Министерство науки и высшего образования Российской федерации

Томский государственный университет
систем управления и радиоэлектроники

К.О. Гусаченко,
А.О. Семкин

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

Методические указания к лабораторной работе
«Библиотека Qt. Процессы и потоки»

Томск
2020

УДК 004.428.4 + 519.683.8
ББК 32.972.1я73

Рецензент:
Перин А.С., доцент кафедры сверхвысокочастотной и квантовой радиотехники ТУСУР, канд. техн. наук

Гусаченко, Ксения Олеговна
Информационные технологии: Методические указания к лабораторной работе «Библиотека Qt. Процессы и потоки» / К.О. Гусаченко, А.О. Семкин. – Томск: Томск. гос. ун-т систем упр. и радиоэлектроники, 2020. – 21 с.

Настоящие методические указания составлены с учетом требований федерального государственного образовательного стандарта высшего образования (ФГОС ВО).

Изложены принципы управления процессами и потоками компьютерных приложений (программ) в среде разработки *Qt Creator*. Представлены методические материалы компьютерной лабораторной работы, посвященной разработке приложений, работающих с вызовом сторонних процессов и имеющих многопоточную структуру.

Предназначены для студентов очной и заочной форм обучения по направлению подготовки бакалавриата 11.03.02 «Инфокоммуникационные технологии и системы связи», 11.03.01 «Радиотехника».

Одобрено на заседании каф. СВЧиКР протокол №_6__ от __30.01.2020

УДК 004.428.4 + 519.683.8
ББК 32.972.1я73

Оглавление

1. Цель работы

Целью данной работы является изучение принципов управления процессами и потоками компьютерных программ при помощи средств разработки *Qt*.

2. Введение

Qt представляет собой комплексную рабочую среду, предназначенную для разработки на C++ межплатформенных приложений с графическим пользовательским интерфейсом по принципу «написал программу – компилируй ее в любом месте». *Qt* позволяет программистам использовать дерево классов с одним источником в приложениях, которые будут работать в системах Windows, Mac OS X, Linux. Solaris, HP-UX и во многих других.

Графический пользовательский интерфейс (GUI — Graphical User Interface) это средства позволяющие пользователям взаимодействовать с аппаратными составляющими компьютера комфортным и удобным для себя образом.

В рамках данной лабораторной работы будут рассмотрены принципы работы с процессами и потоками в среде *Qt Creator*.

3. Процессы и потоки в Qt

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.

Многопоточность позволяет разделять задачи и независимо работать над каждой из них для того, чтобы максимально эффективно задействовать процессор. Написание многопоточных приложений требует больше времени и усложняет процесс отладки, поэтому многопоточность нужно применять тогда, когда это действительно необходимо. Многопоточность удобно использовать для того, чтобы блокировка или зависание одного из методов не стали причиной нарушения функционирования основной программы.

Для использования многопоточности нужно унаследовать класс от *QThread* и перезаписать метод *run()*, в который должен быть помещен код для исполнения в потоке. Чтобы запустить поток, нужно вызвать метод *start()*.

Связь между объектами из разных потоков можно осуществлять при помощи сигналов и слотов или посредством обмена объектами событий.

При работе с потоками нередко требуется синхронизировать функционирование потоков. Причиной синхронизации является необходимость обеспечения доступа нескольких потоков к одним и тем же данным. Для этого библиотека *Qt* предоставляет классы *QMutex*, *QWaitContion* и *QSemaphore*.

В целях эффективности не все классы *Qt* обладают механизмом надежности для потоков.

3.1. *QProcess* – процессы в *Qt*

В том случае, когда пользователь или программа производят запуск другой программы, операционная система всегда создает новый процесс. Процесс — это экземпляр программы, загруженной в память компьютера для выполнения.

По своей сути, процессы — это независимые друг от друга программы, обладающие своими собственными данными. Коротко процесс можно охарактеризовать как общность кода, данных и ресурсов, необходимых для его работы. Под ресурсами подразумеваются объекты, запрашиваемые и используемые процессами в период их работы. Любая прикладная программа, запущенная на вашем компьютере, представляет собой не что иное, как процесс.

Создание процесса может оказаться полезным для использования функциональных возможностей программ, не имеющих графического интерфейса и работающих с командной строкой. Другое полезное свойство — довольно простой запуск других программ из текущей программы. Особенно он полезен для запуска команд или программ, действия которых непродолжительны по времени.

