field contains the name of the executable.

JobLinks

This field has a List Entry structure that links all the Job Objects together.

LockedPagesList

This field points to a list which contains the memory pages locked by the process.

ThreadListHead

This field contains a List Entry structure that links all the Threads part of this Process.

SecurityPort

This field points to the LPC Port Structure used for Security Purposes.

PaeTop

This field contains information about Physical Address Extension for systems with more then 4GB of memory.

ActiveThreads

This field counts the number of active threads in the process.

GrantedAccess

This field contains the access mask of the process.

DefaultHardErrorAction

This field has the default action when an NT Hard Error occurs.

LastThreadExitStatus

This field contains the Exit Status of the last Thread to end.

Peb

This field points to the Process Environment Block (PEB).

PrefetchTrace

This field contains information used by the Prefetcher.

ReadOperationCount

This field contains the number of I/O Read Operations performed.

WriteOperationCount

This field contains the number of I/O Write Operations performed.

OtherOperationCount

This field contains the number of I/O Misc Operations performed.

ReadTransferCount

This field contains the number of I/O Read Transfers performed.

WriteTransferCount

This field contains the number of I/O Write Transfers performed.

OtherTransferCount

This field contains the number of I/O Misc Transfers performed.

CommitChargeLimit

This field contains the maximum memory usage possible.

CommitChargePeak

This field contains the maximum memory usage reached.

AweInfo

This field contains information used by Address Windowing Extension on PAE systems.

SeAuditProcessCreation

This field contains a pointer to an OBJECT_NAME Structure which contains the name of the process that audited the process creation (usually csrss.exe)

Vm

This field is a MMSUPPORT structure describing many Virtual Memory Settings. Described later.

ModifiedPageCount

This field contains information the number of pages that have been modified by the process.

NumberOfVads

This field contains the number of VAD Entries that the process has.

JobStatus

This field contains the current status of the Job that contains this Process, if applicable.

Flags

This field contains several EPROCESS Flags, shown later.

ExitStatus

This field contains the Return Code of the Process.

NextPageColor

This field contains the color of the next memory page.

SubSystemMinorVersion

This field has part of the Subsystem's Version.

SubSystemMajorVersion

This field has part of the Subsystem's Version.

SubSystemVersion

This field has part of the Subsystem's Version.

PriorityClass

This field contains the Process Priority.

WorkingSetIsUnsafe

This field is a dirty flag for the status of the memory working set.

Cookie

Unknown

This completes the full overview of the whole EPROCESS Structure as a whole. The next sections will look at some of the more important structures in detail. Some are documented in the attached modules, but not shown here due to their lack of usefulness for any user-mode purpose.

5.2 Kernel Process (KPROCESS)

The most important structure for the Kernel Itself is the PCB, or KPROCESS Structure. This structure contains all the necessary scheduling, affinity and priority settings.

```
Public Type KPROCESS
      Header                  As DISPATCHER_HEADER
      ProfileListHead         As LIST_ENTRY
      DirectoryTableBase(1)   As Long
      LdtDescriptor           As KGDTENTRY
      Int21Descriptor         As KGDTENTRY
      IopmOffset              As Integer
      Iopl                    As Byte
      Unused                  As Byte
      ActiveProcessors        As KAFFINITY
      KernelTime              As Long
      Usertime                As Long
      ReadyListHead           As LIST_ENTRY
      SwapListEntry           As SINGLE_LIST_ENTRY
```

```
        VdmTrapcHandler         As Long
        ThreadListHead          As LIST_ENTRY
        ProcessLock             As KSPIN_LOCK
        Affinity                As KAFFINITY
        StackCount              As Integer
        BasePriority            As Byte
        ThreadQuantum           As Byte
        AutoAlignment           As Byte
        State                   As Byte
        ThreadSeed              As Byte
        DisableBoost            As Byte
        PowerState              As Byte
        DisableQuantum          As Byte
        IdealNode               As Byte
        Flags                   As Byte
End Type
```

Header

This field is a structure contained information about the Kernel Dispatcher, which is responsible for all scheduling in the system.

ProfileListHead

This field contains a List Entry for KPROFILE structures which describe various Kernel Profiling actions (performance timers, etc…)

DirectoryTableBase

This field contains the Page Table Directory (the physical address) for the current process, which contains all the Page Table Entries that map Virtual to Physical addresses.

LdtDescriptor

This field contains an LDT Descriptor for the Local Descriptor Table used by 16-bit applications running under NT.

Int21Descriptor

This field contains the descriptor in the IDT for the Interrupt 21 Handler used by 16-bit applications running under NT.

IopmOffset

This field contains a pointer to the IO Permission bitMap, which contains the permissions for the I/O port usage by IN and OUT assembly code commands.

Iopl

This field contains the IO Privilege Level, which can either be set to 0 for Ring 0 only (Kernel Mode) or 3 to allow Ring 3 (User Mode) process to access I/O Ports (NT never usually allows this).

ActiveProcessors

This field contains the number of CPUs on the system available to this process.

KernelTime

This field contains the number of time that the process has spent in Kernel Mode.

UserTime

This field contains the number of time that the process has spent in User Mode.

ReadyListHead

This field contains a List Entry of which threads are currently in the ready state.

SwapListEntry

This field contains a List Entry of which threads are currently getting their Contexts swapped.

ThreadListHead

This field contains a List Entry of all the threads created by the process.

ProcessLock

This field contains the Process Lock that was used, usually 0 after loading.

Affinity

This field contains the Process Affinity.

StackCount

This field contains the number of stacks used by the Process.

BasePriority

This field contains the Process Priority, from 0-15.

ThreadQuantum

This field contains the default thread quantum for new Threads created by the Process. This is the time until a thread is switched.

AutoAlignment

This field probably describes if the Process is aligned in memory or not.

State

This field contains the current state of the Process.

ThreadSeed

This field describes if a Thread Seed was used (generated from *KiGetTickCount*)

DisableBoost

This field describes if the thread boost should be disabled.

PowerState

This field contains the Process's power state (should reflect the system's power state).

DisableQuantum

This field describes if the thread quantum should be disabled.

IdealNode

This field is unknown

Flags

This field contains certain flags about the Process, used in Windows XP Service Pack 2 for No-Execute Memory Protection.

As it is possible to see, the KPROCESS structure doesn't offer much useful information to the programmer. The most important field that a programmer would like to change is the IOPL, which would permit the User-Mode application to access I/O ports, which NT disables. This could be used for communicating or for using ancient communications programs that need this functionality (there are however much safer ways). It is also possible to change the Process's scheduling properties, some which are inaccessible by APIs.

5.3 LPC Port (LPC_PORT_OBJECT)

LPC, or Local Procedure Call, is an Inter Process Communication (IPC) method extensively used by NT. It will be discussed in great detail in the future article dealing with IPC methods of NT. The structure of an LPC port is documented below however for the sake of completeness.

```
Public Type LPC_PORT_OBJECT
    ConnectionPort          As Long ' // LPC_PORT_OBJECT
    ConnectedPort           As Long ' // LPC_PORT_OBJECT
    MsgQueue                As Long ' // LPC_PORT_QUEUE
    Creator                 As Long
    ClientSectionName       As Long
    ServerSectionName       As Long
    PortContext             As Long
    ClientThread            As Long ' // ETHREAD
    SecurityQoS             As SECURITY_QUALITY_OF_SERVICE
    StaticSecurity          As SECURITY_CLIENT_CONTEXT
    LpcReplyChainHead       As LIST_ENTRY
    LpcDataInfoChainHead    As LIST_ENTRY
    ServerProcess           As Long ' // EPROCESS
    MappingProcess          As Long ' // EPROCESS
    MaxMessageLength        As Integer
    MaxConnectionInfoLength As Long
    Flags                   As Long
    WaitEvent               As KEVENT
End Type
```

ConnectionPort

This field is a pointer that references back to the Port Object.

ConnectionPort

This field is a pointer that references to another Port Object if the current one is not the one used for the connection.

MsgQueue

This field is a pointer to the Port's Queue (messages waiting in line).

