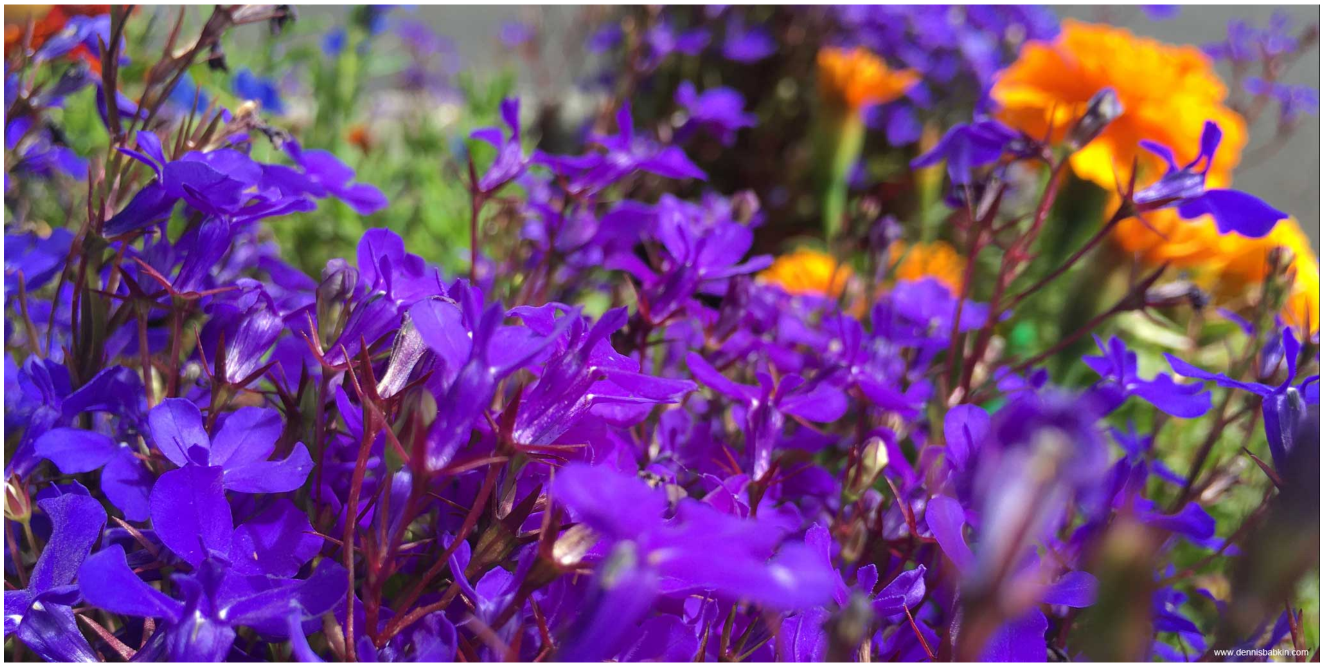
Deep dive into user-mode Asynchronous Procedure Calls in Windows.

dennisbabkin.com/blog



Intro

Asynchronous Procedure Calls, or APC, was an always obscure subject for me. Even though it is documented by Microsoft, the intricacies of its implementation kept me away from it in my own software.

Recently though during our conversation with Rbmm, he pointed out some aspects of APC that were stopping me from using it before. Additionally, when I tried searching online for a comprehensive guide on APC, I didn't find much. Thus, I'm writing this blog post to shed some light on APC and its use when writing Windows native code.

For those that don't like reading blog posts, make sure to check my video recap at the end.

APC Basics

APC, in short, is a *property* of a Windows thread that allows to specify a callback routine to execute asynchronously. In most cases APC will be beneficial for code functions that perform some lengthy operations, usually an input/output (or I/O), such as file operations, web transactions, timers, etc. An APC in Windows is basically the way to attach a callback code to a particular thread.

APC in Windows comes in two flavors: kernel-mode and user-mode. The former is executed primary as an *interrupt* (that we'll discuss in a separate blog post.) The latter one though has some intricacies in the way a thread needs to call certain Windows APIs to ensure that an APC callback can be invoked. This blog post will be about implementation of the user-mode APC.

In my view, the best way to illustrate all these concepts is with code. So let's do it next.

Simple APC Example

Let's create the most basic example of how user-mode APC can be used. We'll be using a console application (but for that matter, this can also be used in the service application as well.)

For simplicity I will be using the C++ *Console Application* template in the Visual Studio.

> Since we're dealing with the Windows-specific content, I will stick with WinAPIs for the code samples below. For simplicity I will stay away from the standard C++ code primitives, as they are not relevant to Windows internals that I will be discussing in this blog post.

Let's modify our main function to create a thread. Keep in mind to do error checks too:

C++

```cpp
int main()
{
        HANDLE hThread = ::CreateThread(NULL, 0, ThreadProc, 0, 0, NULL);
        if (hThread)
        {

                ::Sleep(1000 * 1000);
                ::CloseHandle(hThread);
        }
        else
                wprintf(L"ERROR: (%d) CreateThread\n", ::GetLastError());
}
```

The code above creates a Win32 thread and then goes into a sleep cycle for 1000 seconds. I need this long delay to ensure that our main thread remains "alive" for the duration of our test.

Then the thread procedure itself is just this:

C++

```
DWORD WINAPI ThreadProc(
        _In_ LPVOID lpParameter
)
{
        wprintf(L"[%u] Thread has started\n", ::GetCurrentThreadId());


        ::Sleep(1000 * 1000);
        return 0;
}
```

As you can see I added another diagnostic output into the console to tell us what our thread ID is, and then added another 1000-second delay at the end to make sure that our thread stays alive for the duration of our test. So nothing fancy so far.

