What does this crash in TppRaiseHandleStatus mean?

devblogs.microsoft.com/oldnewthing/20160115-00

January 15, 2016

Raymond Chen

A customer found that their program was crashing with this stack:

```
ntdll!TppRaiseHandleStatus
ntdll!TppSetupNextWait
ntdll!TppWaitCompletion
ntdll!TppWorkerThread
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
```

“Since none of our code is on the stack, it is unclear what may have gone wrong here.”

A naming convention followed by many teams is to come up with a short prefix for a component and put that prefix on every function in that component. For example, all of the I/O manager functions begin with `Io`, then a capital letter. Furthermore, functions intended to be called from other components use the unadorned prefix, and functions that are internal to the component add a `p` (for *private*) to the prefix. For example, all of the functions internal to the I/O manager begin with `Iop`, then a capital letter.

(An older convention is to use the lowercase prefix to indicate internal functions. Under the older convention, the internal functions in the I/O manager would begin with `io` followed by a capital letter.)

Given that information about naming conventions, you can guess that the functions prefixed `Tpp` are functions that are internal to a component whose prefix is `Tp`.

And your guess would be correct. This code is from the thread pool.

One of the purposes of the thread pool is to consolidate waits. If one object wants to do something when handle A is signaled, and another object wants to do something when handle B is signaled, one way to do this is to have each object create a thread, and have that thread wait on the corresponding handle. Unfortunately, that results in two threads, and when you have a hundred objects, this results in a hundred threads, and that doesn’t scale well. Better would be to put one thread in charge of all the waiting, so that it can use `WaitForMultipleObjects`. That way, you need only ⌈n/ `MAXIMUM_WAIT_OBJECTS`⌉ threads to wait on n handles.

Okay, so let's see what this stack is telling us. I am not familiar with the thread pool's internals, so this is all educated guesswork. It's educated by the fact that code is not intentionally written to be impossible to understand. For example, if a function is called `TppSetupNextWait`, it probably sets up the next wait.

The thread pool started its own worker thread with `TppWorkerThread`; that makes sense. And then a wait completed, which was handled by `TppWaitCompletion`. After handling that wait, the thread wants to go back to waiting until the next handle becomes signaled, which is done by the function `TppSetupNextWait`. But something went wrong, and it needs to raise a status, which is done by the `TppRaiseHandleStatus` function. And that's where we crashed.

What could go wrong in `TppSetupNextWait`? The most likely reason in my opinion is that the list of handles it is being asked to wait on is no longer valid, probably because an application closed a handle while it was still registered with the thread pool, so when the thread pool tried to wait on it, it got an `ERROR_INVALID_HANDLE` error.

The documentation for the `SetThreadpoolWait` function says,

> If this handle is closed while the wait is still pending, the function's behavior is undefined.

The customer wrote back that they have another crashing stack that points at their application code, so they have something else to work with to help pin down where the handle went bad.

Raymond Chen

Follow