Creator

This field is a ClientID Structure that defines the Process and Thread ID of the creator of this port.

ClientSectionName

If the Port uses a Memory Mapped Section (happens when transferring large messages), this field contains a pointer to the name of the section in the client Process.

ServerSectionName

If the Port uses a Memory Mapped Section (happens when transferring large messages), this field contains a pointer to the name of the section in the Server Process.

PortContext

This field contains the Port context.

ClientThread

This field is a pointer to an ETHREAD structure of the Client Thread using this Port.

SecurityQoS

This field is a security structure used for private LPC messages.

StaticSecurity

This field is a security structure used for private LPC messages.

LpcReplyChainHead

This field is a List Entry that contains information on how the LPC reply should be handled, used only in COMMUNICATION ports.

LpcDataInfoChainHead

This field is a List Entry that contains information on LPC Data Information, used only in COMMUNICATION ports.

ServerProcess

This field is a pointer to an EPROCESS structure of the Server Process using this Port.

MappingProcess

This field is a pointer to an EPROCESS structure of the Server Process who made the Mapped Section (usually the same as above)

MaxMessageLength

This field contains the maximum LPC message length without using a Memory Mapped Section

MaxConnectionInfoLength

This field contains the maximum LPC Connect Info length.

Flags

This field contains the LPC Port flags.

WaitEvent

This field contains a KEVENT structure that contains information about the LPC Wait event that's generated (like Winsock's).

A Process usually contains only two system-defined LPC Ports in EPROCESS, which are the Exception Port, which sends crashes to CSRSS (The client/server runtime subsystem) and the Debug Port, which also sends debug information to CSRSS if the process is being debugged. As it will be shown in a future article about LPC, there is a limited use in modifying/reading these structures, but it does allow for message interception or sending.

5.4 Handle Table (HANDLE_TABLE)

An important part of the EPROCESS Structure is a pointer to the Process' Handle Table Descriptor, which ultimately points to the Handle Table. In NT, every object that is opened by a program (meaning any file, screen, memory section etc) is given a handle that the Process can use (such as hFile). All these handles are stored in the Process' Handle Table, along with the permissions for each object, and a pointer to the Object's Structure.

```
Public Type HANDLE_TABLE
    Table                   As Long
    QuotaProcess            As Long '// EPROCESS
    UniqueProcessId         As Long
    HandleTableLock(3)      As EX_PUSH_LOCK ' // ERESOURCES
```

```
    HandleTableList          As LIST_ENTRY
    HandleContentionEvent    As EX_PUSH_LOCK ' // KEVENT
    DebugInfo                As Long ' // HANDLE_TRACE_DEBUG_INFO
    ExtraInfoPaces           As Long
    FirstFree                As Long
    LastFree                 As Long
    NextHandleNeedingPool    As Long
    HandleCount              As Long
    Flags                    As Long
End Type
```

Table

This field contains the pointer to the Handle Table itself, which is an array of Object Pointer/Access Mask entries (8 bytes each)

QuotaProcess

This field contains a pointer to the EPROCESS Structure of the Process.

UniqueProcessId

This field contains the Process ID of the Process.

HandleTableLock

This field contains Lock structures used when safely modifying the table by the Kernel.

HandleTableList

This field contains a List Entry towards other Handle Tables.

HandleContentionEvent

This field contains the Event to block on when the Kernel modifies the table.

DebugInfo

This field contains stack trace information.

ExtraInfoPages

This field contains a pointer to a parallel table that is used for auditing.

FirstFree

This field contains the first Handle ID that's free to use.

LastFree

This field contains the last Handle ID that's free to use.

NextHandleNeedingPool

This field contains information about Handles that require memory.

HandleCount

This field contains the number of Handles in use.

Flags

This field contains a flag if a strict First In/First Out order should be kept (FIFO), or if handles can be put non-sequentially (Say you close Handle 18 and you have handles up to 204, and then it becomes free again, should the next Handle be 18 or be 208?

The Handle Table is very useful to read, because it allows the programmer to view in detail all the objects opened by the process, and change their access properties. While the first function can be achieved with Native API, the second is usually impossible.

5.5 Virtual Address Descriptor Table (MM_AVL_TABLE)

Every memory location that a Process can read, write or execute from is mapped in structures called VADs, or Virtual Address Descriptors. These contain the virtual address of the memory region and the permissions. Every time that a process uses *GlobalAlloc*, *VirtualAlloc* or any other memory allocation routine, or that it loads a DLL or anything else in its memory, an entry is created into the VAD. Once again, this structure is useful both for enumeration, and also for changing privileges (the first one cannot be done by API, the second one can, but not with all addresses).

```
Public Type MM_AVL_TABLE
    Root            As MMADDRESS_NODE
    Flags           As Long
    NodeHint        As Long
    NodeFreeHint    As Long
End Type

Public Type MMADDRESS_NODE
    Parent          As Long
    LeftChild       As MMADDRESS_NODE
    RightChild      As MMADDRESS_NODE
```

```
        StartingVpn     As Long
        EndingVpn       As Long
    End Type
```

Root

This field contains the first VAD, in the also called a MmAddressNode.

Flags

This field contains the depth and size of the tree.

Parent

This field contains a pointer to the parent VAD.

LeftChild

This field contains a pointer to the left child VAD.

RightChild

This field contains a pointer to the right child VAD.

StartingVpn

This field contains the starting virtual address of the VAD. This number should be multiplied by 10000.

EndingVpn

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

Walking the VAD Tree is not an easy task, since it can become pretty complex in advanced depths, but As Long as an efficient tree browsing algorithm

5.6 Token (TOKEN)

Security Tokens are perhaps the most important and underlying mechanism of the entire NT Kernel architectures. Because this article deals with Processes and Threads however, only the main Token Structure will be shown. Those interested in more information on Tokens should visit MSDN.

```
Public Type TOKEN
    TokenSource                 As TOKEN_SOURCE
    TokenId                     As LUID
    AuthenticationId            As LUID
    ParentTokenId               As LUID
    ExpirationTime              As LARGE_INTEGER
    TokenLock                   As Long ' // ERESOURCE
    AuditPolicy                 As SEP_AUDIT_POLICY
    ModifiedId                  As LUID
    SessionId                   As Long
    UserAndGroupCount           As Long
    RestrictedSidCount          As Long
    PrivilegeCount              As Long
    VariableLength              As Long
    DynamicCharged              As Long
    DynamicAvailable            As Long
    DefaultOwnerIndex           As Long
    UserAndGroups               As Long ' // SID_AND_ATTRIBUTES
```

```
        RestrictedSids              As Long ' // SID_AND_ATTRIBUTES
        PrimaryGroup                As Long
        Privileges                  As Long ' // LUID_AND_ATTRIBUTES
        DynamicPart                 As Long
        DefaultDacl                 As Long ' // ACL
        TokenType                   As TOKEN_TYPE
        ImpersonationLevel          As SECURITY_IMPERSONATION_LEVEL
        TokenFlags                  As Long
        TokenInUse                  As Byte
        ProxyData                   As Long ' // SECURITY_TOKEN_PROXY_DATA
        AuditData                   As Long ' // SECURITY_TOKEN_AUDIT_DATA
        OriginatingLogonSession     As LUID
        VariablePart                As Long
    End Type
```

VariablePart

This field contains a pointer to the left child VAD.

OriginatingLogonSession

This field contains a pointer to the left child VAD.

AuditData

This field contains a pointer to the left child VAD.

ProxyData

This field contains a pointer to the left child VAD.

TokenInUse

This field contains a pointer to the left child VAD

ImpersonationLevel

This field contains a pointer to the left child VAD.

TokenFlags

This field contains a pointer to the left child VAD.

TokenType

This field contains a pointer to the left child VAD.

DefaultDacl

This field contains a pointer to the left child VAD.

DynamicPart

This field contains a pointer to the left child VAD.

PrimaryGroup

This field contains a pointer to the left child VAD.

Privileges

This field contains a pointer to the left child VAD.

RestrictedSids

This field contains a pointer to the left child VAD.

UserAndGroups

This field contains a pointer to the left child VAD.

