PyQt6 Tutorial

Multithreading PyQt6 applications with QThreadPool

Torres Colorado Juan Daniel

Polytechnic University of Victoria

May-August 2024

Background (1)

Applications based on Qt (like most GUI applications) are *event based.* This means that execution is driven in response to user interaction, signals and timers. In an event-driven application, clicking on a button creates an *event* which your application subsequently *handles* to produce some expected output. Events are pushed onto and taken off an event queue and processed sequentially.

```
app = QApplication([])
window = MainWindow()
app.exec()
```

The event loop is started by calling .exec() on your QApplication object and runs within the same thread as your Python code. The thread which runs this event loop — commonly referred to as the *GUI thread* — also handles all window communication with the host operating system.

Background (2)

By default, any execution triggered by the event loop will also run synchronously within this thread. In practise this means that any time your PyQt application spends *doing something* in your code, window communication and GUI interaction are frozen.

If what you're doing is simple, and returns control to the GUI loop quickly, this freeze will be imperceptible to the user. However, if you need to perform longer-running tasks, for example opening/writing a large file, downloading some data, or rendering some complex image, there are going to be problems. To your user the application will appear to be unresponsive (because it is). Because your app is no longer communicating with the OS, on MacOS X if you click on your app you will see the spinning wheel of death. And, *nobody* wants that.

The solution is simple: get your work out of the *GUI thread* (and into another thread). PyQt (via Qt) provides an straightforward interface to do exactly that.

Preparation (1)

The following code will work with both Python 2.7 and Python 3.

To demonstrate multi-threaded execution we need an application to work with. Below is a minimal stub application for PyQt which will allow us to demonstrate multi-threading, and see the outcome in action. Simply copy and paste this into a new file, and save it with an appropriate filename like multithread.py. The remainder of the code will be added to this file (there is also a complete working example at the bottom if you're impatient).

multithread.py (Part I)

```python
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *

import time

class MainWindow(QMainWindow):


    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)
```

Preparation (2)

multithread.py (Part II)

```python

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def oh_no(self):
        time.sleep(5)

    def recurring_timer(self):
        self.counter +=1
        self.l.setText("Counter: %d" % self.counter)


app = QApplication([])
window = MainWindow()
app.exec_()
```

Run the file as for any other Python application:

```
python3 multithread.py
```

You should see a demonstration window with a number counting upwards. This a generated by a simple recurring time, firing once per second. Think of this as our *event loop indicator*, a simple way to let us known that out application is ticking over normally. There is also a button with the word "**DANGER!**". Push it.

Preparation (3)

You'll notice that each time you push the button the counter stops ticking and your application freezes entirely.

The dumb approach (1)

Don't do this. Ever.

What appears as a *frozen* interface is the main Qt event loop being blocked from processing (and responding to) window events. Your clicks on the window as still registered by the host OS and sent to your application, but because it's sat in your big ol' lump of code (time.sleep), it can't accept or react to them. They have to wait until your code passes control back to Qt.

The simplest, and perhaps most logical, way to get around this is to accept events from within your code. This allows Qt to continue to respond to the host OS and your application will stay responsive. You can do this easily by using the static .processEvents() function on the QApplication class. Simply add a line like the following, somewhere in your long-running code block:

```
QApplication.processEvents()
```

The dumb approach (2)

For example *long running* code time.sleep we could break that down into 5x 1-second sleeps and insert the .processEvents in between. The code for this would be:

```python
def oh_no(self):
    for n in range(5):
        QApplication.processEvents()
        time.sleep(1)
```

Now when you push the button your code is entered as before. However, now QApplication.processEvents() intermittently passes control back to Qt, and allows it to respond to events as normal. Qt will then accept events *and handle* them before returning to run the remainder of your code.

This works, but it’s horrible for a couple of reasons.

The dumb approach (3)

Firstly, when you pass control back to Qt, your code is no longer running. This means that whatever long-running thing you're trying to do will take *longer*. That is definitely not what you want.

