Threads Events QObjects

From Qt Wiki

The article is almost done, but it needs a bit of polishing and some good examples. Any review or contribution is welcome! Discussion about this article happens in this thread (http://forum.qt.io/viewthread/2423/).

En Ar Bg De El Es Fa Fi Fr Hi Hu It Ja Kn Ko Ms Nl Pl Pt Ru Th Tr Uk Zh

Contents

Introduction

> You're doing it wrong. — Bradley T. Hughes

One of the most popular topics on the "#qt IRC channel":irc://irc.freenode.net/#qt is threading: many people join the channel and ask how they should solve their problem with some code running in a different thread.

Nine times out of ten, a quick inspection of their code shows that the biggest problem is the very fact they're using threads in the first place, and they're falling in one of the endless pitfalls of parallel programming.

The ease of creating and running threads in Qt, combined with some lack of knowledge about programming styles (especially asynchronous network programming, combined with Qt's signals and slots architecture) and/or habits developed when using other tookits or languages, usually leads to people shooting themselves in the foot. Moreover, threading support in Qt is a double-edged sword: while it makes it very simple for you to do multithread programming, it adds a certain number of features (especially when it comes to interaction with QObjects) you must be aware of.

The purpose of this document is **not** to teach you how to use threads, do proper locking, exploit parallelism, nor write scalable programs; there are many good books about these topics; for instance, take a look to the recommended reading list on this page (http://doc.qt.io/qt-4.8/threads.html). Instead, this small article is meant to be a guide to introduce users to threading in Qt 4, in order to avoid the most common pitfalls and help them to develop code that is at the same time more robust and with a better structure.

Prerequisites

> Think of it this way: threads are like salt, not like pasta.
>
> You like salt, I like salt, we all like salt. But we eat more pasta. — Larry McVoy

Not being a general-purpose introduction to (threads) programming, we expect you to have some previous knowledge about:

- C++ basics (though most suggestions do apply to other languages as well);
- Qt basics: QObjects, signals and slots, event handling;
- what a thread is and what the relationships are between threads, processes and the operating system;
- how to start and stop a thread, and wait for it to finish, under (at least) one major operating system;
- how to use mutexes, semaphores and wait conditions to create thread-safe/reentrant functions, data structures, classes.

In this document we'll follow the Qt naming conventions (http://doc.qt.io/qt-4.8/threads-reentrancy.html), which are:

- **Reentrant** A class is reentrant if it's safe to use its instances from more than one thread, provided that at most one thread is accessing the same instance at the same time. A function is reentrant if it's safe to invoke it from more than one thread at the same, provided that each invocation references unique data. In other words, this means that users of that class/function must *serialize* all accesses to instances/shared data by means of some *external locking mechanism*.
- **Thread-safe** A class is thread-safe if it's safe to use its instances from more than one thread at the same time. A function is thread-safe if it's safe to invoke it from more than one thread at the same time even if the invocations reference shared data.

Events and the event loop

Being an event-driven toolkit, events and event delivery play a central role in Qt architecture. In this article we'll not give a comprehensive coverage about this topic; we'll instead focus on some thread-related key concepts (see here (http://doc.qt.io/qt-4.8/eventsandfilters.html) and here (https://doc.qt.io/archives/qq/qq11-events.html) for more information about the Qt event system).

An **event** in Qt is an object which represents something interesting that happened; the main difference between an event and a signal is that events are *targeted* to a specific object in our application (which decides what to do with that event), while signals are emitted "in the wild". From a code point of view, all events are instances of some subclass of QEvent (http://doc.qt.io/qt-4.8/qevent.html), and all QObject-derived classes can override the QObject::event() virtual method in order to handle events targeted to their instances.

Events can be generated from both inside and outside the application; for instance:

- QKeyEvent and QMouseEvent objects represent some kind of keyboard and mouse interaction, and they come from the window manager;
- QTimerEvent objects are sent to a QObject when one of its timers fires, and they (usually) come from the operating system;
- QChildEvent objects are sent to a QObject when a child is added or removed, and they come from inside your Qt application.

The important thing about events is that they're not delivered as soon as they're generated; they're instead queued up in an **event queue** and sent sometime later. The dispatcher itself loops around the event queue and sends queued events to their target objects, and therefore it is called the **event loop**. Conceptually, this is how an event loop looks (see the Qt Quarterly article linked above):

```
while (is_active)
{
 while (!event_queue_is_empty)
 dispatch_next_event();

wait_for_more_events();
}
```

We enter Qt's main event loop by running QCoreApplication::exec(); this call blocks until QCoreApplication::exit() or QCoreApplication::quit() are called, terminating the loop.

