РАБОТА С ПОТОКАМИ В QT 5 И ОБРАБОТКА КРИТИЧЕСКИХ СИТУАЦИЙ В МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЯХ

THREADING IN QT 5 AND TREATMENT OF CRITICAL SITUATIONS IN MULTI-THREADED APPLICATIONS

R. Bondarenko
A. Romanenkov

Summary. The Development of multithreaded cross-platform and productive applications is particularly relevant today, as modern services and resource-intensive methods due to their efficiency. High productivity is one of the main factors that affects the quality of the product sold. The speed of the application is directly related to the speed of execution of tasks, which can be increased by parallelizing them using multiple threads. When working with threads, there are many different subtleties. Some of them are working with threads under different operating systems, thread synchronization, memory access control. Their knowledge and understanding in the development is the key to a successful final product.

Keywords: Flow, Multithreading, Qt Framework 5, cross-Platform, Bystrodeistviya, Performance.

Бондаренко Роман Николаевич
Национальный Исследовательский Университет, Московский авиационный институт, Москва
rnikbond@gmail.com
Романенков Александр Михайлович
К.т.н., доцент, Национальный Исследовательский Университет, Московский авиационный институт, Москва
romanaleks@gmail.com

Аннотация. Разработка многопоточных кроссплатформенных и производительных приложений сегодня особо актуальна, так как современные сервисы и методы ресурсоемки в силу своей эффективности. Высокая производительность является одним из основных факторов, который влияет на качество реализуемого продукта. Быстродействие приложения прямо связано со скоростью выполнения поставленных задач, которую можно увеличить за счет их распараллеливания, используя несколько потоков. При работе с потоками существует много различных тонкостей. Некоторые из них — это работа с потоками под различными операционными системами, синхронизация потоков, управление доступа к памяти. Их знание и понимание при разработке является залогом успешного конечного продукта.

Ключевые слова: Поток, Многопоточность, Фреймворк Qt 5, Кроссплатформенность, Быстродейтсвие, Производительность.

Введение

Быстродействие — это один из важнейших критериев любого современного приложения. Сбалансированные алгоритмы с максимальной производительностью повышают скорость работы приложения, но чаще всего они выполняются последовательно, в одном потоке. Используя несколько потоков в одном приложении можно значительно увеличить его производительность за счет распараллеливания выполняемых задач. Иными словами, перемещая задачи в отдельные потоки, от этой обязанности освобождается главный поток, в следствии чего работа происходит параллельно, что позволяет уменьшить необходимое время для выполнения задач и, тем самым, увеличить производительность. Приложения, в которых работает два и более потоков, называются многопоточными. Но, помимо положительной стороны — увеличения производительности, есть и другая сторона — сложность разработки и отладки при синхронизации потоков, управлением одновременного доступа к памяти из нескольких потоков и использованием потоков под различными операционными системами. Фреймворк *Qt 5.** является одним из лучших на сегодняшний день для реализации производительного и кроссплатформенного приложения.

Потоки в Qt 5.* и их реализация

Существует два типа потоков: *thread* и *stream*. При создании многопоточного приложения, ориентированного на быстродействие, используются потоки типа *thread*. Поток (*thread*) — это независимая задача, которая выполняется внутри процесса и разделяет вместе с ним общее адресное пространство, код и глобальные данные. Каждый поток имеет свой собственный стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры и регистры, в которых содержатся текущие рабочие переменные [1]. В отличии от процессов, потоки используют одно и то же адресное пространство (*список адресов в памяти, с которыми происходит работа*).

В *Qt 5.** существует несколько классов, которые позволяют создавать потоки и управлять ими. *QThread* — это основной класс для работы с потоками, в котором реализован весь необходимый функционал, позволяющий управлять потоками независимо от платформы, на которой работает приложение. Данный класс уже включает в себя оболочку над более низкоуровневой реализацией протоков, что значительно упрощает ра-

боту с ними. Рассмотрим несколько способов создания нового потока.