DefaultOwnerIndex

This field contains a pointer to the left child VAD.

DynamicAvailable

This field contains a pointer to the left child VAD.

DynamicCharged

This field contains a pointer to the left child VAD.

VariableLength

This field contains a pointer to the left child VAD.

UserAndGroupCount

This field contains a pointer to the left child VAD.

RestrictedSidCount

This field contains a pointer to the left child VAD.

SessionId

This field contains a pointer to the left child VAD.

PrivilegeCount

This field contains a pointer to the left child VAD.

ModifiedId

This field contains a pointer to the left child VAD.

AuditPolicy

This field contains a pointer to the left child VAD.

TokenLock

This field contains a pointer to the left child VAD.

ExpirationTime

This field contains a pointer to the left child VAD.

AuthenticationId

This field contains a pointer to the left child VAD.

ParentTokenId

This field contains a pointer to the left child VAD.

TokenId

This field contains a pointer to the left child VAD.

TokenSource

This field contains a pointer to the left child VAD.

This completes the documentation on Processes. Although EPROCESS has many other structures, they do not have any real use to a developer and are not described in this document. The structures themselves are included in the documents however for completeness and to allow proper compilation. The SECTION_OBJECT structure will be documented in a later article on Memory Mapped Sections, while the EJOB structure will be shown after the documentation on Threads.

6. User-Mode Thread Structures

6.1 Thread Environment Block (PEB)

The TEB is the Thread Environment Block. It is a high-level user-mode structure that contains some important information about the current Thread:

```
Public Type TEB
    NtTib                           As TIB
    EnvironmentPointer              As Long
    ClientId                        As CLIENT_ID
    ActiveRpcHandle                 As Long
    ThreadLocalStoragePointer       As Long
    ProcessEnvironmentBlock         As Long
    LastErrorValue                  As Long
    CountOwnedCriticalSections      As Long
    CsrClientThread                 As Long ‘ // CSR_THREAD
    Win32ThreadInfo                 As Long ‘ // W32_THREAD
    User32Reserved(25)              As Long
    UserReserved(4)                 As Long
    WOW32Reserved                   As Long ‘ // WOW32_THREAD
    CurrentLocale                   As Long
    FpSoftwareStatusRegister        As Long
    SystemReserved1(53)             As Long
    ExceptionCode                   As Long
    ActivationContextStack          As ACTIVATION_CONTEXT_STACK
    SpareBytes1(23)                 As Byte
    GdiTebBatch                     As GDI_TEB_BATCH
    RealClientId                    As CLIENT_ID
    GdiCachedProcessHandle          As Long
    GdiClientPID                    As Long
    GdiClientTID                    As Long
    GdiThreadLocalInfo              As Long
    Win32ClientInfo(61)             As Long
    glDispatchTable(232)            As Long
    glReserved1(28)                 As Long
    glReserved2                     As Long
    glSectionInfo                   As Long
    glSection                       As Long
    glTable                         As Long
    glCurrentRC                     As Long
    glContext                       As Long
    LastStatusValue                 As Long
    StaticUnicodeString             As UNICODE_STRING
    StaticUnicodeBuffer(260)        As Integer
    DeallocationStack               As Long
    TlsSlots(63)                    As Long
    TlsLinks                        As LIST_ENTRY
    Vdm                             As Long ' // VDM_OBJECTS
    ReservedForNtRpc                As Long
    DbgSsReserved(1)                As Long
    HardErrorsAreDisabled           As Long
    Instrumentation(15)             As Long
```

```
    WinSockData                 As Long
    GdiBatchCount               As Long
    InDbgPrint                  As Byte
    FreeStackOnTermination      As Byte
    HasFiberData                As Byte
    IdealProcessor              As Byte
    Spare3                      As Long
    ReservedForPerf             As Long
    ReservedForOle              As Long
    WaitingOnLoaderLock         As Long
    Wx86Thread                  As Wx86ThreadState
    TlsExpansionSlots           As Long
    ImpersonationLocale         As Long
    IsImpersonating             As Long
    NlsCache                    As Long
    pShimData                   As Long
    HeapVirtualAffinity         As Long
    CurrentTransactionHandle    As Long
    ActiveFrame                 As TEB_ACTIVE_FRAME
    FlsData                     As Long
    SafeThunkCall               As Byte
    BooleanSpare(2)             As Byte
End Type
```

Tib

This field points to the Thread Information Block, which contains stack and exception information used for error handling.

EnvironmentPointer

This field points to the Thread's Environment Block. Often not used.

ClientId

This field contains a structure with the TID and PID of the Thread.

ActiveRpcHandle

This field contains an opaque handle used if the Thread is currently using RPC.

ThreadLocalStoragePointer

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

ProcessEnvironmentBlock

This field contains per-module Thread Local Storage (TLS) blocks.

LastErrorValue

This field contains the last DLL Error Value for the Thread.

CountOwnedCriticalSections

This field counts the number of Critical Sections (a Synchronization mechanism) that the Thread owns.

CsrClientThread

This field points to a CSR_CLIENT structure used by the Client-Server Runtime System Service (CSRSS).

Win32ThreadInfo

This field points to a W32_THREAD structure used by Win32K, the Kernel-Mode Graphical Subsystem.

WOW32Reserved

This field contains to a WOW32_THREAD structure used by Windows-on-Windows virtualization (for 32-bit processes running on 64-bit Windows).

CurrentLocale

This field contains the current locale ID.

FpSoftwareStatusRegister

This field contains a floating point register.

ExceptionCode

This field contains the last exception code generated by the Thread.

ActivationContextStack

This field contains a structure describing the Activation Context Stack. Activation Context was described in the documentation on Processes above.

GdiTebBatch

This field contains a cached copy of GDI Objects used by the Thread in a structure.

RealClientId

This field contains a structure containing the real PID and TID (usually the same as in ClientId).

GdiCachedProcessHandle

This field contains a cached handle to the current Process that GDI uses.

GdiClientPID

This field contains the PID used by GDI.

GdiClientTID

This field contains the TID used by GDI.

GdiThreadLocalInfo

This field contains more GDI Information the Thread.

LastStatusValue

This field contains the last NTSTATUS value (similar to LastError, but used in the Kernel).

StaticUnicodeString

This field contains a UNICODE_STRING structure describing the string that follows below.

StaticUnicodeBuffer

This field contains a buffer for a string buffer used by the PE Loader to save various DLL names when loading them.

DeallocationStack

This field contains the stack of the Thread that should be freed on exit.

TlsSlots

This field contains the Thread Local Storage slots for the Thread.

Vdm

This field contains a pointer to the VDM_OBJECTS Structure used for VDM Threads.

ReservedForNtRpc

This field contains a pointer to an RPC Structure (unknown).

HardErrorsAreDisabled

This field is a flag to whether Hard Errors are disabled or not.

Instrumentation

This field contains various data used by WMI (Windows Management Instrumentation).

WinSockData

This field contains Winsock Stack Data…nothing useful to read.

GdiBatchCount

This field contains the GDI Batch Count Limit, which can be read/set by using APIs.

InDbgPrint

This field contains a flag whether the Thread has issued a Debug Print command.

FreeStackOnTermination

This field contains a flag whether the Thread's Stack should be freed when the thread terminates.

HasFiberData

This field indicates if the Thread created Fibers.

IdealProcessor

This field contains the ideal Process to use (Affinity).

ReservedForPerf

This field contains reserved data for the Performance Manager.

ReservedForOle

This field contains the IObjContext for the current context.

WaitingOnLoaderLock

This field contains a flag if the Thread is waiting on the PE Loader to establish a lock.

Wx86Thread

This field contains information that an ancient x86 emulator called Wx86 needed on NT4.

TlsExpansionSlots

This field contains Thread Local Storage slots.

ImpersonationLocale

This field contains the locale ID that the Thread is impersonating.

IsImpersonating

This field is a flag on whether the Thread is doing any impersionation.

NlsCache

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

pShimData

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

HeapVirtualAffinity

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

CurrentTransactionHandle

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

ActiveFrame

This field contains the ending virtual address of the VAD. This number should be multiplied by 10000.

