



Code less.
Create more.
Deploy everywhere.

Enhancing your Qt application with multiple threads

NOKIA

Thiago Macieira
Qt Developer Days 2008

NOKIA

Who am I?

- Senior Software Engineer at Qt Software / Nokia
- Degrees in Engineering and an MBA
- Almost 2 years with the company
- Part of the Core Team
 - Work mostly with Networking, I/O, IPC, Threading, Tool Classes
- Other official duties:
 - Liaison to the KDE Community
 - Release Manager for Qt



Agenda

Introduction

Evolution of threading in Qt

API, do's and don'ts

Agenda

Introduction

Evolution of threading in Qt

API, do's and don'ts

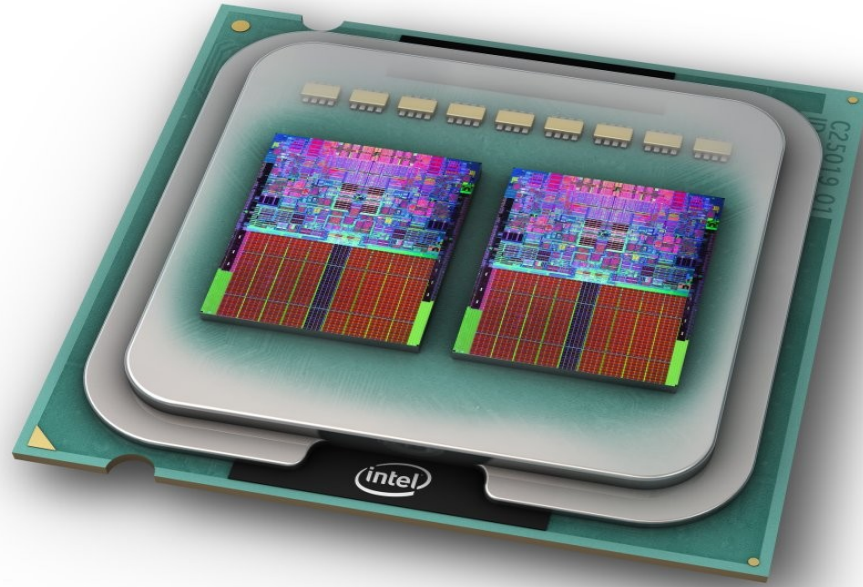
General recommendations

- In general, people advise against threads
 - And give many reasons why
- Threads do make the program more complex
- If your problem doesn't require threads, you don't have to use it

“Threads are for people who can't program a state machine”
-- Alan Cox

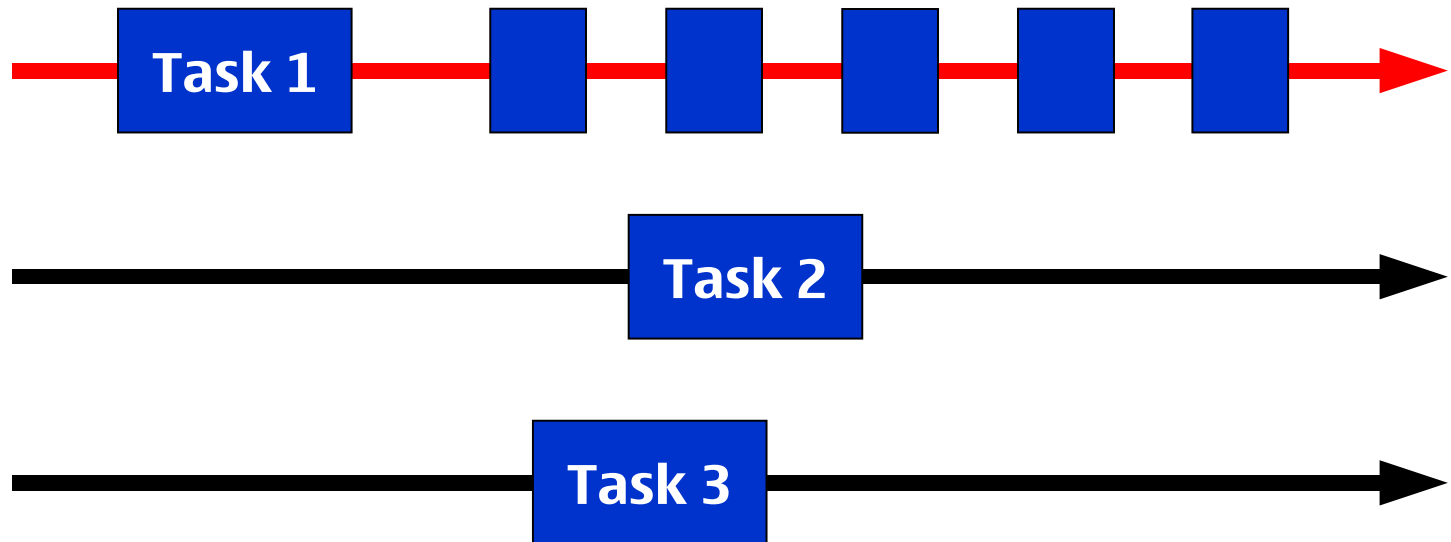
Multi-core hardware...?

