ECE 3574: Applied Software Design

Producer/Consumer Pattern

Today we are going to see how to use a design pattern that works well for concurrency as well as discuss Qt's threading implementation.

- Producer/Consumer Pattern
- C++11 producer/consumer using a thread-safe queue
- Reusing threads: thread *pools*
- Async function calls using QtConcurrent::Run
- QFuture
- QThread
- Qt-based producer/consumer

The producer/consumer pattern divides code into two largely independent pieces.

The *producer* which does the work of creating a product and putting it into a thread-safe data structure.

The *consumer* removes the product from the data structure and does something with it.

Note, all synchronization happens in the data structure.

C++11 producer/consumer using a thread-safe queue

Lets reuse the thread-safe queue from last time to implement an example.
See `cpp11_prodcon.cpp`.

Producer/Consumer is more efficient that async calls because it reuses threads.

How long does it take to create and join a thread?
See `threads_per_sec.cpp`. On my laptop

```
100000 threads in 1.50751 seconds.
66334.6 threads per second.
0.0150751 milliseconds per thread.
```

That seems fast, but compare that to just calling the thread_function.
It is over 1000 times *slower* even with no optimization.

Threads can be reused by creating a *thread pool*

A thread pool is a collection of running threads that can do a variety of work without starting/stopping threads each time.
Lets look at a potential implementation.
See `cpp11_threadpool_ex1.cpp`.
What issues are there with this design?
How might it be improved?

Qt Thread support

Qt has a threading library that is pretty standard, except for how it integrates with the event and signal/slot system:

- `std::async` and `std::future` become `QtConcurrent::run` and `QFuture`
- `std::thread` becomes `QtThread`
- `std::mutex` become `QMutex`

However:

- it uses a *thread pool*,which manages and recycles QThread objects
- threads can have there own event loop running
- you can use the signal/slot mechanism to send/receive signals *between* threads, which provides a thread-safe queued message passing system, and the ability to monitor and control thread execution (pause, resume, cancel).

Using QtConcurrent to run a function in another thread.

This is very similar to C++11 `std::async` usage.
See `qt_concurrent_ex1.cpp`, `qt_concurrent_ex2.cpp`, and `qt_concurrent_ex3.cpp`.

There are two ways to use QThread.

1. Subclass QThread and re-implement `run`. The constructor runs in the old thread while `start/run` executes in the new thread. Unless you call `exec` in the thread yourself there is no event loop. Emits signals when started, terminated, or finished.

See `qthread_ex1.cpp`.

2. Create a QThread object and move an object to it. Calling `start` starts a Qt event loop in the thread to which the object responds.

See `qthread_ex2.cpp`

QThread and signal/slots

You can monitor QThreads by connecting to the signals

- started - emitted when thread starts executing
- finished - emitted when thread is done executing (run returns)
- terminated - emitted when thread is terminated

You can manually managing threads by connecting signals to the slots

- start - start the thread event loop
- terminate - terminate the thread next time it is scheduled by OS (generally a bad idea)
- quit - tell the event loop to exit

Qt-based producer/consumer

Producer/Consumer is easy in Qt since `QtConcurrent::run()` uses a thread pool.
See `qt_concurrent_ex3.cpp`.

Next Actions and Reminders

- Read about the actor model
- Project 3 beta due 4/25 at 8 am