

Windows Vista APC Internals

By Enrico Martignetti

First edition, May 2009

Table of Contents

Introduction And Acknowledgements	4
APC Basic Concepts	4
APC Process Context.....	4
APC Types	5
APC Initialization	6
Initial APC Scheduling.....	8
Special And Regular Kernel Mode APCs.....	9
Scheduling.....	9
Linking _KAPC to Its List.....	9
Directing The Thread to Execute The APC	9
Triggering Thread Dispatching	19
Delivery.....	19
Effect of _KTHREAD.SpecialApcDisable Being Set	20
Kernel APC Delivery When SpecialApcDisable Is Clear.....	21
Special Vs. Regular Kernel APCs	23
User Mode APCs.....	23
Scheduling.....	23
Linking _KAPC to Its List.....	24
Directing The Thread to Execute The APC	24
nt!PsExitSpecialApc and The Special User APC.....	24
Triggering Thread Dispatching	25
Delivery.....	25
nt!KiDeliverApc Invocation.....	25
Effect of User Mode APCs on Kernel Wait Functions.....	26
nt!KiDeliverApc for User APCs	27

nt!KiInitializeUserApc And The User Mode Context	28
User Mode APC Delivery by nt!KiUserApcDispatcher.....	30
nt!NtContinue And The User Mode Context.....	30
Appendix - The Test Driver.....	31
IMPORTANT WARNING	31
Driver Package Contents	31
Driver	32
TestClt.....	32
CommonFile.....	32
Loading And Running The Driver.....	32
What The Driver Can Do.....	32
APC Spurious Interrupt Test.....	32
Sample Trace.....	34
Atypical Traces.....	36
APC Disable Test.....	39
APC User Mode Test And APC User Mode Test #2.....	39
References.....	40

Introduction And Acknowledgements

There are several articles on the subject of Windows APCs and in particular *Inside NT's Asynchronous Procedure Call* by Albert Almeida (Dr. Dobb's Journal, Nov 01, 2002) contains a good deal of information about them. To be fair, that article has guided me throughout the writing of this one. My attempt has been to add a few more details to the concepts presented and a little insight into undocumented Windows features, like the GATEWAIT and DEFERREDREADY thread states, which are tied to how APCs work.

Furthermore, Almeida's article was published when Windows XP was current, so another goal of this work has been to revise the information presented there for Windows Vista. Unless otherwise specified, all the material contained here has been verified against the Windows Vista SP1 Ultimate Edition code (for x86).

APC Basic Concepts

APCs are a Windows functionality that can divert a thread from its regular execution path and direct it to execute some other code. The most important thing about an APC is that when one is scheduled, it is targeted to a specific thread.

APCs are little documented: the kernel APIs to use them are not public and their inner working is only partially covered. What makes them interesting is that they are tied with the way Windows dispatches threads, thus, by analyzing them we can better understand this core Windows feature.

Windows internals books often mention the fact that an APC is scheduled by means of a software interrupt. This raises the question of how can the system guarantee that this interrupt will run in the context of a particular thread. This is, ultimately, the purpose on an APC, but software interrupts can break into the execution of whatever thread is current. We will see in this document how Windows accomplishes this.

Code executed by means of an APC can run, depending on the APC type, at a dedicated IRQL level, which, not surprisingly, is called APC level.

So, what are APCs used for? Among other things:

- The I/O manager uses an APC to complete an I/O operation in the context of the thread which initiated it.
- A special APC is used to break into the execution of a process when it must terminate.
- The kernel implementation of APCs is under the hood of Windows API functions like `QueueUserAPC` and `ReadFileEx/WriteFileEx`, when used to perform asynchronous I/O.

APC Process Context

Normally, a Windows thread executes in the context of the process which created it. However, it is possible for a thread to attach itself to another process, which means it will execute in its context.

Windows accounts for this when managing APCs: they can be scheduled to run in the context of the process which owns the thread, or in the context of whichever process the thread is currently attached to.

A detailed explanation of how this is achieved can be found in [1] in the section titled *APC Environments*. I did not revise this material for Windows Vista, however, all the APC control variables described there are still present and used in the rest of the Vista APC implementation. Below is a summary of the *Environments* section

Windows maintains the state of APCs waiting to execute in the `_KAPC_STATE` data structure, shown below (output from the WinDbg dt command):

```
kd> dt nt!_KAPC_STATE
+0x000 ApcListHead      : [2] _LIST_ENTRY
+0x010 Process         : Ptr32 _KPROCESS
+0x014 KernelApcInProgress : UChar
+0x015 KernelApcPending : UChar
+0x016 UserApcPending  : UChar
```

The main kernel data structure `_KTHREAD` has two members of type `_KAPC_STATE`, named `ApcState` and `SavedApcState`, which are called *APC environments*. `ApcState` is the environment for APCs targeted at the current thread context: regardless of whether the thread is attached to its own process, or another one, this member contains APCs for the current process context and therefore deliverable ones. `SavedApcState` stores APCs for the context which is not current and that must wait. For instance, if the thread is attached to a process other than its owner and there are APCs for the owner process, they go into `SavedApcState` and have to wait until the thread detaches from the other process.

From the explanation above, we can understand that when a thread attaches itself to another process, `ApcState` is copied into `SavedApcState` and re-initialized. When the thread detaches itself, `ApcState` is restored from `SavedApcState`, which is then emptied. Also, we can see that the kernel components responsible for dispatching APCs, always look into `ApcState` for deliverable ones.

The `_KTHREAD` structure also stores an array of 2 pointers called `ApcStatePointer`, whose members hold the addresses of `ApcState` and `SavedApcState` (that is, they point inside `_KTHREAD` itself). The kernel updates these pointers so that the first one always points to the APC environment for the thread owning process and the second one points to the environment for an eventual process the thread is attached to.

For instance, if the thread is not attached to another process, the Environment for the owning process is also currently active, therefore it is stored into `ApcState` and `ApcStatePointer[0]` holds the address of `ApcState`.

Finally, `_KTHREAD.ApcStateIndex` stores the index of the pointer to `ApcState`, i. e. the active environment. Thus, if the thread is attached to another process, the owning process environment is into `SavedApcState`, `ApcStatePointer[0]` points to `SavedApcState`, because it always points to the owning process environment. In turn, `ApcStatePointer[1]` points to `ApcState`, because it always points to the attached process environment. Finally, `ApcStateIndex` is set to 1, because it is `ApcStatePointer[1]` which is pointing to the active environment.

When we schedule an APC, we can specify that we want to add it to the environment for the owning process (i. e. the one pointed by `ApcStatePointer[0]`), to the secondary one (`ApcStatePointer[1]`) or to whichever environment is currently active (i. e. the one at `ApcState`, no matter what).

APC Types

APCs come in three kinds.

Special kernel mode APCs execute kernel mode code at APC IRQL (i. e. 1). They are truly asynchronous events that divert a thread from its normal execution path to the kernel mode function to be executed, which is called *KernelRoutine*.

If an SK APC is queued to a thread which has entered a waiting state by calling one of the four waiting routines KeWaitForSingleObject, KeWaitForMultipleObjects, KeWaitForMutexObject, or KeDelayExecutionThread, the thread is awakened to execute the KernelRoutine and will re-enter its waiting state afterwards.

Normally, SK APCs are always delivered as soon as the target thread is running and its IRQL drops below APC level. However, APC delivery can be disabled on a thread by thread basis. When the SpecialApcDisable member of the _KTHREAD structure is not 0, APC delivery does not occur for the corresponding thread, not even for SK APCs.

Regular kernel mode APCs execute kernel mode code at PASSIVE IRQL. Like SK APCs they can break into the execution of threads and bring them out of waiting states. However, they are dispatched only under more restrictive conditions, which will be detailed later.

User mode APCs call user mode code and are executed under even more restrictive conditions than kernel mode ones: they are dispatched to a thread only when it willingly enters an alertable wait state (except for one special case, detailed later). This happens when a thread calls one of the four waiting routines KeWaitForSingleObject, KeWaitForMultipleObjects, KeWaitForMutexObject, or KeDelayExecutionThread with Alertable = true and WaitMode = User.

Thus, normally, U APCs don't asynchronously break into the execution of a thread. They are more like a form of queueable work items: they can be queued to a thread at any time and the thread code decides when to process them.

APC Initialization

An APC is represented by the _KAPC structure which can be examined in WinDbg with the command dt nt!_KAPC. The output from the command is shown below:

```
kd> dt nt!_KAPC
+0x000 Type           : UChar
+0x001 SpareByte0    : UChar
+0x002 Size          : UChar
+0x003 SpareByte1    : UChar
+0x004 SpareLong0    : Uint4B
+0x008 Thread        : Ptr32 _KTHREAD
+0x00c ApcListEntry  : _LIST_ENTRY
+0x014 KernelRoutine : Ptr32
+0x018 RundownRoutine : Ptr32
+0x01c NormalRoutine : Ptr32
+0x020 NormalContext : Ptr32 Void
+0x024 SystemArgument1 : Ptr32 Void
+0x028 SystemArgument2 : Ptr32 Void
+0x02c ApcStateIndex : Char
+0x02d ApcMode       : Char
+0x02e Inserted      : UChar
```

A _KAPC structure is initialized by calling nt!KeInitializeApc, which has the following (undocumented) prototype:

```

NTKERNELAPI VOID KeInitializeApc (
    IN PRKAPC Apc,
    IN PKTHREAD Thread,
    IN KAPC_ENVIRONMENT Environment,
    IN PKKERNEL_ROUTINE KernelRoutine,
    IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
    IN PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
    IN KPROCESSOR_MODE ApcMode,
    IN PVOID NormalContext
);

```

This prototype was published in [1] and is still valid for Windows Vista SP1.

Calling nt!KeInitializeApc does not schedule the APC yet: it just fills the members of the _KAPC. To actually schedule the APC one more step will be required.

Ke InitializeApc sets the Type field to a constant value (0x12) which identifies this structure as a _KAPC and the Size field to 0x30, i.e. the size of the structure rounded up to a multiple of 4.

The Thread field is set to the corresponding function parameter and contains a pointer to the _KTHREAD for the thread the APC is intended for.

The ApcListEntry is not set by nt!KeInitializeApc. Rather, it is used when the APC is actually scheduled, to chain it to a list of pending APCs for the thread.

The KernelRoutine field, taken from the function input parameter, stores a pointer to the function to be called when the APC is dispatched. This is the actual code which will be executed in the context of the thread targeted by the APC. Every APC has a KernelRoutine and, depending on its type it can have a NormalRoutine as well.

The function pointed by KernelRoutine is called at APC IRQL.

The NormalRoutine and ApcMode function parameters determine the type of the APC.

If NormalRoutine is 0, this is a *special kernel mode APC*.

For this kind of APCs nt!KeInitializeApc sets _KAPC.ApcMode to 0, which stands for kernel mode and _KAPC.NormalContext to 0. The corresponding function parameters are ignored.

If NormalRoutine is not 0 and ApcMode is set to 0, this is a *regular kernel mode APC*; _KAPC.ApcMode and _KACP.NormalContext are set from the function parameters.

This kind of APC will still execute kernel mode code, as denoted by ApcMode = 0, but it is less privileged than a special one, which means it can be executed only under certain conditions, which will be detailed later. NormalRoutine is stored in the corresponding _KAPC member and is the address of a function which will be called when the APC is delivered.

When such an APC is serviced, the KernelRoutine is called first, at APC IRQL. Afterwards, the NormalRoutine is called at PASSIVE IRQL. Both functions execute in kernel mode.

