Multithreading with Qt

Giuseppe D'Angelo, Senior Software Engineer at KDAB
Agenda

- QThread (page 4)
- Synchronization (page 18)
- Thread safety in Qt (page 26)
- Qt and the Standard Library threading facilities (page 38)
Do you know what a thread is?
QThread

- QThread
- Synchronization
- Thread safety in Qt
- Qt and the Standard Library threading facilities
QThread

- QThread is the central class in Qt to run code in a different thread
- It's aQObject subclass
  - Not copiable/moveable
  - Has signals to notify when the thread starts/finishes
- It is meant to manage a thread
QThread usage

- To create a new thread executing some code, subclass QThread and reimplement run()
- Then create an instance of the subclass and call start()
- Threads have priorities that you can specify as an optional parameter to start(), or change with setPriority()
QThread usage

class MyThread : public QThread {
private:
    void run() override {
        // code to run in the new thread
    }
};

MyThread *thread = new MyThread;
thread->start(); // starts a new thread which calls run()
// ...
thread->wait(); // waits for the thread to finish
QThread usage

- The thread will stop running when (some time after) returning from run()
- QThread::isRunning() and QThread::isFinished() provide information about the execution of the thread
- You can also connect to the QThread::started() and QThread::finished() signals
- A thread can stop its execution temporarily by calling one of the QThread::sleep() functions
  - Generally a *bad idea*, being event driven (or polling) is much much better
- You can wait for a QThread to finish by calling wait() on it
  - Optionally passing a maximum number of milliseconds to wait
QThread caveats

From a non-main thread you cannot:

- Perform any GUI operation
  - Including, but not limited to: using any QWidget / Qt Quick / QPixmap APIs
  - Using QImage, QPainter, etc. (i.e. "client side") is OK
  - Using OpenGL may be OK: check at runtime
    QOpenGLContext::supportsThreadedOpenGL()

- Call Q(Core|Gui)Application::exec()
QThread caveats

- Be sure to always destroy all the QObjects living in secondary threads before destroying the corresponding QThread object
- Do not ever block the GUI thread
Ensuring destruction of QObjects

- Create them on `QThread::run()` stack
- Connect their `QObject::deleteLater()` slot to the `QThread::finished()` signal
  - Yes, this will work
- Move them out of the thread
Ensuring destruction of QObjects

```cpp
class MyThread : public QThread {
  private:
    void run() override {
      MyQObject obj1, obj2, obj3;

      QScopedPointer<OtherQObject> p;
      if (condition)
        p.reset(new OtherQObject);

      auto anotherObj = new AnotherQObject;
      connect(this, &QThread::finished,
              anotherObj, &QObject::deleteLater);

      auto yetAnother = new YetAnotherQObject;

      // ... do stuff ...

      // Before quitting the thread, move this object to the main thread
      yetAnother->moveToThread(qApp->thread());
      // Somehow notify the main thread about this object,
      // so it can be deleted there.
      // Do not touch the object from this thread after this point!
    }
};
```
QThread usage

There are two basic strategies of running code in a separate thread with QThread:

- Without an event loop
- With an event loop
QThread usage without an event loop

- Subclass QThread and override QThread::run()
- Create an instance and start the new thread via QThread::start()
QThread usage without an event loop

- Subclass QThread and override QThread::run()
- Create an instance and start the new thread via QThread::start()

```cpp
class MyThread : public QThread {
    private:
    void run() override {
        loadFilesFromDisk();
        doCalculations();
        saveResults();
    }
};

auto thread = new MyThread;
thread->start();
// some time later...
thread->wait();
```
QThread usage with an event loop

- An event loop is necessary when dealing with timers, networking, *queued connections*, and so on.

