DEVELOPER DAYS 2014
NORTH AMERICA

QThread
Are you doing it wrong?

David Johnson
ZONARE Medical Systems

Topics

- Controversy and confusion
- Patterns of threading
- How QThread works
- Pitfalls to avoid
- Examples

Controversy

“You are doing it wrong!”

- Brad Hughes 2010

“You were not doing so wrong”

- Olivier Goffart 2013

“Have you guys figured it out yet?”

- Developer 2014

Who's Right?

- A casual reading of some blog posts leads one to imagine there is a “right” way and a “wrong” way to use QThread
- There are two valid patterns of QThread use
- Both are suitable for different use cases

Confusion Over QThread

- Only one pattern of QThread use prior to 4.4
  - Must subclass QThread and implement run()
- New pattern of QThread use since 4.4
  - Default implementation for run()
  - Default implementation simply calls exec()
- Documentation was not fully updated until Qt 5
- Problems arise when mixing the two patterns

Four Threading Patterns

- Runnables - Pass runnable object to thread
  - QThreadPool, Java
- Callables - Pass callback or functor to thread
  - Qt::Concurrent, std::thread, Boost::thread
- Subclassing - Subclass and implement run()
  - QThread, Java
- Event Driven - Events handled in associated thread
  - QThread

Notice that QThread has two patterns of use

Four Threading Patterns

Let's look at these in the context of Qt

Runnable Pattern

QThreadPool / QRunnable

- Create a QRunnable subclass
- Pass runnable object to QThreadPool::start()
- Very simple, but very low level
- Good for “fire and forget” style tasks
- Good for CPU-bound tasks

Example: Complex calculation

Callable Pattern

QtConcurrent

- Collection of STL like algorithms
  - Operates concurrently over collections and ranges
  - Function pointers, functors, std::bind
- Actual threads are managed by QThreadPool
- Good for parallel tasks

Example: Image processing

Subclassing Pattern

QThread

- Subclass QThread
- Re-implement the run() method
- Call start()
- Good for blocking tasks
- Good for independent tasks

Example: Encrypting data

Event Driven Pattern

QThread

- Move worker object to QThread
  - Creates thread affinity
- Call thread's start() method
- Worker's slots will be executed in its thread
- Good for event driven tasks

Example: Network manager

The Problem

- Two patterns of QThread use
  - Mixing subclassing and event driven patterns
  - Easy pitfall to stumble into
- Older documentation seemed to encourage this

"Oh I see, I need to subclass QThread and give it a bunch of slots!"

-Or-

"Just subclass QThread and move it to itself!"

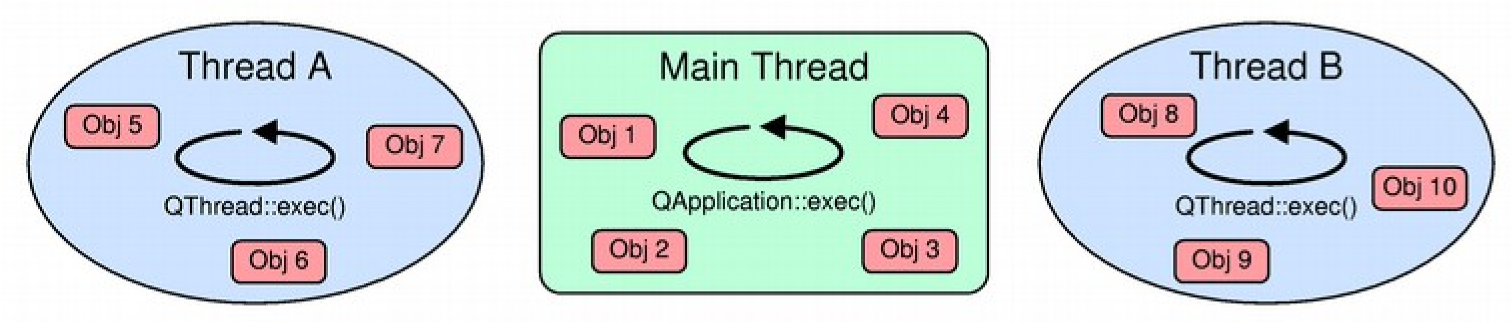
The Problem

What not to do:

```
MyThread::MyThread() : QThread()
{
    moveToThread(this);
    start();
}

void MyThread::run()
{
    connect(Manager, &Manager::newData,
            this, &MyThread::processData);
    ...
}
```