FlsData

This field contains Fiber Local Storage (FLS) information on Windows 2003 and possibly future versions of Windows XP.

SafeThunkCall

This field contains a flag on calling 16-bit functions from 32-bit Threads in a safe way.

As seen above, the TEB is not filled with much useful information that couldn't be read by using normal API calls. The structures which are of interest are unfortunately undocumented, or contain only random data. In the case of the gL and other GDI structures, they have been omitted because they are only used during a graphic operation. Normal reading of those pointers/bytes will usually reveal null information (except for the cached GDI Handle structure). As such, you will find that most fields in the TEB are empty or set to 0 when reading them, and the ones that aren't are usually readable by APIs.

6.2 NT Thread Information Block (TIB)

The TIB, or NT_TIB, contains information most commonly used for SEH (Structured Exception Handling, used in C/C++ with try/catch).

```
Public Type TIB
    ExceptionList               As EXCEPTION_LIST_REGISTRATION_RECORD
    StackBase                   As Long
    StackLimit                  As Long
    SubSystemTib                As Long
    FiberData                   As Long ' // FIBER_CONTEXT
    Version                     As Long
    ArbitraryUserPointer        As Long
    Self                        As Long
End Type
```

ExceptionList

This field contains the Exception Handlers List used by SEH.

StackBase

This field contains a pointer to the beginning of the Thread's Stack.

StackLimit

This field is a pointer to the end of the Thread's Stack.

SubSystemTib

This field contains an optional pointer to a Subsystem TIB (POSIX, OS/2)

FiberData

This field contains a pointer to the Fiber Context, which will be shown in the following chapter.

Version

This field contains the version number of the TIB.

ArbitraryUserPointer

This field contains a user-defined pointer sent when the Thread is created.

Self

This field points back to the TIB; used for ASM code to get a pointer to the TIB faster.

The TIB is actually only useful if the stack of the Thread should be moved or determined, or for changing/reading SEH settings. It's actually internally used extensively by C and VB programs.

6.3 Miscellaneous User-Mode Structures

Because most of the Structures that TEB fields point to are either undocumented or in Kernel-Mode, very few are readable from User-Mode. The most important ones will be shown below:

```
Public Type ACTIVATION_CONTEXT_STACK
    Flags                           As Long
    NextCookieSequenceNumber        As Long
    ActiveFrame                     As Long
    FrameListCache                  As LIST_ENTRY
End Type
```

This structure contains information about the Activation Context Stack (Activation Contexts have been explained previously). This information can be easily recovered with documented APIs on MSDN.

```
Public Type CLIENT_ID
    UniqueProcess                   As Long
    UniqueThread                    As Long
End Type
```

This structure contains the TID of the Thread and the PID of the owning Process.

```
Public Type GDI_TEB_BATCH
    Offset                          As Long
    HDC                             As Long
    Buffer(309)                     As Long
End Type
```

This structure contains a cached hDC (Device Context) as well as a buffer of GDI Objects.

```
Public Type TEB_ACTIVE_FRAME
    Flags                        As Long
    Previous                     As Long ' // TEB_ACTIVE_FRAME
    Context                      As Long ' // TEB_ACTIVE_FRAME_CONTEXT
End Type

Public Type TEB_ACTIVE_FRAME_CONTEXT
   Flags                         As Long
   FrameName                     As Long
End Type
```

These structures contain information about TEB Frames, which are of no use.

This concludes the Chapter on the Thread User-Mode Structures. The next chapter will move on with the Thread Kernel-Mode Structures. Since these will not generally available to the VB Programmer you may skip this section unless you are genuinely interested in the inner workings of Threads. However, unlike with Processes, the ETHREAD contains much more valuable information then the TEB, and is more worthwhile studying then EPROCESS versus PEB.

7. Kernel-Mode Thread Structures

Once again, the Kernel needs to know deep details about the Thread. Unlike the TEB, which was disappointing in useful and documented information, ETHREAD and KTHREAD are two structures which are much more defining and important to the concept of a Thread and contain usable information, as well as pointers to other useful structures.

7.1 Executive Thread (ETHREAD)

The ETHREAD structure contains information about LPC, IRP, Scheduling (and Timers, Semaphores, Events) as well as Create and Exit Times. Its structure is well defined and easy to understand:

```
Public Type ETHREAD
    Tcb                         As KTHREAD
    CreateTime                  As LARGE_INTEGER
    ExitTime                    As LARGE_INTEGER
    LpcReplyChain               As LIST_
    KeyedWaitChain              As LIST_
    ExitStatus                  As NTSTATUS
    OfsChain                    As Long
    PostBlockList               As LIST_ENTRY
    TerminationPort             As Long ' // TERMINATION_PORT
    ReperLink                   As Long ' // ETHREAD
    KeyedWaitValue              As Long
    ActiveTimerLock             As KSPIN_LOCK
    ActiveTimerList             As LIST_ENTRY
    Cid                         As CLIENT_ID
    LpcReplySemaphore           As KSEMAPHORE
    KeyedWaitSemaphore          As KSEMAPHORE
    LpcReplyMessage             As Long ' // LPC_MESSAGE
    LpcWaitingOnPort            As Long ' // LPC_PORT_OBJECT
    ImpersonationInfo           As Long ' // PS_IMPERSONATION_INFORMATION
    IrpList                     As LIST_ENTRY
    TopLeverlIrp                As Long
    DeviceToVerify              As Long ' // DEVICE_OBJECT
    ThreadsProcess              As EPROCESS
    StartAddress                As Long
    LpcReceivedMessageId        As Long
    ThreadListEntry             As LIST_ENTRY
    RundownProtect              As EX_RUNDOWN_REF
    ThreadLock                  As EX_PUSH_LOCK
    LpcReplyMessageId           As Long
    ReadClusterSize             As Long
    GrantedAccess               As ACCESS_MASK
    CrossThreadFlags            As ETHREAD_FLAGS
End Type
```

Tcb

This field points to the Thread Control Block, also called KTHREAD, which contains Thread information that the Kernel uses.

CreateTime

This field contains the time when the Thread was created.

ExitTime

This field contains the time when the Thread was exited.

LpcReplyChain

This field contains a List Entry pointing to LPC Replies.

KeyedWaitChain

This field contains a List Entry pointing to Keyed Wait Events.

ExitStatus

This field contains the Exit Status (NTSTATUS, not Win32 Status Code) of the Thread.

OfsChain

This field contains a List Entry of unknown members.

PostBlockList

This field contains a List Entry of all the Objects that hold a reference to this Thread. The Thread won't be killed until those references are broken.

TerminationPort

This field contains the LPC Port to be used for Thread termination.

ReaperLink

This field is a pointer to itself, used by the Thread reaper when terminating the Thread.

KeyedWaitValue

This field is used for Keyed Wait Synchronization Events.

ActiveTimerLock

This field contains a Spin Lock protecting the Thread's Timers when they are being modified by Kernel routines.

Cid

This field contains a structure containing the PID and TID (same as in the TEB Structure)

ActiveTimerList

This field is a List Entry that points to the Thread's active Timers.

LpcReplyMessage

This field contains a pointer to an LPC_MESSAGE structure, referring to an LPC Message that this Thread will send (or just sent) as an LPC Reply.

KeyedWaitSemaphore

This field contains a semaphore that is used for each Keyed Wait Event.

LpcReplySemaphore

This field contains a semaphore that is used for each LPC Reply.

LpcWaitingOnPort

This field contains a pointer to an LPC_PORT_OBJECT structure which defines the LPC Port on which the Thread is waiting for LPC communications.

ImpersonationInfo

This field contains a pointer to a structure used when the Thread is impersonating another one.

IrpList

This field is a List Entry of the current IRPs (Interrupt Request Packet) associated with this Thread.

TopLeverlIrp

This field contains the first IRP that the Thread must process.

DeviceToVerify

This field contains a pointer to a Device Object structure associated with this Thread.

ThreadsProcess

This field contains a pointer to the EPROCESS Structure of the Process that owns this Thread.