- Qt supports per-thread event loops:

- Each thread-local event loop delivers events for the QObjects living in that thread.
QThread usage with an event loop

- We can start a thread-local event loop by calling QThread::exec() from within run():

```cpp
1 class MyThread : public QThread {
2 private:
3     void run() override {
4         auto socket = new QTcpSocket;
5         socket->connectToHost(...);
6
7         exec(); // run the event loop
8
9     } // cleanup
10 }
11
```

- QThread::quit() or QThread::exit() will quit the event loop

- We can also use QEventLoop
  - Or manual calls to QCOREApplication::processEvents()
QThread usage with an event loop

- The default implementation of QThread::run() actually calls QThread::exec()

- This allows us to run code in other threads without subclassing QThread:

```cpp
auto thread = new QThread;

auto worker = new Worker;

connect(thread, &QThread::started, worker, &Worker::doWork);
connect(worker, &Worker::workDone, thread, &QThread::quit);
connect(thread, &QThread::finished, worker, &Worker::deleteLater);
worker->moveToThread(thread);
thread->start();
```
Synchronization

- QThread
- **Synchronization**
- Thread safety in Qt
- Qt and the Standard Library threading facilities
What is the single most important thing about threads?
**Synchronization**

- Any concurrent access to shared resources **must not result in a data race**

- Two conditions for this to happen:
  1. At least one of the accesses is a write
  2. The accesses are not *atomic* and no access *happens before* the other
Qt has a complete set of cross-platform, low-level APIs for dealing with synchronization:

- QMutex is a mutex class (recursive and non-recursive)
- QSemaphore is a semaphore
- QWaitCondition is a condition variable
- QReadWriteLock is a *shared mutex*
- QAtomicInt is an atomic int
- QAtomicPointer<T> is an atomic pointer to T

There are also RAII classes for lock management, such as QMutexLocker, QReadLocker and so on.
Mutex Example

```cpp
class Thread : public QThread
{
    bool m_cancel;

public:
    explicit Thread(QObject *parent = nullptr)
        : QThread(parent), m_cancel(false) {}

    void cancel() // called by GUI
    {
        m_cancel = true;
    }

private:
    bool isCanceled() const // called by run()
    {
        return m_cancel;
    }

    void run() override { // reimplemented from QThread
        while (!isCanceled())
            doSomething();
    }
};
```
class Thread : public QThread
{

mutable QMutex m_mutex; // protects m_cancel
bool m_cancel;

public:
explicit Thread(QObject *parent = nullptr) : QThread(parent), m_cancel(false) {}

void cancel() { // called by GUI
    const QMutexLocker locker(&m_mutex);
    m_cancel = true;
}

private:
bool isCanceled() const { // called by run()
    const QMutexLocker locker(&m_mutex);
    return m_cancel;
}

void run() override { // reimplemented from QThread
    while (!isCanceled())
        doSomething();
};
QThread's built-in cancel

QThread actually has this already built-in:

- `QThread::requestInterruption()`, to set the flag
- `QThread::isInterruptionRequested()`, to check the flag

```cpp
void run() override { // reimplemented from QThread
    const int checkAtNthIteration = 10;

    int iteration = 0;
    while (true) {
        ++iteration;
        if (iteration == checkAtNthIteration) {
            iteration = 0;
            if (isInterruptionRequested())
                return;
        }
        doSomething();
    }
}
```
Quick Quiz: Mutex Example

In this code:

```cpp
explicit Thread(QObject *parent = nullptr) : QThread(parent), m_cancel(false) {}  
```

don't you need to protect

```cpp
m_cancel(false)  
```

with `m_mutex`, too, like in `cancel()`?

```cpp
void cancel() { // called by GUI  
    const QMutexLocker locker(&m_mutex);  
    m_cancel = true;  
}  
```
Thread safety in Qt

- QThread
- Synchronization
- **Thread safety in Qt**
- Qt and the Standard Library threading facilities
Reentrancy definitions

A function is:
Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization

- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization.

- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization.

- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all.
Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization

- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization

- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all

For classes, the above definitions apply to non-static member functions when invoked on the same instance. (In other words, considering the this pointer as an argument.)
Examples

- **Thread safe:**
  - QMutex
  - QObject::connect()
  - QCoreApplication::postEvent()

- **Reentrant:**
  - QString
  - QVector
  - QImage
  - value classes in general

- **Non-reentrant:**
  - QWidget (including all of its subclasses)
  - QQuickItem
  - QPixmap
  - in general, GUI classes are usable only from the main thread
Thread safety for Qt classes/functions

The documentation of each class / function in Qt has notes about its thread safety:

**QString Class**

The `QString` class provides a Unicode character string. More...

Note: All functions in this class are reentrant.

Unless otherwise specified, classes and functions are non-reentrant.
QObject: thread affinity

What about QObject?
QObject: thread affinity

What about QObject?

