Threading with Qt

Qt 3.0

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

Thread Support in Qt

In version 2.2, Qt introduced thread support to Qt in the form of some basic platform-independent threading classes, a thread-safe way of posting events and a global Qt library lock that allows you to call Qt methods from different threads.

Preface

This document is intended for an audience that has knowledge and experience with multithreaded applications. Recommended reading:

- Threads Primer: A Guide to Multithreaded Programming
- Thread Time: The Multithreaded Programming Guide
- Pthreads Programming: A POSIX Standard for Better Multiprocessing (O'Reilly Nutshell)
- Win32 Multithreaded Programming

Enabling thread support

When Qt is installed on Windows, thread support is an option on some compilers.

On Mac OS X and Unix, thread support is enabled by adding the `-thread` option when running the `configure` script. On Unix platforms where multithreaded programs must be linked in special ways, such as with a special libc, installation will create a separate library, `libqt-mt` and hence threaded programs must be linked against this library (with `-lqt-mt`) rather than the standard Qt library.

On both platforms, you should compile with the macro `QT_THREAD_SUPPORT` defined (eg. compile with `-DQT_THREAD_SUPPORT`). On Windows, this is usually done by an entry in `qconfig.h`.

The Qt thread classes

The most important class is QThread; this provides the means to start a new thread, which begins execution in your reimplementation of QThread::run(). This is similar to the Java thread class.

However, a thread class alone is not sufficient. In order to write threaded programs it is necessary to protect access to data that two threads wish to access at once. Therefore there is also a QMutex class; a thread can lock the mutex, and while it has it locked no other thread can lock the mutex; an attempt to do so will block the other thread until the mutex is released. For instance:

```
class MyClass {
```

```
public:
    void doStuff(int);

private:
    QMutex mutex;
    int a;
    int b;
};

// This sets a to c, and b to c*2

void MyClass::doStuff(int c)
{
    mutex.lock();
    a = c;
    b = c * 2;
    mutex.unlock();
}
```

This ensures that only one thread at a time can be in MyClass::doStuff(), so b will always be equal to c*2.

Also necessary is a method for threads to wait for another thread to wake it up given a condition; the QWaitCondition class provides this. Threads wait for the QWaitCondition to indicate that something has happened, blocking until it does. When something happens, QWaitCondition can wake up all of the threads waiting for that event or one randomly selected thread (this is the same functionality as a POSIX Threads condition variable and is implemented as one on Unix). For example:

```
#include <qapplication.h>
#include <qpushbutton.h>

// global condition variable

QWaitCondition mycond;


// Worker class implementation

class Worker : public QPushButton, public QThread
{
    Q_OBJECT

public:
    Worker(QWidget *parent = 0, const char *name = 0)
      : QPushButton(parent, name)
    {
        setText("Start Working");

        // connect the clicked() signal inherited from QPushButton to our
        // slotClicked() method
        connect(this, SIGNAL(clicked()), SLOT(slotClicked()));

        // call the start() method inherited from QThread... this starts
        // execution of the thread immediately
        QThread::start();
```

```
    }

public slots:
    void slotClicked()
    {
        // wake up one thread waiting on this condition variable
        mycond.wakeOne();
    }

protected:
    void run()
    {
        // this method is called by the newly created thread...

        while(1) {
            // lock the application mutex, and set the caption of
            // the window to indicate that we are waiting to
            // start working
            qApp->lock();
            setCaption("Waiting");
            qApp->unlock();

             // wait until we are told to continue
            mycond.wait();

            // if we get here, we have been woken by another
            // thread... let's set the caption to indicate
            // that we are working
            qApp->lock();
            setCaption("Working!");
            qApp->unlock();

            // this could take a few seconds, minutes, hours, etc.
            // since it is in a separate thread from the GUI thread
            // the gui will not stop processing events...
            do_complicated_thing();
        }
    }
};


// main thread - all GUI events are handled by this thread.

main(int argc, char **argv)
{
    QApplication app(argc, argv);

    // create a worker... the worker will run a thread when we do
    Worker firstworker(0, "worker");

    app.setMainWidget(&worker);
    worker.show();
```

```
    return app.exec();
}
```

This program will wake up the worker thread whenever you press the button; the thread will go off and do some work and then go back to waiting to be told to do some more work. If the worker thread is already working when the button is pressed, nothing will happen. When the thread finishes working and calls QWaitCondition::wait() again, then it can be started.

