Accidentally creating a choke point for what was supposed to hand work off quickly to a background task, part 3

devblogs.microsoft.com/oldnewthing/20190215-00

February 15, 2019

Raymond Chen

Last time, we identified the reason why a function that intended to queue work to a background thread quickly ended up being a bottleneck because it waited until the task started running before returning to its caller.

The reason why the function waited for the task to run was to prevent COM from being uninitialized for the process, because that would cause all of its captured interface pointers to become invalid. What we need is a way to keep COM active even though we don't have a thread that we can use to ensure that it remains active.

Fortunately, there's a way to do it: The `CoIncrementMTAUsage` function lets you keep the MTA alive despite not actually having a thread dedicated to doing it.

```
class CCoMTAUsage
{
public:
 CCoMTAUsage() { CoIncrementMTAUsage(&m_cookie); }
 ~CCoMTAUsage() { Reset(); }
 bool Initialized() { return m_cookie; }
 bool Reset() {
  if (Initialized()) CoDecrementMTAUsage(m_cookie);
  m_cookie = nullptr;
 }

 // Movable but not copyable.
 CCoMTAUsage(const CCoMTAUsage&) = delete;
 CCoMTAUsage& operator=(const CCoMTAUsage&) = delete;
 CCoMTAUsage(CCoMTAUsage&& other) :
  m_cookie(other.m_cookie) { other.m_cookie = nullptr; }
 CCoMTAUsage& operator=(CCoMTAUsage&& other)
  { Reset(); Swap(other); return *this; }

 void Swap(CCoMTAUsage& other)
  { std::swap(m_cookie, other.m_cookie); }
private:
 CO_MTA_USAGE_COOKIE m_cookie = nullptr;
};
```

This helper class provides RAII-style support for managing the MTA usage cookie, and we can use this class to keep the MTA alive while our task is waiting to run. This removes the need to keep a thread hostage for the purpose of keeping the MTA alive.

```
// Error checking has been elided for expository purposes.
struct BackgroundData
{
 std::promise<StreamResult> promise;
 // Put this before the COM objects so it destructs after them.
 CCoMTAUsage m_mtaUsage;
 Microsoft::WRL::AgileRef agileStream;
 int taskId;
};

std::atomic<int> next_available_id = 1;

std::future<StreamResult> ProcessStreamInBackground(IStream* stream)
{
 // Create data that the background task will use.
 auto data = std::make_unique<BackgroundData>();

 var future = data->promise.get_future();

 // Make sure this task gets a unique ID number.
 data->id = next_available_id++;

 // Marshal the stream into the background task.
 Microsoft::WRL::AsAgile(stream, &data->agileStream);

 // Queue up the background task.
 // The background task will free the data when done.
 // The MTA cookie will keep the MTA alive.
 QueueUserWorkItem([](void* context) -> DWORD
 {
  // Initialize COM for this work item.
  CCoInitializeEx init;

  // Take responsibility for freeing the data.
  std::unique_ptr<BackgroundData>
    data{ reinterpret_cast<BackgroundData*>(context) };

  // Unmarshal the stream.
  Microsoft::WRL::ComPtr<IStream> stream;
  data->agileStream.As(&stream);

  // The main thread can resume now.
  // SetEvent(data->startEvent.Get());

  // Do our processing and get a result.
  StreamResult result = ProcessStuff(data.get());

  // Complete the promise.
  data->promise.set_value(result);

  // All done.
  return 0;
```

```
 }, data.release(), 0);

 // Wait for the stream to be unmarshaled.
 // DWORD index;
 // CoWaitForMultipleHandles(COWAIT_DEFAULT, INFINITE,
 //                          1, startEvent.Get(), &index);

 return future;
}
```

We no longer need to wait for the task to start. Just queue the task and return. No waiting.

Raymond Chen

Follow