Now let's try to queue our APC. (In Microsoft jargon this means to *add an APC callback to a thread*.) Doing that should notify the thread to execute our callback at the first available opportunity. We can use the `QueueUserAPC` function to do that.

C++

```
int main()
{
        HANDLE hThread = ::CreateThread(NULL, 0, ThreadProc, 0, 0, NULL);
        if (hThread)
        {
                ::Sleep(1000);

                if(!::QueueUserAPC(Papcfunc, hThread, 123))
                {
                        wprintf(L"ERROR: (%d) QueueUserAPC\n", ::GetLastError());
                }

                ::Sleep(1000 * 1000);
                ::CloseHandle(hThread);
        }
        else
                wprintf(L"ERROR: (%d) CreateThread\n", ::GetLastError());
}
```

As you can see the call to `QueueUserAPC` takes the address of the callback function as the first parameter, the thread handle to associate it with, and the last parameter as a user-defined value to pass into the callback. Let's just choose something random, like 123. We also need to make sure to catch and log all errors.

Additionally, note that I added another 1-second delay right after the call to `CreateThread` and before `QueueUserAPC` in the form as `Sleep(1000)`. The reason I did that was to point out a potential *race condition* in our test code. The way `CreateThread` works is that it is an asynchronous function itself, meaning that it may return before the thread has a chance to start. If that happens quickly, without any further delay our call to `QueueUserAPC` may also succeed before the thread had started running. In that case, to quote the documentation:

> If an application queues an APC before the thread begins running, the thread begins by calling the APC function. After the thread calls an APC function, it calls the APC functions for all APCs in its APC queue.

So it won't be a good test of our callback because it will be executed automatically even before our thread has a chance to start running. The test we're making here is how to queue an APC callback after the thread had begun executing. Thus we added a slight delay to ensure that.

OK. Then our APC callback becomes this:

C++

```
void Papcfunc(
        ULONG_PTR Parameter
)
{
        wprintf(L"[%u] APC callback has fired with param=%Id\n",
::GetCurrentThreadId(), Parameter);
}
```

Again, for the purpose of our test, I'm outputting into console the thread ID with which our APC callback is executing and also the fact that our callback was actually invoked.

So if you run the code above, our thread will start, but the APC callback will not be invoked. And that's what was very confusing to me at first. My question was, why?

The reason our APC callback was not invoked can be gleaned from the documentation:

> When a user-mode APC is queued, the thread is not directed to call the APC function unless it is in an alertable state.

> A thread enters an alertable state by using SleepEx, SignalObjectAndWait, WaitForSingleObjectEx, WaitForMultipleObjectsEx, or MsgWaitForMultipleObjectsEx to perform an alertable wait operation.

What that quote says, is that a thread that has a queued APC needs to be in an `alertable` state to invoke that APC. But what is that? Well, in short, this basically means that a thread needs to call one of those listed waiting APIs to enter that state.

OK, so let's modify our thread procedure. The easiest function for us to call is `SleepEx`:

C++

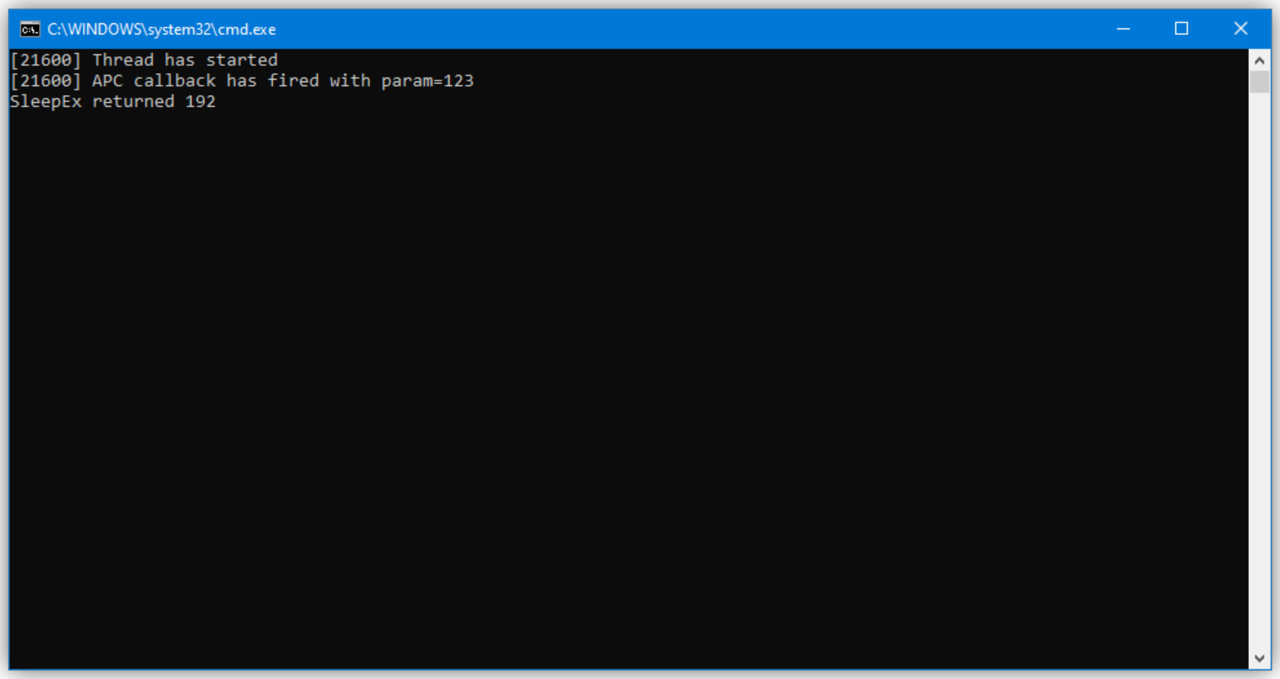
```
DWORD WINAPI ThreadProc(
        _In_ LPVOID lpParameter
)
{
        wprintf(L"[%u] Thread has started\n", ::GetCurrentThreadId());

        DWORD dwR = ::SleepEx(INFINITE, TRUE);
        wprintf(L"SleepEx returned %d\n", dwR);


        ::Sleep(1000 * 1000);
        return 0;
}
```