- QObject itself is thread-aware.

- Every QObject instance holds a reference to the thread it was created into (QObject::thread())
  - We say that the object *lives in*, or has *affinity with* that thread

- We can move an instance to another thread by calling QObject::moveToThread(QThread *)
QObject: thread safety

QObject is **reentrant** according to the documentation, however:
QObject: thread safety

QObject is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)

Thread safety in Qt
QObject: thread safety

QObject is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given QObject happens in the thread it has affinity with
QObject: thread safety

QObject is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given QObject happens in the thread it has affinity with
- All the QObject objects in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent QObject objects created in a thread to the QThread object itself
QObject: thread safety

QObject is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given QObject happens in the thread it has affinity with
- All the QObject(s) in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent QObject(s) created in a thread to the QThread object itself
- You must delete all QObject(s) living in a certain QThread before destroying the QThread instance
QObject: thread safety

QObject is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given QObject happens in the thread it has affinity with
- All the QObjects in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent QObjects created in a thread to the QThread object itself
- You must delete all QObjects living in a certain QThread before destroying the QThread instance
- You can only call `moveToThread()` on a QObject from the same thread the object has affinity with (`moveToThread()` is non-reentrant)
QObject: thread safety

QObject is **reentrant** according to the documentation, however:

- Event-based classes are non-reenentrant (timers, sockets, ...)

- The event dispatching for a given QObject happens in the thread it has affinity with

- All the QObjects in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent QObjects created in a thread to the QThread object itself

- You must delete all QObjects living in a certain QThread before destroying the QThread instance

- You can only call moveToThread() on a QObject from the same thread the object has affinity with (moveToThread() is non-reenentrant)

In practice: **it's easier to think of QObject as non-reenentrant**, as it will make you avoid many mistakes.
QObject: queued connections

- If QObject is non-reentrant, how can I communicate with a QObject living in another thread?
If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?

Qt has a solution: **cross-thread signals and slots**
QObject: queued connections

- If QObject is non-reentrant, how can I communicate with a QObject living in another thread?

- Qt has a solution: **cross-thread signals and slots**

- You can emit a signal from one thread, and have the slot invoked by another thread
  - Not just any thread: the thread the receiver object is living in
QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object **by the right thread**
  - Handling such metacall events means invoking the slot
QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be queued.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object by the right thread
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!
QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.

- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object **by the right thread**
  - Handling such metacall events means invoking the slot

- This requires that the receiver object is living in a thread with a running event loop!

- Also, `qRegisterMetaType()` is required for the argument types passed
QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.

- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object by the right thread
  - Handling such metacall events means invoking the slot

- This requires that the receiver object is living in a thread with a running event loop!

- Also, qRegisterMetaType() is required for the argument types passed

- We can force any connection to be queued:

  ```
  connect(sender, &Sender::signal, receiver, &Receiver::slot, Qt::QueuedConnection);
  ```
class MyThread : public QThread {
    Producer *m_producer;
public:
    explicit MyThread(Producer *p, QObject *parent = nullptr)
        : QThread(parent), m_producer(p) {}
    void run() override {
        Consumer consumer;
        connect(m_producer, &Producer::unitProduced,
               &consumer, &Consumer::consume);
        exec();
    }
};

// in main thread:
auto producer = new Producer;
auto thread = new MyThread(producer);
thread->start();

// Producer::unitProduced gets emitted some time later from the main thread,
// Consumer::consume gets run in the secondary thread
// Same as before, but without the race

auto producer = new Producer;
auto consumer = new Consumer;
auto thread = new QThread;

connect(m_producer, &Producer::unitProduced,
consumer, &Consumer::consume);
connect(thread, &QThread::finished,
consumer, &QObject::deleteLater);

consumer->moveToThread(thread);
thread->start();

// Producer::unitProduced gets emitted some time later from the main thread,
// Consumer::consume gets run in the secondary thread
class MyThread : public QThread {

public:
    explicit MyThread(QObject *parent = nullptr)
    : QThread(parent) {} 

private:
    void run() override {
        emit mySignal();
    }

signals:
    void mySignal();
};