Первый способ — это наследование от класса *QThread*. Новый поток начинает работу после вызова функции *start(Prioritet)*. Для более детального понимания этого способа ниже приведен фрагмент исходного кода функции *start(...)*, в которой поток создается и начинает свою работу [2]:

```
//! —- qthread.h —-!//
class Q_CORE_EXPORT QThread: public QObject {
…
// Приоритет потока
enum Priority {
IdlePriority, // 0 — Планируется тогда, когда другие потоки не запущены.
LowestPriority, // 1 — Запланировано реже, чем LowPriority.
LowPriority, // 2 — Запланировано реже, чем NormalPriority.
NormalPriority, // 3 — Приоритет по умолчанию операционной системы.
HighPriority, // 4 — Запланировано чаще, чем NormalPriority.
HighestPriority, // 5 — Запланировано чаще, чем HighPriority.
TimeCriticalPriority, // 6 — Запланировано как можно чаще.
InheritPriority // 7 — Используйте тот же приоритет, что и поток создани
};
//! —- qthread.cpp —-!//
void QThread:: start(Priority priority) {
Q_D(QThread);
QMutexLocker locker(&d->mutex);
if (d->isInFinish) {
locker.unlock();
wait();
locker.relock();
}
if (d->running) return;
d->running = true;
d->finished = false;
d->exited = false;
d->returnCode = 0;
d->interruptionRequested = false;
d->handle = CreateThread(nullptr, d->stackSize,
reinterpret_cast<LPTHREAD_START_ROUTINE>(QThreadPrivate:: start),
this, CREATE_SUSPENDED, reinterpret_cast<LPDWORD>(&d->id));
if (!d->handle) {
qErrnoWarning("QThread:: start: Failed to create thread");
d->running = false;
d->finished = true;
return;
}
int prio;
d->priority = priority;
switch (d->priority) {
case IdlePriority: prio = THREAD_PRIORITY_IDLE; break;
case LowestPriority: prio = THREAD_PRIORITY_LOWEST; break;
case LowPriority: prio = THREAD_PRIORITY_BELOW_NORMAL; break;
case NormalPriority: prio = THREAD_PRIORITY_NORMAL; break;
case HighPriority: prio = THREAD_PRIORITY_ABOVE_NORMAL; break;
case HighestPriority: prio = THREAD_PRIORITY_HIGHEST; break;
case TimeCriticalPriority: prio = THREAD_PRIORITY_TIME_CRITICAL; break;
case InheritPriority:
default: prio = GetThreadPriority(GetCurrentThread()); break;
}
if (! SetThreadPriority(d->handle, prio))
qErrnoWarning("QThread:: start: Failed to set thread priority");
if (ResumeThread(d->handle) == (DWORD) –1)
qErrnoWarning("QThread:: start: Failed to resume new thread");
}
```

В функции *start(Proirity priority)* поток создается при помощи вызова функции *CreateThread(...)*, которая реализована в библиотеке *windows.h* операционной системы *Windows*, или при помощи вызова функции *pthread_create(...)*, реализованной в библиотеке *pthread.h*, если разработка происходит в системе *unix*. В реализации функции также используется переменная *d* — объект класса *QThreadPrivate*, при обращении к членам которой устанавливаются свойства потока.

Ниже представлен прототип функции *CreateThread(...)*, где третьим параметром передается функция потока [3]:

```
HANDLE CreateThread (
LPSECURITY_ATTRIBUTES, // дескриптор защиты
SIZE_T, // начальный размер стека
LPTHREAD_START_ROUTINE, // функция потока
LPVOID, // параметр потока
DWORD, // опции создания
LPDWORD // идентификатор потока
);
```