Secondly, processing events outside the main event loop (app.exec_()) causes your application to branch off into handling code (e.g. for triggered slots, or events) while within your loop. If your code depends on/responds to external state this can cause undefined behavior. The code below demonstrates this in action:

multithread.py (Part I)

```python
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *

import time

class MainWindow(QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        c = QPushButton("?")
        c.pressed.connect(self.change_message)

        layout.addWidget(self.l)
        layout.addWidget(b)

```

The dumb approach (4)

multithread.py (Part II)

```
        layout.addWidget(c)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

    def change_message(self):
        self.message = "OH NO"

    def oh_no(self):
        self.message = "Pressed"

        for n in range(100):
            time.sleep(0.1)
            self.l.setText(self.message)
            QApplication.processEvents()


app = QApplication([])
window = MainWindow()
app.exec_()

```

If you run this code you'll see the counter as before. Pressing "DANGER!" will change the displayed text to "Pressed", as defined at the entry point to the oh_no function. However, if you press the "?" button while oh_no is still running you'll see that the message changes. State is being changed from outside your loop.

This is a toy example. However, if you have multiple long-running processes within your application, with each calling QApplication.processEvents() to keep things ticking, your application behaviour can be unpredictable.

Threads and Processes (1)

If you take a step back and think about what you want to happen in your application, it can probably be summed up with "stuff to happen at the same time as other stuff happens".

There are two main approaches to running independent tasks within a PyQt application: *threads* and *processes.*

Threads share the same memory space, so are quick to start up and consume minimal resources. The shared memory makes it trivial to pass data between threads, however reading/writing memory from different threads can lead to race conditions or segfaults. In a Python GUI there is the added issue that multiple threads are bound by the same Global Interpreter Lock (GIL) — meaning non-GIL-releasing Python code can only execute in one thread at a time. However, this is not a major issue with PyQt where most of the time is spent outside of Python.

Threads and Processes (2)

Processes use separate memory space (and an entirely separate Python interpreter). This side-steps any potential problems with the GIL, but at the cost of slower start-up times, larger memory overhead and complexity in sending/receiving data.

For simplicity's sake it usually makes sense to use threads, unless you have a good reason to use processes (see caveats later). Subprocesses in Qt are better suited to running and communicating with external programs.

There is nothing stopping you using pure-Python threading or process-based approaches within your PyQt application.

QRunnable and the QThreadPool (1)

Do this.

Qt provides a very simple interface for running jobs in other threads, which is exposed nicely in PyQt. This is built around two classes: QRunnable and QThreadPool. The former is the container for the work you want to perform, while the latter is the method by which you pass that work to alternate threads.

The neat thing about using QThreadPool is that it handles queuing and execution of workers for you. Other than queuing up jobs and retreiving the results there is not very much to do at all.

To define a custom QRunnable you can subclass the base QRunnable class, then place the code you wish you execute within the run() method. The following is an implementation of our long running time.sleep job as a QRunnable.

QRunnable and the QThreadPool (2)

Add the following code to multithread.py, above the MainWindow class definition.

multithread.py

```python
class Worker(QRunnable):
    '''
    Worker thread
    '''

    @pyqtSlot()
    def run(self):
        '''
        Your code goes in this function
        '''
        print("Thread start")
        time.sleep(5)
        print("Thread complete")
```

Executing our function in another thread is simply a matter of creating an instance of the Worker and then pass it to our QThreadPool instance and it will be executed automatically.

Next add the following within the __init__ block, to set up our thread pool.

```python
self.threadpool = QThreadPool()
print("Multithreading with maximum %d \
 threads" % self.threadpool.maxThreadCount()
)
```

QRunnable and the QThreadPool (3)

Finally, add the following lines to our oh_no function.

```python
def oh_no(self):
    worker = Worker()
    self.threadpool.start(worker)
```

Now, clicking on the button will create a worker to handle the (long-running) process and spin that off into another thread via thread pool. If there are not enough threads available to process incoming workers, they'll be queued and executed in order at a later time.

Try it out and you'll see that your application now handles you bashing the button with no problems.

Check what happens if you hit the button multiple times. You should see your threads executed immediately *up* to the number reported by .maxThreadCount. If you hit the button again after there are already this number of active workers, the subsequent workers will be queued until a thread becomes available.