// in main thread:
auto thread = new MyThread;
connect(thread, &MyThread::mySignal, receiver, &Receiver::someSlot);
thread->start();
QObject: queued connections example (3)

```cpp
1 class MyThread : public QThread {
2 public:
3   explicit MyThread(QObject *parent = nullptr)
4     : QThread(parent) {} 
5
6 private:
7   void run() override {
8     emit mySignal();
9   }
10
11 signals:
12   void mySignal();
13 };
14
15 // in main thread:
16 auto thread = new MyThread;
17 connect(thread, &MyThread::mySignal, receiver, &Receiver::someSlot);
18 thread->start();
```

- It is perfectly OK to add signals to QThread
- The connection is queued: the thread that emits the signal is not the thread the receiver has affinity with
- someSlot() gets invoked by the main thread's event loop
class MyThread : public QThread {
    Socket *m_socket;
public:
    explicit MyThread(QObject *parent = nullptr)
        : QThread(parent) {}

private:
    void run() override {
        m_socket = new Socket;
        connect(m_socket, &Socket::connected, this, &MyThread::onConnected);
        m_socket->connectToHost(...);
        exec();
    }

private slots:
    void onConnected() { qDebug() << "Data received:" << m_socket->data(); }
};
QObject: queued connections example (4)

```cpp
class MyThread : public QThread {
    Socket *m_socket;
public:
    explicit MyThread(QObject *parent = nullptr)
        : QThread(parent) {}

private:
    void run() override {
        m_socket = new Socket;
        connect(m_socket, &Socket::connected, this, &MyThread::onConnected);
        m_socket->connectToHost(...);
        exec();
    }

private slots:
    void onConnected() { qDebug() << "Data received:" << m_socket->data(); }
};
```

- QThread is a QObject and as such has its own thread affinity (it's the thread that created the MyThread instance, not itself!)
- The connection is queued: the thread that emits the signal is not the thread the receiver has affinity with
  - *This is not what we wanted!*
- Huge recommendation: **avoid adding slots to QThread**
Qt and the Standard Library threading facilities

- QThread
- Synchronization
- Thread safety in Qt
- **Qt and the Standard Library threading facilities**
General remarks

- It is perfectly possible to mix'n'match Qt and std threading classes.

- The Standard Library is moving extremely fast and Qt will not (and should not) catch up with all of its new developments:
  - parallel algorithms, continuations, latches, barriers, atomic smart pointers, executors, concurrent queues, distributed counters, coroutines, ...

- More and more tooling will start checking for correct usages of std APIs, but not Qt ones (unless they get reimplemented on top of the std ones).

- QThread is still more convenient when dealing with QObjects and event loops.

- A comparison of the APIs is in the next slides.
### QThread vs. std::thread

<table>
<thead>
<tr>
<th></th>
<th>QThread</th>
<th>std::thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No need to subclass it in order to use it</td>
<td>✔️ ¹</td>
<td>✔</td>
</tr>
<tr>
<td>Function (job/task) runner</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Detach support</td>
<td>✗ ²</td>
<td>✔</td>
</tr>
<tr>
<td>Interruption request</td>
<td>✗</td>
<td>✗ ³</td>
</tr>
</tbody>
</table>

¹ only if we go for a signal/event-based design, which likely requires subclassing QObject

² we can emulate that by connecting QThread::finished() to QThread::deleteLater()

³ as shown before, it's trivial to emulate
### QThread vs. std::thread (2)

<table>
<thead>
<tr>
<th>Feature</th>
<th>QThread</th>
<th>std::thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event loop support</td>
<td>✔️</td>
<td>✗ 1</td>
</tr>
<tr>
<td>QObject can be created into</td>
<td>✔️</td>
<td>✔</td>
</tr>
<tr>
<td>QObject can be moved to</td>
<td>✔️</td>
<td>✔ 2</td>
</tr>
<tr>
<td>Signals can be emitted from</td>
<td>✔️</td>
<td>✔</td>
</tr>
<tr>
<td>Slots work in direct connections</td>
<td>✔️</td>
<td>✔</td>
</tr>
<tr>
<td>Slots work in queued connections</td>
<td>✔️</td>
<td>✔</td>
</tr>
</tbody>
</table>

1. But we can use QEventLoop to run a thread-local event loop
2. We can use QThread::currentThread() to get a QThread * (to move a QObject to, etc.)
Synchronization primitives