The "wait_for_more_events()" function blocks (that is, it's not a busy wait) until some event is generated. If we think about it, all that can generate events at that point is some *external* source (dispatching for all internal events is now complete and there were no more pending events in the event queue to delivery). Therefore, the event loop can be woken up by:

- window manager activity (key/mouse presses, interaction with the windows, etc.);
- sockets activity (there's some data available to read, or a socket is writable without blocking, there's a new incoming connection, etc.);
- timers (i.e. a timer fired);
- events posted from other threads (see later).

In a UNIX-like system, window manager activity (i.e. X11) is notified to applications via sockets (Unix Domain or TCP/IP), since clients use them to communicate with the X server. If we decide to implement cross-thread event posting with an internal socketpair(2), all that is left is being woken up by activity on:

- sockets;
- timers;

which is exactly what the **select(2)** system call does: it watches over a set of descriptors for activity *and* it times out (with a configurable timeout) if there's no activity for a certain while. All Qt needs to do is converting what select returns into an object of the right QEvent subclass and queue it up in the event queue. Now you know what's inside an event loop :)

What requires a running event loop?

This isn't an exhaustive list, but if you have the overall picture, you should be able to guess which classes require a running event loop.

- **Widgets painting and interaction**: QWidget::paintEvent() will be called when delivering QPaintEvent objects, which are generated both by calling QWidget::update() (i.e. internally) or by the window manager (for instance, because a hidden window was shown). The same thing holds for all kinds of interaction (keyboard, mouse, etc.): the corresponding events will require an event loop to be dispatched.
- **Timers**: long story short, they're fired when select(2) or similar calls time out, therefore you need to let Qt do those calls for you by returning to the event loop.
- **Networking**: all low-level Qt networking classes (QTcpSocket, QUdpSocket, QTcpServer, etc.) are asynchronous by design. When you call read(), they just return already available data; when you call write(), they schedule the writing for later. It's only when you return to the event loop the actual reading/writing takes place. Notice that they do offer synchronous methods (the waitFor* family of methods), but their use is discouraged because they block the event loop while waiting. High-level classes, like QNetworkAccessManager, simply do not offer any synchronous API and require an event loop.

Blocking the event loop

Before discussing why **you should never ever block the event loop**, let's try to figure out what this "blocking" means. Suppose you have a Button widget which emits a signal when clicked; connected to this signal there's a slot of our Worker object, which does a lot of work. After you click the button, the stack trace will look like this (the stack grows downwards):

1. main(int, char **)**
2. QApplication::exec()
3. […]
4. QWidget::event(QEvent **)**
5. Button::mousePressEvent(QMouseEvent**)**
6. Button::clicked()
7. […]
8. Worker::doWork()

In main() we started the event loop, as usual, by calling QApplication::exec() (line 2). The window manager sent us the mouse click, which was picked up by the Qt kernel, converted in a QMouseEvent and sent to our widget's event() method (line 4) by QApplication::notify() (not shown here). Since Button didn't override event(), the base class implementation (QWidget) is called. QWidget::event() detects the event is actually a mouse click and calls the specialized event handler, that is, Button::mousePressEvent() (line 5). We overrode this method to emit the Button::clicked() signal (line 6), which invokes the Worker::doWork slot of our worker object (line 7).

While the worker is busy working, what's the event loop doing? You should've guessed it: nothing! It dispatched the mouse press event and it's blocked waiting for the event handler to return. We managed to **block the event loop**, which means that no event is sent any more, until we return from the doWork() slot, up the stack, to the event loop, and let it process pending events.

With the event delivery stuck, **widgets won't update themselves** (QPaintEvent objects will sit in the queue), **no further interaction with widgets is possible** (for the same reason),

timers won't fire and **networking communications will slow down and stop**. Moreover, many window managers will detect that your application is not handling events any more and **tell the user that your application isn't responding**. That's why is so important to quickly react to events and return to the event loop as soon as possible!

Forcing event dispatching

So, what do we do if we have a long task to run and don't want to block the event loop? One possible answer is to move the task into another thread: in the next sections we'll see how to do that. We also have the option to manually force the event loop to run, by (repeatedly) calling QCoreApplication::processEvents() inside our blocking task. QCoreApplication::processEvents() will process all the events in the event queue and return to the caller.