Improved QRunnables (1)

If you want to pass custom data into the execution function you can do so via the init, and then have access to the data via self. from within the run slot.

multithread.py

```python
class Worker(QRunnable):
    '''
    Worker thread

    :param args: Arguments to make available to the run code
    :param kwargs: Keywords arguments to make available to the
    run code

    '''

    def __init__(self, *args, **kwargs):
        super(Worker, self).__init__()
        self.args = args
        self.kwargs = kwargs

    @pyqtSlot()
    def run(self):
        '''
        Initialise the runner function with passed self.args,
        self.kwargs.
        '''
        print(args, kwargs)
```

Improved QRunnables (2)

In fact, we can take advantage of the fact that in Python functions are objects and pass in the function to execute rather than subclassing each time. In the following construction we only require a single Worker class to handle all of our execution jobs.

multithread.py (Part I)

```
class Worker(QRunnable):
    '''
    Worker thread

    Inherits from QRunnable to handler worker thread setup,
    signals and wrap-up.

    :param callback: The function callback to run on this worker
        thread. Supplied args and kwargs will be passed through
        to the runner.
```

multithread.py (Part II)

```
    :type callback: function
    :param args: Arguments to pass to the callback function
    :param kwargs: Keywords to pass to the callback function

    '''

    def __init__(self, fn, *args, **kwargs):
        super(Worker, self).__init__()
        # Store constructor arguments (re-used for processing)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs

    @pyqtSlot()
    def run(self):
        '''
        Initialise the runner function with passed args, kwargs.
        '''
        self.fn(*self.args, **self.kwargs)
```

Improved QRunnables (3)

You can now pass in any Python function and have it executed in a separate thread.

```
def execute_this_fn(self):
    print("Hello!")

def oh_no(self):
    # Pass the function to execute
    worker = Worker(self.execute_this_fn) # Any other args, kwargs are passed to the run function

    # Execute
    self.threadpool.start(worker)
```

Thread IO (1)

Sometimes it's helpful to be able to pass back *state* and *data* from running workers. This could include the outcome of calculations, raised exceptions or ongoing progress (think progress bars). Qt provides the *signals and slots* framework which allows you to do just that and is thread-safe, allowing safe communication directly from running threads to your GUI frontend. *Signals* allow you to .emit values, which are then picked up elsewhere in your code by slot functions which have been linked with .connect.

Below is a simple WorkerSignals class defined to contain a number of example signals.

Custom signals can only be defined on objects derived from QObject. Since QRunnable is not derived from QObject we can't define the signals there directly. A custom QObject to hold the signals is the simplest solution.

Thread IO (2)

multithread.py

```python
import traceback, sys

class WorkerSignals(QObject):
    '''
    Defines the signals available from a running worker thread.

    Supported signals are:

    finished
        No data

    error
        tuple (exctype, value, traceback.format_exc() )

    result
        object data returned from processing, anything

    '''
    finished = pyqtSignal()
    error = pyqtSignal(tuple)
    result = pyqtSignal(object)
```

In this example we've defined 3 custom signals:

1. *finished* signal, with no data to indicate when the task is complete.
2. *error* signal which receives a tuple of Exception type, Exception value and formatted traceback.
3. *result* signal receiving any object type from the executed function.

Thread IO (3)

You may not find a need for all of these signals, but they are included to give an indication of what is possible. In the following code we're going to implement a long-running task that makes use of these signals to provide useful information to the user.

multithread.py (Part I)

```python
class Worker(QRunnable):
    '''
    Worker thread

    Inherits from QRunnable to handler worker thread setup,
    signals and wrap-up.

    :param callback: The function callback to run on this worker
        thread. Supplied args and kwargs will be passed through
        to the runner.
    :type callback: function
```

multithread.py (Part II)

```python
    :param args: Arguments to pass to the callback function
    :param kwargs: Keywords to pass to the callback function

    '''

    def __init__(self, fn, *args, **kwargs):
        super(Worker, self).__init__()
        # Store constructor arguments (re-used for processing)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        '''
        Initialise the runner function with passed args, kwargs.
        '''

        # Retrieve args/kwargs here; and fire processing using
        # them
        try:
            result = self.fn(
                *self.args, **self.kwargs
```

Thread IO (4)

multithread.py (Part III)

```python
            )
        except:
            traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
            self.signals.error.emit((exctype, value, \
            traceback.format_exc()))
        else:
            self.signals.result.emit(result)  # Return the
                                              # result of the processing
        finally:
            self.signals.finished.emit()  # Done
```

You can connect your own handler functions to these signals to receive notification of completion (or the result) of threads.

multithread.py

```python
def execute_this_fn(self):
    for n in range(0, 5):
        time.sleep(1)
    return "Done."

def print_output(self, s):
    print(s)

def thread_complete(self):
    print("THREAD COMPLETE!")

def oh_no(self):
    # Pass the function to execute
    worker = Worker(self.execute_this_fn) # Any other args, \
    kwargs are passed to the run function
    worker.signals.result.connect(self.print_output)
    worker.signals.finished.connect(self.thread_complete)

    # Execute
    self.threadpool.start(worker)
```

Thread IO (5)

You also often want to receive status information from long-running threads. This can be done by passing in *callbacks* to which your running code can send the information. You have two options here: either define new signals (allowing the handling to be performed using the event loop) or use a standard Python function.