Note the important thing is that I'm passing the 2nd parameter into `SleepEx` as `TRUE`, that brings that thread into an alartable state, or allows it to process its queued APC callbacks. And again for our debugging purposes I also output the return value from the `SleepEx` function onto the console.

Now if we run this code, the result is what we wanted to achieve and our APC callback is invoked successfully:

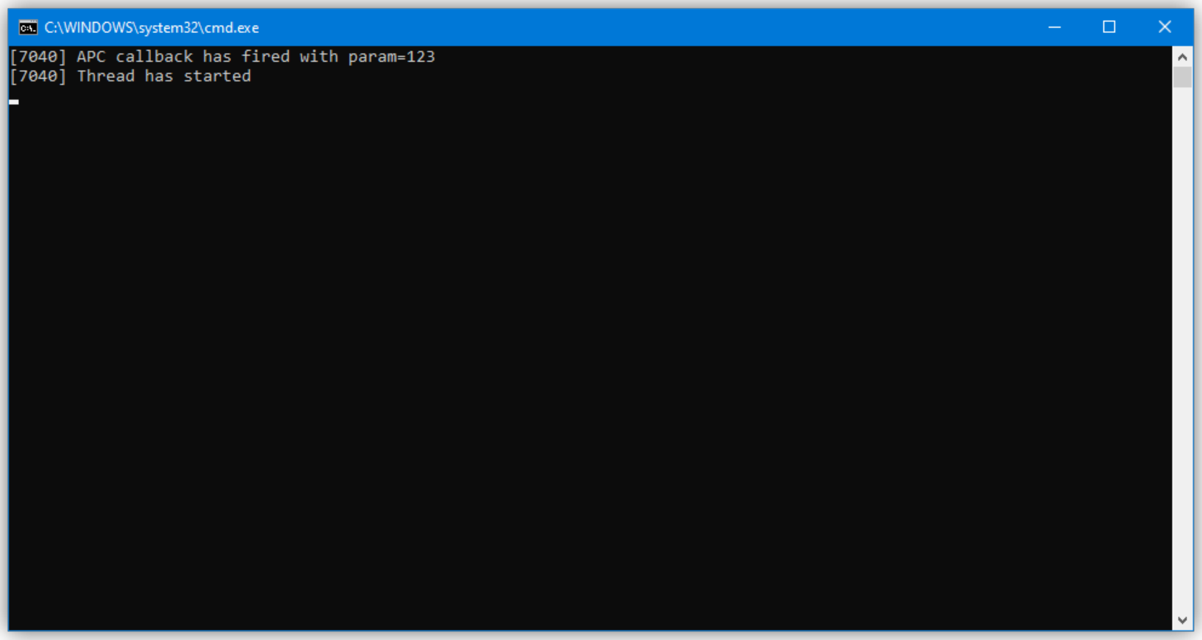


```
[21600] Thread has started
[21600] APC callback has fired with param=123
SleepEx returned 192
```

There are several things to note here:

1. See that the APC callback has been invoked from within the context of the thread itself. We can tell that because they both have the same thread ID.
2. `SleepEx` function call returned the value of 192, which is `WAIT_IO_COMPLETION`, that signifies that the function returned after APC callback was invoked.

3. If you remove, or comment out, the `Sleep(1000)` delay after a call to `CreateThread` and run the code, note that the APC callback may be executed before the code in the thread entry point (i.e. `ThreadProc`) has even started running:



```
[7040] APC callback has fired with param=123
[7040] Thread has started
```

> When we called `QueueUserAPC` right after `CreateThread` the APC callback was executed within initiation of a thread itself. (Remember that a thread is started asynchronously after a call to `CreateThread` returns.) And thus in that case, the thread did not need to be in an alertable state, or call `SleepEx` . But **do not rely** solely on this behavior if you also want your APC callback to be executed after the thread has started running. By doing so you are creating a bug in your code, or a *race condition*, which I demonstrated by adding a one-second delay after a call to `CreateThread` . In your production code such delay may come from some other code that is executed right after you created a thread but before you queued an APC.

> In a situation when you need to execute APC callback before the thread entry point `ThreadProc` , the correct way to do it is to create that thread suspended, by specifying `CREATE_SUSPENDED` flag, then queue an APC using `QueueUserAPC` function call, and resume the thread using `ResumeThread` function. Note that APC callbacks will not be executed while thread is still suspended. When you resume the thread the APC callback(s) will run first, in order that you queued them, and then the thread entry point `ThreadProc` will run next.