As we will see in more detail later, the KernelRoutine has a chance to prevent the execution of the NormalRoutine or to change the address which will be called, before the NormalRoutine comes into play.

If NormalRoutine is not 0 and ApcMode is set to 1, this is a *user mode APC*, which will therefore call the NormalRoutine in user mode.

For user mode APCs, nt!KeInitializeApc sets the _KAPC.ApcMode and _KAPC.NormalContext members to the value of its corresponding input parameters.

The NormalContext field is passed to both the kernel routine and the normal routine when they are called.

SystemArgument1 and 2 are not set by nt!KeInitializeApc. Instead, they are set when an APC is scheduled and they are passed to the callback routines as well. We will see this process in much more detail later.

ApcStateIndex determines the environment the APC is for and will later be used as an index into the ApcStatePointer array of the thread, to select the _KAPC_STATE where to store the APC. If it has any value other than 2, nt!KeInitializeApc stores this value in the corresponding _KAPC field. If it is 2, the function sets _KAPC.ApcStateIndex to the value of _KTHREAD. ApcStateIndex for the input thread. Thus, the value 2 means: schedule the APC for whatever environment is current when nt!KeInitializeApc is called.

Finally, the Inserted field is set to 0 by nt!KeInitializeApc and this represents the fact that this APC has not been queued to its thread yet. nt!KeInsertQueueApc, which is used to schedule APCs, will set this field to 1.

Initial APC Scheduling

All kinds of APCs are queued to their thread by calling nt!KeInsertQueueApc. The steps performed by this function are the same, regardless of the type of APC for which it is called.

This function takes as input 4 stack parameters:

- A pointer to the _KAPC structure for the APC.
- Two user-defined values SystemArgument1 and SystemArgument2
- A priority increment

nt!KeInsertQueueApc begins by acquiring a spinlock stored in the _KTHREAD for the target thread and raising the IRQL to 0x1b, i. e. profile level. Thus the rest of its operation can be interrupted only by the clock and IPI interrupts.

Afterwards, it looks into a bit field at offset 0xb0 of the _KTHREAD for the target thread, to see if bit 6, i. e. ApcQueueable is set. If it's not, it aborts returning 0 (i. e. FALSE) to its caller.

The function then checks whether the Inserted field of the input _KAPC is already set to 1 and, if so, returns FALSE.

If the two checks above are successful, the APC is actually scheduled as follows.

The function copies the input SystemArgument1 and 2 into the corresponding fields of _KAPC. These value will be passed to the APC callbacks and offer a way to pass them context information when scheduling the APC, in contrast with NormalContext, whose value is set at _KAPC initialization time.

Afterwards, it sets _KAPC.Inserted to 1 and calls nt!KeInsertQueueApc with edx set to the input priority increment and ecx set to the address of the _KAPC.

nt!KiInsertQueueApc performs the greater part of the APC scheduling process, which will be analyzed in distinct sections for the different APC kinds.

Special And Regular Kernel Mode APCs

Scheduling

The scheduling performed by nt!KiInsertQueueApc can be split into two main steps: linking the _KAPC to a list of pending APCs and updating control variables for the target thread so that it will be diverted to process the waiting APC.

Linking _KAPC to Its List

The function begins by examining again _KAPC.ApcStateIndex. By now, we know that when nt!KeInitializeApc is called to initialize the _KAPC, this field is set from the Environment parameter and that the value 2 is a pseudo-value, which has the effect to set it to whatever environment is current at the time of the call.

nt!KiInsertQueueApc allows for yet another pseudo-value, 3, which has the same effect: if ApcStateIndex is found to be 3, it is replaced with _KTHREAD.ApcStateIndex at the time of the call.

Thus, 2 means current at the time of nt!KeInitializeApc, while 3 means current at the time of nt!KiInsertQueueApc.

The final value of _KAPC.ApcStateIndex is used to select the environment to which the _KAPC will be linked. nt!KiInsertQueueApc accesses the _KTHREAD for the target thread at offset +0x14c, where we find ApcStatePointer, i. e. the array of two pointers to _KAPC_STATE structures (see [APC Process Context](#)). ApcStateIndex is used as the array index to select the correct pointer and the _KAPC will be linked to the selected _KAPC_STATE.

The _KAPC_STATE structure is as follows:

```
+0x000 ApcListHead      : [2] _LIST_ENTRY
+0x010 Process          : Ptr32 _KPROCESS
+0x014 KernelApcInProgress : UChar
+0x015 KernelApcPending : UChar
+0x016 UserApcPending   : UChar
```

The ApcListHead array contains the head of the two APC lists for kernel mode and user mode. nt!KiInsertQueueApc uses the _KAPC.ApcMode field as an index into it (thus ApcListHead[0] is for kernel mode APCs) and links the _KAPC to the list.

The list for kernel mode APCs, is for both regular and special ones. A special kernel mode APC is linked to the list head, i. e., ahead of any eventual regular kernel mode APC already present. If other special APCs are already in place, the new one goes behind them (but still ahead of any regular one).

Directing The Thread to Execute The APC

In this section we are going to look at how the target thread is diverted to execute the APC. This is where the APC software interrupt *may* be used. As we are going to see, the interrupt is actually used only in particular cases, and control variables are used as well.

The first step of this process is to check whether the APC environment specified by _KAPC.ApcStateIndex is equal to the current environment, specified by _KTHREAD.ApcStateIndex.

If the two are not the same, the function just returns. This looks interesting: nothing is done to divert the target thread from its execution path. On one hand, it makes sense: as long as the thread is attached to a different environment, it can't process the APC being scheduled. On the other hand, this means that the code which switches the thread to a new environment will check whether there are APCs in the list for it and act to divert the thread to service them.

This is the first of many cases where the APC interrupt is *not* used.

This logic means that kernel mode APCs, even special ones, are kept waiting until the thread switches environment on its own accord.

If the environments match, `nt!KiInsertQueueApc` goes on to check whether the APC is for the currently executing thread, i. e. the one inside the function itself.

Kernel APC for The Current Thread

For this scenario, `nt!KiInsertQueueApc` sets `KernelApcPending` to 1 in the `ApcState` member of the target `_KTHREAD`.

As we saw in the section titled [APC Process Context](#), `ApcState` stores the main APC environment, i. e. the one currently active. Since we have already determined that our APC is for the active environment, we access the control variables for it directly from the `ApcState` member, instead of using the environment index and the pointer array `_KTHREAD.ApcStatePointer`.

`KernelApcPending` is a very important flag, which plays a crucial role in breaking into the execution of the thread and making it deliver the APC.

The SpecialApcDisable Flag

After setting this flag, `nt!KiInsertQueueApc` checks the flag `_KTHREAD.SpecialApcDisable`: if it's not 0, the function returns.

This means `SpecialApcDisable` can disable all kinds of kernel mode APCs, including special ones.

When will then the APC be dispatched? Part of the answer can be found in `nt!SwapContext`. This function is executed to load the context of a thread after it has been selected to run and checks the `KernelApcPending` flag: if it is set and if `SpecialApcDisable` is clear, it triggers APC dispatching for the thread. We will see in greater detail later how this happens. For now, let's just keep in mind that having `KernelApcPending` set guarantees to have APC dispatched the next time the thread is scheduled to run (providing that `SpecialApcDisable` has been cleared in the meantime).

This begs the question of what happens if the current thread clears its own `SpecialApcDisable`: do pending APCs have to wait until it passes through `nt!SwapContext` or are they fired right away? If we try setting a breakpoint on write accesses to `SpecialApcDisable` for a thread of the system process, we discover there are several kernel functions which update this field, so, there is no single "EnableApc" function which clears it.

However, all the functions I caught in a random test, update `SpecialApcDisable` according to the following pattern. To disable APCs, they decrement it when it is 0, therefore bringing it to 0xffff. To reenable them, they increment it, then check if the final value is 0. If this is the case, they check whether the list for kernel mode APCs has any and, if so, call a function named `nt!KiCheckForKernelApcDelivery`.

This function triggers kernel APC delivery for the current thread. If the IRQL is passive, the function raises it to APC and calls `nt!KiDeliverApc` (detailed in a later section), which delivers the APCs.

Otherwise, sets `_KTHREAD.ApcState.KernelApcPending = 1` and requests an APC interrupt, which will call `nt!KiDeliverApc` when the IRQL drops below APC. We will add a few more detail about this interrupt in a short while.

Hence, at least in the functions I observed (`nt!NtQueryInformationFile`, `nt!KeLeaveGuardedRegion`, `nt!IopParseDevice`), kernel APCs are serviced as soon as `SpecialApcDisable` is zeroed, if the IRQL allows it.

The APC Interrupt

Now let's get back to `nt!KiInsertQueueApc`. We just analyzed what happens if `_KTHREAD.SpecialApcDisable` is not 0; if, on the other hand, it is 0, the function requests an APC software interrupt by calling `hal!HalRequestSoftwareInterrupt`, then quits.

Before exploring the effect of this interrupt, let's note that the actions performed by `nt!KiInsertQueueApc` when the APC is for the current thread, are the same, both for regular and special kernel mode APCs. Hence, the restrictions on regular APCs, which are dispatched only when certain conditions apply, are implemented into `nt!KiDeliverApc`, as we will see later.

The APC interrupt can't be processed right now, because inside `nt!KiInsertQueueApc` the IRQL is at 0x1b, but will eventually click when the IRQL is lowered inside `nt!KiInsertQueueApc`. When this happens, `nt!KiDeliverApc` is called.

This leads us to an interesting question. In the time between the interrupt request and its servicing, the timer keeps interrupting the processor, given that its IRQL is 0x1c, higher than the current one. The handler for this interrupt could detect that the quantum for the current thread has expired and request a DPC software interrupt to invoke the thread dispatcher.

In this scenario, when the IRQL is lowered to PASSIVE, there are *two* outstanding software interrupts: the DPC and the APC one, with the former having a higher IRQL and therefore taking priority.

Thus, the dispatcher is invoked, which may select a different thread to run. Eventually, the IRQL will drop to PASSIVE and the APC interrupt will click, *in the context of the wrong thread*. However, two steps in the overall APC logic save the day.

The first one is the APC dispatcher, `nt!KiDeliverApc`, which is ultimately invoked by the APC interrupt, and allows for the case of being called and finding that there are no APCs in the list. If this happens, it just returns. Thus, when invoked for the wrong thread it's harmless.

Still, we may think we are at risk of "loosing" the APC for the right thread, given that the interrupt is now gone. This is not the case, because the `KernelApcPending` flag is still set and `nt!SwapContext` will look at it when the original thread will get its chance to run again, re-triggering a call to `nt!KiDeliverApc`.

Can The APC Interrupt Really Break into The Wrong Thread?

The test driver for this article, for which more details can be found in the [appendix](#), shows how the scenario of the previous section can actually happen. The test function for this test is `ApcSpuriousIntTest`.

This function installs hooks at the beginning of `nt!SwapContext` and `nt!KiDeliverApc`, which track when these functions are called, along with some data at the time of the call. Then it raises the IRQL to DISPATCH, schedules the APC and wastes time waiting for the DISPATCH interrupt to be requested. Afterwards, it lowers the IRQL to passive, having both the APC and DPC interrupts outstanding. The driver writes to the debugger console a trace of what happens.