Thread-safe posting of events

In Qt, one thread is always the event thread - that is, the thread that pulls events from the window system and dispatches them to widgets. The static method QThread::postEvent posts events from threads other than the event thread. The event thread is woken up and the event delivered from within the event thread just as a normal window system event is. For instance, you could force a widget to repaint from a different thread by doing the following:

```
QWidget *mywidget;
QThread::postEvent( mywidget, new QPaintEvent( QRect(0, 0, 100, 100) ) );
```

This (asynchronously) makes mywidget repaint a 100x100 square of its area.

The Qt library mutex

The Qt library mutex provides a method for calling Qt methods from threads other than the event thread. For instance:

```
QApplication *qApp;
QWidget *mywidget;

qApp->lock();

mywidget->setGeometry(0,0,100,100);

QPainter p;
p.begin(mywidget);
p.drawLine(0,0,100,100);
p.end();

qApp->unlock();
```

Calling a function in Qt without holding a mutex will generally result in unpredictable behavior. Calling a GUI-related function in Qt from a different thread requires holding the Qt library mutex. In this context, all functions that may ultimately access any graphics or window system resources are GUI-related. Using container classes, strings and I/O classes does not require any mutex if that object is only accessed by one thread.

Caveats

Some things to watch out for when programming with threads:

- Don't do any blocking operations while holding the Qt library mutex. This will freeze up the event loop.
- Make sure you lock a recursive QMutex as many times as you unlock it, no more and no less.
- Lock the Qt application mutex before calling anything except for the Qt container and tool classes.
- Be wary of classes which are implicitly shared; you should probably detach() them if you need to assign them between threads.
- Be wary of Qt classes which were not designed with thread safety in mind; for instance, QPtrList's API is not thread-safe and if different threads need to iterate through a QPtrList they should lock before calling QPtrList::first() and unlock after reaching the end, rather than locking and unlocking around QPtrList::next().
- Be sure to create objects that inherit or use QWidget, QTimer and QSocketNotifier objects only in the GUI thread. On some platforms, such objects created in a thread other than the GUI thread will never receive events from the underlying window system.
- Similar to the above, only use the QNetwork classes inside the GUI thread. A common question asked is if a QSocket can be used in multiple threads. This is unnecessary, since all of the QNetwork classes are asynchronous.
- Never call a function that attempts to processEvents() from a thread other than the GUI thread. This includes QDialog::exec(), QPopupMenu::exec(), QApplication::processEvents() and others.
- Don't mix the normal Qt library and the threaded Qt library in your application. This means that if your application uses the threaded Qt library, you should not link with the normal Qt library, dynamically load the normal Qt library or dynamically load another library that depends on the normal Qt library. On some systems, doing this can corrupt the static data used in the Qt library.

QMutex Class Reference

The QMutex class provides access serialization between threads.

`#include <qmutex.h>`

Public Members

- **QMutex** (bool recursive = FALSE)
- virtual **~QMutex** ()
- void **lock** ()
- void **unlock** ()
- bool **locked** ()
- bool **tryLock** ()

Detailed Description

The QMutex class provides access serialization between threads.

The purpose of a QMutex is to protect an object, data structure or section of code so that only one thread can access it at a time (In Java terms, this is similar to the synchronized keyword). For example, say there is a method which prints a message to the user on two lines:

```
void someMethod()
{
   qDebug("Hello");
   qDebug("World");
}
```

If this method is called simultaneously from two threads then the following sequence could result:

```
Hello
Hello
World
World
```

If we add a mutex:

```
QMutex mutex;

void someMethod()
{
   mutex.lock();
   qDebug("Hello");
   qDebug("World");
   mutex.unlock();
}
```

In Java terms this would be:

```
void someMethod()
{
   synchronized {
     qDebug("Hello");
     qDebug("World");
   }
}
```

Then only one thread can execute someMethod at a time and the order of messages is always correct. This is a trivial example, of course, but applies to any other case where things need to happen in a particular sequence.

When you call lock() in a thread, other threads that try to call lock() in the same place will block until the thread that got the lock calls unlock(). A non-blocking alternative to lock() is tryLock().

See also Environment Classes and Threading.

Member Function Documentation

QMutex::QMutex (bool recursive = FALSE)

Constructs a new mutex. The mutex is created in an unlocked state. A recursive mutex is created if *recursive* is TRUE; a normal mutex is created if *recursive* is FALSE (the default). With a recursive mutex, a thread can lock the same mutex multiple times and it will not be unlocked until a corresponding number of unlock() calls have been made.