Multiple APC Callbacks

Now let's dive a little bit deeper. Can we queue multiple APC callbacks to one thread?

Let's modify our code to accomplish that. I'll try to queue a large number of APCs at once. How about a thousand:

C++

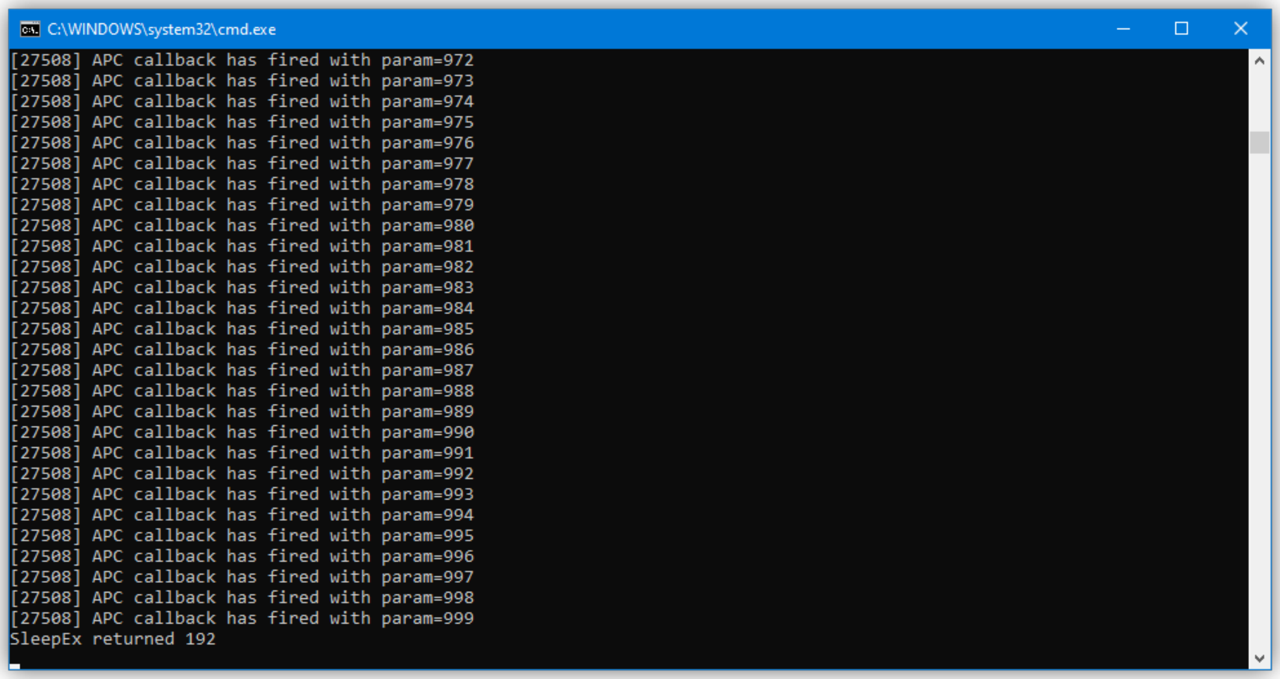
```
int main()
{
        HANDLE hThread = ::CreateThread(NULL, 0, ThreadProc, 0, 0, NULL);
        if (hThread)
        {
                ::Sleep(1000);

                for (int q = 0; q < 1000; q++)
                {
                        if (!::QueueUserAPC(Papcfunc, hThread, q))
                        {
                                wprintf(L"ERROR: (%d) QueueUserAPC with value
q=%d\n", ::GetLastError(), q);
                                break;
                        }
                }


                ::Sleep(1000 * 1000);
                ::CloseHandle(hThread);
        }
        else
                wprintf(L"ERROR: (%d) CreateThread\n", ::GetLastError());
}
```

I modified our call to `QueueUserAPC` to be called in a loop. I also changed its user-defined parameter to an index in that loop, and also modified our error reporting code to notify us of a specific cycle that the function may fail at and break the loop.

If you run that code as-is, it may produce this output:



```
[27508] APC callback has fired with param=972
[27508] APC callback has fired with param=973
[27508] APC callback has fired with param=974
[27508] APC callback has fired with param=975
[27508] APC callback has fired with param=976
[27508] APC callback has fired with param=977
[27508] APC callback has fired with param=978
[27508] APC callback has fired with param=979
[27508] APC callback has fired with param=980
[27508] APC callback has fired with param=981
[27508] APC callback has fired with param=982
[27508] APC callback has fired with param=983
[27508] APC callback has fired with param=984
[27508] APC callback has fired with param=985
[27508] APC callback has fired with param=986
[27508] APC callback has fired with param=987
[27508] APC callback has fired with param=988
[27508] APC callback has fired with param=989
[27508] APC callback has fired with param=990
[27508] APC callback has fired with param=991
[27508] APC callback has fired with param=992
[27508] APC callback has fired with param=993
[27508] APC callback has fired with param=994
[27508] APC callback has fired with param=995
[27508] APC callback has fired with param=996
[27508] APC callback has fired with param=997
[27508] APC callback has fired with param=998
[27508] APC callback has fired with param=999
SleepEx returned 192
```

> So the answer to the original question is yes, we can queue multiple APC callbacks to the same thread. They will be executed sequentially in the order that they were queued. And the number of available APC callbacks that can be queued seems to be only limited by the amount of non-pageable kernel memory in the system.

The tricky thing about our sample above is that by introducing the loop we also introduced another race condition into our code. Did you spot it? (Rbmm had actually pointed that condition out to me.)

