

Таймеры. Процессы и ПОТОКИ

Лекция + практика

Дата в Qt

Работа с датой и временем в Qt осуществляется с помощью классов `QDate`, `QTime` и `QDateTime`. Чаще всего требуется получение текущей даты и времени. Эти классы предоставляют методы для преобразования даты и времени в строку определенного формата. Также есть методы для проведения обратного преобразования — из строки.

Класс `QDate` представляет собой структуру данных для хранения дат и проведения с ними разного рода операций. В конструкторе класса `QDate` передаются три целочисленных параметра (год, месяц, день).

```
QDate date(2023, 12, 25);  
или  
QDate date;  
date.setDate(2023, 12, 25);
```

Метод `toString()` позволяет получить текстовое представление даты.

Можно определить собственный формат даты, передав в метод `toString()` строку-шаблон, описывающую её.

```
QDate date(2023, 12, 25);  
QString str;  
str = date.toString("dd/MM/yy"); //str – «25/12/23»  
str = date.toString("yyyy.MMM.ddd"); //str = "2023.дек.Вс"  
str = date.toString("yyyy.MMMM.dddd");  
//str = "2023.Декабря.воскресенье"
```

```
QDate dateToday = QDate::currentDate();  
QDate dateNewYear(dateToday.year(), 12, 31);  
qDebug() << "Осталось " << dateToday.daysTo(dateNewYear) << " дней до Нового года !";
```

Время в Qt

Для работы со временем библиотека *Qt* предоставляет класс **QTime**. Как и в случае с объектами даты, с объектами времени можно проводить операции сравнения `==`, `!=`, `<`, `<=`, `>` или `>=`. Объекты времени способны хранить время с точностью до миллисекунд. В конструктор класса *QTime* передаются четыре параметра (часы, минуты, секунды, миллисекунды). Третий и четвертый параметры можно опустить, по умолчанию они равны 0.

```
QTime time(20, 4, 23, 3);  
или  
QTime time;  
time.setHMS (20, 4, 23, 3);  
QString str;  
str = time.toString("hh:mm:ss.zzz");  
//str = "20:04:23.003"  
str = time.toString("h:m:s ap");  
//str = "8:4:23 pm"
```

Класс *QTime* предоставляет метод *toString()* для передачи данных объекта времени в виде строки. В этот метод, в качестве параметра, можно передать одно из форматов времени или задать свой собственный.



```
QTime time;  
time.start () ;  
test();  
qDebug() << "Время работы функции test() равно "  
        << time.elapsed() << "миллисекунд" << endl;
```

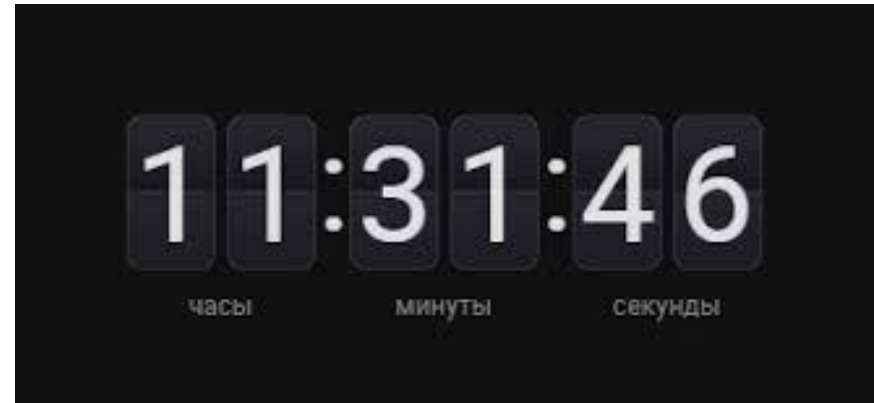
В пример вычисляется
время работы функции
test():

Объекты класса **QDateTime** содержат в себе дату и время. Вызовом метода **date()** можно получить объект даты *QDate*, а вызов **time()** возвращает объект времени *QTime*. Этот класс также содержит методы *toString()* для представления данных в виде строки.

Таймеры в Qt

Таймер – программный механизм, уведомляющий приложение об истечении заданного промежутка времени. События таймера относятся к разряду **внешних прерываний**, т.е. прерываний, вызываемых асинхронными событиями (устройствами ввода/вывода или самим устройством таймера). Интервалы запуска таймера устанавливаются в миллисекундах. Недостаток состоит в том, что если программа занята интенсивными вычислениями, то события таймера могут быть обработаны по окончании процесса вычисления.