StartAddress

This field contains the Thread's Kernel Start Address (Explained in the Expert Chapter).

LpcReceivedMessageId

This field contains a the Message ID of the last LPC Message that was received by the Thread

RundownProtect

This field contains a reference that is used to keep the ETHREAD Structure alive and protect it from rundown while it's being created.

ThreadListEntry

This field is a List Entry of all the other Threads in the Process.

ThreadLock

This field contains a Push Lock used when modifying the Thread's structures.

LpcReplyMessageId

This field contains the LPC Message ID of the LPC Reply that was last sent by the Thread.

ReadClusterSize

This field contains the Memory Cluster Size used by some Mm* Kernel functions.

GrantedAccess

This field contains a the Access Mask of the Access that the Thread has to itself.

CrossThreadFlags

This field contains different Thread Flags (declared in the accompanied files, and commented).

If ETHREAD seems so small and compact, it's because it references a lot of other structures which will be shown later. It is noticeable that ETHREAD doesn't really contain any critical Scheduler information for the Kernel. That's because the Executive doesn't really care about it, because Scheduling is done entirely by the Kernel, as KTHREAD will show.

7.2 Kernel Thread (KTHREAD)

The KTHREAD structure is primarily responsible for delegating all Thread information to the Kernel itself, which is why it's predominantly composed of members describing Priorities, Affinities, Waits, Locks, APCs and IRQLs. Unlike ETHREAD, which could be useful for a programmer, KTHREAD should usually be left alone.

```
Public Type KTHREAD
    Header                  As DISPATCHER_HEADER
    MutantListHead          As LIST_ENTRY
    InitialStack            As Long
    StackLimit              As Long
    TEB                     As Long
    TlsArray                As Long
    KernelStack             As Long
    DebugActive             As Byte
    State                   As Byte
    Alerted(1)              As Byte
    Iopl                    As Byte
    NpxState                As Byte
    Saturation              As Byte
    Priority                As Byte
    ApcState                As KAPC_STATE
    ContextSwitches         As Long
    IdleSwapBlock           As Byte
    Spare0(2)               As Byte
    WaitStatus              As NTSTATUS
    WaitIrql                As Byte
```

```
    WaitMode                As KPROCESSOR_MODE
    WaitNext                As Byte
    WaitReason              As Byte
    WaitBlockList           As KWAIT_BLOCK
    WaitListEntry           As LIST_ENTRY
    SwapListEntry           As SINGLE_LIST_ENTRY
    WaitTime                As Long
    BasePriority            As Byte
    DecrementCount          As Byte
    PriorityDecrement       As Byte
    Quantum                 As Byte
    WaitBlock(3)            As KWAIT_BLOCK
    LegoData                As Long
    KernelApcDisable        As Long
    UserAffinity            As Long
    SystemAffinityActive    As Byte
    PowerState              As Byte
    NpxIrql                 As Byte
    InitialNode             As Byte
    ServiceTable            As Long
    Queue                   As KQUEUE
    ApcQueueLock            As KSPIN_LOCK
    Timer                   As KTIMER
    QueueListEntry          As LIST_ENTRY
    SoftAffinity            As KAFFINITY
    Affinity                As KAFFINITY
    Preempted               As Byte
    ProcessReadyQueue       As Byte
    KernelStackResident     As Byte
    NextProcessor           As Byte
    CallbackStack           As Long
    Win32Thread             As Long
    TrapFrame               As Long ` // KTRAP_FRAME
    ApcStatePointer(1)      As Long ` // KAPC_STATE
    PreviousMode            As Byte
    EnableStackSwap         As Byte
    LargeStack              As Byte
    ResourceIndex           As Byte
    KernelTime              As Long
    UserTime                As Long
    SavedApcState           As KAPC_STATE
    Alertable               As Byte
    ApcStateIndex           As Byte
    ApcQueueable            As Byte
    AutoAlignment           As Byte
    StackBase               As Long
    SuspendApc              As KAPC
    SuspendSemaphore        As KSEMAPHORE
    ThreadListEntry         As LIST_ENTRY
    FreezeCount             As Byte
    SuspendCount            As Byte
    IdealProcessor          As Byte
    DisableBoost            As Byte
End Type
```

Header

This field contains the information used by the Kernel Dispatcher.

MutantListHead

This field contains List Entries for the Mutants (Mutexes) that this Thread owns.

InitialStack

This field contains a pointer to the Kernel-Mode Stack of this Thread.

StackLimit

This field contains the end of the Kernel-Mode Stack of the Thread.

TEB

This field contains a pointer to the Thread's TEB.

TlsArray

This field contains a pointer to the Thread Local Storage information of this Thread.

KernelStack

This field contains a pointer to a Kernel Stack of this Thread.

DebugActive

This field is a flag on whether the Thread is being debugged.

State

This field contains the Thread's current state.

Alerted

This field specifies whether the Thread is currently in an Alerted state.

Iopl

This field contains the I/O Privilege Level for this Thread.

NpxState

This field contains Floating Point status information for this Thread.

Priority

This field contains the Thread's current priority.

Saturation

This field contains the Thread's current priority saturation.

ApcState

This field contains the current APC State of the Thread.

ContextSwitches

This field counts the number of Context Switches that the Thread has gone through (switching Contexts/Threads).

IdleSwapBlock

This field contains an unknown data.

WaitStatus

This field contains the current waiting status for this Thread (used by calls like WaitForSingleObject etc) (in NTSTATUS, not Win32)

WaitIrql

This field contains the IRQL of the current Wait.

WaitMode

This field contains the mode of the current Wait.

WaitNext

This field contains a flag on whether the Thread has been marked for Waiting.

WaitBlockList

This field is a List Entry for the current Wait Blocks.

WaitListEntry

This field is a List Entry for the current Waits.

WaitReason

This field contains the reason for the Wait.

SwapListEntry

This field contains a List Entry for the current Kernel Stack Swaps done.

WaitTime

This field contains the time until a Wait will expire.

DecrementCount

This field is used for synchronizing priority changes.

PriorityDecrement

This field is used for synchronizing priority changes.

BasePriority

This field contains the base priority for this Thread.

Quantum

This field contains the Thread's Quantum (the time before a switch is made).

WaitBlock

This field contains a structure containing the real PID and TID (usually the same as in ClientId).

LegoData

This field contains the "Lego" data to return to a registered "Lego Notify" routine. It's an undocumented way to receive Thread Exit notifications.

KernelApcDisable

This field determines if Kernel-Mode APCs will be disabled for this thread.

UserAffinity

This field contains the Thread's Affinity in User-Mode.

SystemAffinityActive

This field specifies if the System-Wide Affinity is applied to this Thread.

NpxIrql

This field contains the IRQL of the Floating Point area.

PowerState

This field contains the Thread's current Power state.

ServiceTable

This field contains a pointer to the System Call Table for This Thread.

InitialNode

This field contains an unknown value.

Queue

This field contains a Queue for this Thread.

ApcQueueLock

This field is a Spin Lock protecting APC Queue modifications

Timer

This field contains the Timer used for this Thread.

QueueListEntry

This field contains List Entries the Thread's Queues

SoftAffinity

This field contains the Soft Affinity for this Thread.

ProcessReadyQueue

This field is used for Synchronization when the Thread is attached to a Process.

Preempted

This field specifies if the Thread will be preempted or not.

Affinity

This field contains the Thread's Kernel Affinity.

KernelStackResident

This field determines if the Thread's Kernel Stack will remain in memory after the Thread exists.

NextProcessor

This field contains the next processor on which the Scheduler will try to run this Thread on.

CallbackStack

This field contains the stack to be used when coming back from a User-Mode Callback.

Win32Thread

This field contains a pointer to the associated WIN32_THREAD structure.

ApcStatePointer

This field contains pointers for the three possible APC States of the Thread.

TrapFrame

This field contains a pointer to a Kernel Trap Frame used for Exceptions and other Traps.

EnableStackSwap

This field determines if Kernel Stack Swaps are to be used on this Thread.

PreviousMode

This field contains the previous mode of the Thread (Kernel or User). This determines if parameters passed to System Calls will be validated.