Another available option we can use to forcibly reenter the event loop is the QEventLoop (http://doc.qt.io/qt-4.8/qeventloop.html) class. By calling QEventLoop::exec() we reenter the event loop, and we can connect signals to the QEventLoop::quit() slot to make it quit. For instance:

```
QNetworkAccessManager qnam;
QNetworkReply *reply = qnam.get(QNetworkRequest(QUrl(…)));
QEventLoop loop;
QObject::connect(reply, SIGNAL (finished()), &loop, SLOT (quit()));
loop.exec();
/* reply has finished, use it */
```

QNetworkReply doesn't offer a blocking API and requires an event loop to be running. We enter a local QEventLoop, and when the reply has finished, the local event loop quits.

Be very careful when reentering the event loop "by other paths": it can lead to unwanted recursions! Let's go back to the Button example. If we call QCoreApplication::processEvents() inside the doWork() slot, and the user clicks again on the button, the doWork() slot will be invoked **again**:

1. main(int, char**)**
2. QApplication::exec()
3. […]
4. QWidget::event(QEvent **)**
5. Button::mousePressEvent(QMouseEvent**)**
6. Button::clicked()
7. […]
8. Worker::doWork() // **first, inner invocation**
9. QCoreApplication::processEvents() // **we manually dispatch events and…**
10. […]
11. QWidget::event(QEvent *) // **another mouse click is sent to the Button…**
12. Button::mousePressEvent(QMouseEvent *)
13. Button::clicked() // **which emits clicked() again…**
14. […]
15. Worker::doWork() // **DANG! we've recursed into our slot.**

A quick and easy workaround for this is passing QEventLoop::ExcludeUserInputEvents to QCoreApplication::processEvents(), which tells the event loop to not dispatch any user input event (the events will simply stay in the queue).

Luckily, the same thing does **not** apply to **deletion events** (the ones posted in the event queue by QObject::deleteLater()). In fact, they are handled in a special way by Qt, and are processed only if the running event loop has a smaller degree of "nesting" (w.r.t. event loops) than the one where deleteLater was called. For instance:

```
QObject *object = new QObject;
object->deleteLater();
QDialog dialog;
dialog.exec();
```

will not make object a dangling pointer (the event loop entered by QDialog::exec() is more nested than the deleteLater call). The same thing applies to local event loops started with QEventLoop. The only notable exception I've found to this rule (as of Qt 4.7.3) is that if deleteLater is called when NO event loop is running, then the first event loop entered will pick up the event and delete the object. This is pretty much reasonable, since Qt does not know about any "outer" loop that will eventually perform the deletion, and therefore deletes the object immediately.

.

Qt thread classes

> A computer is a state machine. Threads are for people who can't program state machines.
>
> — Alan Cox

Qt has had thread support for many years (Qt 2.2, released on 22 Sept 2000, introduced the QThread class.), and with the 4.0 release thread support is enabled by default on all supported platforms (although it can be turned off, see here (http://doc.qt.io/qt-4.8/fine-tuning-features.html) for more details). Qt now offers several classes for dealing with threads; let's start with an overview.

QThread

QThread (http://doc.qt.io/qt-4.8/qthread.html) is the central, low-level class for thread support in Qt. A QThread object represents one thread of execution. Due to the cross-platform nature of Qt, QThread manages to hide all the platform-specific code that is needed to use threads on different operating systems.

In order to use a QThread to run some code in a thread, we can subclass it and override the QThread::run() method:

```
class Thread : public QThread {
protected:
 void run() {
 /* your thread implementation goes here */
 }
};
```

Then we can use

```
Thread *t = new Thread;
t->start(); // start(), not run()!
```

to actually start the new thread. Note that since Qt 4.4 QThread is no longer an abstract class; now the virtual method QThread::run() instead simply calls QThread::exec();, which starts the *thread's event loop* (more info on this later).

QRunnable and QThreadPool

QRunnable (http://doc.qt.io/qt-4.8/qrunnable.html) is a lightweight abstract class that can be used to start a task in another thread in a "run and forget" fashion. In order to do so, all we have to do is subclass QRunnable and implement its run() pure virtual method:

```
class Task : public QRunnable {
public:
 void run() {
 /* your runnable implementation goes here */
 }
};
```

To actually run a QRunnable object we use the QThreadPool (http://doc.qt.io/qt-4.8/qthreadpool.html) class, which manages a pool of threads. By calling QThreadPool::start(runnable) we put a QRunnable in a QThreadPool's runqueue; as soon as a thread becomes available, the QRunnable will be picked up and run into that thread. All Qt applications have a global thread pool available by calling QThreadPool::globalInstance(), but one can always create a private QThreadPool instance and manage it explicitely.