To spot it, let's add a slight delay after each call to `QueueUserAPC` in our loop. We'll do it with a call to `Sleep(1)` :

C++

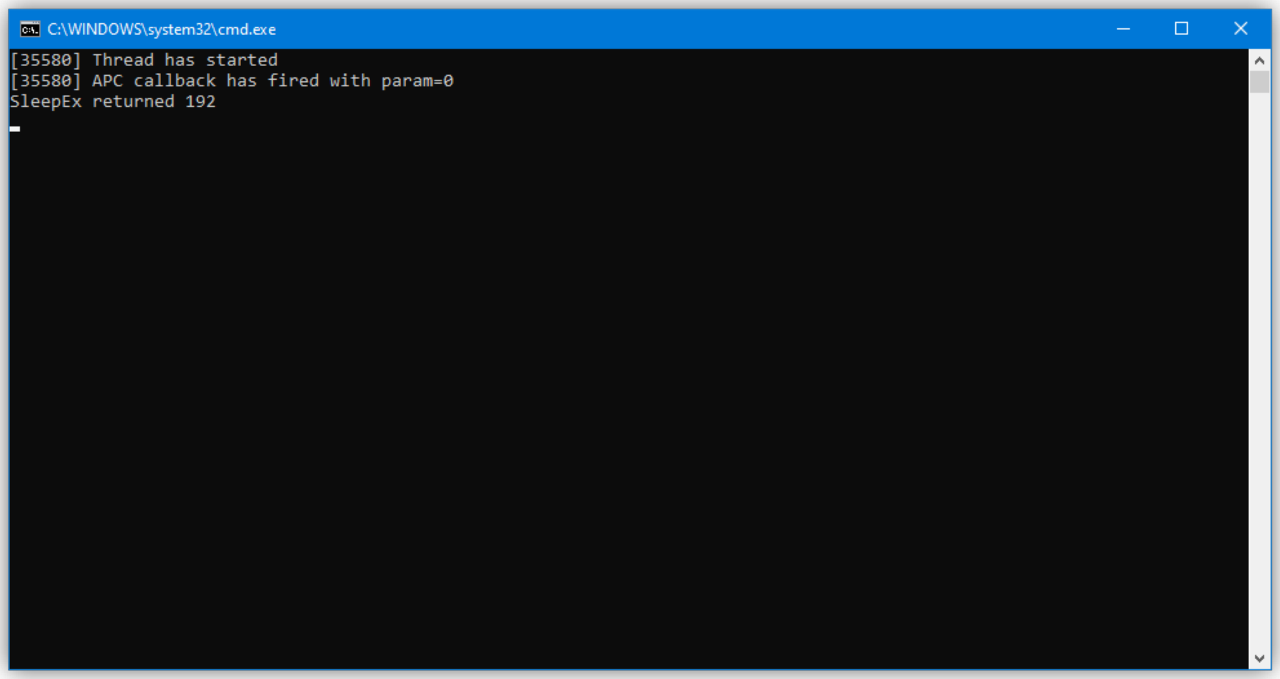
```
int main()
{
        HANDLE hThread = ::CreateThread(NULL, 0, ThreadProc, 0, 0, NULL);
        if (hThread)
        {
                ::Sleep(1000);

                for (int q = 0; q < 1000; q++)
                {
                        if (!::QueueUserAPC(Papcfunc, hThread, q))
                        {
                                wprintf(L"ERROR: (%d) QueueUserAPC with value
q=%d\n", ::GetLastError(), q);
                                break;
                        }

                        ::Sleep(1);
                }


                ::Sleep(1000 * 1000);
                ::CloseHandle(hThread);
        }
        else
                wprintf(L"ERROR: (%d) CreateThread\n", ::GetLastError());
}
```

But now if we run this code, the result may look like this:



```
[35580] Thread has started
[35580] APC callback has fired with param=0
SleepEx returned 192
```

So why did we get only one APC callback with that delay?

Well, this is a classic *race condition*. The reason is that our `ThreadProc` had only one call to `SleepEx`. Let's see what could've happened with and without a delay in our loop:

- Without a delay our loop quickly went through all the calls to `QueueUserAPC`. In that particular instance, the thread executing our loop was able to do so within its own time slice before our `ThreadProc` thread had a chance to run. So in that case all 1000 APCs were queued before `SleepEx` function in our `ThreadProc` ran. But then when it did, it executed them sequentially as we queued them, which made it look like what we wanted to achieve.
- With a delay though our loop was queuing an APC per each time slice of its execution. So after the first call to `QueueUserAPC`, the `SleepEx` function in our `ThreadProc` was invoked, which processed our single queued callback and returned. But after that the `ThreadProc` simply went into its 1000-second delay, which does not put it into an alertable state. And thus we saw only one APC callback in our output.

To fix this timing bug, we need to ensure that we put our thread into an alertable state for as many times as we queue our APCs. To do that in our test example, we can simply call it in an infinite loop like so:

C++

```
DWORD WINAPI ThreadProc(
        _In_ LPVOID lpParameter
)
{
        wprintf(L"[%u] Thread has started\n", ::GetCurrentThreadId());

        for(;;)
        {
                DWORD dwR = ::SleepEx(INFINITE, TRUE);
                wprintf(L"SleepEx returned %d\n", dwR);
        }


        //::Sleep(1000 * 1000);    // becomes redundant
        return 0;
}
```

In your production code though you would probably not call `SleepEx` in your worker thread. Instead you will be using a function such as `WaitForSingleObjectEx`, or `WaitForMultipleObjectsEx` that will not only let you put your thread into an alertable state, but will also let you keep track of some signaling object, like an event, to properly end that thread.