In both cases you'll need to pass these callbacks into your target function to be able to use them. The signal-based approach is used in the completed code below, where we pass an int back as an indicator of the thread's

The complete code (1)

A complete working example is given below, showcasing the custom QRunnable worker together with the worker & progress signals. You should be able to easily adapt this code to any multi-threaded application you develop.

multithread.py (Part I)

```python
from PyQt6.QtGui import *
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *

import time
import traceback, sys


class WorkerSignals(QObject):
    '''
    Defines the signals available from a running worker thread.

```

multithread.py (Part II)

```python
    Supported signals are:

    finished
        No data

    error
        tuple (exctype, value, traceback.format_exc() )

    result
        object data returned from processing, anything

    progress
        int indicating % progress

    '''
    finished = pyqtSignal()
    error = pyqtSignal(tuple)
    result = pyqtSignal(object)
    progress = pyqtSignal(int)


class Worker(QRunnable):
    '''
    Worker thread
```

The complete code (2)

multithread.py (Part III)

```python
    Inherits from QRunnable to handler worker thread setup,
    signals and wrap-up.

    :param callback: The function callback to run on this
        worker thread. Supplied args and kwargs will be
        passed through to the runner.
    :type callback: function
    :param args: Arguments to pass to the callback function
    :param kwargs: Keywords to pass to the callback function

    '''

    def __init__(self, fn, *args, **kwargs):
        super(Worker, self).__init__()

        # Store constructor arguments (re-used for processing)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

        # Add the callback to our kwargs
        self.kwargs['progress_callback'] = self.signals.progress
```

multithread.py (Part IV)

```python
    @pyqtSlot()
    def run(self):
        '''
        Initialise the runner function with passed args, kwargs.
        '''

        # Retrieve args/kwargs here; and fire processing using
        # them
        try:
            result = self.fn(*self.args, **self.kwargs)
        except:
            traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
            self.signals.error.emit((exctype, value,
                traceback.format_exc()))
        else:
            self.signals.result.emit(result)  # Return the
                                              # result of the processing
        finally:
            self.signals.finished.emit()  # Done


class MainWindow(QMainWindow):
```

The complete code (3)

multithread.py (Part V)

```python
    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
```

multithread.py (Part VI)

```python
        print("Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount())

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def progress_fn(self, n):
        print("%d%% done" % n)

    def execute_this_fn(self, progress_callback):
        for n in range(0, 5):
            time.sleep(1)
            progress_callback.emit(n*100/4)

        return "Done."

    def print_output(self, s):
        print(s)

    def thread_complete(self):
        print("THREAD COMPLETE!")
```

The complete code (4)

multithread.py (Part VII)

```python

    def oh_no(self):
        # Pass the function to execute
        worker = Worker(self.execute_this_fn) # Any other args,
                                # kwargs are passed to the run function
        worker.signals.result.connect(self.print_output)
        worker.signals.finished.connect(self.thread_complete)
        worker.signals.progress.connect(self.progress_fn)

        # Execute
        self.threadpool.start(worker)


    def recurring_timer(self):
        self.counter +=1
        self.l.setText("Counter: %d" % self.counter)


app = QApplication([])
window = MainWindow()
app.exec_()
```

Caveats (1)

You may have spotted the slight flaw in this master plan — we are still making use of the event loop (and the *GUI thread*) to process the output of our workers.

This isn't a problem when we're simply tracking progress, completion or returning metadata. However, if you have workers which return large amounts of data — e.g. loading large files, performing complex analysis and need (large) results, or querying databases — passing this data back through the GUI thread may cause performance problems and is best avoided.

Caveats (2)

Similarly, if your application makes use of a large number of threads and Python result handlers, you may come up against the limitations of the GIL. As mentioned previously, when using threads execution of Python is limited to a single thread at one time. The Python code that handles signals from your threads can be blocked by your workers and *vice versa*. Since blocking your slot functions blocks the event loop, this can directly impact GUI responsiveness.

In these cases it is often better to investigate using a pure-Python thread pool (e.g. concurrent futures) to keep your processing and thread-event handling further isolated from your GUI. However, note that *any* Python GUI code can block other Python code unless it's in a separate process.