Notice that, not being a QObject, QRunnable has no built-in means of explicitely communicating something to other components; you have to code that by hand, using low-level threading primitives (like a mutex-guarded queue for collecting results, etc.).

QtConcurrent

QtConcurrent (http://doc.qt.io/qt-4.8/threads-qtconcurrent.html) is a higher-level API, built on top of QThreadPool, useful to deal with the most common parallel computation patterns: map) (http://en.wikipedia.org/wiki/Map_(higher-order_function), reduce) (http://en.wikipedia.org/wiki/Fold_(higher-order_function), and filter) (http://en.wikipedia.org/wiki/Filter_(higher-order_function) ; it also offers a QtConcurrent::run() method that can be used to easily run a function in another thread.

Unlike QThread and QRunnable, QtConcurrent does not require us to use low-level synchronization primitives: all QtConcurrent methods instead return a QFuture (http://doc.qt.io/qt-4.8/qfuture.html) object, which can be used to query the computation status (its progress), to pause/resume/cancel the computation, and that also contains its *results*. The QFutureWatcher (http://doc.qt.io/qt-4.8/qfuturewatcher.html) class can be used to monitor a QFuture progress and interact with it by means of signals and slots (notice that QFuture, being a value-based class, doesn't inherit QObject).

Feature comparison

	QThread	**QRunnable**	**QtConcurrent**[1]
High level API	✘	✘	✔
Job-oriented	✘	✔	✔

Builtin support for pause/resume/cancel	✘	✘	✔
Can run at a different priority	✔	✘	✘
Can run an event loop	✔	✘	✘

Threads and QObjects

Per-thread event loop

So far we've always talked about "*the* event loop", taking somehow per granted that there's only one event loop in a Qt application. This is not the case: QThread objects can start thread-local event loops running in the threads they represent. Therefore, we say that the **main event loop** is the one created by the thread which invoked main(), and started with QCoreApplication::exec() (which *must* be called from that thread). This is also called the **GUI thread**, because it's the only thread in which GUI-related operations are allowed. A QThread local event loop can be started instead by calling QThread::exec() (inside its run() method):

```
class Thread : public QThread {
protected:
 void run() {
 /* … initialize … */

exec();'''
 }
};
```

As we mentioned before, since Qt 4.4 QThread::run() is no longer a pure virtual method; instead, it calls QThread::exec(). Exactly like QCoreApplication, QThread has also the QThread::quit() and QThread::exit() methods to stop the event loop.

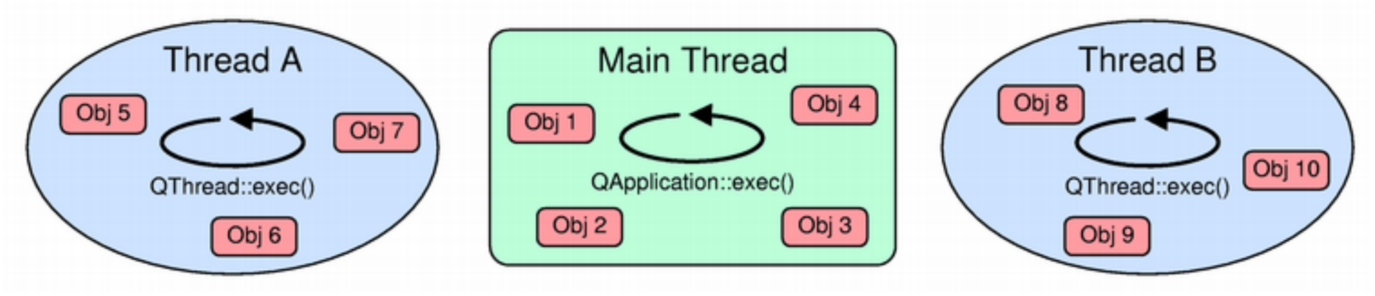
A thread event loop delivers events for all QObjects that are **living** in that thread; this includes, by default, all objects that are created into that thread, or that were moved to that thread (more info about this later). We also say that the **thread affinity** of a QObject is a certain thread, meaning that the object is living in that thread. This applies to objects which are built in the constructor of a QThread object:

```
class MyThread : public QThread
{
public:
 MyThread()
 {
 otherObj = new QObject;
 }

private:
 QObject obj;
 QObject *otherObj;
 QScopedPointer<QObject> yetAnotherObj;
};
```

What's the thread affinity of obj, otherObj, yetAnotherObj after we create a MyThread object? We must look at the thread that created them: it's the thread that ran the MyThread constructor. Therefore, all three objects are **not** living in the MyThread thread, but in the thread that created the MyThread instance (which, by the way, is where the instance is living as well).