We are now going to analyze a trace captured with it. It begins with the APC being initialized, then the first captured event we see is the call to SwapContext:

```
APCTEST - Thr: 0X851D2030; APC initialized: 0X850F7A9C
```

```
APCTEST - SWAP CONTEXT trace
```

```
APCTEST - Current IRQL: 0x1b
APCTEST - Current thread: 0X851D2030
APCTEST - Current thread K APC pending: 1
APCTEST - Current thread K APC list empty: 0
APCTEST - Current thread U APC pending: 0
APCTEST - Current thread U APC list empty: 1

APCTEST - New thread: 0X84CBB460
APCTEST - New thread K APC pending: 0
APCTEST - New thread K APC list empty: 1
APCTEST - New thread U APC pending: 0
APCTEST - New thread U APC list empty: 1

APCTEST - APC INT: 1
```

The SwapContext trace shows the IRQL at which the function runs (0x1b), then some data on the current thread (the one about to be suspended). First of all we see that this thread is indeed the same one which initialized the APC (0X851D2030), then we see that the KernelApcPending flag for it is set to 1 and that its kernel APC list is not empty (this is shown by the 0, i. e. false, value on the corresponding line).

Then we see that the new thread (0X84CBB460), has KernelApcPending clear, an empty kernel APC list and the same goes for the user APC data. This tells us that this thread has no APC to deliver.

Finally, the line labeled "APC INT" shows the status of the APC interrupt flag at the time of the trace: it is set, meaning that the APC interrupt resulting from nt!KeInsertQueueApc is pending.

After the swap context we see a call to nt!KiDeliverApc traced:

```
APCTEST - DELIVER APC trace
```

```
APCTEST - Current IRQL: 0x1
APCTEST - Caller address: 0X81BB42F7
APCTEST - Trap frame: 00000000
APCTEST - Reserved: 00000000
APCTEST - PreviousMode: 0

APCTEST - Thread: 0X84CBB460
APCTEST - Thread K APC pending: 0
APCTEST - Thread K APC list empty: 1
APCTEST - Thread U APC pending: 0
APCTEST - Thread U APC list empty: 1
```

This trace shows that the now current thread is indeed the same that nt!SwapContext was about to resume before, and confirms once more that this thread has no pending APCs of any kind.

Thus, we can see that this APC interrupt really is a spurious one: it occurred because it was pending, but occurred in the wrong thread, so it is useless. It is also harmless, because nt!KiDeliverApc returns without causing any more trouble.

The remainder of the trace shows that nt!SwapContext is called to resume the original thread, which still has its pending APCs, then, even though there is no outstanding APC interrupt, nt!KiDeliverApc gets called and finally delivers our APC. This is confirmed by the fact that we see the trace from the kernel routine:

APCTEST - SWAP CONTEXT trace

```
APCTEST - Current IRQL: 0x1b
APCTEST - Current thread: 0X84CBB460
APCTEST - Current thread K APC pending: 0
APCTEST - Current thread K APC list empty: 1
APCTEST - Current thread U APC pending: 0
APCTEST - Current thread U APC list empty: 1

APCTEST - New thread: 0X851D2030
APCTEST - New thread K APC pending: 1
APCTEST - New thread K APC list empty: 0
APCTEST - New thread U APC pending: 0
APCTEST - New thread U APC list empty: 1

APCTEST - APC INT: 0
```

APCTEST - DELIVER APC trace

```
APCTEST - Current IRQL: 0x1
APCTEST - Caller address: 0X81BB42F7
APCTEST - Trap frame: 00000000
APCTEST - Reserved: 00000000
APCTEST - PreviousMode: 0

APCTEST - Thread: 0X851D2030
APCTEST - Thread K APC pending: 1
APCTEST - Thread K APC list empty: 0
APCTEST - Thread U APC pending: 0
APCTEST - Thread U APC list empty: 1
```

APCTEST - KERNEL ROUTINE trace

```
APCTEST - Thread: 0X851D2030
APCTEST - Thread K APC pending: 0
APCTEST - Thread K APC list empty: 1
APCTEST - Thread U APC pending: 0
```

```
APCTEST - Thread U APC list empty: 1
APCTEST - TRACE MESSAGE: Returned from KeLowerIrql
```

Finally, the last trace line is a message written right after the call to KeLowerIrql returns, indicating that all of the above took place while we were inside this function: as soon as the IRQL was lowered, the thread was preempted by the DISPATCH interrupt, it regain control later, delivered its APC because of the APC interrupt, and finally resumed execution, returning from KeLowerIrql.

This shows how the “state” information that a kernel APC is pending is actually stored in KernelApcPending, which is a per-thread flag and not in the fact that an APC software interrupt is outstanding. The software interrupt is merely an instrument to *attempt* to trigger APC dispatching. In another test performed with the driver, which will be detailed later, we will see how APCs start to fire merely by setting KernelApcPending, without requesting any software interrupt.

Kernel APC for Another Thread

For this scenario, nt!KiInsertQueueApc acquires another spinlock protecting the processor control block (_KPRCB) of the executing processor. Note that inside this function we already have a lock protecting the APC lists for the thread; now we are synchronizing accesses to the processor data structures as well.

Afterwards, the function sets KernelApcPending inside ApcState, which ensures the APC will eventually fire at some point.

It then checks the state of the target thread, which, in this scenario, is not the one executing nt!KiInsertQueueApc.

The thread state is stored in the State member of the _KTHREAD structure and its documented values include READY (1), RUNNING (2), WAIT (5). There are also undocumented values like DEFERREDREADY (7 - for a thread ready to run for which a processor must be found) and GATEWAIT (8 - a different kind of wating state).

The next sections describe the behaviour of nt!KiInsertQueueApc for the different thread states.

APC Targeted at A RUNNING Thread

If the thread state is RUNNING, it must be executing on another processor, because we already know it is a different thread from the one executing nt!KiInsertQueueApc, thus an IPI (Interprocessor Interrupt) is sent to its processor, to trigger an APC interrupt for that thread. As in the previous section, we are not totally sure that this interrupt will be serviced by the target thread, but its KernelApcPending variable will keep track of the APC for as long as it takes.

It is interesting to observe that the IPI is requested by calling nt!KiIpiSend with the following register values:

Ecx = a bitmask corresponding to the number of the processor stored in the NextProcessor field of _KTHREAD for the target thread. This suggests that this field stores the identifier of the processor running the thread (a fact confirmed by an analysis of nt!KiDeferredReadyThread, which is the function responsible to choose a processor for a ready thread).

Edx = 1. Given that an APC interrupt is being requested, with APC IRQL being 1, this suggests that edx stores the IRQL for the interrupt to be sent to the target processor.

Before requesting the IPI, `nt!KiInsertQueueApc` releases the spinlock for the current processor; after the call to `nt!KiIpiSend`, it returns.

APC Targeted at A WAIT THREAD

A waiting thread is not eligible to run because it entered the WAIT state. Under certain conditions, `nt!KiInsertQueueApc` awakens it and sends it to execute the APC. Thus, when possible, the APC does not have to wait until the thread enters the running state to be dispatched.

`nt!KiInsertQueueApc` begins by comparing the `WaitIrql` member of the target `_KTHREAD` with 0, i. e. PASSIVE IRQL. If `WaitIrql` is not 0, `nt!KiInsertQueueApc` exits.

This gives us an interesting insight on this `_KTHREAD` member: when a thread enters the WAIT state, this member records the IRQL it was running at, before transitioning to WAIT.

This is further confirmed by an analysis of `nt!KeWaitForSingleObject`, which stores into this member the current IRQL before calling the dispatcher routines which will put the thread into the WAIT state.

So, while it's true that the IRQL is not related to threads, but, rather, is an attribute of a *processor* (at any given time a processor is running at a given IRQL), it's also true that the IRQL that was current when the thread went into waiting is recorded here.

Hence, the reasoning behind `nt!KiInsertQueueApc` is straightforward: if the thread was not running at PASSIVE it was running at APC and as such APC interrupts were masked for it. Thus, it cannot be sent to service an APC interrupt, because this would break the IRQL-based interrupt masking rule.

As a side note, for a waiting thread `WaitIrql` should either be PASSIVE or APC, because a thread running at IRQL greater than APC cannot enter the WAIT state.

As always, the fact that `KernelApcPending` is set guarantees that when the right conditions will apply, the APC will eventually be delivered.

If the thread is indeed waiting at PASSIVE, `nt!KiInsertQueueApc` checks `SpecialApcDisable`: if it's not 0, it exits without awakening the thread.

If, on the other hand, APCs are enabled and this is a special kernel mode APC, `nt!KiInsertQueueApc` awakens the target thread; it will be resumed by `nt!SwapContext` and will dispatch the APC.

If this is a regular kernel mode APC, two additional checks are performed.

First, the `KernelApcDisable` member of the target `_KTHREAD` is checked: if it's not 0, `nt!KiInsertQueueApc` exits without awakening the thread. Thus, `KernelApcDisable` acts like `SpecialApcDisable`, but for regular APCs only.

The second check involves the `KernelApcInProgress` member of `ApcState`: again, a non-zero value causes `nt!KiInsertQueueApc` to exit.

We will see in `nt!KiDeliverApc` that it sets this flag before calling the normal routine for a regular APC. This test means that if the thread entered the WAIT state while it was in the middle of a regular APC, it will not be hijacked by another regular APC. In other words, regular APC calls can't be nested.

We saw earlier that when the APC is targeted at the current thread, `nt!KiDeliverApc` is invoked (through the interrupt) regardless of the APC type, regular or special, which may seem in contrast with this behaviour. However, `nt!KiDeliverApc` performs the same checks and avoid dispatching a regular APC if either of these two flags is set. This means that not awakening the thread is a performance optimization: even if the thread had been awakened, it would not dispatch regular APCs.

Thread Awakening And nt!KiUnwaitThread

If nt!KiInsertQueueApc decides to awaken the thread, it does so by calling nt!KiUnwaitThread and this unveils an interesting piece of information.

The DDK documentation states that when a waiting thread is dispatched to process a kernel mode APC, it will reenter the WAIT state afterwards, i. e., the thread does not return from the KeWaitFor... function after the APC. On the other hand, the DDK also states that a waiting thread dispatched to process a user mode APC returns from the waiting function with a return value of STATUS_USER_APC.

By analyzing the code for user mode APCs, we can see that nt!KiUnwaitThread is used as well and, before calling it, the value 0xc0 is loaded into edx. This value is the same as STATUS_USER_APC.

On the other hand, for kernel APCs, edx is loaded with 0x100.

The ntstatus.h file defines STATUS_USER_APC as 0xc0 and, immediately below, STATUS_KERNEL_APC (not mentioned in the DDK help) as 0x100. This suggests that edx is loaded with a value that controls how the waiting function will behave: if it is STATUS_USER_APC, the waiting function will return to the caller, passing this value along; if it is STATUS_KERNEL_APC, it will resume the wait, the caller will never see this value and for this reason it is an undocumented one.

Another input parameter to nt!KiUnwaitThread is the priority increment that nt!KiInsertQueueApc received from its caller. Interestingly, if nt!KiInsertQueueApc had to spin in loops waiting for synchronizing locks, this value was incremented on each iteration, meaning the priority is boosted even more, to compensate for the time spent waiting.

Having awakened the target thread, nt!KiInsertQueueApc releases the spinlock for the processor control block and exits.

APC Targeted at A GATEWAIT Thread

GATEWAIT state, whose value is 8, appears to be another kind of waiting state.

For such a thread, nt!KiInsertQueueApc begins by synchronizing on the ThreadLock member of the target thread. This is used like a spinlock, without actually calling the regular spinlock functions: the processor spins in a loop until it succeeds in modifying its value exclusively.

Having acquired the lock, nt!KiInsertQueueApc checks whether the thread state is still GATEWAIT or has changed and, if the latter is true, exits (releasing all the locks). This is consistent with the fact that for states other than RUNNING, WAIT and GATEWAIT, nt!KiInsertQueueApc does nothing more than setting KernelApcPending (which at this point has already been done).