Таблица 1. Приоритеты потока

Название приоритета	Значение приоритета
THREAD_PRIORITY_IDLE	Уровень приоритета 1. Поток работает только тогда, когда операционная система простаивает. Поток с таким приоритетом также его можно назвать «простаивающим».
THREAD_PRIORITY_LOWEST	Уровень приоритета 2. Самый низкий уровень. Такой приоритет характерен для поток, работающих в фоновом процессе. Устанавливая такой приоритет потоку, при нагрузке процессора операционная система приостановит его для выполнения потоков с наиболее высшим приоритетом.
THREAD_PRIORITY_BELOW_NORMAL	Уровень приоритета 3. Приоритет ниже нормального. Такой приоритет также характерен для потоков процесса, работающего в фоновом режиме.
THREAD_PRIORITY_NORMAL	Уровень приоритета 4. При указании такого приоритета новый поток будет создан с таким же приоритетом, как и у потока, создавшего его.
THREAD_PRIORITY_ABOVE_NORMAL	Уровень приоритета 5. Такой приоритет выше нормального.
THREAD_PRIORITY_HIGHEST	Уровень приоритета 6. Наивысший приоритет потока.
THREAD_PRIORITY_TIME_CRITICAL	Уровень приоритета 15. Максимальный приоритет потока.

В представленной выше реализации функции *QThread:: start(...)* при вызове функции *CreateThread(...)* третьим параметром передается статический метод *QThreadPrivate:: start(...)*, объявление и реализация которого представлена ниже.

```
class QThreadPrivate: public QObjectPrivate {
…
static unsigned int __stdcall start(void *) Q_DECL_
NOEXCEPT;
…
}
unsigned int __stdcall QT_ENSURE_STACK_
ALIGNED_FOR_SSE QThreadPrivate:: start(void
*arg) Q_DECL_NOEXCEPT {
QThread *thr = reinterpret_cast<QThread *>(arg);
…
emit thr->started(QThread:: QPrivateSignal());
QThread:: setTerminationEnabled(true);
thr->run(); // Вызов функции run()
finish(arg);
return 0;
}
```

В реализации метода *QThreadPrivate:: start(void *arg)* обратим внимание на подчеркнутую строку в коде — в этой строке происходит вызов функции *run()*, в которой и заключается основная идея описываемого способа создания и использования нового потока.

После наследования от класса *QThread* нужно переопределить функцию *run()* и реализовать в ней тот функционал, который должен выполняться в новом потоке.

Также важным свойством является приоритет потока. Этот свойство передается в виде параметра при вызове функции *start(Prioritet prioritet)* и устанавливается в функции *SetThreadPriority(...)*, которая реализована в библиотеке *windows.h*. Прототип функции *SetThreadPriority(...)* представлен ниже:

```
BOOL SetThreadPriority (
HANDLE hThread,
int nPriority
);
```

Функция принимает два параметра: *HANDLE hThread* — дескриптор потока, значение приоритета которого должно быть установлено и *int nPriority* — определяет значение приоритета для потока. Перечисление *enum Priority*, объявленное в классе *QThread*, и является лишь оберткой над названиями приоритетов, что показано в конструкции *switch* реализации функции *start(...)* класса *QThread*. В этой конструкции объект *d* класса *QThreadPrivate* принимает значение приоритет *(prio)* одного из макросов, представленных в таблице 1.

Приоритет для потока нужно устанавливать исходя из той задачи, для которой он предназначен. Приоритеты до *THREAD_PRIORITY_NORMAL* следует устанавливать для потоков, которые выполняют задачи, незначительно влияющие на быстродействие и работоспособность приложения. Обычно такие задачи называют фоновыми. Более внимательно следует относиться при установке приоритета начиная с *THREAD_PRIORITY_HIGHEST*. Если поток выполняется с наивысшим уровнем приоритета в течение долгого времени, то для других потоков операционная система не сможет выделить процессорное время для выполнения. Если несколько потоков, принадлежащих одному процессу, имеют высокий приоритет, то произойдет потеря эффективности, и, как следствие, производительности, поскольку опера-

ционная система будет приостанавливать каждый из потоков по очереди для назначения процессорного времени для выполнения поставленной задачи. Высокий приоритет должен быть определен для потоков, которые должны реагировать на критичные по времени события (например, обработка пользовательского ввода). Если поток в приложении выполняет одну высокоприоритетную задачу, а остальные потоки имеют обычный приоритет выполнения, то в этом случае стоит временно повысить приоритет потока. Затем, когда поток выполнит поставленную задачу, приоритет следует понизить как минимум до *THREAD_PRIORITY_NORMAL*.