We can query anytime the thread affinity of a QObject by calling QObject::thread(). Notice that QObjects created before a QCoreApplication object have **no thread affinity**, and therefore no event dispatching will be done for them (in other words, QCoreApplication builds up the QThread object that represents the main thread).



We can use the thread-safe QCoreApplication::postEvent() method for posting an event for a certain object. This will enqueue the event in the event loop of the thread the object is living in; therefore, the event will not be dispatched unless that thread has a running event loop.

It is very important to understand that QObject and all of its subclasses **are not thread-safe** (although they can be reentrant); therefore, you can not access a QObject from more than one thread at the same time, unless you serialize all accesses to the object's internal data (for instance, by protecting it with a mutex). Remember that the object may be handling events dispatched by the event loop of the thread it is living in while you're accessing it from another thread! For the same reason, you can't delete a QObject from another thread, but you must use QObject::deleteLater(), which will post an event that will ultimately cause its deletion by the thread the object is living in.

Moreover, QWidget and all of its subclasses, along with other GUI-related classes (even not QObject-based, like QPixmap) **are not reentrant** either: they can be used exclusively from the GUI thread.

We can change a QObject's affinity by calling QObject::moveToThread(); this will change the affinity of the object and of its children. Since QObject is not thread-safe, we must use it from the thread the object is living in; that is, you can only **push** objects from the thread they're living in to other threads, and not **pull** them or move them around from other threads. Moreover, Qt requires that the child of a QObject must live in the same thread where the parent is living. This implies that:

- you can't use QObject::moveToThread() on a object which has a parent;
- you must not create objects in a QThread using the QThread object itself as their parent:

```
class Thread : public QThread {
 void run() {
 QObject *obj = new QObject(this); // WRONG[[Image:|Image:]]!
 }
};
```

This is because the **QThread object is living in another thread**, namely, the one in which it was created.

Qt also requires that all objects living in a thread are deleted before the QThread object that represents the thread is destroyed; this can be easily done by creating all the objects living in that thread on the QThread::run() method's stack.

Signals and slots across threads

Given these premises, how do we call methods on QObjects living in other threads? Qt offers a very nice and clean solution: we post an event in that thread's event queue, and the handling of that event will consist in invoking the method we're interested in (this of course requires that the thread has a running event loop). This facility is built around the method introspection provided by moc: therefore, only signals, slots and methods marked with the Q_INVOKABLE macro are invokable from other threads.

The QMetaObject::invokeMethod() static method does all the work for us:

```
QMetaObject::invokeMethod(object, "methodName",
 Qt::QueuedConnection,
 Q_ARG(type1, arg1),
 Q_ARG(type2, arg2));
```

Notice that since the arguments need to be copied in the event which is built behind the scenes, their types need to provide a public constructor, a public destructor and a public copy constructor, and must be registered within Qt type system by using the qRegisterMetaType() function.

Signals and slots across threads work in a similar way. When we connect a signal to a slot, the fifth argument of QObject::connect is used to specify the connection type:

- a **direct connection** means that the slot is always invoked directly by the thread the signal is emitted from;
- a **queued connection** means that an event is posted in the event queue of the thread the receiver is living in, which will be picked up by the event loop and will cause the slot invocation sometime later;
- a **blocking queued connection** is like a queued connection, but the sender thread blocks until the event is picked up by the event loop of the thread the receiver is living in, the slot is invoked, and it returns;
- an **automatic connection** (*the default*) means that if the thread the receiver is living in is the same as the current thread, a direct connection is used; otherwise, a queued connection is used.

In every case, keep in mind *the thread the emitting object is living in* has no importance at all! In case of an automatic connection, Qt looks at the thread that invoked the signal and compares it with the thread the receiver is living in to determine which connection type it has to use. In particular, the current Qt documentation (http://doc.qt.io/qt-5/threads-qobject.html) **is simply wrong** when it states:

Auto Connection (default) The behavior is the same as the Direct Connection, if the emitter and receiver are in the same thread. The behavior is the same as the Queued Connection, if the emitter and receiver are in different threads.

because the emitter object's thread affinity does not matter. For instance:

```
class Thread : public QThread
{
 Q_OBJECT

signals:
 void aSignal();

protected:
 void run() {
 emit aSignal();
 }
};
```

```
/* … */
Thread thread;
Object obj;
QObject::connect(&thread, SIGNAL (aSignal()), &obj, SLOT (aSlot()));
thread.start();
```

The signal aSignal() will be emitted by the new thread (represented by the Thread object); since it is not the thread the Object object is living in (which, by the way, **is the same thread the Thread object is living in**, just to stress that the sender's thread affinity doesn't matter), a **queued connection** will be used.