LargeStack

This field determines whether a Large Stack was created for this Thread, as a result of becoming a Graphical Thread (using Win32K).

ResourceIndex

This field seems to contain some kind of information about Resources (another Kernel method of locking access to data). Its exact meaning or usefulness is unknown.

KernelTime

This field contains the time that the Thread has spent in Kernel Mode.

UserTime

This field contains the time that the Thread has spent in User Mode.

SavedApcState

This field contains the last saved APC State of the Thread.

Alertable

This field determines if the Thread can be in an alertable state and receive APCs.

ApcStateIndex

This field contains an Index about APC States.

ApcQueueable

This field determines if the APCs for this Thread can be queued.

AutoAlignment

This field contains an unknown flag on whether Auto-Alignment (of what?) should be used.

SuspendApc

This field contains the APC that should be used when the Kernel wants to suspend this Thread.

StackBase

This field contains the Base Address of this Thread's Stack.

SuspendSemaphore

This field contains a Semaphore used when suspending the Thread.

ThreadListEntry

This field is a List Entry pointing to other Threads of this Process.

FreezeCount

This field contains a count on how many times the Thread has been frozen.

SuspendCount

This field contains a count on how many times the Thread has been suspended.

IdealProcessor

This field determines the Ideal Processor on which the Thread should run on.

DisableBoost

This field determines if Priority Boosting should be allowed for this Thread.

At first glance, KTHREAD seems much more massive then ETHREAD, compared to EPROCESS versus KPROCESS. This is because ETHREAD contains only information that the Executive would need since a Thread, as mentioned before, is very tightly related to the CPU, while instead Processes are the ones more OS-dependent (Because they use OS/Executive Facilities). As such, the OS cares about Threads mostly in their scheduling and code execution. Threads rarely own Executive Objects. On the contrary, with Processes, almost everything is done at a layer above the CPU (except the Memory Allocation), so the Kernel itself cares little about the Process, therefore KPROCESS isn't very important.

7.3 Impersonation (PS_IMPERSIONATION_INFORMATION)

Impersonation is the ability of a thread to execute using different security information than the process that owns the thread. Typically, a thread in a server application impersonates a client. This allows the server thread to act on behalf of that client to access objects on the server or validate access to the client's own objects. The data for this is kept into the PS_IMPERSIONATION_INFORMATION, linked from ETHREAD:

```
Public Type PS_IMPERSONATION_INFORMATION
    Token                        As Long ` // TOKEN
    Flags                        As Long
    ImpersonationLevel           As SECURITY_IMPERSONATION_LEVEL
End Type
```

The Token member of this structure will point to a TOKEN structure which contains the Impersonation Token for this Thread.

7.4 APC State (KAPC_STATE)

An APC is an Asynchronous Procedure Call, which is a way for the Kernel to asynchronously call functions. There are 2 possible modes, Kernel-Mode and User-Mode, each with their own APC State and APCs in a List Entry. The KAPC_STATE Structure is also linked to the Process that owns it:

```
Public Type KAPC_STATE
    ApcListHead(1)               As LIST_ENTRY
    Process                      As Long ' // KPROCESS
    KernelApcInProgress          As Byte
    KernelApcPending             As Byte
    UserApcPending               As Integer
End Type
```

This concludes the Chapter on the Thread Kernel-Mode Structures. Notice that many of the structures in ETHREAD and KTHREAD have not been documented. This is because most of them refer to Kernel Synchronization Objects like Semaphores, Locks or LPC Messages/Ports. Because these structures and objects will be dealt with in upcoming articles, it is not of use to describe them here. Note however that the structure definitions have been included in the accompanying source files, which will allow you to read them properly.

The next Chapter will deal with how the Job Object exists in Kernel-Mode (there is no User-Mode Job Structure) and what its fields represent.

8. Kernel-Mode Job Structure

A Job has only a single Executive Job Structure, called EJOB. Because the Kernel doesn't need to know anything about Jobs, since everything is at the Executive Level, there is no KJOB Structure. Furthermore, because Jobs could potential disable their security features if the Job structure would exist in User-Mode, it is stored in Kernel-Mode to be inaccessible except by API Calls.

8.1 Executive Job (EJOB)

A Job has only a single Executive Job Structure, called EJOB. Because the Kernel doesn't need to know anything about Jobs, since everything is at the Executive Level, there is no KJOB Structure. Furthermore, because Jobs could potential disable their security features if the Job structure would exist in User-Mode, it is stored in Kernel-Mode to be inaccessible except by API Calls.

```
Public Type EJOB
   Event                        As KEVENT
   JobLinks                     As LIST_ENTRY
   ProcessListHead              As LIST_ENTRY
   JobLock                      As ERESOURCE
   TotalUserTime                As LARGE_INTEGER
   TotalKernelTime              As LARGE_INTEGER
   ThisPeriodTotalUserTime      As LARGE_INTEGER
   ThisPeriodTotalKernelTime    As LARGE_INTEGER
   TotalPageFaultCount          As Long
   TotalProcesses               As Long
   ActiveProcesses              As Long
   TotalTerminatedProcesses     As Long
   PerProcessUserTimeLimit      As LARGE_INTEGER
   PerJobUserTimeLimit          As LARGE_INTEGER
   LimitFlags                   As Long
   MinimumWorkingSetSize        As Long
   MaximumWorkingSetSize        As Long
   ActiveProcessLimit           As Long
   Affinity                     As Long
   PriorityClass                As Long
   UIRestrictionsClass          As Long
   SecurityLimitFlags           As Long
   Token                        As Long
   Filter                       As Long  ' // PS_JOB_TOKEN_FILTER
```

```
    EndOfJobTimeAction          As Long
    CompletionPort              As Long
    CompletionKey               As Long
    SessionId                   As Long
    SchedulingClass             As Long
    ReadOperationCount          As LARGE_INTEGER
    WriteOperationCount         As LARGE_INTEGER
    OtherOperationCount         As LARGE_INTEGER
    ReadTransferCount           As LARGE_INTEGER
    WriteTransferCount          As LARGE_INTEGER
    OtherTransferCount          As LARGE_INTEGER
    IoInfo                      As IO_COUNTERS
    ProcessMemoryLimit          As Long
    JobMemoryLimit              As Long
    PeakProcessMemoryUsed       As Long
    PeakJobMemoryUsed           As Long
    CurrentJobMemoryUsed        As Long
    MemoryLimitsLock            As FAST_MUTEX
    JobSetLinks                 As LIST_ENTRY
    MemberLevel                 As Long
    JobFlags                    As Long
End Type
```

Event

This field contains the Event used when Job Times expire or need to be checked.

ProcessListHead

This field is a List Entry pointing to the EPROCESS Structures of the Processes that this Job contains.

JobLinks

This field is a List Entry pointing to all the other EJOB Structures on the current system.

JobLock

This field contains the Lock used when the Job Object is modified by the Kernel.

TotalKernelTime

This field contains the accumulated time spent in Kernel-Mode by the Processes contained by this Job.

TotalUserTime

This field contains the accumulated time spent in User-Mode by the Processes contained by this Job.

ThisPeriodTotalUserTime

This field contains the accumulated time spend in Kernel-Mode by the Processes contained by this Job, for a specific time interval.

ThisPeriodTotalKernelTime

This field contains the accumulated time spend in Kernel-Mode by the Processes contained by this Job, for a specific time interval.

TotalPageFaultCount

This field contains the accumulated Page Faults by the Processes contained by this Job.

TotalProcesses

This field counts the number of Processes in this Job.

ActiveProcesses

This field counts the number of active Processes in this Job.

TotalTerminatedProcesses

This field counts the number of terminated Processes in this Job.

PerProcessUserTimeLimit

This field determines the maximum number of CPU Time that each individual Process part of this Job can spend in User-Mode.

PerJobUserTimeLimit

This field determines the maximum number of CPU Time that all Processes part of this Job can spend in User-Mode.

LimitFlags

This field specifies the Job Limits enabled for this Job.

MinimumWorkingSetSize

This field determines the minimum Memory usage of this Job.