This suggests that ThreadLock protects the thread State: functions changing State do so only after having acquired the lock.

Afterwards, nt!KiInsertQueueApc goes through the same checks it does for WAIT-ing threads: WaitIrql must be PASSIVE, SpecialApcDisable must be clear and either this is a special APC or KernelApcDisable is clear, along with KernelApcInProgress.

If all the checks pass, nt!KiInsertQueueApc does a sort of unwaiting of the thread all by itself. It unchains the thread from a list to which is linked by means of its WaitBlock[0] member and puts it into DEFERREDREADY state, like nt!KiUnwaitThread does for WAIT threads.

nt!KiInsertQueueApc also sets the WaitStatus member of _KTHREAD to 0x100, i. e. the same "waiting return" value passed to nt!KiUnwaitThread for WAIT threads. nt!KiUnwaitThread does something

similar: stores the value it receives into `WaitStatus`, albeit by ORing the new value (passed into `edx`) with the current one. It could be that there are higher bits in the `WaitStatus` field which `nt!KiUnwaitThread` needs to preserve.

Furthermore, `nt!KiInsertQueueApc` chains the thread to a list pointed by the `DeferredReadyListHead` member of the `_KPRCB` structure for the executing processor.

An analysis of `nt!KiReadyThread` and `nt!KiDeferredReadyThread` shows that `DEFERREDREADY` is a state in which a thread ready to run is placed, before choosing the processor on which it will run; from `DEFERREDREADY` the thread will go into `READY` if the chosen processor is busy running another thread or into `STANDBY` if the processor can immediately start executing the thread.

Thus, both `nt!KiUnwaitThread` and `nt!KiInsertQueueApc` start the same transition which will result in the thread becoming `RUNNING` and dispatching its APCs.

WAIT Threads vs GATEWAIT Ones

It is interesting to compare the steps `nt!KiUnwaitThread` goes through for `WAIT` threads, with the one taken by `nt!KiInsertQueueApc` for `GATEWAIT` ones.

Thread Waiting Lists

For a `WAIT` thread, `nt!KiUnwaitThread` unlinks the thread from the objects it is waiting on. The thread is linked to them through a list pointed by its `WaitBlockList` member, so `nt!KiUnwaitThread` walks this list and unchains each node from its object.

These wait blocks, are explained in [2], p. 164. They are used to link a thread to the objects (events, mutexes, etc) it is waiting on, allowing for the fact that a thread can wait on multiple objects and each object can have more than one thread waiting on it.

To implement this m:n relationship, each thread has a list of wait blocks, where each node represents an object the thread is waiting on. A wait block is a `_KWAIT_BLOCK` structure with this layout:

```
+0x000 WaitListEntry      : _LIST_ENTRY
+0x008 Thread            : Ptr32 _KTHREAD
+0x00c Object            : Ptr32 Void
+0x010 NextWaitBlock     : Ptr32 _KWAIT_BLOCK
+0x014 WaitKey           : Uint2B
+0x016 WaitType          : UChar
+0x017 SpareByte         : UChar
```

The `NextWaitBlock` field chains the wait blocks of a single thread to one another.

The `Object` field points to the object the thread is waiting on.

When there are more than one thread waiting on the same object, the wait blocks pointing to that object are chained in a list by means of their `WaitListEntry` member.

In other words, a wait block may be part of two lists: the list of wait blocks of a thread and the list of wait blocks of an object.

`nt!KiUnwaitThread` walks the thread list through the `NextWaitBlock` member and unchains each `_KWAIT_BLOCK.WaitListEntry` from the list of the object.

`nt!KiInsertQueueApc` does something different for `GATEWAIT` threads.

It uses the GateObject member of _KTHREAD which points to a _KGATE structure. _KGATE, in turn has just one member: Header, which is a _DISPATCHER_HEADER.

The first byte of Header is used to synchronize with other processors, in the same way as a spinlock would be: the function spins in a loop until it succeeds in exclusively updating it. _KGATE is therefore used as a synchronization lock.

Having acquired the lock, nt!KiInsertQueueApc unlinks the thread from a list to which is chained by means of its WaitBlock[0] member. Thus GATEWAIT threads are also part of a list, but pointed by a different member of _KTHREAD.

WaitBlock[0] is itself of type _KWAIT_BLOCK, i. e. the same type used for WAIT thread. Here, however, WaitBlock[0] is embedded into _KTHREAD, while for WAIT threads a list of _KWAIT_BLOCK is pointed by _KTHREAD.WaitBlockList.

Furthermore, a GATEWAIT thread is simply unchained from the list pointed by WaitBlock[0].WaitListEntry. For a WAIT thread, the entire list of wait blocks is walked (through NextWaitBlock) and each node is unchained from the list for the target object.

It is also interesting to note that the _KTHREAD.GateObject member used to access the synchronizing gate is actually the same member which stores _KTHREAD.WaitBlockList: the definition for _KTHREAD places both fields in union at offset 0x64:

```
...
+0x064 WaitBlockList      : Ptr32 _KWAIT_BLOCK
+0x064 GateObject         : Ptr32 _KGATE
+0x068 KernelStackResident : Pos 0, 1 Bit
...
```

We can understand that nt!KiInsertQueueApc is treating offset 0x64 as a pointer to a _KGATE, because it uses its first byte for synchronization purposes. If the pointer were for a _KWAIT_BLOCK, the first byte would be part of a pointer variable; instead, for a _KGATE, the first byte is a byte value named Type (it's actually the first byte of the embedded _DISPATCHER_HEADER).

This means that the same pointer has two different meanings, depending on the thread state.

Thread Priority Boost

nt!KiUnwaitThread also applies a priority boost to thread; nt!KiInsertQueueApc does nothing similar for a GATEWAIT thread.

Checking for Outswapped Processes And Stacks

nt!KiUnwaitThread brings the thread out of WAIT by calling nt!KiReadyThread. This function does more than blindly changing the state to DEFERREDREADY. It checks whether the owning process is outswapped and, if this is the case, initiates inswapping for it.

If the process is resident, nt!KiReadyThread checks whether the thread stack is outswapped, by testing _KTHREAD.KernelStackResident. If it's not, puts the thread into TRANSITION and queues it for stack inswapping.

Otherwise if the stack is resident, nt!KiReadyThread puts (at last!) the thread into DEFERREDREADY.

On the other hand, `nt!KiInsertQueueApc` does not bother with any of these checks for GATEWAIT threads: it just sets their state to DEFERREDREADY. This means GATEWAIT-ing thread cannot have their stack outswapped and cannot be part of outswapped processes.

APC Targetd at Threads in Other States

For all other thread states, `nt!KiInsertQueueApc` exits.

However, it does so after having already set the `KernelApcPending` flag.

Triggering Thread Dispatching

As we saw earlier, `nt!KeInsertQueueApc` is the function used to schedule APCs, which internally calls `nt!KiInsertQueueApc`. When the latter returns, `nt!KeInsertQueueApc` has some more work to do.

We saw that `nt!KiInsertQueueApc` may awaken a thread that was in WAIT or GATEWAIT state. When this happens, the thread is placed into the DEFERREDREADY state and chained to the list of deferred ready threads for the executing processor (`_KPRCB.DeferredReadyListHead`). This means that the thread can run, i. e. nothing is blocking it, but a processor must be chosen to run it. To accomplish this, `nt!KeInsertQueueApc` calls `nt!KiExitDispatcher`.

The logic of `nt!KiExitDispatcher` is not covered in detail here, because it could fill an article by itself; however, this function calls `nt!KiDeferredReadyThread` for each thread in the deferred ready list, assigning it to a processor. Some of these threads may be assigned to a processor which is currently executing another thread with a lower priority of the one being assigned. If this is the case, the former thread must be preempted, thus, `nt!KiDeferredReadyThread` sends an IPI (interprocessor interrupt) to the chosen processor. It may also occur that `nt!KiDeferredReadyThread` assigns the awakened thread to the executing processor (the one executing `nt!KeInsertQueueApc`) and determines that the current thread must be preempted. This is taken care of by the remainder of `nt!KiExitDispatcher`, which, in this case, calls `nt!KiSwapContext` (which in turn calls `nt!SwapContext`).

The overall effect is that the thread dispatcher re-assesses which threads are going to run, to account for the threads awakened by DPCs.

Delivery

After `nt!KeInsertQueueApc` has run, `nt!KiDeliverApc` will eventually be called to process the APC.

This may happen because an APC software interrupt has been requested, or because `nt!SwapContext` detects it is resuming a thread with `ApcState.KernelApcPending` set and `SpecialApcDisable` clear.

In the latter situation, `nt!SwapContext` checks the IRQL at which the thread about to be resumed was running, when it was suspended. This IRQL value is stored on the thread stack, so `nt!SwapContext` pops it from there while restoring the thread context.

The popped IRQL can be either PASSIVE or APC (a thread can't be preempted when is running above APC). If `nt!SwapContext` finds out that the thread is returning to PASSIVE, returns 1, which tells its caller to call `nt!KiDeliverApc` right away. If, on the other hand, the thread is returning to APC, `nt!SwapContext` returns 0, but before leaving requests an APC interrupt. When the IRQL will drop to PASSIVE, the APC interrupt will fire and `nt!KiDeliverApc` will be called.

The APC interrupt may still fire after the thread has been preempted again. This is not a problem, however, because the next time `nt!SwapContext` will resume the thread, the process will be repeated.

nt!KiDeliverApc is called at APC IRQL and takes three stack parameters. We will call them PreviousMode, Reserved and TrapFrame, as in [1]. For kernel mode APCs only the first one is used and it is set to KernelMode (i. e. 0) to tell nt!KiDeliverApc it is being invoked to deliver kernel mode APCs.

One of the first things nt!KiDeliverApc does, is copying into a local the address of the _KPROCESS for the current process context (_KTHREAD.ApcState.Process). Before leaving, it will compare the saved value with the current one, taken again from _KTHREAD and, if they are not the same, it will bring the system down with the bug check code INVALID_PROCESS_ATTACH_ATTEMPT.

Effect of _KTHREAD.SpecialApcDisable Being Set

nt!KiDeliverApc begins by zeroing ApcState.KernelApcPending and, *only afterwards*, it checks if _KTHREAD.SpecialApcDisable is set. If this is the case, it returns without dispatching APCs and leaving ApcState.KernelApcPending set to 0.

This is an important issue: when KernelApcPending is 0, nt!SwapContext does not call nt!KiDeliverApc anymore. Even if _KTHREAD.SpecialApcDisable is later cleared, the pending APCs are not delivered until someone sets KernelApcPending again (or requests an APC interrupt).

The test driver accompanying this article demonstrate this behaviour, in the ApcDisableTest function.

This function initializes a special kernel mode APC, then chains it to the APC list and sets _KTHREAD.ApcState.KernelApcPending, without calling nt!KiInsertQueueApc (it updates these data structures directly). It also sets _KTHREAD.SpecialApcDisable.

ApcDisableTest then invokes a wait function, to ensure thread dispatching takes place and nt!SwapContext gets a chance to run for the thread.

The APC KernelRoutine writes a message on the debugger console, so we see that the APC is not delivered during the wait, due to SpecialApcDisable being set.

After the wait, ApcDisableTest shows that KernelApcPending is still set, which is consistent with the fact that nt!KiDeliverApc is not even called for this thread.

ApcDisableTest goes on by explicitly calling nt!KiDeliverApc. We see that the APC is not delivered, but after nt!KiDeliverApc returns, KernelApcPending is clear. We have thus reached the situation where, until someone sets it again, the pending APC will not be delivered.