Процессы можно создавать с помощью класса *QProcess*, который определен в заголовочном файле *QProcess*. Благодаря тому, что этот класс унаследован от класса *QIODevice*, объекты этого класса в состоянии считывать информацию, выводимую запущенными процессами, и даже подтверждать их запросы на ввод информации. Этот класс содержит методы для манипулирования системными переменными процесса. Работа с объектами класса *QProcess* производится в асинхронном режиме, что позволяет сохранять работоспособность графического интерфейса программы в моменты, когда запущенные процессы находятся в работе. При появлении данных или других событий объекты класса *QProcess* высылают сигналы. Например, при возникновении ошибок объект процесса вышлет сигнал *error()* с кодом этой ошибки.

Для создания процесса его нужно запустить. 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 *exitStatis()*, который возвращает только два значения: *NormaiExit* (нормальное завершение) и *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()*.

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*.

Приложение, изображенное на рисунке 3.1, иллюстрирует применение некоторых методов класса *QProcess*. В текстовом поле ***Command*** (Команда) может быть введена любая команда, соответствующая операционной системе. Если запущенная команда или программа осуществляют вывод на консоль, то отображение будет производиться в виджете многострочного тестового поля.

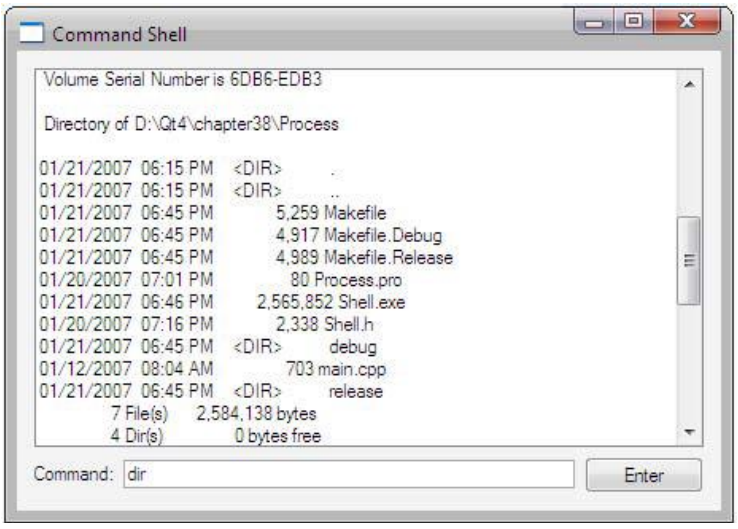

Рисунок 3.1 – Простейшая командная оболочка

Файл *shell.h*

```
class Shell : public QWidget {
Q_OBJECT
private:
    QProcess*  m_process;
    QLineEdit* m_ptxtCommand;
    QTextEdit* m_ptxtDisplay;
public:
    Shell(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        m_process     = new QProcess(this);
        m_ptxtDisplay = new QTextEdit;

        QLabel* plbl = new QLabel("&Command:");

        m_ptxtCommand = new QLineEdit("dir");
        plbl->setBuddy(m_ptxtCommand);

        QPushButton* pcmd = new QPushButton("&Enter");

        connect(m_process,
                SIGNAL(readyReadStandardOutput()),
```

```
                SLOT(slotDataOnStdout())
               );
        connect(m_ptxtCommand,
                SIGNAL(returnPressed()),
                SLOT(slotReturnPressed())
               );
        connect(pcmd, SIGNAL(clicked()), SLOT(slotReturnPressed()));

        //Layout setup
        QHBoxLayout* phbxLayout = new QHBoxLayout;
        phbxLayout->addWidget(plbl);
        phbxLayout->addWidget(m_ptxtCommand);
        phbxLayout->addWidget(pcmd);

        QVBoxLayout* pvbxLayout = new QVBoxLayout;
        pvbxLayout->addWidget(m_ptxtDisplay);
        pvbxLayout->addLayout(phbxLayout);
        setLayout(pvbxLayout);
    }

public slots:
    void slotDataOnStdout()
    {
        m_ptxtDisplay->append(m_process->readAllStandardOutput());
    }

    void slotReturnPressed()
    {
        QString strCommand = "";
#ifdef Q_WS_WIN
        strCommand = "cmd /C ";
#endif
        strCommand += m_ptxtCommand->text();
        m_process->start(strCommand);
    }
};
```

В конструкторе класса *Shell* производится создание объекта класса *QProgress*. Его сигнал *readyReadStandardOutput()* соединяется со слотом *slotDataOnStdout()*, в котором вызывается метод *readAllstandardOutput()* для считывания всего содержимого стандартного потока. После считывания эти данные добавляются, вызовом метода *append()*, в виджет многострочного текстового поля *m_ptxtDisplay*.

