Qt

..and modern (future) C++ asynchrony

Why use Qt?

- I recently heard remarks saying it’s “so old”..

Why use Qt?

Portability

Cross-platform

- Works the same on
  - Windows
  - Linux
  - Mac
  - Android
  - iOS
  - Various RTOSes (Integrity, QNX, VxWorks)

Multiple UI back-ends

- The GUI is HW-accelerated on
  - DirectX
  - OpenGL (and OpenGL/ES)
  - Metal
  - Vulkan
- Adding more of these didn’t break API/ABI.

Declarative UI

- I find it odd that ROOT examples using Qt are using QWidgets..
- ..which work fine, but are still relatively tedious to use,
- instead of using Qt Quick, i.e. Qt Declarative.

Declarative UI

- UI definitions are QML, which is javascript.
- A simple box model where layout is very simple to do, you can get very far by using Rows and Columns, and some anchors.

Declarative UI

- I haven’t used QML UIs with Cling, but I certainly plan to
  - That allows very rapid prototyping of user interfaces
  - And would be useful for Qt customers and users as well.

Qt keeps up with new C++ standards

- I’m going to show you a practical example of this with the real meat of this presentation, which is modern C++ asynchrony..
- ..and I’ll also give you a brief recap on what coroutines are and what they do.

Senders and Receivers (and Schedulers)

- Senders represent asynchronous work that can send a value, report an error, or report that the work has been canceled.

Senders and Receivers (and Schedulers)

- Receivers react to what Senders send, i.e. either (an exclusive or) a value, an error, or cancellation.

Senders and Receivers (and Schedulers)

- Schedulers represent an execution context, and can produce a Sender that sends a nullary value, i.e. like calling void f().

Senders and Receivers (and Schedulers)

- With these generic building blocks in place, we can now use generic adapters that
  - tie continuations to senders
  - cause senders to initiate their work on a particular context
  - cause sender continuations to run on a particular context.

Senders and Receivers (and Schedulers)

- This allows us to
  - synchronously, without any concurrency to worry about, build a Work Graph where the continuations are bound and context switches are also bound..
  - ..and then separately start and run the work, so that it runs asynchronously.

Senders and Receivers (and Schedulers)

- What we gain out of this is that
  - the continuation setup is generic and reusable
  - the context switch handling is generic and reusable
  - the actual domain-specific work is separated from the generic parts.

Schedulers and Senders in Qt

- a QThread is a Scheduler.
- the main event loop is also a scheduler, since it has a QThread in it, the loop is actually in Qthread.
- all QObjects are Senders.
  - That is, you can take any signal and convert the combination of the QObject and the signal into a Sender.

Coroutines

- Coroutines are functions that can do two additional things:
  - they can suspend, and return control back to the caller
  - they can be resumed, and continue execution from where they suspended.

Coroutines

- For the purposes of asynchrony, this means that coroutines can
  - initiate asynchronous work and suspend while “waiting” for the completion of that work
  - be resumed when that completion occurs, continuing their work after the asynchronous work is done.

Coroutines

- What this buys us is that
  - you don’t need to bind callbacks. Of any kind. Not even lambdas.
  - you initiate asynchronous work, and suspend..
  - ..and the code that follows is your continuation, it will run when the coroutine is resumed.

Senders and coroutines

- Senders are coroutines.
- coroutines are Senders.
- therefore, all QObject signals are coroutines..
- ..and you **can** bind continuations to coroutines if you want to, and you often do want to.

Senders and coroutines

- So all the reusable work-graph utilities (Sender algorithms) work with coroutines..
- ..which gives us the missing part of coroutine support in standard C++, aka the standard library support for them.

The plan for Senders&Receivers in Qt

- The first deliverable is integration of Qt with libunifex, which is an older (but feature-rich) implementation of Senders and Receivers by Facebook.

The plan for Senders&Receivers in Qt

- The second deliverable is integration of Qt with stdexec, which is a newer (but less feature-rich) implementation of Senders and Receivers by NVidia.

The plan for Senders&Receivers in Qt

- The third deliverable is integration of Qt with the NVidia HPC SDK, which contains an implementation of stdexec that can offload work to GPGPUs. In other words, it has GPU (CUDA) Schedulers.
- You can take your Senders, and have them initiate work on a GPU, or run their continuations on a GPU.