Thread pool threads are like preschool: Leave things the way you found them

devblogs.microsoft.com/oldnewthing/20250613-00/?p=111268

June 13, 2025

A customer wanted to use the Windows thread pool, but they also wanted to use COM from their work item. They saw in the COM documentation that each thread must call `CoInitialize(Ex)` before using COM, so they planned on doing something like this:

```
thread_local bool isComInitialized = false;

auto DoWorkOnBackground()
{
    return TrySubmitThreadpoolCallback(
        WorkFunction, nullptr, nullptr);
}

void CALLBACK WorkFunction(
    [[maybe_unused]] PTP_CALLBACK_INSTANCE instance,
    [[maybe_unused]] void* context)
{
    if (!isComInitialized) {
        CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED);
        isComInitialized = true;
    }

    ⟦ do some work ⟧
}
```

The idea is that before each work function runs, it checks whether it has already initialized COM in apartment threaded mode for this thread. If not, it does the initialization, and then it remembers that the initialization has been done so it won't do it again.

The problem with this approach is that it initializes COM on a thread pool thread but fails to clean up the initialization before returning the thread to the thread pool. The nickname for a thread in the thread pool that has been left in a bad state is *poisoned*.

When the next task runs on that thread, it will be running on a COM single-threaded apartment even though it didn't expect to. That thread might perform a long blocking operation like `WaitForSingleObject`, thinking that it's safe to do so because it's on a background thread. Unfortunately, it's secretly running on a COM single-threaded

apartment, which is required to pump messages while waiting. The result is that you now have a UI thread that has stopped pumping messages, and the system will mark it as unresponsive. And the system might be broadcasting a message to all windows, and that includes the helper window that COM created to manage inbound calls to the single-threaded apartment. The broadcast hangs, and now you have problems like hangs in the `SystemParametersInfo` function or 30-second hangs when opening documents waiting for the DDE timeout.

Indeed, the problem occurs even before the thread is used to run another task. If there are no tasks waiting to run, then the thread is returned to the thread pool, where the thread simply blocks waiting for work. This block happens without pumping messages because the thread pool has no expectation that anybody just left windows lying around on the thread. And then you get the same problem of broadcast hangs.

Even if you don't get a broadcast hang, the next task that runs on the thread pool thread might do a `CoInitializeEx(COINIT_MULTITHREADED)` to initialize the thread in the multithreaded apartment, and it will get the error `RPC_E_CHANGED_MODE`. There is really no recovery from this, so the task will fail, and then the poisoned thread pool thread gets returned to the thread pool, ready to terrorize the next task.

So leave thread pool threads the same way you found them. If you initialize COM, then uninitialize it. If you change the thread priority, then set it back. If you change the thread execution state to "continuous display required", then change it back. That thread does not belong to you. You were merely given permission to borrow it. You are a guest in someone else's house: If you want to put up your own posters, remember to take them down when you leave.

Bonus chatter: Don't forget to do your cleanup even if the task fails. Using a C++ RAII type makes it easy to ensure that the cleanup occurs no matter how the function exits.