# Практическое занятие. Потоки. Синхронизирующие объекты Windows.

### Тестовое приложение Qt

Создайте GUI-приложение на базе QMainWindow.

В главном окне разместите:

* Кнопки для запуска потоков (кроме обычных PushButton, для задания приоритетов потоков возможно, понадобятся еще кнопки-переключатели - RadioButton)
* Виджет для отображения работы потоков (QWidget или QFrame)

Все потоки будут «рисовать» посредством механизма сигналов/слотов в одном и том же виджете, предназначенном для отображения взаимодействия потоков. Рисование текущей «точки» будет производиться в главном (интерфейсном) потоке по сигналу, поступающему из «рабочего» потока.

Примерный вид приложения:



Синхронизация потоков в разных приложениях с использованием семафора

Очистка области рисования

использование приоритета

использование семафора

использование мьютекса

Запуск без синхронизации

Каждый поток будет «рисовать» точки:

* своим цветом
* на своем уровне (координата “y” будет у каждого потока своя)
* все потоки будут использовать (разделять) общую координату “x” для того, чтобы было видно, как ОС переключает потоки

### Структура (класс) для передачи данных потоковой функции

Чтобы что-нибудь подобное можно было бы нарисовать в потоковой функции, ей нужно передать довольно много параметров, поэтому логично в качестве параметра передавать указатель на объект Вашего класса, который:

1. Наследует от QObject и поддерживает механизм сигналов/слотов
2. В объект можно запаковать все необходимые данные.

class MyData:public QObject

{

Q\_OBJECT

int y; // индивидуальная для потока координата Y

QColor col; // индивидуальный цвет

int maxX; // «докуда» рисовать

HANDLE hObject; // дескриптор синхронизирующего объекта

…

};

1. А также реализовать сигнал о том, что добавлено новое данное и рисунок требует перерисовки:   
   void signalAddPoint (MyPoint);

**Подсказка:** как можно/нужно формировать данные для передачи в потоковую функцию?

### Вспомогательный класс для хранения

Так как рисуемые потоками точки будут разного цвета, с разными координатами – создаем вспомогательный класс MyPoint:

сlass MyPoint

{

public:

MyPoint(const QPoint &pt = QPoint(),QColor = Qt::black);

void draw(QPainter & painter);

private:

QPoint m\_pt;

QColor m\_col;

};

**Замечание:**

В методе void draw(QPainter & painter)должно производиться рисование цветной точки. Для большей наглядности можно рисовать вертикальные линии с заданной верхней координатой.

### Демонстрация переключения потоков ОС.

Для хранения и отображения результатов работы потоков потребуется виджет. Создаем пользовательский класс, производный от QWidget или QFrame, например, MyDemoWidget.

1. В классе **MyDemoWidget** заведите контейнер с объектами MyPoint, например QVector<MyPoint>m\_vector;
2. Для обработки сигнала о добавлении точки, в классе MyDemoWidget создайте слот, например, void MyDemoWidget ::slotAddPoint(MyPoint);
3. в слоте slotAddPoint (MyPoint) нужно:
4. принять очередной объект MyPoint,
5. добавить в контейнер,
6. перерисовывать весь рисунок (repaint())
7. Для рисования содержимого контейнера m\_vector перегрузите виртуальный обработчик события QPaintEvent void MyDemoWidget::*paintEvent*(QPaintEvent \*) и
8. Выполните отрисовку всех точек, сохраненных в контейнере.

Замечание:

механизм сигналов/слотов при взаимодействии между потоками реализуется посредством событий => чтобы параметр, передаваемый в сигнале по **значению** (в нашем случае MyPoint), «завернуть» в событие, мета-объектная система должна «знать» о пользовательском типе =>пользовательский тип нужно зарегистрировать посредством шаблона  
**int qRegisterMetaType** (const char \**typeName*), где *typeName - тип*

В нашем случае:  
qRegisterMetaType<MyPoint>(“MyPoint”);

Это необходимо сделать до использования типа MyPoint в механизме сигналов-слотов, например, в конструкторе **MyDemoWidget**.