События таймера можно использовать и в мульти поточном программировании, для каждого потока, имеющего цикл сообщений (**event loop**). Для запуска цикла сообщений в потоке нужно вызвать метод **QThread::exec()**



Каждый класс, унаследованный от *QObject*, содержит свои собственные встроенные таймеры. Вызов метода **QObject::startTimer()** производит запуск таймера. В качестве параметра ему передается интервал запуска в миллисекундах. Метод *startTimer()* возвращает идентификатор, необходимый для распознавания таймеров, используемых в объекте. По истечении установленного интервала запуска генерируется событие **QTimerEvent**, которое передается в метод *timerEvent()*. Вызвав метод **QTimerEvent::timerId()** объекта события *QTimerEvent*, можно узнать идентификатор таймера, инициировавшего это событие. Идентификатор можно использовать для уничтожения таймера, передав его в метод **QObject::killTimer()**.

QTimer

Класс таймера **QTimer**, являющийся непосредственным наследником класса **QObject**.

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    MyProgram myProgram;
    QTimer::singleShot(5 * 60 * 1000, &app,
                      SLOT(quit()));

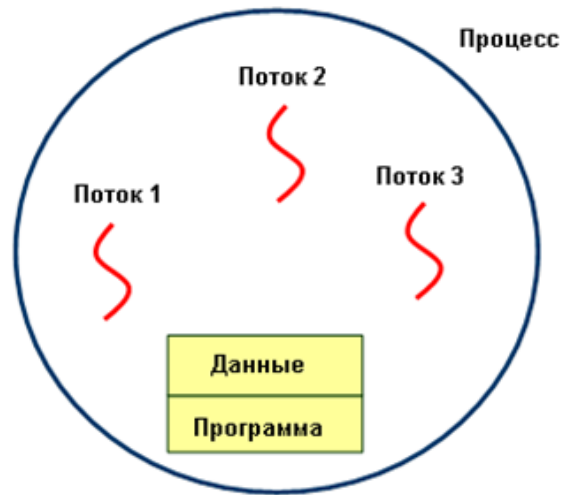
    myProgram.show();
    return app.exec();
}
```

```
#include <QtGui>
class Clock : public QLabel {
    Q_OBJECT
public:
    Clock(QWidget* pwgt = 0) : QLabel(pwgt){ {
        QTimer* ptimer = new QTimer(this);
        connect(ptimer, SIGNAL(timeout()),
                SLOT(slotUpdateDateTime()));