QMutex::~QMutex () [virtual]

Destroys the mutex.

void QMutex::lock ()

Attempt to lock the mutex. If another thread has locked the mutex then this call will *block* until that thread has unlocked it.

See also unlock() [p. 10] and locked() [p. 10].

bool QMutex::locked ()

Returns TRUE if the mutex is locked by another thread; otherwise returns FALSE.

Warning: Due to differing implementations of recursive mutexes on various platforms, calling this function from the same thread that previously locked the mutex will return undefined results.

See also lock() [p. 9] and unlock() [p. 10].

bool QMutex::tryLock ()

Attempt to lock the mutex. If the lock was obtained, this function returns TRUE. If another thread has locked the mutex, this function returns FALSE, instead of waiting for the mutex to become available, i.e. it does not block.

The mutex must be unlocked with unlock() before another thread can successfully lock it.

See also lock() [p. 9], unlock() [p. 10] and locked() [p. 10].

void QMutex::unlock ()

Unlocks the mutex. Attempting to unlock a mutex in a different thread to the one that locked it results in an error. Unlocking a mutex that is not locked results in undefined behaviour (varies between different Operating Systems' thread implementations).

See also lock() [p. 9] and locked() [p. 10].

QSemaphore Class Reference

The QSemaphore class provides a robust integer semaphore.

`#include <qsemaphore.h>`

Public Members

- **QSemaphore** (int maxcount)
- virtual **~QSemaphore** ()
- int **available** () const
- int **total** () const
- int **operator++** (int)
- int **operator--** (int)
- int **operator+=** (int n)
- int **operator-=** (int n)
- bool **tryAccess** (int n)

Detailed Description

The QSemaphore class provides a robust integer semaphore.

A QSemaphore can be used to serialize thread execution, in a similar way to a QMutex. A semaphore differs from a mutex, in that a semaphore can be accessed by more than one thread at a time.

For example, suppose we have an application that stores data in a large tree structure. The application creates 10 threads (commonly called a thread pool) to perform searches on the tree. When the application searches the tree for some piece of data, it uses one thread per base node to do the searching. A semaphore could be used to make sure that two threads don't try to search the same branch of the tree at the same time.

A non-computing example of a semaphore would be dining at a restuarant. A semaphore is initialized to have a maximum count equal to the number of chairs in the restuarant. As people arrive, they want a seat. As seats are filled, the semaphore is accessed, once per person. As people leave, the access is released, allowing more people to enter. If a party of 10 people want to be seated, but there are only 9 seats, those 10 people will wait, but a party of 4 people would be seated (taking the available seats to 5, making the party of 10 people wait longer).

When a semaphore is created it is given a number which is the maximum number of concurrent accesses it will permit. This amount may be changed using operator++(), operator--(), operator+=() and operator-=(). The number of accesses allowed is retrieved with available(), and the total number with total(). Note that the incrementing functions will block if there aren't enough available accesses. Use tryAccess() if you want to acquire accesses without blocking.

See also Environment Classes and Threading.

Member Function Documentation

QSemaphore::QSemaphore (int maxcount)

Creates a new semaphore. The semaphore can be concurrently accessed at most *maxcount* times.

QSemaphore::~QSemaphore () [virtual]

Destroys the semaphore.

int QSemaphore::available () const

This function returns the number of accesses currently available to the semaphore.

int QSemaphore::operator++ (int)

Postfix ++ operator.

Try to get access to the semaphore. If available() == 0, this call will block until it can get access, i.e. until available() > 0.

int QSemaphore::operator+= (int n)

Try to get access to the semaphore. If available() < *n*, this call will block until it can get all the accesses it wants, i.e. until available() >= *n*.

int QSemaphore::operator-- (int)

Postfix — operator.

Release access of the semaphore. This wakes all threads waiting for access to the semaphore.

int QSemaphore::operator-= (int n)

Release *n* accesses to the semaphore.

int QSemaphore::total () const

This function returns the total number of accesses to the semaphore.

bool QSemaphore::tryAccess (int n)

Try to get access to the semaphore. If available() < *n*, this function will return FALSE immediately. If available() >= *n*, this function will take *n* accesses and return TRUE. This function does *not* block.

QThread Class Reference

The QThread class provides platform-independent threads.