### Предписываем всем потокам выполняться на одном ядре:

1. В файле main.cpp подключить <windows.h>
2. После создания очередного потока вызвать системную функцию  
   SetThreadAffinityMask (<дескриптор потока>, 1);

### Доработка класса mainWindow

1. В режиме дизайнера преобразуйте виджет QWidget (или QFrame ) в MyDemoWidget
2. В классе MainWindow предусмотреть переменную int m\_X, которая будет содержать общую для всех потоков координату «x»; - подумайте, как нужно объявить такую переменную? .
3. Если потоковая функция является статическим методом класса, то имеет доступ ко всем статическим данным класса => в потоковой функции:

* Формируем очередную точку MyPoint
* Эмитируем сигнал о добавлении новой точки, например:  
  void signalAddPoint (MyPoint);
* Инкрементируем общую для всех потоков координату “x”
* Ставим программную задержку (имитируем работу)

1. Создайте слот, вызываемый при нажатии кнопки “Threads go!”, для запуска трех «рисующих» потоков с приоритетом по умолчанию, каждый из которых будет в цикле выполнять «работу». При работе этих потоков не будет использоваться синхронизация.
2. В слоте:

* Установите начальное значение координаты X =0
* Запустите три потока , передав каждому свои параметры
* соедините сигнал MyData::signalAddPoint(MyPoint) каждого потока со слотом MyDemoWidget::slotAddPoint (MyPoint) .
* В качестве последнего параметра метода connect надо использовать Qt::QueuedConnection (при условии, что сигнал и слот находятся в разных потоках, это значение берется по умолчанию).

Подсказка: можно всю работу по подготовке и запуску потоков вынести в отдельный метод, принимающий в качестве параметров, дескриптор синхронизирующего объекта и указатель на потоковую функцию.

### Общие замечания по оформлению потоковой функции

1. Потоковая функция не может быть обычным методом класса! Почему??? Зато может быть статическим методом класса.
2. Вид потоковой функции предопределен:  
   DWORD WINAPI MainWindow::ThreadFunc( LPVOID lpParam ){}
3. В потоковой функции Вам понадобится задержка (для того, чтобы один поток не изрисовал всю отведенную область за свой квант времени):
   1. величину задержки подбираете исходя из возможностей Вашего процессора
   2. задержку не стоит оформлять как цикл с пустым телом, так как оптимизирующий компилятор такую задержку скорее всего «выкинет» => имитация деятельности

### Запуск рабочих потоков без синхронизации.

#### Запуск потоков с одинаковым приоритетом

Требуется запустить несколько потока с одинаковым приоритетом и «увидеть» как операционная система переключает потоки в соответствии с отведенными каждому потоку квантами времени.



Запуск потоков – системная Win32 API функция CreateThread ()

HANDLE CreateThread(//возвращаемое значение==0 - ошибка

LPSECURITY\_ATTRIBUTES lpsa, // 0- защита по умолчанию и дочерние процессы не наследуют данный поток

SIZE\_T cbStack, // размер стека или 0 (по умолчанию)

LPTHREAD\_START\_ROUTINE lpStartAddr, // указатель на функцию (точку входа) потока  
- **DWORD WINAPI ThreadF(LPVOID lpvThreadParm);**Не может быть NULL! Каждый поток внутри родительского процесса начинает свое выполнение с вызова специальной функции, называемой *потоковой функцией*. Выполнение потока продолжается до тех пор, пока не завершится его потоковая функция.

LPVOID lpvThreadParm, // значение или указатель на любые данные. Используется по замыслу программиста. Эти данные должны **гарантированно существовать** пока их использует потоковая функция, так как стек у каждого потока свой!!!

DWORD fdwCreate, // дополнительные флаги создания:  
**0** (начинает выполняться немедленно)  
**CREATE\_SUSPENDED** - поток создается “остановленным”, то есть ожидающим запуска. Для того чтобы запустить такой поток, кто-то должен вызвать функцию **ResumeThread()**

LPDWORDlpIDThread// сюда функция вернет идентификатор (**в NT – если Вам не нужно значение, можно передать 0**)

);

Замечание: для «чистоты эксперимента» возможно, стоит создавать потоки в «приостановленном» (suspended) состоянии, а затем (когда все сформированы) возобновить их выполнение (ResumeThread)

#### Управление приоритетами потоков

Поэкспериментируйте с приоритетами потоков.

В GUI-приложении проще и нагляднее для изменения приоритетов потоков ввести для каждого потока переключатели (достаточно трех radio buttons для каждого потока: нормальный, чуть повыше нормального, чуть повыше)

Функция для установки приоритета - SetThreadPriority()

### Синхронизация потоков посредством критической секции



**Основные понятия:**

1. сформировать структуру данных типа CRITICAL\_SECTION (она может быть глобальной, локальной или динамической – главное, чтобы она существовала, пока ею кто-нибудь пользуется).