        ptimer->start(500);
        slotUpdateDateTime();
    }
public slots:
    void slotUpdateDateTime()
    {
        QString str =
            QDateTime::currentDateTime().toString(Qt::SystemLocaleDate);
        setText("<H2><CENTER>" + str + "</CENTER></H2>");
    }
};
#endif // _Clock_h_
```

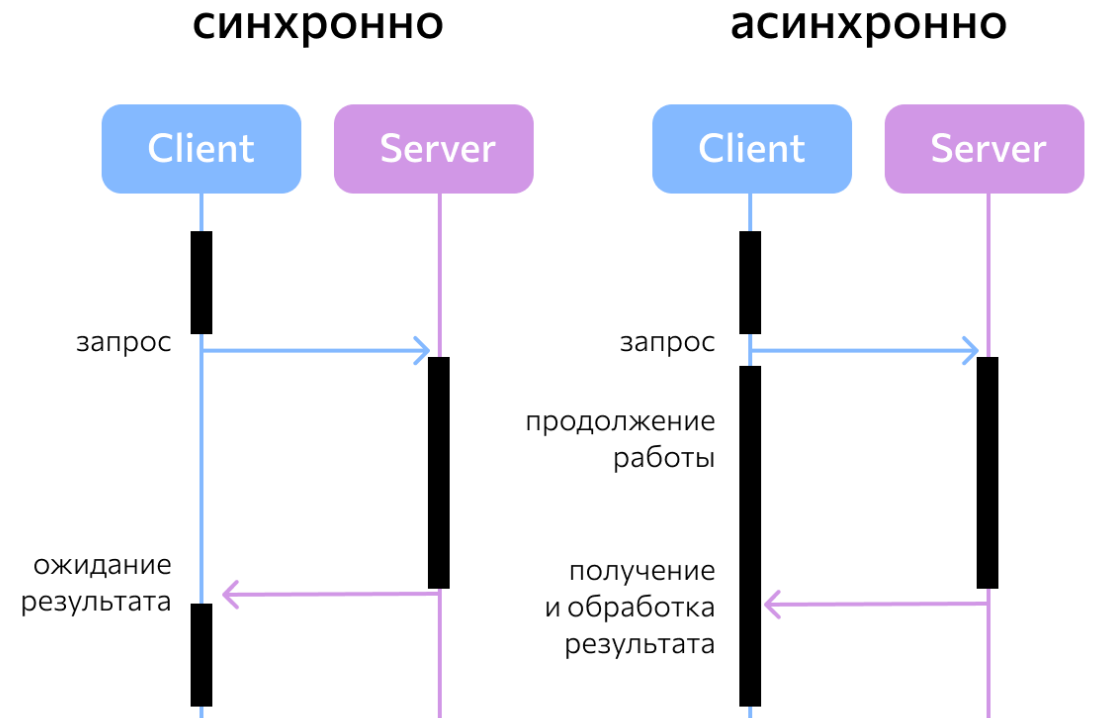
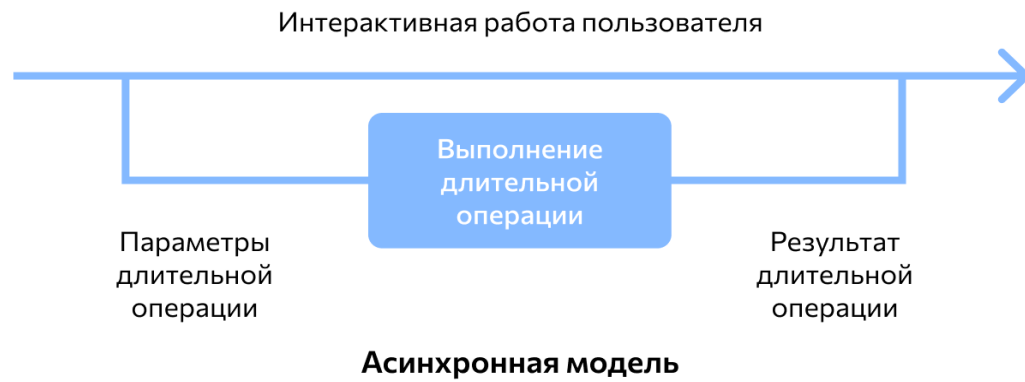
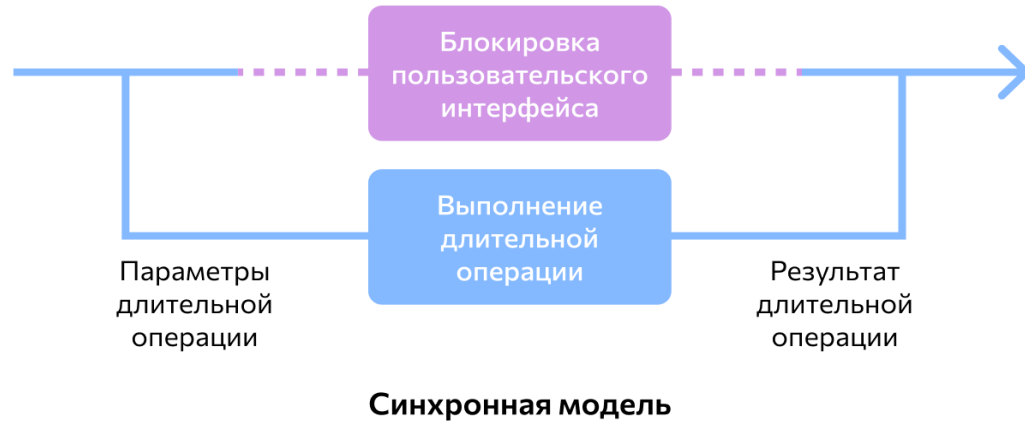
Процессы

Процессы представляют собой **программы**, независимые друг от друга и **загруженные для исполнения**. Каждый процесс должен создавать хотя бы один поток, называемый **основным**. *Основной поток* процесса создается в момент запуска программы. Однако сам **процесс** может создавать **несколько потоков одновременно**.



Запуск другой программы, может осуществить как пользователь так и другая программа, а операционная система всегда создает при этом новый процесс.

Синхронность и асинхронность в программировании



QProcess - Процессы

Запускать другие ПО из текущей Qt-программы можно используя класс **QProcess**(наследник класса *QIODevice*).

Работа с объектами этого класса происходит в **асинхронном режиме**, это позволяет сохранять работоспособность *GUI*, когда запущенные процессы находятся в работе. При появлении данных или других событий объекты *QProcess* отправляют сигналы.

Считывать и записывать данные в процесс можно с помощью методов класса *QIODevice::write()*, *read()*, *readLine()* и *getChar()*. Также для чтения можно воспользоваться методами, привязанными к конкретным каналам: **readAllStandardOutput()** и **readAllStandardError()**. Эти методы считывают данные в объекты класса *QByteArray*.

Запуск процесса выполняется методом **start()**, в который необходимо передать имя команды и список ее аргументов, либо все вместе — команду и аргументы одной строкой. Как только процесс будет запущен, высылается сигнал **started()**, а после завершения его работы высылается сигнал **finished()**. Вместе с сигналом **finished()** высылается код и статус завершения работы процесса. Для получения статуса выхода можно вызвать метод **exitStatus()**, который возвращает либо **NormalExit** (нормальное завершение) либо **CrashExit** (аварийное завершение).

Для чтения данных запущенного процесса класс *QProcess* предоставляет два разделенных канала: канал стандартного вывода (*stdout*) и канал ошибок (*stderr*). Эти каналы можно переключать с помощью метода **setReadChannel()**. Если процесс готов предоставить данные по текущему установленному каналу, то высылается сигнал **readyRead()**. Также высылаются сигналы для каждого канала в отдельности: **readyReadStandardOutput()** и **readyReadStandardError()**.