`#include <qthread.h>`

Inherits Qt [Additional Functionality with Qt].

Public Members

- **QThread** ()
- virtual **~QThread** ()
- bool **wait** (unsigned long time = ULONG_MAX)
- void **start** ()
- bool **finished** () const
- bool **running** () const

Static Public Members

- Qt::HANDLE **currentThread** ()
- void **postEvent** (QObject * receiver, QEvent * event)
- void **exit** ()

Protected Members

- virtual void **run** ()

Static Protected Members

- void **sleep** (unsigned long secs)
- void **msleep** (unsigned long msecs)
- void **usleep** (unsigned long usecs)

Detailed Description

The QThread class provides platform-independent threads.

A QThread represents a separate thread of control within the program; it shares data with all the other threads within the process but executes independently in the way that a separate program does on a multitasking operating system. Instead of starting in main(), QThreads begin executing in run(). You inherit run() to include your code. For example:

```
class MyThread : public QThread {

public:

  virtual void run();

};

void MyThread::run()
{
  for( int count = 0; count < 20; count++ ) {
    sleep( 1 );
    qDebug( "Ping!" );
  }
}

int main()
{
    MyThread a;
    MyThread b;
    a.start();
    b.start();
    a.wait();
    b.wait();
}
```

This will start two threads, each of which writes Ping! 20 times to the screen and exits. The wait() calls at the end of main() are necessary because exiting main() ends the program, unceremoniously killing all other threads. Each MyThread stops executing when it reaches the end of MyThread::run(), just as an application does when it leaves main().

See also Thread Support in Qt [p. 3], Environment Classes and Threading.

Member Function Documentation

QThread::QThread ()

Constructs a new thread. The thread does not actually begin executing until start() is called.

QThread::~QThread () [virtual]

QThread destructor. Note that deleting a QThread object will not stop the execution of the thread it represents. Deleting a running QThread will probably result in a program crash.

Qt::HANDLE QThread::currentThread () [static]

This returns the thread handle of the currently executing thread. The handle returned by this function is used for internal reasons and should *not* be used in any application code. On Windows, the returned value is a pseudo handle for the current thread, and it cannot be used for numerical comparison.

void QThread::exit () [static]

Ends the execution of the calling thread and wakes up any threads waiting for its termination.

bool QThread::finished () const

Returns TRUE is the thread is finished; otherwise returns FALSE.

void QThread::msleep (unsigned long msecs) [static protected]

System independent sleep. This causes the current thread to sleep for *msecs* milliseconds

void QThread::postEvent (QObject * receiver, QEvent * event) [static]

Provides a way of posting an event from a thread which is not the event thread to an object. The *event* is put into a queue, then the event thread is woken which then sends the event to the *receiver* object. It is important to note that the event handler for the event, when called, will be called from the event thread and not from the thread calling QThread::postEvent().

Since QThread::postEvent() posts events into the event queue of QApplication, you must create a QApplication object before calling QThread::postEvent().

The event must be allocated on the heap since the post event queue will take ownership of the event and delete it once it has been posted.

See also QApplication::postEvent() [Additional Functionality with Qt].

void QThread::run () [virtual protected]

This method is pure virtual, and it must be implemented in derived classes in order to do useful work. Returning from this method will end execution of the thread.

See also wait() [p. 16].

bool QThread::running () const

Returns TRUE if the thread is running; otherwise returns FALSE.

void QThread::sleep (unsigned long secs) [static protected]

System independent sleep. This causes the current thread to sleep for *secs* seconds.

void QThread::start ()

This begins actual execution of the thread by calling run(), which should be reimplemented in a QThread subclass to contain your code. If you try to start a thread that is already running, this call will wait until the thread has finished, and then restart the thread.

void QThread::usleep (unsigned long usecs) [static protected]

System independent sleep. This causes the current thread to sleep for *usecs* microseconds

bool QThread::wait (unsigned long time = ULONG_MAX)

This provides similar functionality to POSIX pthread_join. A thread calling this will block until either of these conditions is met:

- The thread associated with this QThread object has finished execution (i.e. when it returns from run()). This function will return TRUE if the thread has finished. It also returns TRUE if the thread has not been started yet.
- *time* milliseconds has elapsed. If *time* is ULONG_MAX (the default), then the wait will never timeout (the thread must return from run()). This function will return FALSE if the wait timed out.