Another common pitfall is the following one:

```
class Thread : public QThread
{
 Q_OBJECT

slots:
 void aSlot() {
 /* … */
 }

protected:
 void run() {
 /''' … */
 }
};

/''' … */
Thread thread;
Object obj;
QObject::connect(&obj, SIGNAL (aSignal()), &thread, SLOT (aSlot()));
thread.start();
obj.emitSignal();
```

When "obj" emits its aSignal() signal, which kind of connection will be used? You should've guessed it: a **direct connection**. That's because the Thread object is living in the thread that emits the signal. In the aSlot() slot we could then access some Thread's member variable while they're being accessed by the run() method, which is running concurrently: this is the perfect recipe for disaster.

Yet another example, probably the *most important* one:

```
class Thread : public QThread
{
 Q_OBJECT

slots:
 void aSlot() {
 /* … */
 }

protected:
 void run() {
 QObject *obj = new Object;
 connect(obj, SIGNAL (aSignal()), this, SLOT (aSlot()));
 /* … */
 }
};
```

In this case a **queued connection** is used, therefore you're required to run an event loop in the thread the Thread object is living in.

A solution you'll often found in forums, blog posts etc. is to add a moveToThread(this) to the Thread constructor:

```
class Thread : public QThread {
 Q_OBJECT
public:
 Thread() {
 moveToThread(this); // WRONG
 }

/* … */
};
```

which indeed *will work* (because now the affinity of the Thread object changed), but it's a very bad design. What's wrong here is that we're misunderstanding the purpose of a thread object (the QThread subclass): *QThread objects are not threads*; they're control objects around a thread, therefore meant to be used from another thread (usually, the one they're living in).

A good way to achieve the same result is splitting the "working" part from the "controller" part, that is, writing a QObject subclass and using QObject::moveToThread() to change its affinity:

```
class Worker : public QObject
{
 Q_OBJECT

public slots:
 void doWork() {
 /* … */
 }
};

/* … */
QThread *thread = new QThread;
Worker *worker = new Worker;
connect(obj, SIGNAL (workReady()), worker, SLOT (doWork()));
worker->moveToThread(thread);
thread->start();
```

DOs and DON'Ts

You can...

... add signals to a QThread subclass. It's perfectly safe and they'll do the "right thing" (see above; the sender's thread affinity does not matter).

You shouldn't ...

- ... use moveToThread(this).
- ... force the connection type: this usually means that you're doing something wrong, like mixing the control interface of QThread with the program logic (which should stay in a separate object which lives in that thread).
- ... add slots to a QThread subclass: they'll be invoked from the "wrong" thread, that is, not the one the QThread object is managing, but the one that object is living in, forcing you to specify a direct connection and/or to use moveToThread(this).
- ... use QThread::terminate.

You must not...

- ... quit your program when threads are still running. Use QThread::wait to wait for their termination.

- ... destroy a QThread while the thread that it's managing is still running. If you want some kind of "self-destruction", you can connect the finished() signal with the deleteLater() slot.

When should I use threads?

When you have to use a blocking API

If you need to use a library or other code that doesn't offer a non-blocking API (by means of signals and slots, or events, or callbacks, etc.), then the only viable solution in order to avoid freezing the event loop is to spawn a process or a thread. Since creating a new worker process, having it doing the job and communicating back the results is definetely harder and more expensive than just starting a thread, the latter is the most common choice.

A good example of such an API is **address resolution** (just to show you that we're not talking about 3rd-party crappy API. This is something included in every C library out there), which is the process of taking an host name and converting it into an address. This process involves a query to a (usually remote) system — the Domain Name System, or DNS. While, usually, the response is almost instantaneous, the remote servers might fail, some packet might get lost, the network connection might break, and so on; in short, it might take dozens of seconds before we get a reply from our query.

The only standard API available on UNIX systems is *blocking* (not only the old-fashioned gethostbyname(3), but also the newer and better getservbyname(3) and getaddrinfo(3)). QHostInfo (http://doc.qt.io/qt-4.8/qhostinfo.html), the Qt class that handles host name lookups, uses a QThreadPool to enable the queries to run in the background (see here (http://code.qt.io/cgit/qt/qt.git/tree/src/network/kernel/qhostinfo.cpp) ; if thread support is turned off, it switches back to a blocking API).