Понижение приоритета не означает, что поток всегда будет последним, выбранным для получения временного интервала. Для каждого потока предназначено процессорное время, зависящее от его приоритета. Поток с высоким приоритетом будет выполняться дольше и чаще по сравнению с потоком, у которого низкий приоритет. Кроме того, при изменении состояния основного потока в режим ожидания, в это время будут запущены другие потоки, даже если процессорное время основного потока не еще вышло. При использовании многоядерного процессора, операционная система распределит нагрузку между ядрами процессора. Независимо от того, насколько высокий приоритет имеет поток, он будет работать только на одном ядре.

Второй способ создания нового потока — это использование функции *moveToThread(...)*. Эта функция реализована в классе *QObject*. Пример реализации может быть следующего вида:

```
QThread* newThread = new QThread;
Worker* worker = new Worker; // Worker — класс, наследуемый от QObject
worker->moveToThread(newThread);
newThread ->start();
```

В данном примере создается объект класса *QThread*. Важное примечание — объект, перемещаемый в отдельный поток не должен иметь родительского объекта [2]:

```
void QObject:: moveToThread(QThread *targetThread)
{
…
if (d->threadData->thread == targetThread) // object is already in this thread
return;
if (d->parent!= 0) {
qWarning("QObject:: moveToThread: Cannot move objects with a parent");
return;
}
if (d->isWidget) {
qWarning("QObject:: moveToThread: Widgets cannot be moved to a new thread");
return;
}
…
}
```

Как видно из фрагмента исходного кода реализации функции *moveToThread(...)*, перемещение объекта в новый поток не произойдет по трем условиям:

1. Если объект уже находится в этом потоке.
2. Если у перемещаемого объекта есть родительский поток, что реализовано в строке *if (d->parent!= 0) {...; return;}*. Данное условие необходимо по следующей причине — объект, который перемещается в новый поток, больше не принадлежит потоку, создавшего его. Если же перемещаемый объект будет иметь родительский объект, то при удалении родительского объекта он будет должен удалить все дочерние объекты. Как следствие, при наличии родительского объекта он удалит дочерний объект, который находится в другом потоке, что может привести к нарушению логики работы программы, и, как следствие, ее некорректной работе.
3. Если объект наследован от класса *QWidget*, поскольку работа с объектами графического интерфейса *GUI* возможна только из главного потока приложения.

При создании объекта происходит выделение памяти под него. Поскольку память была выделена — ее нужно освободить. Так как объект, перемещаемый в новый поток, не имеет родительского объекта, то при его удалении автоматически дочерний объект не будет удален. Для избежания учетки памяти следует использовать конструкцию соединения сигнала *quit()* со слотом *deleteLater()*: *connect(newThread, SIGNAL(quit()), newThread, SLOT(deleteLater()))*.

Когда поток завершит свое выполнение, он будет удален, и будут удалены объекты, которые ему принадлежат.

При использовании таймеров необходимо обратить внимание на то, что все активные таймеры для объекта при его перемещении в новый поток будут сброшены. Таймеры сначала останавливаются в текущем потоке и перезапускаются с тем же интервалом в новом потоке. В результате, если постоянно перемещать объект между потоками, то может произойти ситуация, когда события таймера постоянно будут откладываться.

Оба описанных способа создают новый поток, и при использовании любого из них приложение будет уже являться многопоточным. Очевидно, что создание различ-

Таблица 2. Значения priv_ptr

Значение *priv_ptr*	Смысловое значение
0x0	Мьютекс разблокирован
0x1	Мьютекс заблокирован и нет потоков, ожидающих разблокировку
Другой адрес	Указатель на существующий *QMutexPrivate*

ного функционала с одинаковым результатом смысла не имеет, поэтому есть существенное различие, которое необходимо учитывать при разработке.