<table>
<thead>
<tr>
<th>Qt</th>
<th>Standard Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMutex</td>
<td>std::mutex</td>
</tr>
<tr>
<td></td>
<td>std::timed_mutex</td>
</tr>
<tr>
<td></td>
<td>std::recursive_mutex</td>
</tr>
<tr>
<td></td>
<td>std::recursive_timed_mutex</td>
</tr>
<tr>
<td>QSemaphore</td>
<td>✗</td>
</tr>
<tr>
<td>QReadWriteLock</td>
<td>std::shared_mutex</td>
</tr>
<tr>
<td></td>
<td>std::shared_timed_mutex</td>
</tr>
<tr>
<td>QWaitCondition</td>
<td>std::condition_variable</td>
</tr>
<tr>
<td>✗</td>
<td>std::call_once</td>
</tr>
<tr>
<td>Q_GLOBAL_STATIC</td>
<td>✗</td>
</tr>
</tbody>
</table>
Synchronization primitives: remarks

- QMutex and QReadWriteLock are faster than the std equivalents
- A non-recursive QMutex never allocates nor throws exceptions on Linux
- QMutex in 5.8 models the TimedLockable concept
  - Can be used together with std lock managers
- No std compatibility functions in QReadWriteLock (yet)
- std::condition_variable(_any) more generic / convenient than QWaitCondition
  - Supports any BasicLockable
  - Pass predicate to test in wait() call, instead of using the mandatory while loop
- Q_GLOBAL_STATIC is superseded by C++11's semantics for thread-safe function statics (and/or std::call_once)
## Lock management

<table>
<thead>
<tr>
<th>Qt</th>
<th>Standard Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMutexLocker</td>
<td>std::lock_guard</td>
</tr>
<tr>
<td>QReadLocker</td>
<td>std::shared_lock</td>
</tr>
<tr>
<td>QWriteLocker</td>
<td>std::lock_guard</td>
</tr>
<tr>
<td></td>
<td>std::unique_lock</td>
</tr>
<tr>
<td></td>
<td>std::lock()</td>
</tr>
</tbody>
</table>
Lock management: remarks

Standard Library lock management is much more powerful and flexible

- Movable lock guards (`std::unique_lock`) to return a managed lock
- Lock managers also have timed `try_lock()`s
- Tag classes to decide what a lock manager should do with the lock
- In C++17 `std::lock_guard` manages multiple locks (in a deadlock-free fashion)
  - `QOrderedMutexLocker` is C++17's `std::lock_guard` for two `QMutexes`
    - private API

Unless you're dealing with `QReadWriteLock`, prefer the std alternatives
## Atomics

<table>
<thead>
<tr>
<th>Qt</th>
<th>Standard Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>QBasicAtomicInteger&lt;T&gt;</td>
<td>std::atomic&lt;T&gt;</td>
</tr>
<tr>
<td>QAtomicInteger&lt;T&gt;</td>
<td></td>
</tr>
<tr>
<td>QAtomicInt</td>
<td></td>
</tr>
<tr>
<td>QBasicAtomicPointer&lt;T&gt;</td>
<td>std::atomic_operation()</td>
</tr>
<tr>
<td>QAtomicPointer&lt;T&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Qt and the Standard Library threading facilities
Atomics: remarks

- Starting with Qt 5.7, Qt atomics actually uses C++11 atomics under the hood
  - Except on MSVC, since it doesn't (properly) implement them yet

- The std atomics support extra (advanced) features compared to the Qt ones
  - Consume, acq+rel memory ordering
  - Different memory orderings available for success/failure in read-modify-write operations

- The non-member atomic operations allow for generic code and specializations

  ```cpp
  std::atomic_store(std::shared_ptr<T> *p, std::shared_ptr<T> q)
  ```

- If you do use atomics, start thinking to move towards the Standard Library
Thread-local storage

<table>
<thead>
<tr>
<th>Qt</th>
<th>Standard Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>QThreadStorage</td>
<td>thread_local</td>
</tr>
</tbody>
</table>

- Same functionality, different syntaxes
- Both lazy initialized
- QThreadStorage allows checking / skipping initialization
Questions?
Thank you!

www.kdab.com

giuseppe.dangelo@kdab.com