QThread - Потоки

Поток — это независимая задача, которая выполняется внутри **процесса** и разделяет вместе с ним общее адресное пространство, код и глобальные данные.

Процесс, сам по себе, не является исполнительной частью программы, поэтому для исполнения программного кода он должен иметь хотя бы один поток - **основной поток**. Можно создавать и более одного потока. Вновь созданные потоки начинают выполняться сразу же, параллельно с главным потоком, при этом их количество может изменяться — одни создаются, другие завершаются.

Завершение основного потока приводит к **завершению процесса**, независимо от того, существуют другие потоки или нет.

Для реализации потоков Qt предоставляет класс QThread.

Создание нескольких потоков в процессе получило название **многопоточность**.

Многопоточность требуется для выполнения действий в фоновом режиме, параллельно с действиями основной программы, и позволяет разбить выполнение задач на параллельные потоки, которые могут быть абсолютно независимы друг от друга. А если приложение выполняется на компьютере с несколькими процессорами, то разделение на потоки может значительно ускорить работу всей программы, так как каждый из процессоров получит отдельный поток для выполнения.

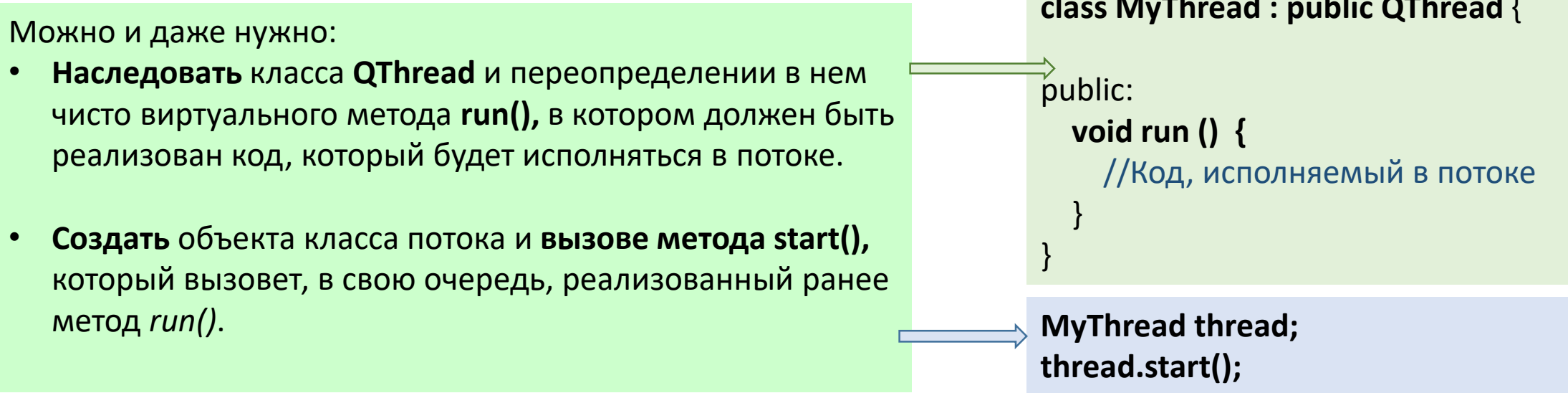
Использование потоков должно быть обоснованным. И если вы не можете сформулировать причину, по которой следует сделать приложение *многопоточным*, то лучше отказаться от использования многопоточности и заменить ее на `QCoreApplication::processEvents()`

Правила работы с потоками в Qt

- **! Нельзя** в потоке (кроме основного) создавать **объекты** от **QWidget** или вызывать **их методы**.
- **! Нельзя** в потоке создавать **объекты** класса **QPixmap**, а только **QImage**;
- **! Нельзя** вызывать **QCoreApplication::exec()**, **QApp::exec()**;
- **! Нельзя** **блокировать** основной поток приложения – иначе “замрет” пользовательский интерфейс.

Можно и даже нужно:

- **Наследовать** класса **QThread** и переопределении в нем чисто виртуального метода **run()**, в котором должен быть реализован код, который будет исполняться в потоке.
- **Создать** объекта класса потока и **вызове метода start()**, который вызовет, в свою очередь, реализованный ранее метод *run()*.



