How do I disassociate a thread from an I/O completion port?

devblogs.microsoft.com/oldnewthing/20210120-00

January 20, 2021

Raymond Chen

A thread becomes associated with an I/O completion port when it calls `GetQueued-CompletionStatus` to retrieve a completion from that port. I/O completion ports keep track of the threads that are associated with them, because one of the rules for I/O completion ports is that the I/O completion port will not dispatch new work if the number of associated threads that are active (not blocked on anything) has reached the concurrency maximum.[1]

But what if you want to dissociate a thread from an I/O completion port, because you want to use the thread for something else?

Here's an artificial program that demonstrates the problem. As is typical of Little Programs, it does no error-checking.

```
#include <windows.h>
#include <stdio.h> // horrors! Mixing C and C++!

DWORD CALLBACK ThreadProc(HANDLE iocp)
{
    DWORD bytes;
    ULONG_PTR key;
    LPOVERLAPPED overlapped;
    GetQueuedCompletionStatus(iocp, &bytes,
                              &key, &overlapped, INFINITE);
    printf("thread %d received %d bytes\n", GetCurrentThreadId(), bytes);
    while (true) {} // hard spin loop
}

int main(int argc, char** argv)
{
    auto iocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 1);
    PostQueuedCompletionStatus(iocp, 1, 0, nullptr);
    PostQueuedCompletionStatus(iocp, 2, 0, nullptr);

    DWORD id;
    CloseHandle(CreateThread(nullptr, 0, ThreadProc, iocp, 0, &id));
    CloseHandle(CreateThread(nullptr, 0, ThreadProc, iocp, 0, &id));
    Sleep(5000);
    return 0;
}
```

This program creates an I/O completion port with a concurrency maximum of one (last parameter) and posts two completions to it.

Next, the program creates two threads, each of which calls `GetQueuedCompletionStatus` to pull a completion from the completion port. The act of doing this also associates the thread with the completion port.

What happens is that one of the threads will pick up the first completion, and then go into a hard spin loop, simulating some computationally-intensive operation. This spin loop is not blocked in the operating system, so the thread is considered by the I/O completion port to be doing productive work, and the I/O completion port will count it as an active thread.

This means that the I/O completion port has hit its concurrency maximum, and the system will not dispatch new work. There is a second thread ready to accept work, but it will not receive any because the I/O completion port was configured to consume CPU on at most one thread.

But maybe what really happened is that the first thread isn’t really doing work on behalf of the I/O completion port any more. Perhaps the work item it received was a message that the code is using to reduce the number of threads in the I/O completion port due to some internal throttling logic. The thread receives the “you are no longer needed” message and stops doing I/O completion port work, and instead goes off to do something else.

Even though the code has told the thread, “You are no longer needed,” the operating system still thinks it’s part of the I/O completion port. The code intends the remaining thread in the I/O completion port to assume the work, but the operating system won’t let it. You need to get the first thread to dissociate from the I/O completion port.

According to the documentation, once a thread is associated with an I/O completion port, it remains associated until one of three things happens: The thread exits, the I/O completion port is closed, or the thread associates with a new I/O completion port.

We want the thread to continue running, just without its I/O completion port, so exiting the thread is not an option. And closing the I/O completion port is not an option either, because the I/O completion port is still being used to process work. We just don’t want this thread to be part of it.

That leaves us the last option: Associate it with another I/O completion port. And then once we do that, we can close it.

```
void DissociateThreadFromIoCompletionPort()
{
    auto iocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 1);
    DWORD bytes;
    ULONG_PTR key;
    LPOVERLAPPED overlapped;
    GetQueuedCompletionStatus(iocp, &bytes, &key, &overlapped, 0);
    CloseHandle(iocp);
}
```

This function creates a new I/O completion port and immediately associates the thread with it by calling `GetQueuedCompletionStatus`. We pass a timeout of zero, so the call returns immediately. This dissociates the thread from the old completion port.

And then we close the handle to the new I/O completion port, causing the thread to become completely dissociated from *any* I/O completion port.

It’s sort of the I/O completion port version of buying something so you can set it free.

We can now use this function in our artificial example:

```
DWORD CALLBACK ThreadProc(HANDLE iocp)
{
    DWORD bytes;
    ULONG_PTR key;
    LPOVERLAPPED overlapped;
    GetQueuedCompletionStatus(iocp, &bytes,
                              &key, &overlapped, INFINITE);
    printf("thread %d received %d bytes\n", GetCurrentThreadId(), bytes);
    DissociateThreadFromIoCompletionPort();
    // now do some work that isn't charged to the I/O completion port
    while (true) {} // hard spin loop
}
```

You might do this if you want to do some work without having it charged to the I/O completion port, and then rejoin the I/O completion port when the side work is complete. You can dissociate from the completion port temporarily, do the side work, and then reassociate with the completion port when you go back and call `GetQueuedCompletionStatus` again.

Bonus chatter: You can use `FileReplaceCompletionInformation` to change the I/O completion port associated with a file.

[1] I've had a partly-written series on the philosophy of I/O completion ports sitting in my "unfinished" pile since 2012. I need to dust that off someday and finish it already. The fact that we now have coroutines in C++ makes this easier to demonstrate, except that I now have to finish my series on coroutines first!

Raymond Chen

Follow