Различие при использовании этих способов заключается в использовании системы слотов(*slots*) и сигналов(*signals*). Сигналы *Qt* создают события типа *QMetaCallEvent*. Сигнало-слотовая система по-разному работает в разных потоках.

Как известно, для связывания сигнала и слота используется статическая функция *connect(…)*, реализованная в классе *QObject*, одним из аргументов которой является перечисление типа *Qt:: ConnectionType*. Прототип данной функции представлен ниже.

```
QObject:: connect(
const QObject* sender,
const char* signal,
const QObject* receiver,
const char* slot,
Qt:: ConnectionType type = Qt:: AutoConnection
);
```

При связывании слота и сигнала, которые находятся в разных потоках, важны два элемента перечисления: *DirectConnection* и *QueuedConnection*. При использовании *DirectConnection*, когда испускается сигнал, то немедленно вызывается в каждый связанный с ним слот. Данная схема аналогична обычному вызову функции. В случае же использования *QueuedConnection*, при испускании сигнала будет создано событие при помощи метода *postEvent(…)*, в результате чего оно поместится в очередь событий, и связанный с сигналом слот будет вызван только тогда, когда до него дойдет очередь.

При использовании первого способа — наследование от класса *QThread* и переопределении метода *run()*, используется тип соединения *QueuedConnection*. При использовании второго способа — *moveToThread(…)*, используется тип соединения *DirectConnection*.

При реализации многопоточного приложения необходимо обратить внимание на безопасную работу с ресурсами. Поскольку все потоки приложения используют единое адресное пространство процесса, то может возникнуть конфликтная ситуация, когда несколько потоков обращаются к одной области памяти. Например, два потока одновременно изменяют значение одной и той же переменной. Такие операции довольно проблематично отслеживаются в процессе отладки. Для избежания подобного рода конфликтов используется класс *QMutex*.

Класс *QMutex* блокирует одновременный доступ к ресурсам из нескольких потоков, в следствии чего критическая секция будет использоваться только одним потоком. Блокировка ресурсов осуществляется в методе класса *QMute:: lock()*. Разблокировка доступа к памяти осуществляется в методе этого же класса *unlock()*. Также в классе *QMutex* есть метод *tryLock()*, в котором проверяется, заблокирован ли в данный момент ресурс.

```
class QMutex
{
…
QAtomicPointer<QMutexPrivate> *priv_ptr;
…
};
```

В примере реализации выше класса *QMutex* показана самая идейная часть класса — указатель *priv_ptr*. Суть заключается в том, что этот указатель не всегда указывает на существующий *QMutexPrivate*. Данный указатель может принимать значения, перечисленные в таблице 2 ниже.

Блокировка ресурсов в классе *QMutex*. Разница между методами *lock()* и *tryLock()* заключается в том, что при вызове метода *lock()*, если ресурс заблокирован, то поток приостанавливает свою работу до тех пор, пока ресурс не освободится. Метод *tryLock()* не переводит поток к режим ожидания, если ресурс заблокирован, и он продолжает свою работу, пропуская заблокированный ресурс. Если же ресурс находится в разблокированном состоянии, методы *lock()* и *tryLock()* работают одинаково.

Разблокировка ресурсов в классе *QMutex*. Для разблокировки ресурсов используется метод *unlock()*. В этом методе происходит атомарное изменение значения *priv_ptr* на *0x0*, и, что самое важное, возобновление работы потоков, ожидающих доступа к ресурсам. Пер-

вый поток, получивший доступ к ресурсам, установит *priv_ptr* в значение, отличное от *0x0*, а остальные потоки снова будут приостановлены.