Other simple examples are **image loading** and **scaling**. QImageReader (http://doc.qt.io/qt-4.8/qimagereader.html) and QImage (http://doc.qt.io/qt-4.8/qimage.html) only offer blocking methods to read an image from a device, or to scale an image to a different resolution. If you're dealing with very large images, these processes can take up to (tens of) seconds.

When you want to scale with the number of CPUs

Threads allow your program to take advantage from multiprocessor systems. Since each thread is scheduled independently by the operating system, if your application is running on such a machine the scheduler is likely to run each thread on a different processor **at the same time**.

For instance, consider an application that generates thumbnails from a set of images. A **thread farm** of *n* threads (that is, a thread pool with a fixed number of threads), one per each CPU available in the system (see also QThread::idealThreadCount()), can spread the work of scaling down the images into thumbnails on all the threads, effectively gaining an almost linear speedup with the number of the processors (for simplicity's sake, we consider the CPU being the bottleneck).

When you don't want to be possibly blocked by others

MEH. BETTER START WITH AN EXAMPLE.

This is quite an advanced topic, so feel free to skip it for now. A nice example of this use case comes from QNetworkAccessManager usage inside WebKit. WebKit is a modern browser engine, that is, a set of classes to lay out and display web pages. The Qt widget that uses WebKit is QWebView.

QNetworkAccessManager is a Qt class that deals with HTTP requests and responses for all purposes, we can consider it to be the networking engine of a web browser. Before Qt 4.8, it does not make use of any worker threads; all networking is handled in the same thread QNetworkAccessManager and its QNetworkReplys are living in.

While not using threads for networking is a very good idea, it has also a major drawback: if you don't read data from the socket as soon as possible, the kernel buffers will fill up, packets will begin to be dropped, and the transfer speed will decrease considerably.

Socket activity (i.e., availability of some data to read from a socket) is managed by Qt's event loop. Blocking the event loop will therefore lead to a loss of transfer performance, because nobody will be notified that there are data to read (and thus nobody will read them).

But what could block the event loop? The sad answer is: WebKit itself! As soon as some data are received, WebKit uses them to start laying out the web page. Unfortunately, the layout process is quite complicated and expensive, therefore it blocks the event loop for a (short) while, enough to impact on ongoing transfers (broadband connections play their role here, filling up kernel buffers in a small fraction of second).

To sum it up, what happens is something like this:

- WebKit issues a request;
- some data from the reply begin to arrive;
- WebKit starts to lay out the web page using the incoming data, blocking the event loop;
- without a running event loop, data are received by the OS, but not read from QNetworkAccessManager sockets;
- kernel buffers will fill up, and the transfer will slow down.

The overall page loading time is therefore worsened by this self-induced transfer slowness.

Notice that since QNetworkAccessManagers and QNetworkReplys are QObjects, they're not thread-safe, therefore you can't just move them to another thread and continue using them from your thread, because they may be accessed at the same time by two threads: yours and the one they're living in, due to events that will be dispatched to them by the latter thread's event loop.

As of Qt 4.8, QNetworkAccessManager now handles HTTP requests in a separate thread by default, so the result of unresponsive GUI and OS buffers filling up too quickly should be cured.

When shouldn't I use threads?

> If you think you need threads then your processes are too fat.
>
> — Rob Pike

Timers

This is perhaps the worst form of thread abuse. If we have to invoke a method repeatedly (for instance, every second), many people end up with something like this:

```
// VERY WRONG
while (condition) {
 doWork();
 sleep(1); // this is sleep(3) from the C library
}
```

Then they figure out that this is **blocking the event loop**, therefore decide to bring in threads:

```
// WRONG
class Thread : public QThread {
protected:
 void run() {
 while (condition) {
 // notice that "condition" may also need volatiness and mutex protection
 // if we modify it from other threads (!)
 doWork();
 sleep(1); // this is QThread::sleep()
 }
 }
};
```

A much **better and simpler way** of achieving the same result is simply using timers, i.e. a QTimer (http://doc.qt.io/qt-4.8/qtimer.html) object with a 1s timeout, and make the doWork() method a slot:

```
class Worker : public QObject
{
 Q_OBJECT

public:
 Worker() {
 connect(&timer, SIGNAL (timeout()), this, SLOT (doWork()));
 timer.start(1000);
 }

private slots:
 void doWork() {
 /* … */
 }

private:
 QTimer timer;
};
```

All we need is a running event loop, then the doWork() method will be invoked each second.

