Accidentally creating a choke point for what was supposed to hand work off quickly to a background task, part 1

devblogs.microsoft.com/oldnewthing/20190213-00

February 13, 2019

Raymond Chen

A customer was investigating performance issues in their program and after much effort was able to identify this function as one source of their problems.

```
// Error checking has been elided for expository purposes.
class CCoInitializeEx
{
public:
 CCoInitializeEx(DWORD dwCoInit = COINIT_MULTITHREADED) :
   m_hr(CoInitializeEx(nullptr, dwCoInit)) { }
 ~CCoInitializeEx() { if (SUCCEEDED(m_hr)) CoUninitialize(); }
 operator HRESULT() const { return m_hr; }
 HRESULT m_hr;
};

struct BackgroundData
{
 std::promise<StreamResult> promise;
 Microsoft::WRL::AgileRef agileStream;
 Microsoft::WRL::Wrappers::Event startEvent;
 int taskId;
};

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.
 std::lock_guard<std::mutex> guard(some_mutex);
 data->id = next_available_id++;

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

 // Create a manual-reset event, initially unset.
 Microsoft::WRL::Wrappers::Event startEvent;
 startEvent.Attach(CreateEvent(nullptr, true, false, nullptr));

 // Share it with the background task.
 DuplicateHandle(startEvent.Get(), GetCurrentProcess(),
                 data->startEvent.GetAddressOf(), GetCurrentProcess(),
                 0, false, DUPLICATE_SAME_ACCESS);

 // Queue up the background task.
 // The background task will free the data when done.
 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;
}
```

The idea is that this function takes the passed-in stream, marshals it into the background task, processes the stream to produce a result, and then returns that result back to the caller asynchronously. For expository purposes, I've used a `std::promise`, but in the original code, it used a custom object, the mechanics of which are not relevant to the discussion.

The subtlety here is that there is a window of time between the time the work item is queued to the thread pool and the time the work item actually starts running. During that time, the calling thread may have uninitialized COM. Even worse, the calling thread may have been the last thread that was using COM, so its uninitialization of COM for the thread also uninitializes COM for the entire process, which means that COM has wiped out all its memories of what it had done.

If COM gets uninitialized for the process, then that causes the agile reference we captured in the `BackgroundData` to become orphaned. When the background task starts running and tries to convert the agile reference back into a stream, COM says, "Huh? What is this thing? I've never seen you before."

The way the code solves this problem is by holding the main thread hostage until the background task initializes COM. This closes the danger window and prevents COM from being uninitialized for the process.

Profiling revealed that this function became a bottleneck in the program. The intent was to divert processing to a background thread, leaving the calling threads free to do other work. But in practice, all the calls to this function ended up serializing against each other. Furthermore, each thread took turns just waiting around. A loop that intended to queue up a thousand streams for background processing ended up stalled for nearly the entire time the background work was occurring.

One reason for this is easy to spot:

```
// Make sure this task gets a unique ID number.
std::lock_guard<std::mutex> guard(some_mutex);
data->id = next_available_id++;
```

The lock is held for the entire remainder of the function, which means that only one thread at a time gets to queue a background thread. To fix this, we can scope the lock:

```
{
 // Make sure this task gets a unique ID number.
 std::lock_guard<std::mutex> guard(some_mutex);
 data->id = next_available_id++;
} // release the mutex
```

Or we can go lock-free and switch to an interlocked operation:

```
data->id = InterlockedIncrement(&next_available_id);
```

Or use a `std::atomic`.

```
std::atomic<int> next_available_id{ 1 };
...
 // Make sure this task gets a unique ID number.
 // std::lock_guard<std::mutex> guard(some_mutex);
 data->id = next_available_id++;
```

This gets rid of the lock that serializes the creation of background tasks, but it doesn't explain why creating background tasks stalls for so long. We'll look at that next time.

Raymond Chen

Follow