> In case you queued more than one APC callback, they will run in order specified. Also note that only one callback function will run at a time. Each APC callback will be executing in the context of the thread that it was queued for.

Next let's review some additional gotchas that may come up with APCs - how to handle them in a GUI app.

APC With GUI Apps

A GUI app in Windows behaves in a slightly different manner than a console app or a service. GUI app comes with a message loop, that by itself does not allow processing of APC callbacks.

If you create a stock *Windows Desktop Application* for C++ in Visual Studio, its message loop in the `wWinMain` function may look like this:

C++

```
MSG msg;

while (GetMessage(&msg, nullptr, 0, 0))
{
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
}

return (int)msg.wParam;
```

> Note that the way Microsoft uses `GetMessage` function in the stock sample above is incorrect because it may return three values: `0` if it receives `WM_QUIT` message, `-1` if it fails, or other value if it receives some other message. In other words, the `while` loop should account for an error condition, as described here.

In the loop above `GetMessage` function waits indefinitely for a message and then returns it in `msg` when one arrives.

So let's see what happens when we try to queue an APC to that thread. In this example we will stay with a single-threaded nature of our GUI app.

Let's create a helper function that will queue an APC for us:

C++

```
void Papcfunc(
        ULONG_PTR Parameter
)
{
        HWND hWnd = (HWND)Parameter;
        ::MessageBox(hWnd, L"APC callback fired OK", L"Success", MB_ICONINFORMATION);
}

void set_test_APC(HWND hWnd)
{
        if (!::QueueUserAPC(Papcfunc, ::GetCurrentThread(), (ULONG_PTR)hWnd))
        {
                ::MessageBox(hWnd, L"ERROR: QueueUserAPC failed", L"ERROR",
MB_ICONERROR);
        }
}
```

As before, we're using `QueueUserAPC` but instead of starting a new thread we will use the same thread that we're running in. Additionally, for the output we will use a GUI message box to tell us if queuing of APC succeeded or failed.

Lastly, we can invoke our `set_test_APC` as a handler from our main window menu.

But if we compile and run our GUI app, and then test our `set_test_APC` function, the APC callback will not be invoked. Why?

The reason is still the same as in the first console example above. Our main thread, that we queued our APC to, does not enter an alertable state by invoking those "magic APIs" that Microsoft listed in their documentation.

To make this work we need to adjust our message loop, and namely unfold the `GetMessage` function, to enter an alertable state. So let's see how our message loop will look then:

C++

```
MSG msg;
int nExitCode = 0;

for (;;)
{
        DWORD dwR = ::MsgWaitForMultipleObjectsEx(0, NULL, INFINITE, QS_ALLINPUT,
MWMO_ALERTABLE | MWMO_INPUTAVAILABLE);
        if (dwR == WAIT_FAILED)
        {
                //Error
                assert(false);
                nExitCode = -1;
                break;
        }

        while (::PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
                if (msg.message == WM_QUIT)
                {
                        //Normal exit
                        return (int)msg.wParam;
                }

                if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }
}

return nExitCode;
```

Note the addition of a new function, `MsgWaitForMultipleObjectsEx`, that is actually doing all the heavy lifting, of not just waiting for an incoming message, but also processing of APC callbacks for us. In it, we requested to wait for all messages by specifying `QS_ALLINPUT` flag, and then also requested to enter an alertable state by using `MWMO_ALERTABLE` , and also to return when messages are available by using `MWMO_INPUTAVAILABLE` .

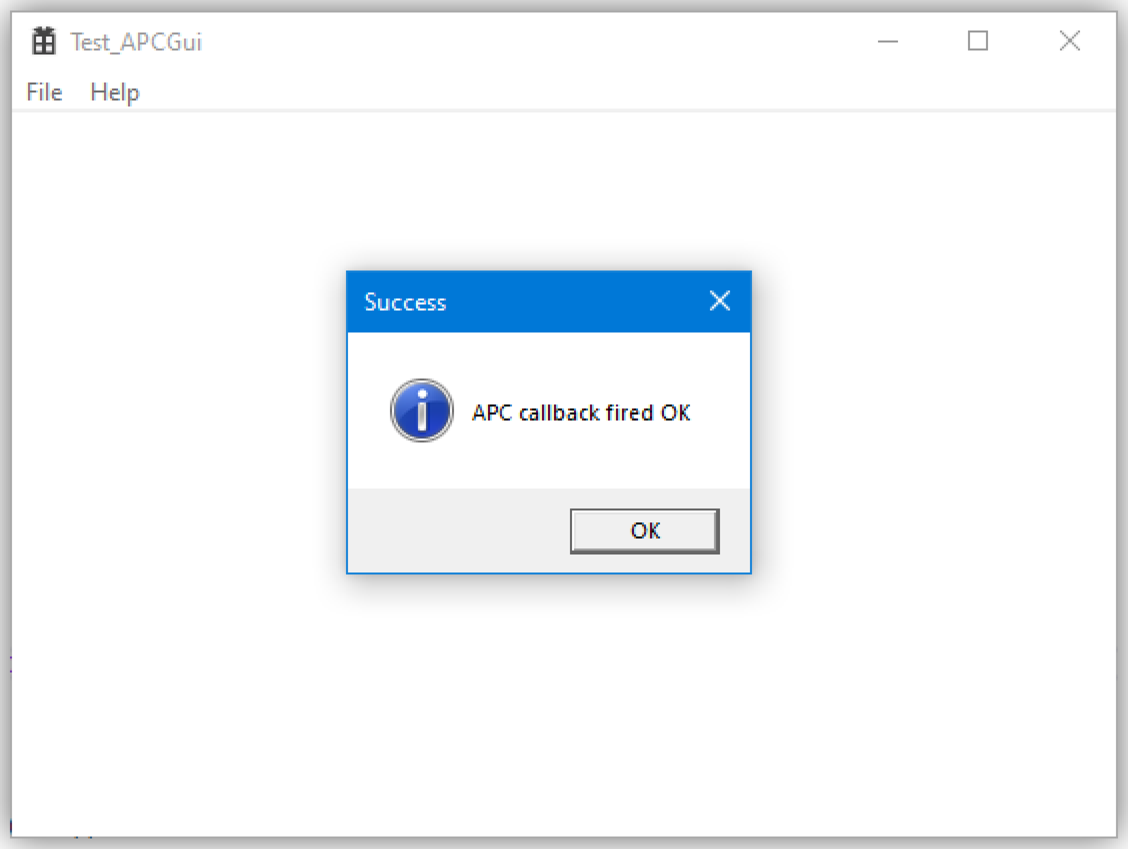
Also note that we're then using `PeekMessage` to retrieve a message and then to remove it from the queue by specifying the `PM_REMOVE` flag.