Слот *slotReturnPressed()* соединен с сигналом кнопки *clicked()* (указатель *pcmd*) и с сигналом однострочного текстового поля *returnPressed()* (указатель *m_txtCommand*). Некоторые команды *ОС Windows*, например — *dir*, не являются отдельными программами, поэтому они должны быть исполнены посредством командного интерпретатора *cmd*. Поэтому для *ОС Windows* в командную строку сначала добавляется строка *"cmd /C"*. Во всех остальных, введенная в однострочном текстовом поле строка передается как есть, без дополнений. Для запуска процесса вызывается метод *start()*.

3.2. *QThread* – потоки в *Qt*

Потоки становятся все более популярными. Для реализации потоков *Qt* предоставляет класс *QThread*. Но давайте сначала разберемся, что же собой представляют 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 (далее — основной поток). Конечно, 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. 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.

Приложения, имеющие один поток, могут выполнять только одну определенную операцию за один промежуток времени, а все остальные операции ждут ее окончания. Например, такие операции, как вывод на печать, считывание большого файла, ожидание ответа на посланный запрос или выполнение сложных математических вычислений, могут привести к блокировке или зависанию всей программы. При помощи многопоточности можно решить такую проблему, запустив подобные операции в отдельно созданных потоках. Тем самым, при зависании одного из потоков, функционирование основной программы не будет нарушено.

Среди разработчиков встречается разное отношение к многопоточному программированию. Некоторые стараются сделать все свои программы многопоточными, а других многопоточность пугает. Важно учитывать и то обстоятельство, что использование потоков существенно усложняет процесс отладки, а также написание самого приложения. Но при правильном и обоснованном применении многопоточности можно существенно улучшить скорость приложения. Неправильное же ее применение, наоборот, может привести к снижению скорости приложения. Поэтому использование потоков должно быть обоснованным. А это значит, что если вы не можете сформулировать причину, по которой следует сделать приложение многопоточным, то лучше отказаться от использования многопоточности. Если вы сомневаетесь, то постарайтесь, на начальных стадиях, создавать два прототипа для тестирования — один многопоточный, а другой нет. Тем самым, прежде чем тратить время на разработку программы, можно

определить, является ли многопоточность решением поставленной задачи или нет. В том случае, если достижения того же результата можно добиться без использования многопоточности, то стоит отдать предпочтение последнему варианту. Но перед тем как начать написание программы с использованием многопоточности, очень важно разобраться и понять фундаментальные принципы и подходы программирования потоков, применяемые в *Qt*.

Так с чего же все-таки начинается многопоточное программирование? Оно начинается с наследования класса *QThread* и переопределении в нем чисто виртуального метода *run()*, в котором должен быть реализован код, который будет исполняться в потоке.