```
class MyThread : public QThread {  
public:  
    void run () {  
        //Код, исполняемый в потоке  
    }  
}
```

```
MyThread thread;  
thread.start();
```

Приоритеты

У каждого потока есть **приоритет**, указывающий процессору, как должно протекать выполнение потока по отношению к другим потокам. Приоритеты разделяются по группам:

InheritPriority, TimeCriticalPriority

Потоки с этими приоритетами нужно создавать в случаях крайней необходимости. Эти приоритеты нужны для программ, напрямую общающихся с аппаратурой или выполняющих операции, которые ни в коем случае не должны прерваться.

HighestPriority, HighPriority,

Пользуйтесь такими приоритетами с большой осторожностью. Обычно эти потоки большую часть времени ожидают какие-либо события;

NormalPriority, LowPriority, LowestPriority, IdlePriority

Они подходят для решения задач, которым процессор требуется только время от времени, например, для фоновой печати или для каких-нибудь несрочных действий;

А для того чтобы узнать, с каким приоритетом был запущен поток, нужно вызвать метод **priority()**. Приоритет можно предварительно установить при помощи метода **setPriority()**.

```
MyThread thread;  
thread.start(QThread::IdlePriority);
```

запустить
поток с
нужным
приоритетом

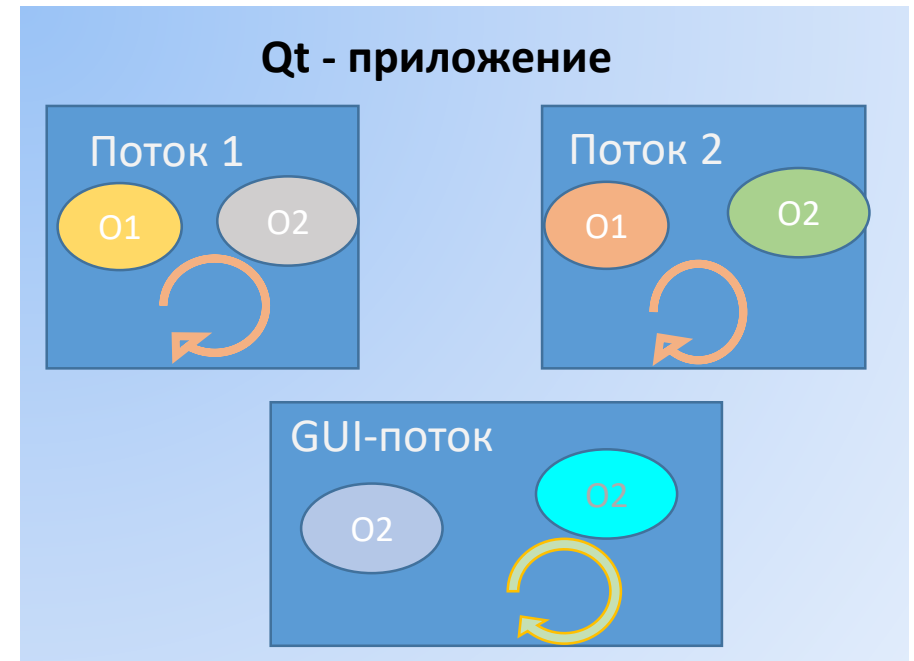
Обмен сообщениями.

Каждый поток может иметь свой собственный цикл событий (**exec()**). Благодаря этому можно осуществлять связь между объектами:

- соединения сигналов и слотов;
- обмена событиями;

Чтобы запустить собственный цикл обработки событий в потоке, нужно поместить вызов метода **exec()** в методе **run()**. Цикл обработки событий потока можно завершать посредством слота **quit()** или метода **exit()**. Это очень похоже на то, как мы обычно поступаем с объектом приложения в функции *main()*.

Каждый объект, произведенный от унаследованного от *QObject* класса, располагает ссылкой на поток, в котором он был создан. Эту ссылку можно получить вызовом метода **QObject::thread()**. Потоки осведомляют свои объекты. Благодаря этому каждый объект знает, к какому потоку он принадлежит.



Обработка событий производится из контекста принадлежности объекта к потоку, то есть обработка его событий будет производиться в том потоке, которому объект принадлежит. Объекты можно перемещать из одного потока в другой с помощью метода **QObject::moveToThread()**.

SIGNAL- SLOT соединение

Можно соединить сигнал объекта одного потока и слотом объекта другого потока. Соединение с помощью метода **connect()** предоставляет дополнительный параметр, обозначающий режим обработки и равный, по умолчанию, значению *Qt::AutoConnection*, которое соответствует автоматическому режиму.

Как только происходит высылка сигнала, Qt проверяет — происходит связь в одном и том же или разных потоках.

*Если это **один и тот же поток**, то высылка сигнала приведет к **прямому вызову метода**.*

*В том случае, если это **разные потоки**, сигнал будет преобразован в **событие** и доставлен нужному объекту.*