**CRITICAL\_SECTION cs; //поля этой структуры недокументированны. Пользоваться ею можно только посредством соответствующих Win32 API**

1. прежде чем синхронизировать потоки с помощью критической секции, нужно ее инициализировать вызовом функции:

**InitializeCriticalSection(&cs);**

1. перед каждым (в обоих потоках) блоком, который модифицирует данные, вызвать

**EnterCriticalSection(&cs)**;//блокирует данный поток, если эта критическая секция уже «занята» другим потоком. Это означает, что поток не может выполнить код, который «защищен» критической секцией => отправляется ОС в спячку.

Функция, анализируя поля структуры cs (некоторый счетчик ссылок), выясняет – вызвана ли она в первый раз (если счетчик нулевой). В этом случае функция увеличивает счетчик и разрешает выполнение потока дальше (то есть выполняется блок, модифицирующий данные). Допустим, в это время истекает квант времени, отпущенный данному потоку, или он вытесняется более приоритетным потоком, использующим те же данные: поток выполняется, пока не встречает функцию EnterCriticalSection(), функция выясняет, что объект cs уже «занят», она приостанавливает данный поток (он «засыпает»), а остаток процессорного времени система передает другому потоку.

1. после выполнения этого блока вызвать

**LeaveCriticalSection(&cs);**

Эта функция уменьшает счетчик ссылок. Как только поток “освобождает” критическую секцию (счетчик ссылок становится 0), система “будит” ожидающий поток, снимая защиту от модификации данных.

1. когда надобность в синхронизации потоков отпадает, следует вызвать

**DeleteCriticalSection(&cs);**

Эта функция освобождает все ресурсы, включенные в критическую секцию.

### Синхронизация потоков посредством объекта – mutex



**Основные понятия:**

**HANDLE CreateMutex(//если мьютекс создать не удалось - 0**

**LPSECURITY\_ATTRIBUTES lpMutexAttributes,**

**BOOL bInitialOwner,** //TRUE - позволяет вызывающему функцию потоку немедленно вступить во владение мьютексом **при его создании** (этот флаг игнорируется, если мьютекс уже существует)  
если **TRUE**, поток, создающий мьютекс, изначально имеет доступ к ресурсу, контролируемому мьютексом => мьютекс оказывается в занятом состоянии=> любой другой поток, ожидающий данный мьютекс, будет приостановлен, пока поток, создавший этот объект, не освободит его. Если параметр равен **FALSE**, это означает, что мьютекс не принадлежит ни одному из потоков => первый же поток из числа ожидающих этот объект может занять его и тем самым продолжить свое выполнение.

**LPCTSTR lpName );** //имя объекта ядра или 0

*Замечание:* если Вы знаете, что существует мьютекс с данным именем, и Вы хотите сделать этот объект доступным другим процессам, нет нужды обращаться к CreateMutex(). Есть способ, получить описатель уже существующего мьютекса:

**HANDLE OpenMutex( DWORD dwDesiredAccess,** //стандартные (DELETE, READ\_CONTROL, SYNCHRONIZE, WRITE\_DAC, and WRITE\_OWNER) + специфические для мьютекса - MUTEX\_ALL\_ACCESS и MUTEX\_MODIFY\_STATE

**BOOL bInheritHandle,** //если TRUE, дочерний процесс //наследует возвращаемый описатель мьютекса

**LPCTSTR lpName** ); //указатель на строку и именем мьютекса

При вызове OpenMutex() система сканирует существующие объекты-мьютексы, проверяя – нет ли среди них объекта с именем, указанным в lpName. Обнаружив таковой, она создает описатель объекта, специфичный для данного процесса, и возвращает его вызвавшему потоку. В дальнейшем любой поток из данного процесса может использовать этот описатель при вызове любой функции, требующей такой описатель. А если объекта с таким именем нет, функция возвращает 0.

Когда требуется освободить объект –

**BOOL ReleaseMutex(HANDLE hObject**);//Замечание: если поток не владеет данным мьютексом - FALSE

Когда мьютекс Вам больше не нужен, вызовите функцию

**BOOL CloseHandle(HANDLE hObject);** //где hObject – описатель мьютекса, возвращенный функцией CreateMutex()

#### Именованный мьютекс

Посредством именованного мьютекса синхронизируйте выполнение потоков в разных процессах.