- Desktop processors are usually multi-core now
 - Dual cores, quad cores, dual quad cores, etc.
 - (This laptop is dual core)
- Embedded hardware to follow suit very soon



... Threaded software!

- To use all cores, you need threads
- To use threads, you need to identify tasks



What can you use threads for?



What can you use threads for?

- Time-consuming tasks
- Blocking operations
- Parallelising independent operations
- More...



Agenda

Introduction

Evolution of threading in Qt

API, do's and don'ts

In the beginning, there was nothing...

- Qt 1.x had no support for threading
- Qt 2.x had QThread, but not by default
- Qt 3.x had QThread by default



Qt 4.0 made threads first-class citizens

- Tool classes reentrant
- Thread-safe atomic detaching
- Event loops in auxiliary threads
- Cross-thread signal-slot connections

Cross-thread signal-slot connection

- Signals delivered immediately (DirectConnection)
- Or delivered later (QueuedConnection)
 - Qt copies the arguments (QVariant)
 - Posts an event to receiver object
 - Events were already thread-safe
- Qt 4.3: BlockingQueuedConnection



Qt 4.4 brought threading to a new level

- Basic atomic operations
- New classes abstracting thread manipulation
- QtConcurrent
- Painting from multiple threads
- Thread-safe text rendering on X11
- Thread-safe rich-text and SVG support



Agenda

Introduction

Evolution of threading in Qt

API, do's and don'ts

Agenda

Introduction

Evolution of threading in Qt

API, do's and don'ts

- Threads & Concurrent
- Locking & Atomic

Things to remember about QThread

- The QThread object is **not** in that thread
 - Think of QThread as a manager object
 - Constructor is run in another thread
 - Be careful with signals and slots
- Event loop is not automatic
 - To receive signals in that thread, you have to call exec()



The basic: starting a thread

```
class MyThread: public QThread
{
protected:
    void run()
    {
        // thread code here
    }
};
```

```
void somewhereElse()
{
    MyThread *thr =
        new MyThread;
    thr->start();
}
```

What if you need a thread, any thread?

- You don't need the thread running all of the time
- You just have a task that has to be run

```
class MyTask: public QRunnable
{
protected:
    void run()
    {
        // task code here
    }
};
```

```
QThreadPool *threadPool;
void somewhereElse()
{
    MyTask *task =
        new MyTask;
    threadPool->start(task);
}
```

QThread vs. QRunnable

- QThread derives from QObject
- Signals and slots
- QObject is “heavy”
- Cost of creating a thread
- Existed since Qt 2.x
- QRunnable has no base class
- No signals or slots
- Light-weight
- Runs on any free thread
- New in Qt 4.4

What if you just need to run a function?

- The function already exists
- But no QThread or QRunnable exists for it

```
int myFunction()
{
    // task code here
    return 42;
}
```

```
void somewhereElse()
{
    QFuture<int> result =
        QtConcurrent::run(myFunction);

    // do some other work
    result.waitForFinished();
}
```

QFuture<T>

- Represents an asynchronous computation
- Useful functions:
 - isFinished
 - isRunning
 - isStarted
 - waitForFinished

QFuture's family

- Multiple QFuture can be combined in a QFutureSynchronizer
- For non-blocking synchronisation, there is QFutureWatcher
- Signals and slots

```
QFuture<int> f1 = ...;  
QFuture<int> f2 = ...;  
  
QFutureSynchronizer<int> sync;  
sync.addFuture(f1);  
sync.addFuture(f2);  
sync.waitForFinished();
```

```
QFuture<int> future = ...;  
QFutureWatcher<int> watcher;  
watcher.setFuture(future);  
  
connect(&watcher,  
        SIGNAL(finished()),  
        this, SLOT(slotFinished()));
```

Passing an argument?

