How can I emulate the REG_NOTIFY_THREAD_AGNOSTIC flag on systems that don't support it? part 5

devblogs.microsoft.com/oldnewthing/20201225-00

December 25, 2020

Raymond Chen

We complete our somewhat pointless exercise of emulating the `REG_NOTIFY_THREAD_AGNOSTIC` flag by making our coroutine resilient to failure partway through. This requires you to accept the anachronism of using C++20 coroutines, while also dropping Windows XP support, since we will be relying on thread pool features new to Windows Vista.

```
auto RegNotifyChangeKeyValueAsync(
  HKEY hkey,
  BOOL bWatchSubtree,
  DWORD dwNotifyFilter,
  HANDLE hEvent)
{
  struct awaiter
  {
    awaiter(awaiter const&) = delete;
    void operator=awaiter(awaiter const&) = delete;

    HKEY m_hkey;
    BOOL m_bWatchSubtree;
    DWORD m_dwNotifyFilter;
    HANDLE m_hEvent;
    LONG m_result;
    PTP_WORK m_completionWork = nullptr;
    std::experimental::coroutine_handle<> m_handle;

    ~awaiter()
    {
      if (m_completionWork) CloseThreadpoolWork(m_completionWork);
    }

    bool await_ready() const noexcept { return false; }

    bool await_suspend(std::experimental::coroutine_handle<> handle)
    {
      m_completionWork = CreateThreadpoolWork(Complete, this, nullptr);
      if (!m_completionWork) {
        m_result = static_cast<LONG>(GetLastError());
        return false;
      }

      m_handle = handle;

      if (!QueueUserWorkItem(
          Register,
          this,
          WT_EXECUTEINPERSISTENTTHREAD)) {
        m_result = static_cast<LONG>(GetLastError());
        return false;
      }

      return true;
    }

    LONG await_ready() const noexcept { return m_result; }

    DWORD CALLBACK Register(void* param)
    {
      auto self = reinterpret_cast<awaiter*>(param);
```

```
      self->m_result = RegNotifyChangeKeyValue(
        self->m_hkey,
        self->m_bWatchSubtree,
        self->m_dwNotifyFilter,
        self->m_hEvent,
        TRUE);
      SubmitThreadpoolWork(m_completionWork);
      return 0;
    }

    DWORD CALLBACK Complete(void* param)
    {
      auto self = reinterpret_cast<awaiter*>(param);
      self->m_handle();
      return 0;
    }
  };

  return awaiter(hkey, bWatchSubtree, dwNotifyFilter, hEvent);
}
```

The idea here is that we have two work items. The first (for which we use `QueueUserWorkItem`) is scheduled onto a persistent thread. When that first work item runs (`Register`), we register the notification and save the result. And then we submit the second work item, which brings us to a normal thread pool thread, which is where we resume the caller by invoking its coroutine handle.

As before, if anything goes wrong during the set-up, we save the error and declare that the caller shouldn't suspend. That way, it picks up the error immediately.

There's a subtlety here: You might be tempted to clean up the completion work item as soon as `SubmitThreadpoolWork` returns, but that would be wrong. There is a race condition where the submitted work runs to completion before `SubmitThreadpoolWork` returns. In that case, the coroutine has already resumed, and the `awaiter` has already destructed. The subsequent call to `CloseThreadpoolWork(m_completionWork);` is accessing an object after it has been destroyed.

Bonus chatter: Commenter Paul Jackson observed that the thread which executes `WT_EXECUTEINPERSISTENTTHREAD` work items can exit if there are no pending I/O requests. Is RegNotifyChangeKeyValue a pending I/O request? I'm not sure. So maybe `WT_EXECUTEINPERSISTENTTHREAD` doesn't solve the problem after all. Fortunately, this was all a pointless exercise.

Raymond Chen

Follow