Например:

```
class MyThread : public QThread
{

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

Второй шаг заключается в создании объекта класса потока и вызове метода *start()*, который вызовет, в свою очередь, реализованный нами метод *run()*.

Например:

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

4. Приоритеты потоков в *Qt*

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

- в первую входят четыре наиболее часто применяемых приоритета. Их значимость распределяется по возрастанию — *IdlePriority*, *LowestPriority*, *LowPriority*, *NormaiPriority*. Они подходят для решения задач, которым процессор требуется только время от времени, например, для фоновой печати или для каких-нибудь несрочных действий;
- во вторую группу входят два приоритета — *HighPriority*, *HighestPriority*. Пользуйтесь такими приоритетами с большой осторожностью. Обычно эти потоки большую часть времени ожидают какие-либо события;
- в третью входят два приоритета — *TimeCriticalPriority*, *InheritPriority*. 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.

Для того чтобы запустить поток с нужным приоритетом, необходимо передать одно из приведенных выше значений в метод *start()*.

Например:

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

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

В *Linux* ядро системы запустит поток даже в том случае, если в нем уже запущены хоть 100 потоков с максимальным приоритетом. Ядро системы не потеряет работоспособность, и вы будете в состоянии запускать и другие потоки. Но в *Windows* все обстоит иначе, если вы запустите поток с самым высоким приоритетом, то поток с самым низким приоритетом будет просто проигнорирован и это может быть плачевно для тех, кто использует потоки с самым низким приоритетом, для проведения каких-либо вспомогательных операций, например, загрузки данных с диска.

5. Обмен сообщениями между потоками

Один из важнейших вопросов при многопоточном программировании — это обмен сообщениями. Действительно, если вы, например, в одном потоке создаете растровое изображение и хотели бы переслать его объекту другого потока, то каким образом вы можете это сделать?

Каждый поток может иметь свой собственный цикл событий. Благодаря этому можно осуществлять связь между объектами. Такая связь может производиться двумя способами: при помощи соединения сигналов и слотов или обмена событиями.

Рисунок 5.1 – Потоки с объектами и собственными циклами обработки информации

Примечание. Если сигнально-слотовое соединение осуществляется между объектами разных потоков, то внутри оно преобразуется в событие.

Использование в потоках собственных циклов обработки событий

позволяет снять ограничение, связанное с применением классов, способных работать только в одном цикле событий, и использовать, параллельно, нужное количество объектов этих классов. К таким классам относятся класс *QTimer* и сетевые классы.

Для того 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()*.

Класс *QObject* реализован так, что обладает близостью к потокам. 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()*.

5.1. Сигнально-слотовые соединения

Итак, мы можем взять сигнал объекта одного потока и соединить его со слотом объекта другого потока. Как мы уже знаем, 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.

Сигналы и слоты в *Qt* реализованы с механизмом надежности работы в потоках, а это означает, что вы можете высылать сигналы и получать, не заботясь о блокировке ресурсов. Вы можете перемещать объект, созданный в одном потоке, в другой. Если вдруг получится так, что высылающий объект окажется в одном потоке с принимающим, то высылка сигнала будет сведена к прямой обработке соединения.

Программа, показанная на рисунке, демонстрирует использование сигналов и слотов в потоке. После ее запуска производится отсчет таймера от 10 к 0, после чего программа завершает свое выполнение.

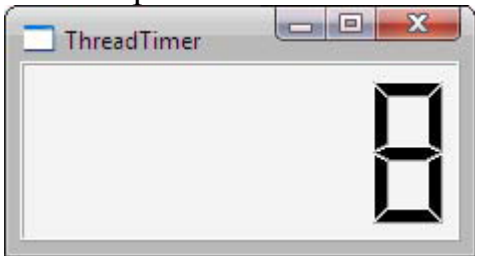

Рисунок 5.2 – Сигнально-слотовые соединения в потоках

Файл *MyThread.h*. Класс *MyWorker*.

```
class MyThread : public QThread {
Q_OBJECT
private:
    int m_nValue;

public:
    MyThread() : m_nValue(10)
    {
    }

    void run()
    {
        QTimer timer;
        connect(&timer, SIGNAL(timeout()), SLOT(slotNextValue()));
        timer.start(1000);

        exec();
    }

signals:
    void finished    (   );
    void currentValue(int);

public slots:
    void slotNextValue()
    {
        emit currentValue(--m_nValue);