How QThread Works



First Rule of QThread*

- QThread is not a thread, QThread manages a thread
  - Threads are code execution, they are not objects. This is true for every framework.
  - With one exception, every QThread method runs outside of the thread of execution

“All of the functions in QThread were written and intended to be called from the creating thread, not the thread that QThread starts” - Bradley T. Hughes

*You are allowed to talk about QThread!

The Basics of QThread

- QThread manages one thread of execution
- The thread itself is defined by run()
  - Note that run() is a *protected* method
- Calling start() will create and start the thread
- QThread inherits from QObject
  - Support properties, events, signals and slots

- Several threading primitive classes
  - QMutex, QSemaphore, QWaitCondition, etc.

QThread Event Loop

- Default implementation of run()

```
void QThread::run()
{
    (void) exec();
}
```

- The only method that runs in the thread context
- Starts an event loop running in the thread

- The event loop is key to thread communication

Thread Affinity

- Objects can have affinity with a QThread
- Object is associated with the thread event loop
  - Events sent to an object are queued in the associated event loop
- Objects have affinity with the thread that created them
  - Can change affinity with QObject::moveToThread()
- QThreads don't have affinity with themselves!

Threaded Events

- Can use custom QEvents to communicate between QObjects in different threads
- Event loop of the receiver must be running
  - Event loop of the sender is optional
- Bad form to use shared data in the event object

```
void MainWindow::onActivity()
{
    ...
    ActivityEvent *event = new ActivityEvent(a, b);
    QApplication:postEvent(worker, event);
}
```

Signal/Slot Connections

- Qt::DirectConnection
  - Slots are called directly
  - Synchronous behavior
- Qt::QueuedConnection
  - Signals are serialized to an event
  - Asynchronous behavior
- Qt::AutoConnection (default)
  - If both objects have same thread affinity: Direct
  - If objects have different thread affinity: Queued

Cross Thread Signals/slots

- Default connection between objects of different thread affinity is Qt::QueuedConnection
  - Sender's signal is serialized into an event
  - Event is posted to the receiver's event queue
  - Event is deserialized and the slot is executed
- Communication between threads is easy!

p.s. This is why QThread self-affinity is usually wrong. It implies you want to send cross-thread signals to yourself.

Cross Thread Signals/Slots

- Cross thread signals are really events
  - The receiver needs a running event loop
  - The sender does NOT need an event loop
  - Signals are placed in the event queue
- All threads can emit signals regardless of pattern
  - Only threads with running event loops should have in-thread slots

Back to Threading Patterns

Let's look closer at the two threading patterns for QThread...

The Subclassing Pattern

Use when task...

- ... doesn't need an event loop
- ... doesn't have slots to be called
- ... can be defined within run()

  - Most classic threading problems
  - Standalone independent tasks

Subclassing Example

```
class WorkProducer : public QThread
{
public:
    WorkProducer();
protected:
    virtual void run();
    WorkItem produceWork();
...
};

class WorkConsumer : public QThread
{
public:
    WorkConsumer(int id);
protected:
    virtual void run();
    void consumeWork(const WorkItem &item);
...
};
```

Subclassing Example, cont.

```
void WorkProducer::run()
{
    while (true) {
        if (isInterruptionRequested()) {
            workcondition.wakeAll();
            return;
        }

        WorkItem item = produceWork();

        QMutexLocker locker(&queuelock);
        workqueue.enqueue(item);
        if (worknumwaiting > 0) {
            workcondition.wakeOne();
            worknumwaiting--;
        }
    }
}
```