Когда **высылающий** объект окажется в одном потоке с **принимающим**, то высылка сигнала будет сведена к **прямой обработке соединения**.

Сигналы и слоты в Qt реализованы с механизмом надежности работы в потоках, а это означает, что можно высылать сигналы и получать, не заботясь о блокировке ресурсов. Вы так же можете перемещать объект, созданный в одном потоке, в другой.



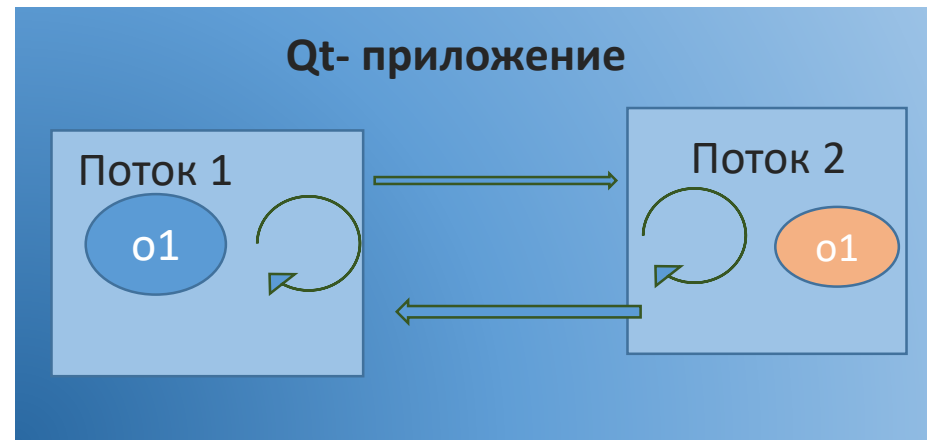
Отправка событий

Существует два метода для высылки событий:
QCoreApplication::postEvent() и **QCoreApplication::sendEvent()**.

Высылка событий методом **postEvent()** обладает надежностью в потоках, а методом **sendEvent()** — нет.

Поток может высылать события другому потоку, который, в свою очередь, может ответить другим событием и т. д. Сами же события, обрабатываемые циклами событий потоков, будут принадлежать тем потокам, в **которых они были созданы**.

Если сравнить реализации программы при помощи сигналов и событий, то заметно, что подход с использованием сигналов более компактный.

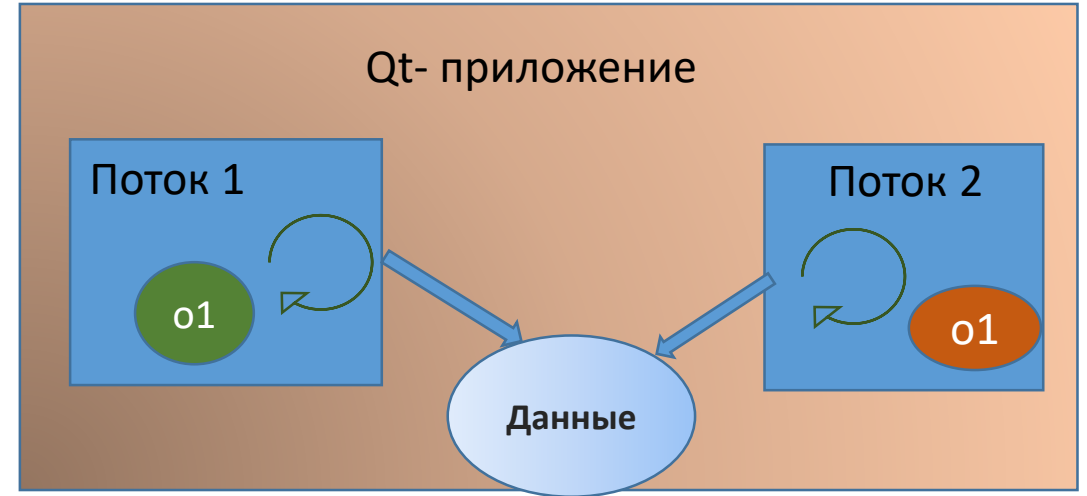


Чтобы объект потока был в состоянии обрабатывать получаемые события, в классе потока нужно реализовать метод **QObject::event()**.

Синхронизация

Основные сложности возникают тогда, когда потокам нужно совместно использовать **одни и те же данные**. Так как несколько потоков могут одновременно обращаться и записывать данные в одну область, то это может привести к нежелательным последствиям.

Синхронизация позволяет задавать критические секции (*critical sections*), к которым в определенный момент имеет доступ **только один из потоков**. Это гарантирует то, что данные ресурса, контролируемые критической секцией, будут невидимы другими потоками и они не изменят их. И только после того, как поток выполнит всю необходимую работу, он освобождает ресурс, и, затем, доступ к этому ресурсу может получить любой другой поток.



Для предотвращения подобных ситуаций требуется механизм, позволяющий блокировать данные, когда один из потоков намеревается их изменить. Этот механизм получил название **синхронизация**.

QMutex - мьютексы

Мьютексы (*mutex*) обеспечивают взаимоисключающий доступ к ресурсам, гарантирующий то, что *критическая секция* будет обрабатываться только одним потоком. Поток, владеющий мьютексом, обладает эксклюзивным правом на использование ресурса, защищенного мьютексом.

Метод **lock()** класса **QMutex** производит блокировку ресурса. Для разблокировки имеется метод **unlock()**, который открывает закрытый ресурс. Метод **tryLock()** можно использовать для того, чтобы проверить, заблокирован ресурс или нет.

Объект класса **QMutexLocker** бывает удобно использовать совместно с объектом *QMutex*. Для создания объекта этого класса в его конструктор необходимо передать указатель на объект *мьютекса*. В конструкторе этого класса сразу же производится блокировка ресурса, а в деструкторе — разблокировка. Это означает, что нам не нужно явно вызывать метод для разблокирования ресурса, как мы это делали в методе *push()*, потому что завершение метода приведет к разрушению этого объекта и произведет вызов его деструктора.