MaximumWorkingSetSize

This field determines the maximum Memory usage of this Job.

Affinity

This field determines the affinity for this Job.

ActiveProcessLimit

This field determines maximum number of Processes that can be part of this Job.

PriorityClass

This field determines the priority of this Job.

UIRestrictionsClass

This field determines the GUI Restrictions in effect for this Job.

Token

This field contains a pointer to the Security Token of this Job.

SecurityLimitFlags

This field determines the Security Restrictions in effect for this Job.

Filter

This field contains a Job Object Filter Structure which determines which Objects were filtered from this Job.

EndOfJobTimeAction

This field determines the Job Action to perform when the Job has gone past its maximum runtime.

CompletionPort

This field determines the Completion Port used when an I/O Completion Event occurs.

SchedulingClass

This field determines the Scheduling class for this Job.

SessionId

This field contains the Session ID that the current Job is running on.

CompletionKey

This field determines the Completion Key to be used when an I/O Completion Event occurs.

ReadOperationCount

This field contains the number of I/O Read Operations performed.

WriteOperationCount

This field contains the number of I/O Write Operations performed.

OtherOperationCount

This field contains the number of I/O Misc Operations performed.

ReadTransferCount

This field contains the number of I/O Read Transfers performed.

WriteTransferCount

This field contains the number of I/O Write Transfers performed.

OtherTransferCount

This field contains the number of I/O Misc Transfers performed.

IoInfo

This field contains a structure which contains the same members and values as the members shown on this page (Transfer/Operation Counts)

ProcessMemoryLimit

This field determines the maximum Memory that each Process can use.

PeakProcessMemoryUsed

This field contains the Peak Memory Usage of a Process in this Job.

PeakJobMemoryUsed

This field contains the Peak Memory Usage of all Processes in this Job.

JobMemoryLimit

This field determines the Ideal Processor on which the Thread should run on.

MemoryLimitsLock

This field determines the Ideal Processor on which the Thread should run on.

CurrentJobMemoryUsed

This field determines the Ideal Processor on which the Thread should run on.

JobSetLinks

This field determines the Ideal Processor on which the Thread should run on.

JobFlags

This field determines the Ideal Processor on which the Thread should run on.

MemberLevel

This field determines the Ideal Processor on which the Thread should run on.

10. Process Creation (*CreateProcessExW*) [EXPERT]

The following chapter will detail Process Creation to its most intricate details, both in User-Mode and Kernel-Mode APIs. APIs that another API uses will sometimes be also explained, leading to more APIs following after that. While it might seem confusing at first, everything is organized into subsections.

All the Kernel 32 *CreateProcessXxx* APIs end up at this API, which is the one that actually starts Creating the Process. The Xxx stubs perform the job of finding the image file and converting it from ANSI to Unicode. The file is then opened with a call to *NtOpenFile*. The handle is passed on to *NtCreateSection*, with the SEC_IMAGE parameter.

10.1 NtCreateSection (*SEC_IMAGE*)

The first step for the Native API, as usual, is to validate and check the parameters that were sent. Once this is done, the real function, part of the Memory Management system, *MmCreateSection* is called, which creates the Section Object. Next, *CcWaitForUninitializeCacheMap* is called, which synchronizes the data section with the NT Cache Manager.

Next, a temporary control area is created, and an ERESOURCE lock is acquired, which lets the kernel synchronize with the File System.

Moving on, *MiFindImageSectionObject* is now called to check if the file has already been Memory Mapped into a Section Object.

10.1.1 MiFindImageSectionObject

This API will use the FCB (File Control Block) that the File system must implement and check if the file is already cached somewhere in memory, by walking all the control areas and see if any correspond.

Once the check has been done, *MiLockPfnDatabase* is called to make a lock on the Page Frame Number so no other code can touch the Virtual Memory. An error handler code then runs to make sure that no other Kernel Thread is conflicting with the current execution, and that the Control Area hasn't been deleted. *MiUnlockPfnDatabase* is called to unlock the PFN.

Because the file is already mapped, the new Section Object will share the same control area, and a new reference will be added. *MiFlushDataSection* is called to flush the file data, the temporary control area created before is destroyed, and the ERESOURCE file system lock is released.

If the file has not yet been mapped however, the temporary control area created will be used instead. *MiInsertImageSectionObject* will be called to insert this Control Area into the File Object.

10.1.2 MiInsertImageSectionObject

This API simply inserts the control area it received as a parameter into the File's Section Object pointers, linking the file to the Section. This is added to a list.

Next, *MiCreateImageFileMap* is called to actually do the mapping and create the real control area.

10.1.3 MiCreateImageFileMap

The first thing this API does is call *FsRtlGetfileSize* to find out the size of the file. The Image Header is then read and validated. Memory is allocated, and an Even Object is initialized. *MiGetPageForHeader* is first called to allocate a page (Virtual Memory) for the Image Header. Then, *MiFlushDataSection* flushes the data section. *IoPageRead* can now be called, and the File, MDL (the memory allocated) and Event are sent as parameters. The API then receives an event that the read was completed, and *MiMapImageHeaderInHyperSpace* is called to map the Image Header in a Kernel Memory area called Hyperspace, where process data is stored. The Image Header is checked again, and is finally being read to verify the size of the Image that will be required in memory. The API calculates how much memory will be needed to map the file, and PTEs (Page Table Entries, they map Virtual Memory to Physical Memory) are created accordingly. The API returns with the memory location where the file was mapped.

Now that the file has been mapped in memory, *KeAcquireQueuedSpinLock* is called to once again ensure that no other code is messing around with what the API is accessing. *MiRemoveImageSectionObject* is called, with the old control area as a parameter, since we can now discard it, because *MiCreateImageFileMap* returned the true control area, which is then passed on to *MiInsertImageSectionObject* again (it does the same thing as before, but in the new control area). The old control area is deleted, and more checks are made in regards to having exclusive access to what is going on. If the checks work out well, *KeReleaseQueuedSpinLock* is called to release the Spinlock created earlier. *ObCreateObject* is called to create a skeleton Kernel Section Object, and the structure it returns is filled out with all the info acquired from the API calls executed until now. This is then passed on to *ObInsertObject*, which creates the final Section Object.

Back to *CreateProcess*, the file is now loaded in memory. Some checks are then made, in regards to VDM (DOS programs), WoW64 (Windows on Windows 64-bit VM), restrictions, or CMD files. *NtQuerySection* is then called to get the ImageInformation data from the PE header, and then *LdrQueryImageFileExecutionOptions* will read the registry and check if PE Loader debugging is enabled. Some special handling is also performed if the file is POSIX (UNIX). At this point *NtCreateProcessEx* is called and creates the Kernel Process Object.

10.1 NtCreateProcessEx

This Native API is the first that commences the Kernel Mode Process Creation, and is responsible for almost all the work (although it calls many other APIs). First, *ObCreateObject* is called to create the basic Process Object, which is actually an EPROCESS Structure. *PspInheritQuota* is called to set up the Process Quotas, followed by *ObInheritDeviceMap* which creates the DosDevices for the Process's Device Map (so that the process can access LPT1, AUX ports, for example). If the process is being forked (cloned), such as is the case with POSIX applications, the Virtual Memory is also cloned, and some settings are copied. If a debug or exception LPC port were passed, they are also referenced in EPROCESS. *PspInitializeProcessSecurity* follows, which creates all the necessary security information for the process. *MmCreateProcessAddressSpace* is now called to create the Address Space for the Process (the Virtual Memory Allocation).

10.2.1 MmCreateProcessAddressSpace

The function first creates some locks, such as the Working Set Lock (the Working Set is the memory that the process will use), and allocates a Page Directory for the Process (this is an area where Physical->Virtual Memory conversion tables will be located. Hyperspace is then initialised for this process, and the page directory is mapped inside it. At this point, the process is added to the Memory Manager's internal Process List. The System Page Directories are then filled (containing information about core kernel memory locations), and the locks are released.