        if (!m_nValue) {
            emit finished();
        }

    }
};
```

Класс *MyThread* представляет собой класс для управления потоком. Он должен быть унаследован от класса *QThread*. Обратите внимание, что в определении нашего класса потока указан макрос *Q_OBJECT*, что необходимо для использования сигналов и слотов. Конструктор нашего класса пуст и служит только для инициализации атрибута *m_nValue*. Одна из самых интересных частей нашего класса — это метод *run()*, который должен быть переопределен — поместим в него код, выполняющийся в потоке. В этом методе мы создаем объект таймера и соединяем его сигнал *timeout()* со слотом *slotNextValue()*, запускаем таймер с интервалом в одну секунду и приводим в действие цикл обработки событий вызовом метода *exec()*.

Вас не должно смущать то, что объект таймера (объект *timer*) был создан статически, а не динамически с помощью оператора new, так как метод *run()* является методом для исполнения кода в потоке и его разрушение произойдет только при завершении работы потока. После вызова метода *exec()* произойдет запуск цикла событий, который произведет блокировку исполнения всех дальнейших команд метода *run()*, если бы таковые имелись. Этот метод можно образно сравнить с функцией *main()*, ведь в ней мы

поступаем аналогичным образом, когда реализуем основной поток приложения, без которого не была бы возможна работа ни одного *Qt*-приложения с пользовательским интерфейсом.

Слот *slotNextValue()* уменьшает значение атрибута *m_nValue* на единицу и высылает сигнал *currentValue()* с его актуальным значением. Если значение станет нулевым, то будет произведена высылка сигнала *finished()*, информирующая о конце работы.

Файл *main.cpp*. Функция *main()*.

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLCDNumber   lcd;
    MyThread     thread;

    QObject::connect(&thread, SIGNAL(currentValue(int)),
                     &lcd,    SLOT(display(int))
                    );
    QObject::connect(&thread, SIGNAL(finished()),
                     &app,    SLOT(quit())
                    );

    lcd.setSegmentStyle(QLCDNumber::Filled);
    lcd.display(10);
    lcd.resize(220, 90);
    lcd.show();
    thread.start();

    return app.exec();
}

#include "main.moc"
```

В основной программе, мы создаем виджет электронного индикатора *lcd* и объект потока *thread*. Для отображения значений, высылаемых потоком, мы соединяем сигнал *currentValue()* со слотом виджета индикатора *display()*. Также мы соединяем сигнал *finished()* объекта потока со слотом приложения *quit()* для того, чтобы приложение завершило свою работу. После некоторых настроек, влияющих на показ виджета электронного индикатора, мы запускаем поток методом *start()*. Ввиду того, что весь исходный код у нас реализован в одном файле *main.cpp*, нам необходимо включить метаинформацию, которая будет сгенерирована *MOC*, при помощи директивы *include*.

В только что продемонстрированном примере, слоты нашего потока не соединяются с объектами другого потока, и, в этом случае, производится прямая высылка сигналов без внутреннего генерирования событий. Теперь давайте изменим наш пример таким образом, чтобы слот нашего потока был соединен с сигналом из другого потока. Для этого просто уберите три строки, связанные с таймером, из метода *run()* и перенесите их в реализацию основного потока (в функцию *main()*). Метод *run()* должен выглядеть следующим образом:

```
oid run()
{
    exec();
}
```

А в функцию *main()* добавьте строки:

```
QTimer timer;
QObject::connect(&timer,          SIGNAL(timeout          O),          &thread,
SLOT(slotNextValue()));
timer.start(1000);
```

Теперь наш поток соединен с объектом таймера, созданным в основном потоке, и отсылка сигналов должна приводить к внутренней генерации событий. Откомпилируйте и запустите пример. Вы убедитесь в том, что он работает так же, как и его предшественник.

Если поток должен только высылать сигналы, то запуск цикла обработки событий не нужен. Теперь давайте создадим более простой пример потока, без цикла сообщений. В приложении, показанном на рисунке, используется поток, высылающий сигналы со значениями для индикатора прогресса.

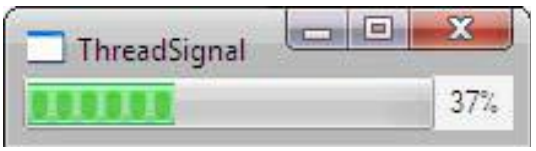

Рисунок 5.3 – Поток, отправляющий сигналы

Файл main.cpp. Класс управления потоком MyThread

```
class MyThread : public QThread {
Q_OBJECT

public:
    void run()
    {
        for (int i = 0; i <= 100; ++i) {
            usleep(100000);
            emit progress(i);
        }
    }

signals:
    void progress(int);
};
```

В основном методе нашего потока *run()* мы запускаем цикл от 0 до 100, в котором через каждые 10 миллисекунд высылается сигнал *progress()* с актуальным значением переменной цикла. Остановка выполнения цикла на десять миллисекунд производится при помощи метода *QThread::usleep()*.

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

```
    MyThread     thread;

    QObject::connect(&thread, SIGNAL(progress(int)),
                     &prb,    SLOT(setValue(int))
                    );

    prb.show();

    thread.start();

    return app.exec();
}

#include "main.moc"
```

После создания виджета прогресса производится создание одного потока — *thread*, а его сигнал *progress()* соединяется, для отображения высылаемых им значений, со слотом *setValue()* виджета индикатора прогресса. Запуск потока приводится в действие методом *start()*.