- QtConcurrent allows you to pass arguments to functions

```
int myTask(QString val)
{
    // task code here
    return val.toInt();
}
```

```
void somewhereElse()
{
    QFuture<int> result =
        QtConcurrent::run(myTask, "42");
}
```

Passing an argument from a list?

- Run the function once for every entry in the sequence

```
void myTask(int id)
{
    // task code here
}
```

```
void somewhereElse(QList<int> ids)
{
    QFuture<void> result =
        QtConcurrent::mapped(ids,
            myTask);
}
```

Retrieving returned values

- Each function's result is stored in QFuture
- Call results() to obtain a QList with the values

```
qulonglong myTask(int id)
{
    // task code here
    return id;
}
```

```
QList<qulonglong>
somewhereElse(QList<int> ids)
{
    QFuture<qulonglong> future =
        QtConcurrent::mapped(ids,
            myTask);
    future.waitForFinished();
    return future.results()
}
```

Agenda

Introduction

Evolution of threading in Qt

API, do's and don'ts

- Threads & Concurrent
- Locking & Atomic

Global data: locking

- Access to global data must be protected

```
static MyObject globalData;  
static QMutex mutex;
```

```
void task1()  
{  
    mutex.lock();  
    // use globalData  
    mutex.unlock();  
}
```

```
void task2()  
{  
    QMutexLocker locker(&mutex);  
    // use globalData  
}
```

More refined locking

- Depends on the type of access

```
static MyObject globalData;  
static QReadWriteLock mutex;
```

```
void task1()  
{  
    mutex.lockForRead();  
    // use globalData  
    mutex.unlock();  
}
```

```
void task2()  
{  
    mutex.lockForWrite();  
    // modify globalData  
    mutex.unlock();  
}
```

Sometimes no locking is needed

- If the operation can be atomic

```
static QAtomicInt counter;
```

```
void task1()
{
    int id = counter
        .fetchAndAddRelaxed(1);
    // operation
    counter
        .fetchAndAddRelaxed(-1);
}
```

```
void task2()
{
    int currentCount = counter;
    qDebug() << currentCount;
}
```

Why can't you use int?

- Operations on small integer types are atomic

```
static int counter;
```

```
void task1()
{
    int id = ++counter;

    // operation

    --counter;
}
```

```
void task2()
{
    int currentCount = counter;
    qDebug() << currentCount;
}
```

Why can't you use int?

- Operations on small integer types are atomic

```
static volatile int counter;
```

```
void task1()
{
    int id = ++counter;

    // operation

    --counter;
}
```

```
void task2()
{
    int currentCount = counter;
    qDebug() << currentCount;
}
```

QAtomicInt & QAtomicPointer<T>

- Atomic operations supported:
 - Test-and-set (a.k.a. compare-and-exchange)
 - Fetch-and-store (a.k.a. exchange)
 - Fetch-and-add
- Each operation supports 4 memory ordering semantics
 - If the hardware supports it
- QAtomicInt is very useful for reference counting



QAtomic feature testing

- Level of functionality varies according to hardware
- Tests at compile time (macros)
or at run-time (static functions)
- Operations can be:
 - Native or emulated
 - Wait-free or looping



Even for slightly complex structures

- The pop() function is left as an exercise for the reader

```
class LockFreeStack
{
    QAtomicPointer<MyType> stack;
public:
    void push(MyType *t)
    {
        do {
            t->next = stack;
        } while (!stack.testAndSetRelease(t->next, t));
    }
};
```

Agenda

Summary

Sometimes you don't need threads

- Analyse your problem
- Can it be done without threads?

- For example, signals and slots
- State machine!

Using threads should be easy

- We give you all the rope you need and a little more
- API is there and constantly being improved
- We're probably not done yet...

*Just because you can, it
doesn't mean you should.*