Asynchronous operations

Niklas Gustafsson

Microsoft Corp.

Premise

- Synchronous, blocking, APIs are bad for a number of reasons:
  - Handling I/O using synchronous APIs wastes resources and limit scalability by waiting for completion inefficiently
  - Synchronous APIs make programming responsive graphical user interfaces complicated
  - By waiting synchronously, cancellation of outstanding work is complicated
- Asynchronous APIs address these problems and are becoming more and more ubiquitous
  - AJAX
  - Silverlight / .NET
  - Windows 8
  - Boost ASIO
  - Node.js

Problem

- The synchronous paradigms let developers presume that when a function returns, its result is available and its side-effects are complete

- The asynchronous paradigm is that the result will *eventually* be available and the side-effects will *eventually* be complete
  - This delay introduces tremendous complexity; thus, asynchronous programming is <u>hard</u>
  - There is no <u>standard</u> way of representing asynchronous operations in C++ (but there are in other languages)

Major Asynchronous Patterns

- Direct Callbacks
  - Pass a function object into the function initiating the operation
  - Used in Boost ASIO, Windows, .NET 4 (with modifications)
- Callback Interfaces
  - Pass a reference to an interface implementing the callback logic
  - Used in Windows 8
- Futures
  - Initiating function returns an object to which handlers can be attached
  - Used by JavaScript, .NET 4.5, many others

std::future / std::shared_future

- std::future *does* allow functions to represent a return value’s eventual availability
- Completely avoids having to pass callbacks or interfaces down
- but …

std::future / std::shared_future

- … just moves the synchronization to another location, the call to get()

- … does not allow the calling code to compose multiple operations into one

- … defines no “canonical” API for cancellation

- … does nothing to optimize for immediately available (prompt) values

- … provides no mechanism for making sophisticated scheduling choices

std::future “v2”

- Add an “asynchronous get(),” called “then(),” to allow chaining of code together by supplying a continuation function object
- Add when_all() and when_any() for parallel composition
- Add create_value<T>() / create_void() to create a “prompt” future
- Add is_done() to test whether a value is available to retrieve without blocking
- Adds a canonical abstract scheduling interface to implement custom scheduling logic

std::future “v2”

```
// From a Windows 8 / Metro-style game

auto ctx = windows::context::use_current();

m_client.request(methods::PUT, buf.str()).then(
    [this](std::future<http_response> tsk)
    {
        try
        {
            InterpretResponse(tsk.get);
        }
        catch (utilities::win32_exception &exc)
        {
            InterpretError(exc.error_code());
        }
    }, ctx);
```

Cancellation

- The need to cancel specific outstanding work is a common and important use case
  - Asynchronous operations make this a whole lot easier than synchronous
- Either consumer or producer may initiate
  - Producer: call set_exception()
  - Consumer: ?

Cancellation

- Proposal:
  - Add the concept of a “cancellation token”
    - Associate a token with each future/promise
    - Allow independent tokens to be created and associated with multiple futures/promises
  - Listeners (producers) register with the token
    - Operation creation functions have overload taking a token
  - Initiators call ‘cancel()’ on the token
    - Event is signaled to all present and future listeners

Scheduling

- The addition of continuation chaining (then()) requires a formalized notion of *scheduling*
  - Programmers may need control of what resources are used to execute the continuation code.
    - *Throttling* in a server-based scenario
    - Scalable mutual exclusion
    - Scheduling *on the GUI thread* in a client scenario
  - For example

    Boost ASIO: IO Service

    .NET: Synchronization Context

- Doesn’t have to be complex, but needs to be abstract

Feasibility

- This approach is currently taken by .NET, JavaScript, and other language environments
  - C# and VB are even building in language support for it

- PPL tasks, shipping in the next release of Visual Studio, makes this model available and is promoted as the *preferred* way to compose Windows 8 asynchronous operations in C++

Backup