5.2. Связь между потоками с помощью высылки событий

Высылка событий — это еще одна из возможностей для осуществления связи между объектами. Как мы знаем, есть два метода для высылки событий: *QCoreApplication::postEvent()* и *QCoreApplication::sendEvent()*. Здесь есть небольшой нюанс, который нужно знать: высылка событий методом *postEvent()* обладает надежностью в потоках, а при помощи метода *sendEvent()* — нет. Поэтому при работе с разными потоками всегда используйте метод *postEvent()*. На рисунке показано, как с помощью механизма обмена событиями разных потоков можно осуществлять связь между двумя потоками. 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.

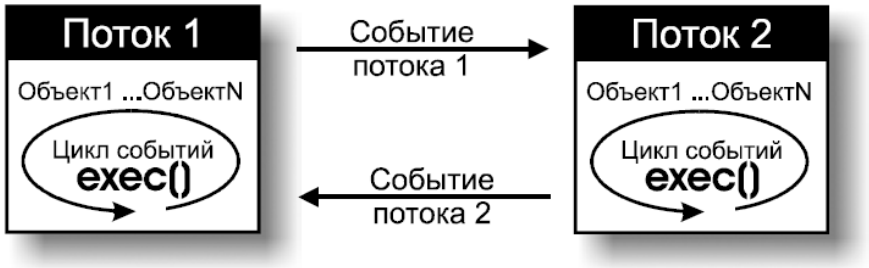

Рисунок 5.4 – Обмен событиями

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

Если поток предназначен исключительно для высылки событий, а не для их получения, то реализацию методов обработки событий и запуск цикла обработки событий можно опустить. Для сравнения высылки событий с высылкой сигналов давайте реализуем программу высылки событий из потока.

Файл *main.cpp*. Класс события *ProgressEvent*

```
class ProgressEvent : public QEvent {
private:
    int m_nValue;

public:
    enum {ProgressType = User + 1};

    ProgressEvent() : QEvent((Type)ProgressType)
    {
    }

    void setValue(int n)
    {
        m_nValue = n;
    }

    int value() const
    {
        return m_nValue;
    }
};
```

Первое, что нам нужно сделать, — это создать класс для нашего события, которое мы будем высылать из потока. Наш класс наследуется от класса *QEvent* и определяет атрибут целого типа *m_nValue*. Для получения и установки значений этого атрибута класс содержит метод *value()* и *setValue()*. В конструкторе мы задаем тип нашего события, передавая его целочисленный идентификатор в конструктор класса *QEvent()*.

Файл *main.cpp*. Класс потока *MyThread*

```
class MyThread : public QThread {
private:
    QObject* m_pobjReceiver;

public:
    MyThread(QObject* pobjReceiver) : m_pobjReceiver(pobjReceiver)
    {
    }

    void run()
    {
        for (int i = 0; i <= 100; ++i) {
            usleep(100000);

            ProgressEvent* pe = new ProgressEvent;
            pe->setValue(i);
            QApplication::postEvent(m_pobjReceiver, pe);
        }
    }
};
```

Определяем в классе атрибут, указывающий на объект-получатель нашего события, создаем объект класса события и высылаем его объекту-получателю с помощью метода *postEvent()*. Создание нашего события в методе *run()* очень похоже на утечку памяти (*memory leak*), но ею не является, так как

после обработки все объекты событий удаляются.

Примечание. Очевидно, что если бы нам понадобилось выслать событие еще одному объекту, то нам пришлось бы создать второй объект класса и повторить действия, проделанные для первого события. Для третьего пришлось бы еще раз повторить код и т. д. При высылке сигналов нам этого делать не нужно, так как объект-получатель задается методом *QObject::connect()*. Поэтому решение с использованием сигналов, в данном случае, будет смотреться более элегантно. Мы всего лишь один раз высылаем сигнал, вне зависимости от числа его получателей.