Additionally, we need to catch the moment when our GUI app is exiting. This will happen when the logic in it calls `PostQuitMessage` function, that in turn sends us the `WM_QUIT` message, that our updated message loop will catch and then exit from the `wWinMain` function with the *exit code* supplied to `PostQuitMessage` .

Lastly, we still need to take care of the error handling. In case of a GUI app for debugging-stage error handling I prefer to use visual assertions, provided by the `assert.h` library, as the `assert` macro. I used it in case `MsgWaitForMultipleObjectsEx` fails, to give us a visual indication of a problem.

> Note that `assert` macros are compiled only in the `Debug` build configuration and are very handy for debugging GUI applications.

So if we run the program with our updated message loop, after we invoke our `set_test_APC`, the APC callback should be called and we should see our visual indicator:



APC callback fired OK

As you see, it wasn't that much code to unfold our message loop.

But next let's see what happens when we don't have access to a message loop.

APC With a Dialog Box

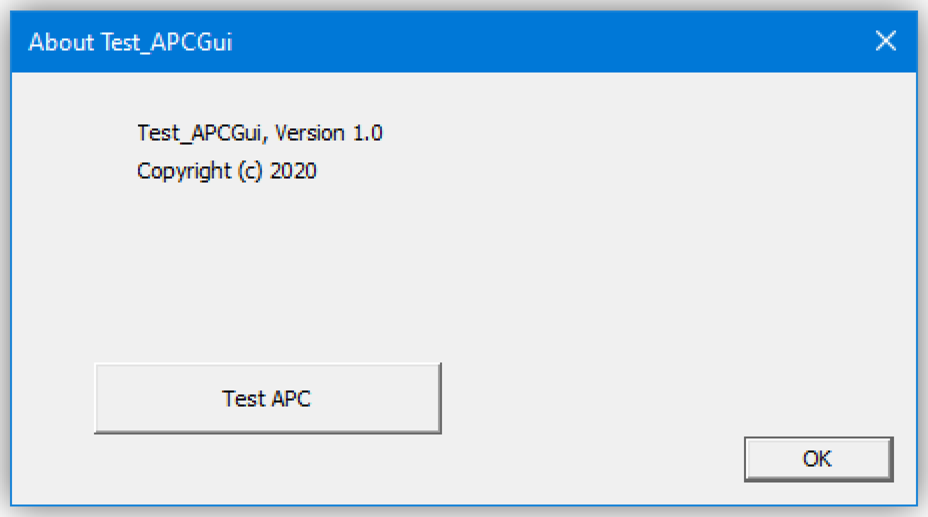
A dialog box in Windows parlance is a window that is created internally by specifying a special layout of its controls in the format of a *resource*. In your app, you may be creating many of your windows this way, using the Visual Studio's resource editor. The following code demonstrates creation of such a dialog in our stock Win32 GUI app:

C++

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
```

It's easy to overlook the simplicity of the `DialogBox` macro, that in one line can create, process and destroy a new window.

To try our APC callback with that dialog box, lets try to add a button to it (using Visual Studio resource editor) and then add a handler to it to invoke our `set_test_APC` function:



But when we invoke our `set_test_APC` from a dialog box, nothing happens. Why?

The reason we don't have our APC callback invoked from a dialog box is because internally it uses its own message loop. To address it, we will need to unfold the call to `DialogBoxParam`, or the function that is called by the `DialogBox` macro.

This task is a little bit more involved and requires the use of a small *undocumented hack*. Let's review the code:

C++

```
HWND hDlg = ::CreateDialog(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
if (hDlg)
{
        ::ShowWindow(hDlg, SW_SHOW);

        //Disable parent window to make ours into a modal dialog
        ::EnableWindow(hWnd, FALSE);

