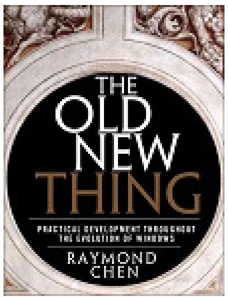
The case of the invalid handle error when a handle is closed while a thread is waiting on it

devblogs.microsoft.com/oldnewthing/20250620-00/?p=111291

June 20, 2025



A customer tracked one of their crashes to an invalid handle exception being raised when one thread closed a handle that another thread was waiting for. Or at least that's how they presented the problem.

The stack trace in the crash dump said

```
ntdll!KiRaiseUserExceptionDispatcher+0x3a
KERNELBASE!WaitForMultipleObjectsEx+0x123
KERNELBASE!WaitForMultipleObjects+0x11
contoso!Widget::WaitUntilReadyAsync$_ResumeCoro$1+0x1316
contoso!std::experimental::coroutine_handle<void>::resume+0xc
contoso!std::experimental::coroutine_handle<void>::operator()+0xc
contoso!winrt::impl::resume_background_callback+0x10
ntdll!TppSimplepExecuteCallback+0x14d
ntdll!TppWorkerThread+0x819
kernel32!BaseThreadInitThunk+0x17
```

Here's a simplified version of the code:

```
struct Widget : std::enable_shared_from_this<Widget>
{
    wil::unique_event m_readyEvent{ wil::EventOptions::ManualReset };
    wil::unique_event m_shutdownEvent{ wil::EventOptions::ManualReset };

    winrt::IAsyncOperation<bool> WaitUntilReadyAsync()
    {
        co_await winrt::resume_background();

        HANDLE events[] = { m_readyEvent.get(), m_shutdownEvent.get() };
        auto status = WaitForMultipleObjects(ARRAYSIZE(events), events,
                        FALSE /* bWaitAll */, INFINITE);

        switch (status) {
        case WAIT_OBJECT_0:
            co_return true; // the ready event is set

        case WAIT_OBJECT_0 + 1:
            co_return false; // the shutdown event is set

        case WAIT_FAILED:
            FAIL_FAST_LAST_ERROR();

        default:
            FAIL_FAST();
        }
    }
};
```

The customer's debugging showed that the `Widget` object had already destructed. (The coroutine should have done a `auto lifetime = shared_from_this()` to ensure that the `Widget` did not destruct while it was still in use.) The destructor of `wil::unique_event` closes the event handle, so we have a case of closing a handle while a thread was waiting on it. The `WaitForMultipleObjecs` documentation calls this out:

> If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

The customer noted that the behavior was undefined, but nevertheless wondered why they were crashing at the `WaitForMultipleObjects` call. When they tried to reproduce the error in-house by forcing the object to destruct during the wait, but they couldn't get the crash that their clients were getting.

The first thing to note is that "undefined behavior" means that anything can happen. Maybe it crashes, maybe it hangs, maybe it returns "The event is set" even though it isn't, maybe it just seems to work okay. There's no requirement that the undefined behavior be consistent from one instance to another.

But you don't need to use the "undefined behavior" escape hatch to explain the behavior.

If you have a case where one thread closes a handle after another thread has started waiting for it, then you also have a case where one thread closes a a handle *before* another thread has started waiting for it.

If this is possible…

Thread 1	Thread 2
`WaitForMultipleObjects(...);`	
	`CloseHandle(...);`

Then this is also possible because there is no synchronization between the two threads, so one CPU might just get lucky and execute a tiny bit faster than the other:

Thread 1	Thread 2
	`CloseHandle(...);`
`WaitForMultipleObjects(...);`	

In this case, it's clear that the `WaitForMultipleObjects` will fail with "invalid handle" since the handle had already been closed by the time we try to wait on it.

Therefore, my guess was that we are in the second case, where the `CloseHandle` raced ahead of the `WaitForMultipleObjects`, causing the `WaitForMultipleObjects` to wait on an invalid handle. We know that it's possible, and the exception code of "invalid handle" is consistent with that theory.

The customer was not entirely convinced that the object was being destroyed before the wait. They observed that the `status` variable always held the value 41. What does 41 mean?

The `status` variable contains the value 41, not because `WaitForMultipleObjects` returned 41, but because `WaitForMultipleObjects` *never returned a value*. This process had enabled the strict handle checking policy by using `SetProcessMitigationPolicy`, which means that a use of an invalid handle raises an exception rather than just failing with `ERROR_INVALID_HANDLE`. You can see this in the stack trace where the call to `Wait-ForMultipleObjectsEx` raised an exception rather than returning.

Therefore, the `status` variable has yet to be initialized, so it contains garbage. The value 41 doesn't mean anything; it's a value left over from previous computations.

The customer was still unconvinced. "Are you sure that it's garbage? In the case of garbage, we would have expected it to contain a random number, but in the crash dumps, the value is consistently 41."

Garbage is not the same as random.

If you look in my garbage can as I take it out to the curb, you will always find a bag of lint on top. That's because the laundry room is the room closest to where I keep the garbage can, so the last thing I do before taking out the garbage can is grab the trash from the laundry room and put it into the garbage can, and the result is that the lint always winds up on the top.

But it's still garbage.

In the case of the program that is crashing, the storage assigned to that variable happens to have always had a leftover value of 41. Why? Who knows. I guess you could debug it further if you are really curious, but really, it's not of any consequence.

The code could fix the race condition by taking a strong reference to the `Widget` to ensure that it doesn't destruct while it's still in use.

```
winrt::IAsyncOperation<bool> WaitUntilReadyAsync()
{
    auto lifetime = shared_from_this();

    co_await winrt::resume_background();

    ⟦ ... rest as before ... ⟧
}
```

Bonus chatter: I found it a bit ironic that the customer simultaneously believed that "unpredictable behavior should always behave the same" when they were trying to reproduce the problem, yet also believed that "unpredictable behavior should never behave the same" when they were trying to explain the consistent presence of the value 41.