QWaitCondition Class Reference

The QWaitCondition class allows waiting/waking for conditions between threads.

`#include <qwaitcondition.h>`

Public Members

- **QWaitCondition** ()
- virtual **~QWaitCondition** ()
- bool **wait** (unsigned long time = ULONG_MAX)
- bool **wait** (QMutex * mutex, unsigned long time = ULONG_MAX)
- void **wakeOne** ()
- void **wakeAll** ()

Detailed Description

The QWaitCondition class allows waiting/waking for conditions between threads.

QWaitConditions allow a thread to tell other threads that some sort of condition has been met; one or many threads can block waiting for a QWaitCondition to set a condition with wakeOne() or wakeAll(). Use wakeOne() to wake one randomly selected event or wakeAll() to wake them all. For example, say we have three tasks that should be performed every time the user presses a key; each task could be split into a thread, each of which would have a run() body like this:

```
QWaitCondition key_pressed;

for (;;) {
   key_pressed.wait(); // This is a QWaitCondition global variable
   // Key was pressed, do something interesting
   do_something();
}
```

A fourth thread would read key presses and wake the other three threads up every time it receives one, like this:

```
QWaitCondition key_pressed;

for (;;) {
   getchar();
```

```
    // Causes any thread in key_pressed.wait() to return from
    // that method and continue processing
    key_pressed.wakeAll();
}
```

Note that the order the three threads are woken up in is undefined, and that if some or all of the threads are still in do_something() when the key is pressed, they won't be woken up (since they're not waiting on the condition variable) and so the task will not be performed for that key press. This can be avoided by, for example, doing something like this:

```
QMutex mymutex;
QWaitCondition key_pressed;
int mycount=0;

// Worker thread code
for (;;) {
    key_pressed.wait(); // This is a QWaitCondition global variable
    mymutex.lock();
    mycount++;
    mymutex.unlock();
    do_something();
    mymutex.lock();
    mycount--;
    mymutex.unlock();
}

// Key reading thread code
for (;;) {
    getchar();
    mymutex.lock();
    // Sleep until there are no busy worker threads
    while( count > 0 ) {
      mymutex.unlock();
      sleep( 1 );
      mymutex.lock();
    }
    mymutex.unlock();
    key_pressed.wakeAll();
}
```

The mutexes are necessary because the results of two threads attempting to change the value of the same variable simultaneously are unpredictable.

See also Environment Classes and Threading.

Member Function Documentation

QWaitCondition::QWaitCondition ()

Constructs a new event signalling, i.e. wait condition, object.

QWaitCondition::~QWaitCondition () [virtual]

Deletes the event signalling, i.e. wait condition, object.

bool QWaitCondition::wait (unsigned long time = ULONG_MAX)

Wait on the thread event object. The thread calling this will block until either of these conditions is met:

- Another thread signals it using wakeOne() or wakeAll(). This function will return TRUE in this case.
- *time* milliseconds has elapsed. If *time* is ULONG_MAX (the default), then the wait will never timeout (the event must be signalled). This function will return FALSE if the wait timed out.

See also wakeOne() [p. 19] and wakeAll() [p. 19].

bool QWaitCondition::wait (QMutex * mutex, unsigned long time = ULONG_MAX)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Release the locked *mutex* and wait on the thread event object. The *mutex* must be initially locked by the calling thread. If *mutex* is not in a locked state, this function returns immediately. If *mutex* is a recursive mutex, this function returns immediately. The *mutex* will be unlocked, and the calling thread will block until either of these conditions is met:

- Another thread signals it using wakeOne() or wakeAll(). This function will return TRUE in this case.
- *time* milliseconds has elapsed. If *time* is ULONG_MAX (the default), then the wait will never timeout (the event must be signalled). This function will return FALSE if the wait timed out.

The mutex will be returned to the same locked state. This function is provided to allow the atomic transition from the locked state to the wait state.

See also wakeOne() [p. 19] and wakeAll() [p. 19].

void QWaitCondition::wakeAll ()

This wakes all threads waiting on the QWaitCondition. The order in which the threads are woken up depends on the operating system's scheduling policies, and cannot be controlled or predicted.

See also wakeOne() [p. 19].

void QWaitCondition::wakeOne ()

This wakes one thread waiting on the QWaitCondition. The thread that is woken up depends on the operating system's scheduling policies, and cannot be controlled or predicted.

See also wakeAll() [p. 19].

Index