Файл *main.cpp*. Класс виджета индикации процесса *MyProgressBar*

```
class MyProgressBar : public QProgressBar {
public:
    MyProgressBar(QWidget* pwgt = 0) : QProgressBar(pwgt)
    {
    }

    void customEvent(QEvent* pe)
    {
        if ((int)pe->type() == ProgressEvent::ProgressType) {
            setValue(((ProgressEvent*)(pe))->value());
        }
        QWidget::customEvent(pe);
    }
};
```

Для того чтобы виджет был в состоянии получать и правильно интерпретировать высылаемые потоком события, у нас есть две возможности. Первая возможность заключается в реализации класса фильтра событий и установки его в виджете. Второй способ заключается в наследовании класса виджета и перезаписи в нем метода *customEvent()*.

В методе *customEvent()* мы проверяем тип полученного события, и если он соответствует типу нашего события *ProgressType*, то мы приводим его к классу *ProgressEvent* для того, чтобы вызвать метод *value()*. Возвращаемое этим методом значение передается, для отображения виджетом индикации прогресса, методу *QProgressBar::setValue()*.

Файл *main.cpp*. Функция *main()*

```
int main(int argc, char** argv)
{
    QApplication  app(argc, argv);
    MyProgressBar prb;
    MyThread      thread(&prb);

    prb.show();

    thread.start();

    return app.exec();
}
```

В основной программе мы создаем виджет индикации прогресса и объект

потока, после чего производим запуск потока вызовом метода *start()*.

Если сравнить реализации программы при помощи сигналов и событий, то заметно, что подход с использованием сигналов более компактный. Но, в целом, оба подхода равнозначны и имеют право на существование. Какой из подходов будет использоваться зависит от вас, насколько удобным вы найдете их применение в конкретных ситуациях.

6. Методические указания по выполнению работы

Лабораторную работу необходимо выполнить в следующей последовательности:

1. Изучите теоретические разделы 2-5, выполните на компьютере все приведенные в данных разделах примеры, скомпилируйте и запустите на исполнение полученные программы. В качестве дополнительных источников информации, воспользуйтесь указанными в перечне используемой литературы (раздел 7) пособиями, а также открытыми источниками в сети Интернет.
2. Самостоятельно выполните задание, описанное ниже в разделе 6.1.
3. Подготовьте исходный код программ в Qt Creator покажите его преподавателю.
4. В случае отсутствия видимых ошибок в коде, скомпилируйте программы и запустите их на выполнение.
5. Получите оценку.

6.1. Индивидуальное задание

Полученная в п. 3.1 программа может быть использована для копирования и сохранения различной системной информации, поскольку получает доступ к системной командной строке. Такой функционал может быть оценен как вредоносный, поскольку позволяет прикладному приложению получать информацию о системе, на которой оно запускается.

Дополните программу, полученную в п. 3.1 следующим функционалом:

1. В описание класса Shell добавьте слот `void slotInfoSteal();`.
2. В реализации слота `void slotInfoSteal()` последовательно:
   - закройте процесс (метод `close()`);
   - создайте файл (класс `QFile`) и откройте его в режиме записи;
   - создайте текстовый поток ввода-вывода и привяжите его к созданному файлу (`QTextStream stream(&file);`)
   - запустите процесс с текстовой строкой `"cmd /C systeminfo";`
   - выполнение сбора данных о системе требует времени, поэтому нужно выдержать паузу перед считыванием данных из процесса с помощью метода `m_process->waitForReadyRead(3000);`
   - считайте данные из процесса и запишите их в созданный текстовый поток ввода/вывода;
   - очистите текстовое поле программы (`m_ptxtDisplay->clear();`);

- закройте процесс (`m_process->close();`)

3. Поместите вызов созданного слота в конец конструктора класса Shell.
4. Проверьте содержимое создаваемого при запуске программы файла, он должен содержать сведения о системе пользователя. При этом процесс копирования этих сведений в файл от пользователя скрыт.

7. Рекомендуемая литература

1. Qt 5.3. Профессиональное программирование на C++: Наиболее полное руководство / М. Шлее. - СПб. : БХВ-Петербург, 2015. – 915 с.
2. Бланшет Ж., Саммерфилд М. Qt 4: программирование GUI на C++. Пер. с англ. 2-е изд., доп. – М.: КУДИЦ-ПРЕСС, 2008. – 736 с.
3. Процессы и потоки. [Электронный ресурс]. - Режим доступа: http://qt-doc.ru/qt-process.html (дата обращения: 02.02.2020).