```
class ThreadSafeStringStack
{
private:
    QMutex      m_mutex;
    QStack<QString> m_stackString;
public:
    void push(const QString& str)
    {
        m_mutex.lock();
        m_stackString.push(str);
        m_mutex.unlock();
    }
    QString pop(const QString& str)
    {
        QMutexLocker locker (&m_mutex);
        return m_stackString.empty() ? QString() :
            m_stackString.pop();
    }
};
```

QSemaphore - Семафоры

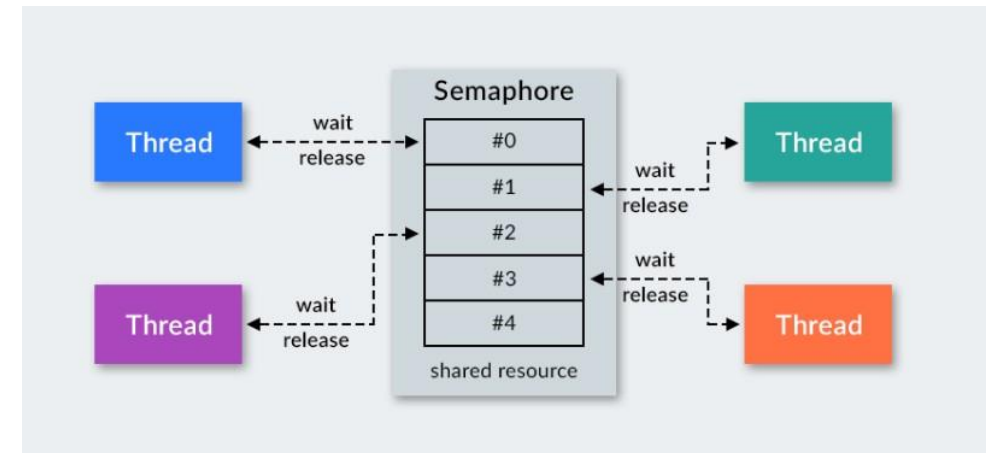
Семафоры являются обобщением *мьютексов*. Как и мьютексы, они служат для защиты *критических секций*, чтобы доступ к ним одновременно могло иметь *определенное число потоков*.

Предположим, что программа поддерживает пять ресурсов одного и того же типа, одновременный доступ к которым может быть предоставлен только пяти потокам. Как только все пять ресурсов будут заблокированы, следующий поток, запрашивающий ресурс данного типа, будет приостановлен до освобождения одного из них.

Принцип действия семафоров очень прост. Они начинают действовать с установленного значения счетчика. Каждый раз, когда поток получает право на владение ресурсом, значение этого счетчика уменьшается на единицу. И наоборот, когда поток уступает право владения этим ресурсом, счетчик увеличивается на единицу. При значении счетчика равном нулю семафор становится недоступным.

```
QMutex::lock() ~ QSemaphore::acquire()  
QMutex::unlock() ~ QSemaphore::release()
```

```
QSemaphore sem(1);  
Sem.acquire();
```



QWaitCondition – Ожидание условий

Класс **QWaitCondition**, так же обеспечивает возможность координации потоков.

Если поток намеревается дождаться разблокировки ресурса, то он вызывает метод **QWaitCondition::wait()** и, тем самым, входит в режим ожидания. Выводится он из этого режима в том случае, если поток, который заблокировал ресурс, вызовет метод **QWaitCondition::wakeOne()** или **QWaitCondition::wakeAll()**.