Now that the Address Space is created, a check is made if the process is being forked, in which case the Object Table also gets cloned. *KeInitializeProcess* is called to initialise the Process in the Kernel Scheduler, which means that the Priority and Affinity are passed on as parameters. The scheduling is then saved into EPROCESS. If the process is being forked, *ObInitProcess* is called. Next, *MmInitializeProcessAddressSpace* is called. Depending on the type of Process (Boot, System, Forked or New), the Address Space is set up accordingly. The ClientID is created for the process (used for debugging) with *ExCreateHandle*, and the process is added to a Job, if it's part of one (see the last Chapter).

The PEB can now be created by using *MmCreatePeb*.

10.2.2 MmCreatePeb

This API attaches itself to the target process' memory, and prepares to write the PEB structure. First, it maps the NLS Tables (Font/Character Set/Language Data) and calls *MmCreatePebOrTeb*, a helper function which simply allocates some Virtual Memory, locks it, maps it, and then unlocks it. The PEB is then initialised, with values from the default system TEB, the NLS tables and the Image Header. The API then de-attaches itself from the process, and returns the address of the PEB.

The PEB is created, so it is safe to add the process to the internal Kernel Process List, called PsActiveProcessHead. *SeCreateAccessStateEx* is called to create an AccessState structure for the permissions of the Process. Then, ObInsertObject is called with the EPROCESS structure, the AccessState, and the DesiredAccess as parameters, which returns a handle to the process. This handle is the PID, and is written into User-Mode (PEB). *ObGetObjectSecurity* gets a SecurityDescriptor for the Process, and it gets passed on to *SeAccessCheck* to verify the process's rights. Finally, *KeQuerySystemTime* is called to save the Process's Create Time in the PEB. The Process is now created!

Back in CreateProcess, after the Kernel Process Object has been created, and the PEB is also loaded, *NtSetInformationProcess* is called, with the ProcessPriorityClass parameter and the priority that the process should run in. Also, if CREATE_DEFAULT_ERROR_MODE is a flag in dwCreationFlags when the CreateProcess API was called, the *NtSetInformationProcess* API is called again, with the ProcessDefaultHardErrorMode parameter. These parameters simply specify what the process should do when the system generates a Hard Error. Next, *BasePushProcessParameters* is called, which pushes some parameters into the new process.

10.3 BasePushProcessParameters

Back in User-Mode, this API basically receives a bunch of parameters and writes them into the PEB, and also does some work by itself. Firstly, the DLL and EXE search path is built, as well as the Command Line, the Current Directory, the Desktop Info and the Window Title. This information is all sent to the API function *RtlCreateProcessParameters*, which puts them in an RTL_USER_PROCESS_PARAMETERS Structure which is also shown in this article. The API then calls *NtAllocateVirtualMemory* to allocate a buffer for the environment block (not the PEB, but the string structure with information that the Environ$ VB function returns, such as %SYSDIR%). *NtWriteVirtualMemory* is called to write the buffer to the process. The Process Parameter Block then gets filled in with other information, such as the console handles, the Profile flags and the window settings, if applicable. The PEB is modified to point to the structure. Following that, the Application Compatibility Data is allocated and created, and the pointer is once again written in the PEB.

BaseCreateStack follows, which creates the user-mode stack, followed by *BaseInitializeContext*, creating the initial thread context.

10.4 BaseCreateStack

Nothing too complicated goes on here. First the Maximum Stack Size is read from the Image Header, and the Minimum Stack Size is read from the PEB. That memory is then allocated with *NtAllocateVirtualMemory*, and the StackTop is calculated and committed with a call to the *NtAllocateVirtualMemory* API again (this time with MEM_COMMIT) as a parameter. If enough space is available, a guard page is created with *NtProtectVirtualMemory*.

The process is now created, but as said before, it is simply a dumb memory structure. No code has been executed, or even loaded. For this, the main thread must be created... *CreateProcess* calls *NtCreateThread*.

11. Thread Creation (*CreateProcessExW* Part 2) [EXPERT]

Thread Creation is the cornerstone of any code execution on the OS. Without a thread, the executable code from the image file would never be loaded, nor would any DLLs. Basically, you wouldn't even notice a file was executed, since nothing would happen. The main API responsible for creating a thread is *NtCreateThread*.

11.1 NtCreateThread

Because memory allocation has already taken place, creating the main thread is a much less nasty procedure then creating the process, and is a lot simpler to understand. First of all, a reference to the Process Object is taken, and then the Thread Object itself is created, returning an ETHREAD structure. The structure is then pointed to its process, and other fields inside it, used in various places like the Memory Manager, I/O Manager and Schedulers, are initialized. *MmCreateTeb* is then called to create the TEB, much in the same way that *MmCreatePeb* works. The exact details of this will not be mentioned, as they are extremely similar to the Process Object Creation and the author does not wish to bore the reader. The starting address of the thread is saved, and *KeInitThread* is called to initialize the thread.

11.1.1 KeInitThread

The API sets the priority and affinity of the parent Process, and initializes the Thread Context. The EIP (the starting address to run) is set to *PspUserThreadStartup*, which is a Kernel Stub that will later call the real entry point of the executable file. The Thread State is now set to Initialized.

PspLockProcessExclusive is called to temporarily lock the Process while new operations are being done. The Active Threads field of EPROCESS is incremented by one, and the Thread is added to the Thread List of the process. *KeStartThread* is then called to set up the Thread.

11.1.2 KeStartThread

Doing slightly more work then *KeInitThread*, this API also sets up a bunch of fields in ETHREAD, such as the Quantum and Scheduling Boosts. This time, the Priority and Affinity are being applied, and the IdealProcessor field chosen. The StackCount is also incremented by one.

The Process Lock is released with *PspUnlockProcessExclusive*, and checks are made to see if this is the first thread. If that's the case, then means that a process has just been created, and the Process Creation Notification is called if a driver has registered it. Likewise, if this is a Job, then it means the first element of the Job has been created, and the notification is sent. Finally, notifications for Thread Creation are also sent. If the thread was created with the Suspended flag, then *KeSuspendThread* is called.

Just like when creating a Process, *SeCreateAccessStateEx* is called, followed by *ObInsertObject* to create the final Thread Object. *KeReadyThread* is then called, which means that the Kernel can begin executing the code any second (however, not in the case of a Main Thread, because it is created with the suspended flag).

Back to *CreateProcess*, the thread is now ready so *CsrClientCallServer* is called with the BasepCreateProcess parameter, in order to register this new process and thread with the CSRSS subsystem (see my previous article on Native API for a brief overview of CSRSS).

11.2 CsrClientCallServer

Almost there! The only thing that remains left to do is to register the Process with the CSRSS Subsystem, and this API will do all the work. To begin, it will first call *AcquireProcessStructureLock* and duplicate the handles to the process. Then, it will create an internal CSRSS Structure for the process, and copy whichever information it needs from the PEB. It will set the CsrApiPort to the process's exception port, so that it can use LPC to communicate with the process and be aware of any crashes or other exceptions. If the process is being debugged, it will also set up the debug port. The internal CSRSS Thread Structure is created, and the ThreadCount and ThreadList parameters are changed accordingly, as well as in the internal CsrThreadHashTable. A PID and TID are then written in the structure, as well as the process and thread handles. Every DLL loaded in CSRSS is then notified about the creation of this new process, and the Kernel is told that the background process has been created. Finally, *ReleaseProcessStructureLock* is called.

Finally, if the Process is part of a Job (see the last Chapter), the restrictions are applied. Now that all is ready, *NtResumeThread* is called, so that the main thread can run.

11.3 NtResumeThread

NtResumeThread is the final API when loading an executable (although it can always be called later when the process creates more threads). It basically sets the suspended state of the Thread to false, unfreezes it, and calls *KiWaitTest*, which puts the Thread in the Scheduler's Queue. At any point now, the Kernel can decide to run the Thread. The API then calls *KiExitDispatcher*, and new threads can be scheduled. When the Scheduler runs the thread, it will execute at *PspUserThreadStartup* (remember that's what *NtCreateThread* set the initial execution pointer to).

The startup routine will call the PE Loader to actually load the code, map the DLLs, and execute the file, by calling *LdrInitialize*, so this is the last API remaining to look at.