This is confirmed by the fact that ApcDisableTest clears SpecialApcDisable, then waits for a few seconds; during the wait, the APC is not delivered.

Finally, ApcDisableTest, sets KernelApcPending and waits again. We see the KernelRoutine message telling us that the APC has been delivered.

This test also confirms that when KernelApcPending is set, APC delivery occurs without even requesting the APC interrupt, providing that APCs are not disabled.

Normally, when SpecialApcDisable is set, nt!KiDeliverApc is not even called. For instance, nt!SwapContext does not trigger nt!KiDeliverApc in this situation. The test confirmed this, when we saw that KernelApcPending was still set after the first wait.

However, the fact that nt!KiDeliverApc bothers to check SpecialApcDisable suggests that this scenario could happen; when this is the case, the pending APCs will stay there until something sets again ApcState.KernelApcPending.

In a previous section titled [The SpecialApcDisable Flag](#), we saw a behaviour of the functions which clear SpecialApcDisable consistent with how the flag is treated here. These functions check whether there are kernel APCs pending by inspecting the list, not by checking KernelApcPending. This accounts for the fact that this flag might have been cleared by nt!KiDeliverApc. Furthermore, these functions call nt!KiCheckForKernelApcDelivery, which either calls nt!KiDeliverApc or raises an APC interrupt. In the latter case, the function sets *KernelApcPending* before requesting the interrupt. The logic behind this is that when SpecialApcDisable is cleared, KernelApcPending is set for good measure, in case it had been cleared by nt!KiDeliverApc. As a final note, when nt!KiCheckForKernelApcDelivery calls nt!KiDeliverApc directly, does not bother to set KernelApcPending, because it's useless: this flag is used when we want nt!SwapContext to call nt!KiDeliverApc at some later time and the latter clears it in its early stages.

Kernel APC Delivery When SpecialApcDisable Is Clear

nt!KiDeliverApc goes on by checking whether there really are APCs in the list for kernel mode ones (ApcState.ApcListHead[KernelMode]). If the list is empty, it just returns (assuming it had been invoked with PreviousMode = KernelMode).

Otherwise, the function acquires the spinlock stored at _KTHREAD.ApcQueueLock (for the current thread) and makes sure that there still are pending APCs, leaving if there are none.

It then goes on by copying into local stack variables KernelRoutine, NormalRoutine, NormalContext, SystemArgument1, SystemArgument2.

If this is a special APC, it is then unchained from the list; afterwards the spinlock is released and the KernelRoutine called, with pointers to the local copies of the APC parameters.

Since the APC has already been unchained and the pointers passed to the KernelRoutine are for local copies of the APC data, the KernelRoutine can safely deallocate the _KAPC instance, if it wants to do so.

Afterwards, the kernel APC list is checked again, to see if there are any more waiting APCs and, if so, the process repeats itself from the spinlock acquisition.

Conversely, if a regular kernel mode APC is found, before unchaining it from the list _KTHREAD.KernelApcInProgress and _KTHREAD.KernelApcDisable are checked: if either one is not 0, the function terminates. We must remember that nt!KiDeliverApc can assume there are no more special APCs when it finds a regular one, because these are inserted in the list behind the special ones, so the function can terminate.

We will resume this point later, but the overall logic is easier to understand if we first look at what happens when these two flags are clear.

As is the case for a special APC, the regular one is unchained from the list, the spinlock is released and the KernelRoutine is called with addresses of local copies of the APC parameters. Indeed, regular APCs have a KernelRoutine, just like special ones.

When the KernelRoutine returns, nt!KiDeliverApc checks to see whether it has zeroed the NormalRoutine pointer (KernelRoutine received the pointer address, so it is free to modify it). If this happened, nt!KiDeliverApc loops to the next APC in the list, so for this one nothing more is done.

Otherwise, nt!KiDeliverApc sets _KTHREAD.KernelApcInProgress (one of the two flags tested before), lowers the IRQL to PASSIVE, and calls the NormalRoutine passing it NormalContext, SystemArgument1 and SystemArgument2. Interestingly, the address of NormalRoutine is taken from

the same pointer passed to KernelRoutine, which, therefore, has a chance to change the pointer and cause a different NormalRoutine to be executed.

When NormalRoutine returns, nt!KiDeliverApc raises the IRQL back to APC, clears KernelApcInProgress and loop to the next APC, leaving if there are none.

Let's focus back on the two checks on KernelApcDisable and KernelApcInProgress.

If either is found set, nt!KiDeliverApc terminates with ApcState.KernelApcPending clear, thus APC delivery won't happen again until someone sets it.

If this happens because KernelApcDisable was set, it means that the code which clears the flag will probably take care of setting ApcState.KernelApcPending again.

If this happens because KernelApcInProgress was set, the scenario is a bit different. Given that KernelApcInProgress is set by this very function before calling the APC normal routine, when we find it set, it means we have re-entered nt!KiDeliverApc while it was in the middle of a normal APC dispatching.

We therefore return from this nested call to nt!KiDeliverApc, the outer call will at some point be resumed and the APC dispatching loop will go on. Put in other words, leaving KernelApcPending clear does not matter, because we are still in the middle of nt!KiDeliverApc – the outer call.

But how come that nt!KiDeliverApc can be re-entered? Why is not protected by the IRQL rules? After all, the code invoking nt!KiDeliverApc does so only when the IRQL is below APC and this should prevent re-entrancy issues.

The point is that, as we just saw, nt!KiDeliverApc lowers the IRQL to PASSIVE to call the NormalRoutine, therefore unprotecting itself from nested APC interrupts. When the NormalRoutine executes at PASSIVE *anything* can happen: there is no IRQL protection in place.

In this nt!KiDeliverApc is quite special: it is not so common for a function to lower the IRQL *below* the value at which it is called. Normally, a function will perhaps raise it, then bring it back to its value. Setting the IRQL below the initial value means unmasking interrupts the caller of the function *might not expect*. Apparently, Windows is built so that nt!KiDeliverApc is always called in situations where bringing the IRQL all the way down to PASSIVE does no harm.

Given how this works, the KernelApcInProgress flag looks like a way to protect NormalRoutines from re-entrancy: even though the IRQL is not protecting them (they run at PASSIVE, after all), nt!KiDeliverApc does. A NormalRoutine writer can safely assume his routine will not be re-entered in the context of the same thread. It is nonetheless possible for it to be re-entered in the context of another thread, because at PASSIVE thread dispatching is free to occur and ApcState.KernelApcInProgress is a per-thread flag.

A further point of interest is that KernelApcDisable is checked in the middle of the dispatching loop, which means an APC KernelRoutine or NormalRoutine could conceivably set it and cause the loop to be aborted.

Finally, it's worth noting that special APCs are chained ahead of regular ones, therefore, when we find the first regular APC, which might cause the loop to be aborted because of a set KernelApcDisable, all the special APCs have already been processed.

Special Vs. Regular Kernel APCs

On the subject of how regular APCs are limited, `nt!KiDeliverApc` has the final say: it is this function which decides when such an APC will be delivered.

The function enforces the following rules:

- When the `NormalRoutine` of a regular APC is in progress, (`ApcState.KernelApcInProgress != 0`), other regular APCs are not delivered to the same thread. Even if the thread catches an APC interrupt and enters `nt!KiDeliverApc`.
- When `ApcState.KernelApcDisable != 0`, regular APCs are not delivered.

If `nt!KiDeliverApc` is called with `ApcState.KernelApcDisable != 0`, `KernelApcPending` is left zeroed, which prevents further calls to the function on context switches. `nt!KiDeliverApc` will be called again when `KernelApcPending` is set, perhaps by queuing another APC.

- Regular APCs have both a `KernelRoutine` and a `NormalRoutine`. The first is called at APC IRQL, the second at PASSIVE.

The `KernelRoutine` is called first and can change the address of the `NormalRoutine` that will be invoked or avoid the call altogether, by zeroing the `NormalRoutine` pointer.

Furthermore, the logic of `nt!KiInsertQueueApc` implies that:

- When a regular APC is queued to a waiting thread, the thread is not awakened if:
 - The `NormalRoutine` of another regular APC is in progress on the thread (`ApcState.KernelApcInProgress != 0`)

This implies that the `NormalRoutine` can enter a waiting state and know for sure that the thread will not service other regular APCs during the wait.

- `KernelApcDisable` has a non-zero value.

It is interesting to note how the DDK states that the conditions under which a waiting thread receives regular APCs are:

“thread not already in an APC, thread not in a critical section”

Thus, “in a critical section” `KernelApcDisable` is set to a non-zero value.

User Mode APCs

This section analyzes the scheduling and delivery of user mode APCs.

The main difference between these APCs and kernel mode ones is that their `NormalRoutine` is executed in user mode, so the APC implementation has to switch to ring 3 before calling `NormalRoutine`.

Scheduling

As for kernel mode APCs, the scheduling performed by `nt!KiInsertQueueApc` can be split into two main steps: linking the `_KAPC` to a list of pending APCs and updating control variables for the target thread so that it will process the waiting APC, when the right conditions apply.

Linking _KAPC to Its List

The APC environment is selected in the same way this is done for kernel mode APCs: here too `ApcStateIndex = 3` specifies the environment which is current when `nt!KiInsertQueueApc` is executing. The APC is then chained to the tail of the list for the selected environment (except for the special user APC detailed later). Note that each environment has two distinct lists: one for kernel mode APCs, another for user mode ones.

Directing The Thread to Execute The APC

As for kernel mode APCs, this stage begins by checking whether the APC is for the current environment and, in case it's not, the function leaves. User mode APCs directed at another environment remain pending, probably until the next environment switch.

If the APC is for the current thread, the result is the same: `nt!KiInsertQueueApc` simply exits. This makes sense if we recall when user mode APCs are dispatched: when a thread is waiting and alertable. Since the current thread is executing `nt!KiInsertQueueApc`, it is obviously not waiting, so nothing can be done at this time.

This leads us to a question: what if this thread attempts to enter an alertable wait after `nt!KiInsertQueueApc` has returned? It's likely that the wait functions will have to account for this possibility, otherwise the thread could be left waiting alertably, yet with pending APCs. We will see in a later section how this scenario is handled.

If the APC is for another thread, `nt!KiInsertQueueApc` goes on by acquiring the spinlock at `LockQueue` in the `_KPRCB` of the executing processor, and checking whether the target thread is in `WAIT` state. For any other thread state, `nt!KiInsertQueueApc` leaves. This again shows how user APCs have an immediate effect only if the target thread is waiting, otherwise they are left pending.

If, however, the thread is waiting, the `WaitMode` field is compared with 1: if it has a different value, `nt!KiInsertQueueApc` stops. Indeed, the DDK states that a waiting thread receives user mode APCs only if it is waiting in `UserMode`, which is defined to be 1 in `wdm.h`, so the compare we are seeing implements this criteria.

The next step is to check whether the `Alertable` bit in the `MiscFlags` member of `_KTHREAD` is set: if this is true, `_KTHREAD.UserApcPending` is set to 1 and the thread is awakened by calling `nt!KiUnwaitThread`, with `edx` set to `0c0h` (`STATUS_USER_APC`); `nt!KiUnwaitThread` also receives the priority boost passed to `nt!KiInsertQueueApc`, which will be applied to the awakened thread.

The steps above mean that when the thread is waiting in user mode and alertable, it is awakened with `UserApcPending` set to 1. As we are about to see, this flag will trigger the APC delivery process.

nt!PsExitSpecialApc and The Special User APC

`nt!KiInsertQueueApc` treats in a special way a user mode APC whose `KernelRoutine` is `nt!PsExitSpecialApc`.