        MSG msg;
        BOOL bStopStop = FALSE;

        for (; !bStopStop;)
        {
                DWORD dwR = ::MsgWaitForMultipleObjectsEx(0, NULL, INFINITE,
QS_ALLINPUT,
                        MWMO_ALERTABLE | MWMO_INPUTAVAILABLE);
                if (dwR == WAIT_FAILED)
                {
                        //Error
                        assert(false);
                        break;
                }

                while (::PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
                {
                        //Hack to ensure processing of EndDialog() calls
                        if (msg.message == WM_NULL && msg.hwnd == hDlg)
                        {
                                //Normal exit
                                bStopStop = true;
                                break;
                        }

                        if (!::IsDialogMessage(hDlg, &msg))
                        {
                                if(msg.message >= WM_KEYFIRST && msg.message <=
WM_KEYLAST)
                                {
                                        TranslateMessage(&msg);
                                }

                                DispatchMessage(&msg);
                        }
                }
        }

        ::DestroyWindow(hDlg);
        hDlg = NULL;

        //Re-enable parent window
        ::EnableWindow(hWnd, TRUE);
}
```

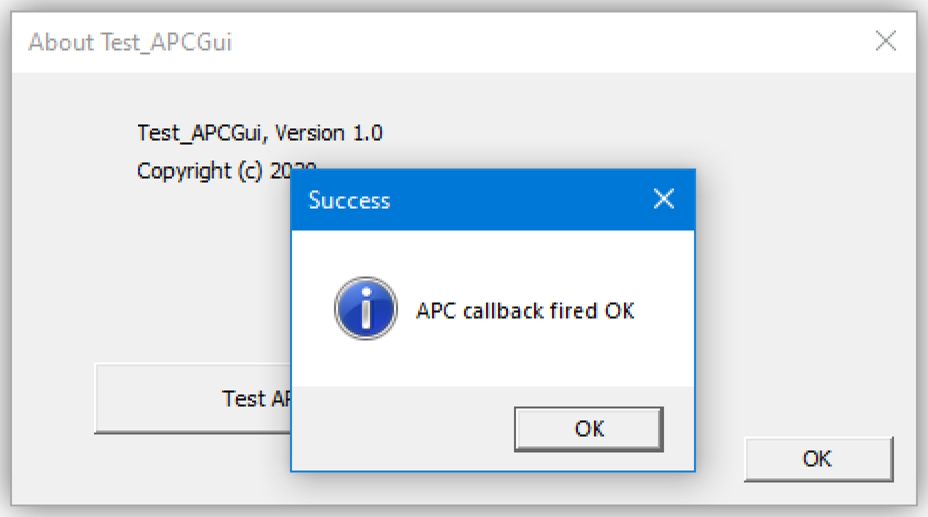
```
else
        assert(false);
```

As you can see, there's way more code. It's somewhat similar to how we handled the main message loop, but also has its own intricacies. To name a few:

- First, to get access to the message loop we need to create our dialog as *modeless*. (Note that `DialogBoxParam` creates *modal* dialog boxes. So we need to emulate that.) We can achieve this by calling the `CreateDialog` macro.
- To convert our *modeless* dialog box into a *modal* one, we need to disable the parent window for the duration when our dialog is shown, and then remember to re-enable it back. We can do that using the `EnableWindow` function.
- Our updated message loop is similar to what we've done before. The call to `MsgWaitForMultipleObjectsEx` is exactly the same. Again, we need it to ensure that our thread can process APC callbacks. But additionally it also waits for messages in a queue and returns if there are any.
- Because it's a dialog box, we needed to differentiate between messages that are specific for our dialog window using the `IsDialogMessage` function, and not to process them in our message loop logic. This is needed to ensure that dialog specific key combinations continue to work.
- Alternatively we call `TranslateMessage` to convert keyboard strokes into specific messages. We do so only if we detect that its a keyboard message.
- And lastly, the most critical part of our message loop, is how we end it. The issue with the *modal* dialog box is that it is destroyed using the `EndDialog` function. Internally, this function simply hides the dialog window, and then sets a flag to end the internal message loop. After that it sends the `WM_NULL` message to invoke execution inside the said message loop.
  Since we can't access the internal flag inside its message loop, we can only rely on the presence of the `WM_NULL` message to end our own loop. This is definitely a *hack* though. Ideally you would not rely on it and instead use some internal variable to signal when the dialog window needs to close, and set it before calling `EndDialog`. Then you would check it after confirmation that `msg.message == WM_NULL`.

- Lastly, since we created our *modeless* window we need to remember to destroy it. We do it at the end using `DestroyWindow` function.

Now, with our modified message loop, when we invoke our `set_test_APC` from the dialog box, we get our APC callback to fire just fine:



APC callback fired OK

Caveats

Note that even though we were able to emulate most common message loops in GUI apps, this doesn't keep us out of the woods yet, when it comes to queuing user-mode APCs. There are still a few cases when that poses a challenge, that you need to be aware of. All of them have their own internal message loops that we do not have access to. To name just a few examples:

- **MessageBox** - creates a popup message box that is a *modal* dialog window that also hides its own message loop. Unfortunately there's no easy solution to replace it. Thus if you need to rely on APC callbacks when a message box can be shown, it is better to either write your own implementation of a message box, or to use some other notification technique other than APC.
- **PropertySheet** - any property sheet, or a wizard is usually created as a *modal* dialog window and will have similar issues processing APC callbacks.
- **GetOpenFileName**, **GetSaveFileName**, **SHBrowseForFolder**, **ChooseFont**, **PrintDlgEx** - are just a few functions that come to mind that display a dialog window with its own internal message loop that will prevent processing of APCs. The workaround here is simply to use some other notification technique other than APC.