Управление памятью. Для управления доступа к памяти используется техника подсчета ссылок, чтобы быть уверенными, что во время освобождения указателя *QMutexPrivate* ресурс не используется ни одним потоков. Но сам указатель *QMutexPrivate* никогда не удаляется. Он используется повторно, поскольку *QMutexPrivate* нужен только для блокировки и разблокировки ресурсов в многопоточном приложении, и, как следствие, таких объектов не может больше, чем самих потоков.

Практический пример

Рассмотрим пример многопоточного клиент-серверного приложения обмена сообщениями, в котором реализованы оба описанных метода создания потоков. В данном приложении каждый экземпляр программы является одновременно как клиентом, так и сервером, что является удобным при использовании в одной локальной сети, поскольку приложение не требует наличия базы данных и каких-либо настроек, кроме установки. Для работы приложения используется 3 потока:

1. Главный поток, через который происходит взаимодействие с *GUI* (далее — поток 1).
2. Поток, созданный при помощи первого описанного метода — создание класса, наследованного *QThread* (далее — поток 2).
3. Поток, созданный при помощи второго описанного метода, создание которого происходит при помощи функции *QObject:: moveToThread(...)* (далее — поток 3).

В потоке 2 реализован метод поиска доступных серверов-клиентов. Ниже представлена реализация класса.

```
class NetworkScan: public QThread {
...
public:
NetworkScan(QObject * parent = nullptr);
protected:
void run();
signals:
void AddNewHost(QString IP);
};
void NetworkScan:: run()
{
while(true)
{
// Connected To Host
}
}
```

В переопределенном методе *run()* реализована проверка доступа к клиенту при помощи методов класса *QAbstractSocket*. В случае, если клиент будет доступен, произойдет испускание сигнала: *emit AddNewHost(IP)*, который связан со слотом в потоке 1, в следствии чего будет установлено соединение с клиентом при помощи методов класса *QTCPSocket*. Как было сказано ранее, сигналы, испускаемые в потоках, созданных данным способом, преобразуются в события *QMetaObject* и обрабатываются в цикле обработки событий. При создании потока также был установлен приоритет *THREAD_PRIORITY_LOWEST* для того, чтобы проверка доступности новых клиентов происходила в фоновом режиме.

Поток 3 используется для получения и отправки сообщений. Поскольку быстродействие при использовании данного приложения является важным критерием, ожидание обработки событий для отправки и получения сообщения является неприемлемым. Именно поэтому данный функционал был реализован при помощи метода *QObject:: moveToThread(...)*, так как сигналы такого потока обрабатываются немедленно (*DirectConnection*).

Заключение

Каждый описанный метод создания потока подходит для решения множества задач, решение которых необходимо для любого современного приложения, нацеленного на быстродействие. Для достижения наилучшего результата необходимо тщательное планирование архитектуры многопоточного приложения, в процессе которого нужно знать и учитывать особенности работы каждого из методов. Как следствие, обоснованные и правильно реализованные способы создания потоков позволяют приложению работать в такой связке, как быстродействие и стабильность, что является залогом успешного современного продукта.

ЛИТЕРАТУРА

1. НИЯУ «МИФИ» Процессы и потоки: [Электронный ресурс]. 2016. URL: http://ftp.csdep.mephi.ru/kiselev/CAI%202017/Module%2001/ OS_course/OS_course/OS-05_K.pdf. (Дата обращения: 02.02.2019).
2. Qt5 module: [Электронный ресурс]. 2019. URL: https://github.com/ qt/qt5. (Дата обращения: 20.01.2019).

3. Функция CreateThread: [Электронный ресурс]. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi /nf-processthreadsapi-createthread. (Дата обращения: 25.01.2019).
4. QThread — потоки в Qt: [Электронный ресурс]. 2011–2012. URL: http://qt-doc.ru/qthread-potoki-v-qt.html. (Дата обращения: 10.02.2019).
5. Qt 5.10. Профессиональное программирование на C++.— СПб.: БХВ-Петербург, 2018.— 1072 с.: ил.— (В подлиннике)
6. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. Пер. с англ. Слинкин А. А.— М.: ДМК Пресс, 2012.— 672с.: ил.

Московский авиационный институт