First of all, sets `_KTHREAD.ApcState.UserApcPending` to 1, regardless of the state of the thread.

We will see later how this variable is the one which diverts the thread from its execution path when it enters user mode, sending it to deliver user APCs. However, for other user APCs, this variable is set only if the thread is found in an alertable wait state and is awakened. For this particular `KernelMode` routine instead, the variable is set regardless of the thread state, which means it will be hijacked as soon as it will switch from kernel mode to user mode.

Thus, this APC is allowed to asynchronously break into the target thread, something user APCs can't normally do.

Furthermore, for this KernelRoutine, the APC is queued at the head of the list of pending user APCs, while APCs for other KernelRoutines are chained in FIFO order.

Finally, for this APC, if `nt!KiInsertQueueApc` finds out that the target thread is in WAIT with `WaitMode = UserMode`, it awakens the thread, regardless of the value of the `Alertable` bit. Remember that with regular user mode APCs, the thread is awakened only if the bit is set.

These special steps allow the APC to be delivered as soon as possible: if the thread can be awakened without breaking kernel mode rules (e. g., a wait with `WaitMode = KernelMode` cannot be aborted, no matter what), it is. At any rate, having `UserApcPending` set, the thread will service its APCs as soon as it goes from kernel mode to user mode.

Given that `Ps` is the prefix for process support routine, `nt!PsExitSpecialApc` looks like a function involved in terminating a process, so perhaps it receives this special treatment to signal in a timely fashion to a process that it must terminate.

Triggering Thread Dispatching

As is the case for kernel mode APCs, `nt!KiInsertQueueApc` returns to `nt!KeInsertQueueApc`, which invokes the thread dispatcher. The code of `nt!KeInsertQueueApc` is the same for both kernel and user APCs and it has already been covered in the corresponding section for kernel mode ones.

Delivery

nt!KiDeliverApc Invocation

We saw in the previous section that for user mode APC no interrupt is requested, nor is `KernelApcPending` set, so we must ask ourselves how will `nt!KiDeliverApc` be invoked.

To answer this question, we must mention a few things about the system service dispatcher.

Many functions of the Windows API need to call executive services to do their job and do so by calling the system service dispatcher, whose task is to switch the processor to ring 0 and figure out the service routine to call.

The SSD switches to ring 0 by executing a `sysenter` instruction, which transfers control to `nt!KiFastCallEntry`. Before `sysenter`-ing, `eax` is loaded with a number identifying the executive service to be called.

`nt!KiFastCallEntry` calls the relevant executive service routine, then resumes ring 3 execution by means of a `sysexit` instruction.

Before doing this, `nt!KiFastCallEntry` checks whether `_KTHREAD.ApcState.UserApcPending` is set and, if so, calls `nt!KiDeliverApc` to deliver user APCs, before resuming the user mode execution context.

Thus, if `UserApcPending` is set, it hijacks the thread when it is about to return to ring 3.

For instance, a thread which entered an alertable wait state from user mode, by calling `WaitForSingleObjectEx`, resulted in the invocation of an executive service. When the thread is unwaited by `nt!KiInsertQueueApc`, it returns to ring 3 through the SSD code described above and is then sent to process user APCs, before finally returning from the wait function.

Effect of User Mode APCs on Kernel Wait Functions

The previous section brings us to an interesting detail about the documented DDK wait functions.

Consider, for instance, the DDK help about KeWaitForSingleObject: among the possible return values, STATUS_USER_APC is listed with the following description:

“The wait was interrupted to deliver a user asynchronous procedure call (APC) to the calling thread.”

The point is, the user APC has *not yet* been delivered: it will be, when the thread will re-enter user mode.

Thus, when KeWaitForSingleObject returns, the APC has merely aborted the wait, but it is still to be delivered. The wait has been aborted so that the thread can return to user mode and, thanks to UserApcPending being set, deliver the APC.

If the thread were to stay in kernel mode, for instance by entering another wait, the APC would remain pending.

User Mode APC Delivery Test in The Sample Driver

The sample driver accompanying this article shows this behaviour, by testing different wait cases. The driver dispatch function for this test is called ApcUserModeTest.

This function creates a separate thread which will queue the APC to the one executing it, then enters an alertable user-mode wait. The APC-requesting thread executes a function named ApcRequester.

When ApcRequester queue the APC to the original thread, the latter returns from KeWaitForSingleObject with STATUS_USER_APC. However, the APC has not yet been delivered and we can confirm this, because the APC kernel routine, when called, writes “Hello from the USER APC kernel routine” on the debugger console. We will see that this happens only *after* ApcUserModeTest has returned to user mode.

Before returning, ApcUserModeTest shows a few more interesting cases. First, it tries again to enter an alertable user-mode wait with KeWaitForSingleObject. However, the function *immediately* returns with STATUS_USER_APC. Thus, once a thread has user APCs pending, it simply can't enter such a wait.

It then tries a non-alertable user mode wait and, interestingly, the result is the same. This means alertability matters only when APCs are requested for a thread which is *already* waiting: the thread is only awakened if the wait is alertable. On the other hand, alertable or not, a thread can't enter a user mode wait if it already has pending user APCs.

Finally, ApcUserModeTest tries a non-alertable *kernel mode* wait and this time the thread is put to sleep, regardless of the pending APCs.

After all these tests, ApcUserModeTest terminates, the thread returns to user mode and only then the APC is dispatched and we see the debug message from its KernelMode routine.

Earlier, when examining nt!KiInsertQueueApc for user APCs, we reasoned that the wait functions would have had to account for the fact that a thread could call them after APCs have been queued for it. This follows from the fact that, for not-waiting threads, nt!KiInsertQueueApc does nothing more than chaining the APCs to its list; the target thread is thus free to attempt to enter an alertable user mode wait after this.

Kernel mode waiting functions handle this case simply by setting `_KTHREAD.ApcState.UserApcPending` to 1 and returning immediately `STATUS_USER_APC`. Thus, the thread can immediately return to user mode and, in doing this, will deliver the APCs thanks to the set `UserApcPending`.

The sample driver shows this in his `ApcUserModeTest2` function, which confirms that, before attempting the wait, `UserApcPending` is clear, then when the wait function (`KeWaitForSingleObject` in this case) is called, it immediately returns `STATUS_USER_APC` and `UserApcPending` is found set.

Conclusions on User APC Vs. Kernel Wait Functions

The DDK help explains how waits initiated by kernel wait functions like `KeWaitForSingleObject` are not aborted for kernel APC: the APCs are delivered, if the right conditions apply, then the thread resumes its wait. On the other hand, the same documentation details how these wait functions abort the wait for user APCs.

Now we understand why the wait is aborted: inside the wait function, the thread does not deliver user APCs. Instead, it returns from the wait function, so that the code following the wait can return to user mode, where the APCs will actually be delivered.

This is probably done because it would be too complex to actually deliver user mode APCs while inside the wait function. After all, we are talking about *user* APCs, whose `NormalRoutine` must be called in user mode. The kernel would have to somehow switch to user mode, deliver the APC, then switch back to kernel mode and resume the wait function – not an easy feat.

The implication is that kernel mode code (e. g. a driver) must “know” this, i. e. it must be written to handle the `STATUS_USER_APC` return code and the fact that it must actually return to ring 3 to allow user APCs to be serviced.

On the other hand, kernel mode APCs can be delivered without aborting the wait functions, because they don't require a switch from kernel mode to user mode.

nt!KiDeliverApc for User APCs

We are now going to see what happens when `nt!KiDeliverApc` is finally called for user APCs.

First of all, we need to remember that it has three stack parameters: `PreviousMode`, `Reserved` and `TrapFrame`.

When it is called for user APCs, `PreviousMode` is set to `UserMode`, i. e. 1, and `TrapFrame` points to a `_KTRAP_FRAME` structure storing the ring 3 context of the thread.

The first part of `nt!KiDeliverApc` does the kernel APC dispatching, regardless of the value of `PreviousMode`. In other words, whenever this function is called, it dispatches kernel mode APCs if there are any, then, only if `PreviousMode = UserMode`, checks for user mode ones.

All the actions explained in the section [Delivery](#) for kernel mode APCs are taken. If `_KTHREAD.SpecialApcDisable` is set, neither kernel nor user APCs are delivered.

Also, if `nt!KiDeliverApc` finds regular kernel mode APCs to deliver and then finds either `KernelApcDisable` or `KernelApcInProgress` set, it stops delivering APCs and returns, without processing user APCs.

However, when all the kernel APCs have been delivered and their list is empty, `nt!KiDeliverApc` sets to work on user APCs.

First of all, it checks `_KTHREAD.ApcState.UserApcPending`: if it is equal to 0, the function returns.

Otherwise, `nt!KiDeliverApc` acquires the spinlock stored at `_KTHREAD.ApcQueueLock`.

Then it sets `UserApcPending` to 0 and copies to locals the APC parameters `KernelRoutine`, `NormalRoutine`, `NormalContext`, `SystemArgument1`, `SystemArgument2`. Afterwards, it unchains the APC from its list.

Now the `_KAPC` structure cannot be reached by anyone else, because it is not in the list anymore, so `nt!KiDeliverApc` releases the spinlock.

After that, the kernel routine is called with the addresses of `NormalRoutine`, `NormalContext`, `SystemArgument1` and `SystemArgument2`.

When `KernelRoutine` returns, it may have changed the pointer to `NormalRoutine`, so `nt!KiDeliverApc` checks to see if it is equal to 0. Let's see what happens when this is not the case, first.

The function calls `nt!KiInitializeUserApc` passing to it the input `_KTRAP_FRAME` address, `NormalRoutine`, `NormalContext`, `SystemArgument1` and `SystemArgument2`. `nt!KiInitializeUserApc` will alter the thread user mode context stored in the `_KTRAP_FRAME` and send it to deliver the APC. This process will be analyzed in a later section. After this call, `nt!KiDeliverApc` exits.

We can now highlight an important detail: `nt!KiDeliverApc` only delivers *one* user APC; it does not spin in a loop delivering all the APCs in the list. Furthermore, it leaves `UserApcPending` clear so, unless somebody else sets it again, the SSD will not call `nt!KiDeliverApc` again.

This is in apparent contradiction with the MSDN documentation about the Win32 APIs for APCs. The following excerpt is for `QueueUserAPC`:

“When a user-mode APC is queued, the thread is not directed to call the APC function unless it is in an alertable state. After the thread is in an alertable state, the thread handles all pending APCs in first in, first out (FIFO) order, and the wait operation returns `WAIT_IO_COMPLETION`”

We will see how this works in a short while.

Now, let's go back to the point where `KernelRoutine` has been called, and see what happens when `KernelRoutine` zeroes the pointer to `NormalRoutine`: there is nothing more to do for the current APC, so `nt!KiDeliverApc` checks whether there are other user APCs pending and, if so, sets `UserApcPending` back to 1, then exits. Again, `nt!KiDeliverApc` does not loop through all the user APCs. However, this situation is different from when the `NormalRoutine` runs, because `UserApcPending` is set to 1.

By examining the SSD code, we see that, after `nt!KiDeliverApc` returns, the SSD checks `UserApcPending` and, if set, calls `nt!KiDeliverApc` again. Thus, if there are more APCs to process, the next one gets its turn.

nt!KiInitializeUserApc And The User Mode Context

The job of `nt!KiInitializeUserApc` is to modify the user mode context of the thread so that, when it returns to user mode, it is hijacked to call the `NormalRoutine`. The user mode context is stored in the `_KTRAP_FRAME` whose address is passed to `nt!KiInitializeUserApc`.