Networking / state machines

A very common design pattern when dealing with network operations is the following one:

```
socket->connect(host);
socket->waitForConnected();

data = getData();
socket->write(data);
socket->waitForBytesWritten();

socket->waitForReadyRead();
socket->read(response);
```

```
reply = process(response);

socket->write(reply);
socket->waitForBytesWritten();
/* … and so on … */
```

Needless to say, the various waitFor**() calls block the caller without returning to the event loop, freezing the UI and so on. Notice that the above snippet does not take into account any error handling, otherwise it would have been even more cumbersome. What is very wrong in this design is that we're forgetting that** networking is asynchronous by design**, and if we build a synchronous processing around we're shooting ourselves in the foot. To solve this problem, many people simple move this code into a different thread.**

Another more abstract example:

```
result = process_one_thing();

if (result->something())
 process_this();
else
 process_that();

wait_for_user_input();
input = read_user_input();
process_user_input(input);
/* … */
```

Which has more or less the same pitfalls of the networking example.

Let's take a step back and consider from an higher point of view what we're building here: we want to create a **state machine** that reacts on inputs of some sort and acts consequently. For instance, with the networking example, we might want to build something like this:

- Idle → Connecting (when calling connectToHost());
- Connecting → Connected (when connected() is emitted);
- Connected → LoginDataSent (when we send the login data to the server);
- LoginDataSent → LoggedIn (the server replied with an ACK)
- LoginDataSent → LoginError (the server replied with a NACK)

and so forth.

Now, there are several ways to build a state machine (and Qt even offers a class for that: QStateMachine (http://doc.qt.io/qt-5/qstatemachine.html)), the simplest one being an enum (i.e. an integer) used to remember the current state. We can rewrite the above snippets like this:

```
class Object : public QObject
{
 Q_OBJECT

enum State {
 State1, State2, State3 /* and so on */
 };

 State state;

public:
 Object() : state(State1)
 {
 connect(source, SIGNAL (ready()), this, SLOT (doWork()));
 }

private slots:
 void doWork() {
 switch (state) {
 case State1:
 /* … */
 state = State2;
 break;
 case State2:
 /* … */
 state = State3;
 break;
 /* etc. */
 }
 }
};
```

What the "source" object and its "ready()" signal are? Exactly what we want them to be: for instance, in the networking example, we might want to connect the socket's QAbstractSocket::connected() and the QIODevice::readyRead() signals to our slot. Of course, we can also easily add more slots if that suits better in our case (like a slot to manage error situations, which are notified by the QAbstractSocket::error() signal). This is a true asynchronous, signal-driven design!

Jobs splittable in chunks

Suppose that we have a long computation which can't be easily moved to another thread (or that it can't be moved at all, because for instance it *must run* in the GUI thread). If **we can split the computation in small chunks**, we can return to the event loop, let it dispatch events, and make it invoke the method that processes the next chunk. This can be easily done if we remember how queued connections are implemented: an event is posted in the event loop of the thread the receiver object is living in; when the event is delivered, the corresponding slot is invoked.

We can use QMetaObject::invokeMethod() to achieve the same result by specifying Qt::QueuedConnection as the type of the invocation; this just requires the method to be invokable, therefore it must be either a slot or marked with the Q_INVOKABLE macro. If we also want to pass parameters to the method, they need to be registered within the Qt metatype system using qRegisterMetaType(). The following snippet shows this pattern:

```
class Worker : public QObject
{
 Q_OBJECT
public slots:
 void startProcessing()
 {
 processItem(0);
 }

void processItem(int index)
 {
 /* process items[index] … */

 if (index < numberOfItems)
 QMetaObject::invokeMethod(this,
 "processItem",
 Qt::QueuedConnection,
 Q_ARG(int, index + 1));

 }
};
```

Since there are no threads involved, it's easy to pause/resume/cancel such a computation and collect the results back.

References

Bradley T. Hughes: You’re doing it wrong… (http://blog.qt.io/blog/2010/06/17/youre-doing-it-wrong/), Qt Labs blogs, 2010-06-17

- Bradley T. Hughes: Threading without the headache (http://blog.qt.io/blog/2006/12/04/threading-without-the-headache/), Qt Labs blogs, 2006-12-04

1. ↑ Except QtConcurrent::run, which is implemented using QRunnable and therefore shares its pros and cons.