```
QString pop(const QString& str)
{
    QMutexLocker locker (&m_mutex);
    while (m_stackString.empty())
    {
        waitCondition.wait(&m_mutex);
    }
    return m_stackString.pop();
}
```

Разница этих двух методов в том, что первый выводит из состояния ожидания только один поток, а второй — все сразу. Также для потока можно установить время, в течение которого он может ожидать разблокировки данных. Для этого нужно передать в метод **wait()** целочисленное значение, обозначающее временной интервал в миллисекундах.

Блокировки чтения/записи.

Существуют особые случаи синхронизации потоков, при которых **некоторому количеству** потоков разрешено получать доступ к ресурсу **только для чтения**, а только **одному** разрешено получать доступ к ресурсу **для записи**. А также и такие случаи, когда один из потоков записывает данные и нужно, чтобы в это время ни один другой из потоков не смог получить доступ к этим данным для чтения. В подобных случаях лучше всего использовать класс **QReadWriteLock**.

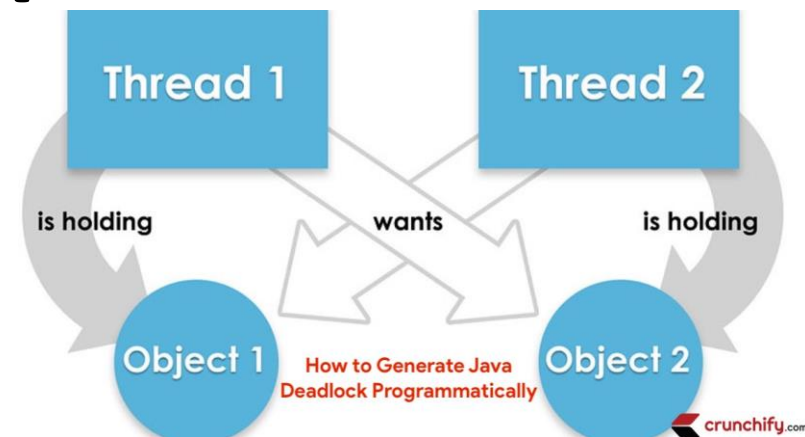
Этот класс предоставляет метод **lockForRead ()** - для блокировки чтения ресурса и метод **lockForWrite ()** - для блокировки записи ресурса.

```
QString ThreadSafeStringStack::push()
{
    readWriteLock . lockForRead();
    m_stackString.push(str);
    readWriteLock . unlock();
}
```

Взаимные блокировки - deadlock

Возможном возникновении тупиковых ситуаций, когда потоки могут заблокировать друг друга.

Поток заблокировал ресурс **A**, а после работы над ним собирается работать с ресурсом **B**. Другой же поток заблокировал ресурс **B** и по окончании намеревается работать с ресурсом **A**. И вот один из потоков, закончив работу, обнаружил, что нужный ему ресурс заблокирован другим потоком. Он переходит в режим ожидания, надеясь дожидаться разблокировки ресурса, но то же самое делает и другой поток. В итоге — оба ждут друг друга. Если ни один из этих потоков не освободит занятый им ресурс, то оба "зависнут" и не смогут продолжать свою работу дальше. Это явление получило название взаимной блокировки (**deadlock**).



Для решения можно так организовать работу потока, чтобы, в том случае, если поток не сможет получить доступ к необходимому ресурсу, он просто произвел бы освобождение занятых им ресурсов, а позже повторил попытку захвата необходимых ресурсов.

QtConcurrent

Для решения сложностей в работе с потоками в Qt было добавлено новое пространство имен *QtConcurrent*.

Это фреймворк высокого уровня, который создает уровень абстракции для управления потоками и синхронизацией. Он значительно упрощает написание мультипоточных приложений, что приводит к более быстрой разработке и уменьшению программного кода.

Для того чтобы его задействовать, необходимо включить в **pro-файл** строку:

QT += concurrent

В самом простом случае, когда нужно запустить функцию в отдельном потоке, ее нужно передать в качестве аргумента функции **QtConcurrent::run ()**.

```
#include <QtCore>
#include <QtConcurrent/QtConcurrent>
QString myToUpper(const QString& str){
    return str.toUpperCase();
}
int main(int argc, char** argv){
    QApplication app(argc, argv);
    QFuture<QString> future =
QtConcurrent::run(myToUpper, QString("test"));
    future.waitForFinished();
    qDebug () << future.result ();
    return 0;
}
```

```
int main(int argc, char** argv){
    QApplication app(argc, argv);
    QStringList lst (QStringList () << "one" << "two" << "three");
    QFuture<QString> future =
QtConcurrent::mapped(lst.begin(), lst.end(), myToUpper);
    future.waitForFinished();
    qDebug() << future.results();
    return 0;
}
```

Практика

1. «NewYearTimer». В проекте доделать счетчик времени для часов, минут, сек.

2. «Shell». Проверить работу в ОС Linux GNU.

3. «MyThread». Переместить все виджеты на одну форму.
Сделать закрывание формы по работе последнего/первого потока

4. «MultyThread». Изучить работу