To do its job, `nt!KiInitializeUserApc` allocates room for a `_CONTEXT` structure on the ring 3 stack, whose `esp` is stored in the `_KTRAP_FRAME`. While doing this, it probes the user mode stack with the documented `ProbeForWrite` function, to ensure it can be written to.

If the stack is not writable, ProbeForWrite raises an exception, which is handled by nt!_except_handler4. I suppose it aborts the process, but I did not analyze it.

The _CONTEXT structure is then filled with the user mode context taken from _KTRAP_FRAME.

nt!KiInitializeUserApc actually reserves some more room on the user mode stack, below the _CONTEXT structure, where it will store the following data before leaving: SystemArgument2, SystemArgument1, NormalContext, pNormalRoutine (listed in decreasing addresses order).

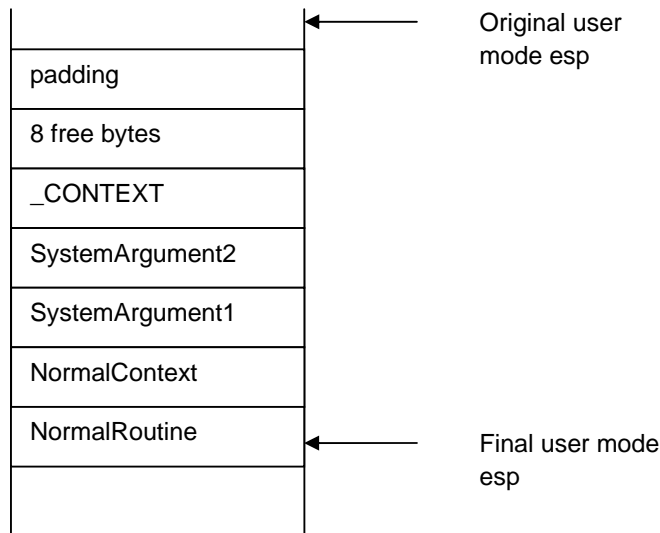
To keep things consistent, nt!KiInitializeUserApc updates the user mode esp into _KTRAP_FRAME so that it points at the beginning of the allocated data. This way it is as if the new data had been pushed on the stack by user mode code.

Afterwards, it sets the ring 3 eip stored inside _KTRAP_FRAME to the address of the user mode function ntdll!KiUserApcDispatcher and leaves, returning to nt!KiDeliverApc.

It is important to note that now we have two values for the ring 3 eip: one inside _KTRAP_FRAME, which points to ntdll!KiUserApcDispatcher, the other, inside the _CONTEXT on the user mode stack, which has the original user mode eip.

The eip copy inside _KTRAP_FRAME is the one which matters when execution goes back from kernel mode to user mode, so the thread will resume ring 3 execution inside ntdll!KiUserApcDispatcher. The _CONTEXT left on the user mode stack will be used by ntdll!KiUserApcDispatcher to resume the original user mode context later, after the NormalRoutine has been called.

The picture below shows the user mode stack after nt!KiInitializeUserApc has run:



The padding is added to align the rest of the stack to 4 bytes boundaries.

nt!KiInitializeUserApc returns to nt!KiDeliverApc, which in turn returns to the SSD. The SSD then resumes user mode execution, causing ntdll!KiUserApcDispatcher to be called.

User Mode APC Delivery by ntdll!KiUserApcDispatcher

This user mode function pops the address of NormalRoutine and calls it. NormalRoutine finds on the stack NormalContext, SystemArgument1 and SystemArgument2 as if they had been pushed by the call.

Before actually calling the NormalRoutine, ntdll!KiUserApcDispatcher uses the 8 free bytes on the user mode stack above _CONTEXT to build an exception handler registration, which is chained at the head of the handlers chain. This registration stores the address of ntdll!KiUserApcExceptionHandler, which will handle any exception caused by the NormalRoutine.

When the NormalRoutine returns, ntdll!KiUserApcDispatcher removes the exception registration from the handlers chain and calls ntdll!NtContinue, which will set the user mode context from the _CONTEXT structure on the stack. Since the eip value inside _CONTEXT is the original pre-APC eip, user mode execution will resume at the point it was before the APC was delivered.

ntdll!NtContinue And The User Mode Context

ntdll!NtContinue is actually implemented as an executive service, so it just loads into eax the service number (0x37) and invokes the SSD, which, in turn will transfer control to the kernel mode function nt!NtContinue.

As usually, invoking the SSD results in the creation of a _KTRAP_FRAME storing the user mode context. nt!NtContinue modifies this _KTRAP_FRAME so that, when returning to user mode, the context actually loaded will be the one from the original _CONTEXT structure passed to ntdll!NtContinue.

In most cases, execution will resume inside the user mode portion of the wait function which caused the APCs to be delivered in the first place.

We are finally ready to understand how Windows manage to deliver all the user APCs before actually returning from the wait function. This is the question we left open at the end of the section titled [nt!KiDeliverApc for User APCs](#)

After having restored the user mode context and just before returning to user mode, nt!NtContinue checks whether there are user mode APCs in the thread list and, if so, sets _KTHREAD.ApcState.UserApcPending. It then transfers control back to the SSD.

Thus, the SSD finds the following situation:

- The user mode context is the original pre-APC one
- UserApcPending is set if there still are APCs to deliver

The thread is thus in the same situation it was, before delivering the first APC. If UserApcPending is set, the SSD will invoke again nt!KiDeliverApc and the whole APC delivery process will be repeated.

This cycle will go on until eventually all user mode APCs have been delivered, the SSD finds UserApcPending clear and finally resumes the original user mode context.

At first sight, this process may seem unnecessarily complex: why have nt!NtContinue set UserApcPending again instead of simply processing all the APCs inside nt!KiDeliverApc? After all, this is what happens with kernel mode APCs.

However, since user mode APC must run in, well, user mode, nt!KiDeliverApc can't simply call them: it has to alter the user mode context by calling nt!KiInitializeUserApc and then it must allow execution

to return to user mode, before the next APC can be dealt with. After all, there is only one `_KTRAP_FRAME` storing the user mode context, so, once it has been manipulated to deliver the first APC, it cannot be altered again for the second one. Instead, execution must be allowed to return to user mode to deliver the APC and only afterwards the process can be repeated, if there are more.

Appendix - The Test Driver

IMPORTANT WARNING

The test driver which comes with this article does several unorthodox things and is therefore evil, nasty, dangerous and likely to crash the system. After consulting the most competent law firms around, I have been advised to include the following disclaimer:

If you are dumb enough to put yourself in a situation where you lose money because of my driver, there's only one thing you can do which is even dumber: asking me your money back. Whatsoever, aforementioned – there you go, any serious disclaimer worth its salt must say "whatsoever" and "aforementioned" at least once.

BUT, if you are willing to risk a blue screen here and there, you can reproduce the tests explained in this article.

Seriously, though, the most risky test to run is the spurious interrupt test. The other ones are almost safe.

Furthermore, this test works ONLY for Windows Vista SP1 with no Windows updates applied. *It will crash any other version of `ntoskrnl.exe`.* It can work on an updated Vista SP1, if the `ntoskrnl.exe` is still the original one.

Also, the test works only on a uniprocessor machine. Before you start rummaging through your attic in search of that old single core PC, remember you can use the "onecpu" bcdedit datatype to boot with one cpu only, or, much safer, you can use a virtual machine.

Driver Package Contents

The package contains both the executables and the source for the driver. It consists of a .zip file with two main directories: executables and sources.

The executables are:

<code>ApcTest.sys</code>	the driver
<code>TestClnt.exe</code>	the client program

The client program calls the driver with the `DeviceIoControl` API, passing the code of the test to perform.

The sources directory contains a Visual C++ Express solution.

You don't need to have VS installed to edit the code and build the driver. The solution projects can just be regarded as subfolders of the overall source tree and all the building is done with the DDK build utility or the SDK `nmake` tool. VS can be used as a text editor and to access the various files through the Solution Explorer window, but this is not required.

The solution is made up of the projects described below.

Driver

This project contains the driver code. See the ReadMe.txt file for details on how to build the driver.

TestClt

Project containing the client to invoke the driver. See the ReadMe.txt file for details on how to build the client.

CommonFile

Shared files.

Loading And Running The Driver

The test client does not load the driver by itself, so some loading utility is needed.

The one I use and I find very handy is w2k_load.exe by Sven B. Schreiber, originally found on the companion CD of his book, Undocumented Windows 2000 Secrets. The CD image can be downloaded from Mr. Schreiber site at:

<http://undocumented.rawol.com/>

In spite of having been written for Windows 2000, w2k_load still works like a breeze under Vista.

Loading the driver is as simple as entering

```
W2k_load apctest.sys
```

And to unload it:

```
W2k_load apctest.sys /unload
```

Once the driver is loaded, run TestClt to call it, specifying a command line switch to select the test to run. TestClt without parameters gives the following output, listing the switch values

Usage:

```
TestClt /d - APC disable test
TestClt /u - APC user mode test
TestClt /2 - APC user mode test #2
TestClt /s - APC spurious int test - VISTA SP1 ONLY. OTHER VER => BSOD!!!

TestClt /h - This help
```

What The Driver Can Do

The driver performs four different APC tests, implemented as dispatch routines for custom I/O control codes. Each test writes its output to the kernel debugger, so you will need to run Windows in a kernel debug session to see the results.

APC Spurious Interrupt Test

This test is described in the section [Can The APC Interrupt Really Break into The Wrong Thread?](#) and shows how a DPC interrupt can pass ahead of an APC one, causing the latter to occur in the wrong thread.

The code for this test does dangerous things and can crash the system.

This code works only with the ntoskrnl.exe which is part of Vista SP1. It will almost certainly crash any other version of ntoskrnl.exe.

This code works correctly only on a uniprocessor system. It is possible to boot a multiprocessor system in single processor mode with the "onecpu" data type of bcdedit.

All the output from this test is written to the debugger console, so it is required to run Windows in a debug session to see the result.

To perform this test run:

```
TestClnt /s
```

Test Code Description

Hooked Kernel Functions

To verify that the APC interrupt occur in the wrong thread, the driver installs two hooks in the kernel: one at the beginning of nt!SwapContext, the other one at the beginning of nt!KiDeliverApc.

These hooks are installed by overwriting the first 8 bytes of the functions with an absolute jmp instruction to the hook routine. This job is done by assembly language functions enclosed in KernelHooks.asm. For instance, SetKDAPHook installs the hook into nt!KiDeliverApc.

While the hook is being set, interrupts are disabled to prevent the processor from being hijacked, which would put as at risk of executing the same machine instructions which are being overwritten. However if the system has more than one processor, another processor may stumble on the function while it is being modified and crash the system. This is the reason why this code only works on uniprocessor systems.

The hook routines, which are also part of KernelHooks.asm, save the processor state, then call a C function (one for each hook) which traces the call. When the C function returns, the hook routine restores the processor state, executes a copy of the original instructions that were at the beginning of the hooked function, then jumps into the function itself, which will go on executing normally.

Such an hook is specific to one version of ntoskrnl (in our case, Vista SP1) because the hook routine contain a copy of the first instructions of the hooked function and jumps after them to a fixed offset into the function. Any variation in the function executable image will result in inconsistent code and a jump to a wrong address.

Furthermore, nt!SwapContext and nt!KiDeliverApc are not exported, so I used their offsets from ntoskrnl starting address to compute their run-time address. A different ntoskrnl will not match with these offsets, so the hook bytes will be written at a wrong address.

The hook routines are KDAPHook and SWCNHook inside KernelHooks.asm.