Subclassing Example, cont.

```
void WorkConsumer::run()
{
    while (true) {
        if (isInterruptionRequested()) return;

        QMutexLocker locker(&queuelock);
        if (workqueue.isEmpty()) {
            worknumwaiting++;
            workcondition.wait(locker.mutex());
        } else {
            WorkItem item = workqueue.dequeue();
            locker.unlock();

            consumeWork(item);

            // send an inter thread signal to the GUI thread
            emit newValue(item.result());
        }
    }
}
```

Subclassing Example, cont.

```
WorkQueue::~WorkQueue() {
    // interrupt and wait for each thread
    producer->requestInterruption();
    producer->wait();
    producer->deleteLater();

    foreach (WorkConsumer *consumer, consumerlist) {
        consumer->requestInterruption();
        consumer->wait();
        consumer->deleteLater();
    }
}
```

Pitfalls of Subclassing

- Providing slots for the subclass
  - Since thread affinity has not changed, and there is no event loop, slots will be executed in the caller's thread.
- Calling moveToThread(this)
  - Frequently used ''solution'' to the pitfall above
  - Threads must never have affinity with themselves

The Event Driven Pattern

Use when task...

- ... is naturally event driven
- ... needs to receive communications
- ... does not have a single entry point
  - Inter-dependent tasks
  - “Manager” tasks

Event Driven Example

```
void Window::onStartClicked()
{
    QThread *thread = new QThread(this);
    mWorker = new Worker();
    mWorker->moveToThread(thread);

    connect(thread, &QThread::finished, mWorker, &Worker::deleteLater);
    connect(thread, &QThread::finished, thread, &QThread::deleteLater);

    connect(this, &Window::quit, thread, &QThread::quit);

    connect(this, &Window::newData, mWorker, &Worker::process);
    connect(mWorker, &Worker::status, this, &Window::processStatus);

    thread->start();
}
```

Event Driven Example, cont.

```
class Worker : public QObject
{
    Q_OBJECT
public:
    Worker(QObject *parent = 0);

public slots:
    void process(const QString &filename);

signals:
    void status(int);


...
};
```

Event Driven Example, cont.

```
void Worker::process(const QString &data)
{
    QFile file(filename);
    if (!file.open(QIODevice::ReadOnly) {
        emit status(PROCESS_ERROR);
        return;
    }

    int percent = 0;
    while (!file.atEnd()) {
        QByteArray line = file.readLine();
        // process line
        ...
        emit status(percent);
    }
}
```

Event Driven Example, cont.

```
void Window::closeEvent(QCloseEvent*)
{
    ...
    if (mWorker) {
        QThread *thread = mWorker->thread();

        if (thread->isRunning()) {
            thread->quit();
            thread->wait(250);
        }
    }
}
```

Pitfalls of Event Driven

- Does your task really need to be event driven?
- Event starvation
  - Compute intensive tasks take too long to return to event loop
- Fake event loop anti-pattern
  - An entry point never exits to the event loop

```
void Worker::start()
{
    while (true) {
        ...
    }
}
```

Event Driven + Subclassing

- Choice of pattern is not an either/or decision

```
MyThread::run()
{
    ... initialize objects
    ... setup workers
    ... handshaking protocols

    exec();

    ... cleanup thread resources
    ... wait on child threads
}
```

What About QThreadPool?

- QThreadPool is ideal for managing “fire and forget” tasks

However...

- Hard to communicate with the runnables
  - Requires either threading primitives...
  - Or requires multiple inheritance from QObject

What about QtConcurrent?

- Higher level threading API

However...

- Mostly limited to parallel computations
  - Requires knowledge of parallel algorithms
  - Requires knowledge of advanced C++ features

QThread!

Remains the workhorse of Qt threading

- General purpose threading solution
- Mid-level threading API
- QObject wholesome goodness

Summary

There's no one right way to use QThread

But there are two good ways to use QThread

Subclassing Pattern

Event Driven Pattern

Now You're Doing it Right!

Thank you!