C Trace Routines

The hook routines are an interface between the hooked kernel function and the rest of the driver. They call C routines which trace the call to the hooked function. KDAPHook calls KiDeliverApcTrace and SWCNHook calls SwapContextTrace.

These tracing functions are called with interrupts disabled and they don't call any kernel DDI, to be as invisible as possible to the system.

To trace the call, these routines store information in a static buffer in non-paged memory. The functions managing this buffer are enclosed in `CircBuffer.cpp`.

Both tracing functions check a static flag named `bHookTraceOn`: if it's false, they don't write anything into the buffer.

`KiDeliverApcTrace` clears this flag when it detects it has been called in the context of the thread which scheduled the APC. This stops the trace when the APC is about to be delivered.

Both trace routines write "records" into the buffer, i. e. copies of C structures containing the traced information. There are different structure types for different trace data and all types begin with a `RecordType` member identifying them.

Main Test Function

The main test function is `ApcSpuriousIntTest` and begins by initializing a special kernel mode APC.

Then it raises the IRQL to `DISPATCH`, preventing thread dispatching.

Afterwards, `ApcSpuriousIntTest` schedules the APC by calling `nt!KeInsertQueueApc`, which, in turn, requests the APC interrupt, because the APC is for the current thread. This interrupt remains pending, because it is masked by the IRQL being `DISPATCH`.

Then the function spins in a loop waiting until a DPC interrupt is pending as well. This is bound to happen when the timer interrupt handler determines that the thread has used all its time slice and must be preempted. This interrupt also is masked by the DPC IRQL.

To detect when a DPC interrupt is pending, we rely on the fact that at `+ 0x95` inside the `_KPCR` for the executing processor is a byte array of interrupt flags.

When the byte at `0x95 + 1` is 1, an IRQL 1 (APC) interrupt is pending

When the byte at `0x95 + 2` is 1, an IRQL 2 (`DISPATCH`) interrupt is pending

The presence of this array can be inferred by analyzing the code of functions like `hal!HalpCheckForSoftwareInterrupt`, `hal!HalpDispatchSoftwareInterrupt`, `hal!HalRequestSoftwareInterrupt`.

When the DPC interrupt arrives, `ApcSpuriousIntTest` install the hooks and lowers the IRQL to passive, allowing the interrupts to fire. The hooks trace the execution of `nt!SwapContext` and `nt!KiDeliverApc`, along with the necessary data at the time of the hooked call.

The APC kernel routine (`ApcKernelRoutineSIT`) writes a record into the buffer as well, so that we can know when it is called with respect to the other traced events.

Finally, `ApcSpuriousIntTest` writes a record into the buffer after `KeLowerIrql` returns.

Before leaving, `ApcSpuriousIntTest` removes the hooks and calls `DumpTrace` which writes the records stored in the buffer to the debugger console.

Sample Trace

The following is sample trace obtained with the driver:

```
APCTEST - Thr: 0X84AD0D78; *** APC SPURIOUS INTERRUPT TEST ***
```

APCTEST - Thr: 0X84AD0D78; APC initialized: 0X84FB11A4

APCTEST - SWAP CONTEXT trace

APCTEST - Current IRQL: 0x1b
APCTEST - Current thread: 0X84AD0D78
APCTEST - Current thread K APC pending: 1
APCTEST - Current thread K APC list empty: 0
APCTEST - Current thread U APC pending: 0
APCTEST - Current thread U APC list empty: 1

APCTEST - New thread: 0X84D689F0
APCTEST - New thread K APC pending: 0
APCTEST - New thread K APC list empty: 1
APCTEST - New thread U APC pending: 0
APCTEST - New thread U APC list empty: 1

APCTEST - APC INT: 1

APCTEST - DELIVER APC trace **Spurious APC int in the wrong thread**

APCTEST - Current IRQL: 0x1
APCTEST - Caller address: 0X8181F2F7
APCTEST - Trap frame: 00000000
APCTEST - Reserved: 00000000
APCTEST - PreviousMode: 0

APCTEST - Thread: 0X84D689F0
APCTEST - Thread K APC pending: 0
APCTEST - Thread K APC list empty: 1
APCTEST - Thread U APC pending: 0
APCTEST - Thread U APC list empty: 1

APCTEST - SWAP CONTEXT trace **Swap Context back to the initial thread**

APCTEST - Current IRQL: 0x1b
APCTEST - Current thread: 0X84D689F0
APCTEST - Current thread K APC pending: 0
APCTEST - Current thread K APC list empty: 1
APCTEST - Current thread U APC pending: 0
APCTEST - Current thread U APC list empty: 1

APCTEST - New thread: 0X84AD0D78
APCTEST - New thread K APC pending: 1
APCTEST - New thread K APC list empty: 0
APCTEST - New thread U APC pending: 0
APCTEST - New thread U APC list empty: 1

```

APCTEST -          APC INT:                0

APCTEST - DELIVER APC trace  APC delivery

APCTEST -          Current IRQL:           0x1
APCTEST -          Caller address:         0X8181F2F7
APCTEST -          Trap frame:             00000000
APCTEST -          Reserved:               00000000
APCTEST -          PreviousMode:           0

APCTEST -          Thread:                  0X84AD0D78
APCTEST -          Thread K APC pending:    1
APCTEST -          Thread K APC list empty: 0
APCTEST -          Thread U APC pending:    0
APCTEST -          Thread U APC list empty: 1

APCTEST - KERNEL ROUTINE trace

APCTEST -          Thread:                  0X84AD0D78
APCTEST -          Thread K APC pending:    0
APCTEST -          Thread K APC list empty: 1
APCTEST -          Thread U APC pending:    0
APCTEST -          Thread U APC list empty: 1
APCTEST - TRACE MESSAGE: Returned from KeLowerIrql
APCTEST - Thr: 0X84AD0D78; WaitCycles = 765646

```

Atypical Traces

While testing the driver, I observed two types of atypical traces, which are worth mentioning.

Trace Without Context Switch

The first type are traces where the first event traced is the APC delivery in the correct thread. There is no traced `nt!SwapContext`, so no interrupt in the wrong thread. We know that when the IRQL is lowered a DISPATCH interrupt is pending, because the driver checks this condition, however we don't see a context switch.

The explanation is probably that the DISPATCH interrupt did not cause a context switch, perhaps because the thread dispatcher chose to let our thread run for another time slice. In general, not every DISPATCH interrupt results in a context switch. For instance, we might have come across an interrupt raised by a driver which scheduled a DPC, while the thread still had some of its time slice left

Here is an example of this kind of trace

```

APCTEST - Thr: 0X84E96030; *** APC SPURIOUS INTERRUPT TEST ***

APCTEST - Thr: 0X84E96030; APC initialized: 0X8511E0EC

APCTEST - DELIVER APC trace  The first event is nt!KiDeliverApc

```

```
APCTEST - Current IRQL: 0x1
APCTEST - Caller address: 0X8181F2F7
APCTEST - Trap frame: 00000000
APCTEST - Reserved: 00000000
APCTEST - PreviousMode: 0
```

```
APCTEST - Thread: 0X84E96030
APCTEST - Thread K APC pending: 1
APCTEST - Thread K APC list empty: 0
APCTEST - Thread U APC pending: 0
APCTEST - Thread U APC list empty: 1
```

APCTEST - KERNEL ROUTINE trace

```
APCTEST - Thread: 0X84E96030
APCTEST - Thread K APC pending: 0
APCTEST - Thread K APC list empty: 1
APCTEST - Thread U APC pending: 0
APCTEST - Thread U APC list empty: 1
APCTEST - TRACE MESSAGE: Returned from KeLowerIrql
APCTEST - Thr: 0X84E96030; WaitCycles = 243152
```

Trace With APC Interrupt Not Outstanding

The second type of trace is similar to the first sample, where the context switch did occur, but shows an interesting difference.

In all the traces, the nt!SwapContext data also include whether an APC interrupt is pending at the time of the call. This is done in order to confirm that nt!SwapContext is actually called while the APC interrupt is still outstanding.

For this type of trace, however, we see nt!SwapContext being called, with the pending APC, but without the interrupt:

```
APCTEST - Thr: 0X85145388; APC initialized: 0X850AF0EC
```

APCTEST - SWAP CONTEXT trace

```
APCTEST - Current IRQL: 0x1b
APCTEST - Current thread: 0X85145388
APCTEST - Current thread K APC pending: 1 the APC is pending
APCTEST - Current thread K APC list empty: 0
APCTEST - Current thread U APC pending: 0
APCTEST - Current thread U APC list empty: 1

APCTEST - New thread: 0X84D689F0
APCTEST - New thread K APC pending: 0
```

```

APCTEST -      New thread K APC list empty:      1
APCTEST -      New thread U APC pending:         0
APCTEST -      New thread U APC list empty:      1

APCTEST -      APC INT:                          0 no APC int

```

*** No APC delivery trace afterwards, b/c no APC int.

APCTEST - SWAP CONTEXT trace

```

APCTEST -      Current IRQL:                      0x1b
APCTEST -      Current thread:                    0X84D689F0
APCTEST -      Current thread K APC pending:      0
APCTEST -      Current thread K APC list empty:   1
APCTEST -      Current thread U APC pending:      0
APCTEST -      Current thread U APC list empty:   1

APCTEST -      New thread:                        0X84F62D78
APCTEST -      New thread K APC pending:          0
APCTEST -      New thread K APC list empty:       1
APCTEST -      New thread U APC pending:          0
APCTEST -      New thread U APC list empty:       1

APCTEST -      APC INT:                          0

```

[...]

We know that:

- nt!KiInsertQueueApc is called when the IRQL is already DISPATCH, so the APC interrupt cannot be serviced before we lower the IRQL to PASSIVE.
- We lower the IRQL to passive after having set the hooks, so we cannot miss the APC interrupt
- The call to nt!SwapContext cannot have occurred while nt!KiInsertQueueApc was in the process of queuing the APC, i. e. before it raised the APC interrupt, because we called nt!KiInsertQueueApc at DISPATCH, so no context switch can occur inside it. Besides, even if we did not set the IRQL at DISPATCH, nt!KiInsertQueueApc sets it to profile (0x1b) before calling nt!KiInsertQueueApc.

The explanation may be that nt!KiInsertQueueApc found SpecialApcDisable set. When this happens, the function does not raise the APC interrupt, although it sets KernelApcPending, which is the situation in the trace above.

But who then set SpecialApcDisable? Possibly, the code handling an hardware interrupt which occurred while the test was in progress. Such interrupts were not masked by the DISPATCH IRQL.

APC Disable Test

This test has been introduced in the section [Effect of KTHREAD.SpecialApcDisable Being Set](#). It shows the effect of the SpecialApcDisable flag on nt!KiDeliverApc. To perform this test run the client with the command:

```
TestClnt /d
```

APC User Mode Test And APC User Mode Test #2

Introduced in the section [User Mode APC Delivery Test in The Sample Driver](#), shows the interaction between user mode APCs and the kernel wait functions. To run them:

```
TestClnt /u  
TestClnt /2 for #2
```

All the output is sent to the kernel debugger.

References

[1] - Albert Almeida; Inside NT's Asynchronous Procedure Call; Dr. Dobb's Journal, Nov 01,2002.
Available at <http://www.ddj.com/windows/184416590>

[2] – David A. Solomon, Mark E. Russinovich; Inside Microsoft Windows 2000; Third Edition; Microsoft Press

[3] - Doing Things "Whenever" - Asynchronous Procedure Calls in NT; The NT Insider,Vol 5, Issue 1, Jan-Feb 1998; Available at <http://www.osronline.com/article.cfm?id=75> (registration required).