### Синхронизация потоков посредством объекта – semaphore



**Основные понятия:**

Объект-семафор создается функцией:

**HANDLE CreateSemaphore**(//если семафор создать не удалось, возвращаемое значение 0

**LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes, //**указатель на структуру типа SECURITY\_ATTRIBUTES или 0

**LONG lInitialCount, //** позволяет задать - сколько из этих ресурсов доступно изначально (на момент создания). Это значение должно лежать в диапазоне 0- lMaximumCount. **Семафор свободен, если его счетчик больше нуля и занят, если счетчик равен 0.**

**LONG lMaximumCount, //** максимальное значение счетчика (должно быть больше нуля)

**LPCTSTR lpName);** //Если этот параметр совпадает с именем уже существующего самафора, параметры lInitialCount и lMaximumCount **игнорируются**.  
Если lpName равен **0**, семафор создается без имени, и им можно пользоваться только в пределах одного процесса.

Получить доступ к уже существующему объекту:

HANDLE **OpenSemaphore**( DWORD fdwAccess, BOOL bInhentHandle, PCTSTR pszName);

Когда семафор в данном потоке больше не нужен, не забудьте вызвать функцию:

BOOL **CloseHandle(**HANDLE hObject**);** //где hObject – описатель семафора, возвращенный функцией CreateSemaphore()

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию *ReleaseSemaphore*

BOOL **ReleaseSemaphore**( HANDLEhSem, LONG lReleaseCount, PLONG pPreviousCount);

Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но это вовсе не обязательно. Функция возвращает исходное значение счетчика ресурсов в *\*plPreviousCount* Если Вас не интересует это значение (а в большинстве программ так оно и есть), передайте в параметре *plPrviousCount* значение 0.

#### Именованный семафор

Синхронизация потоков разных процессов посредством именованного семафора.

### Синхронизация потоков посредством объекта – event

Событие - самая примитивная разновидность синхронизирующих объектов – они просто уведомляют об окончании какой-либо операции.

**События с автоматическим сбросом** переводятся функциями WaitForSingleObject() или WaitForMultipleObjects() в занятое состояние. Таким образом, если в какой-то момент времени событие с автоматическим сбросом оказывается свободным, то только один из потоков, ожидающих данный объект-событие, пробуждается (а событие переводится системой опять в занятое состояние) => остальные продолжают спать. Если ожидающих потоков нет, объект-событие остается в свободном состоянии.

**События с ручным сбросом** автоматически не переустанавливаются в занятое состояние функциями WaitForSingleObject() и WaitForMultipleObjects().У Вас может быть несколько потоков, ждущих возникновения одного события. Когда событие наконец происходит, система пробуждает все ожидающие данное событие потоки.

**Создание объекта-событие.** Объекты-события создаются функцией:

**HANDLE CreateEvent**(//если объект создать не удалось - 0

**LPSECURITY\_ATTRIBUTES lpEventAttributes, //**указатель на структуру SECURITY\_ATTRIBUTES или 0

**BOOL bManualReset, //**TRUE – создастся событие со сбросом вручную, FALSE – с автосбросом.

**BOOL bInitialState, //** начальное состояние объекта-события: TRUE – свободен (событие произошло – потоки, желающие получить доступ к ресурсу посредством WaitForSingleObject() или WatForMultipleObjects() выполняются), FALSE – занят (событие не произошло – все потоки, желающие получить доступ к ресурсу, засыпают).

**LPCTSTR lpName );** //Если объект-событие с данным именем уже существует, то возвращаемый описатель характеризует существующий объект-событие, а поля bManualReset и bInitialState игнорируются, так как они уже были установлены при создании объекта.

*Замечание*: если Вы хотите получить описатель существующего объекта-события – пользуйтесь функцией

HANDLE **OpenEvent**( DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName );

Когда объект-событие в данном потоке больше не нужен, не забудьте вызвать функцию:

**CloseHandle(HANDLE hObject);** //где hObject – описатель события, возвращенный функцией CreateEvent()

#### Событие с ручным сбросом

Позволяет уведомить об окончании операции сразу несколько потоков.

**События этого типа не переводятся в занятое состояние функциями ожидания** (WaitForSingleObject…)!!!.

Событие с ручным сбросом переводится в свободное состояние функцией:

BOOL SetEvent(HANDLE hEvent ); //все ожидающие потоки выполняются

После перевода объекта в свободное состояние он пребывает в нем, пока какой-нибудь поток явно не сбросит событие в занятое состояние вызовом:

BOOL ResetEvent(HANDLE hEvent );

*Замечание*: иногда после вызова SetEvent() (событие освобождается) тут же его нужно опять «занять», то есть вызвать ResetEvent(). Для такой ситуации предусмотрена специальная функция:

BOOL PulseEvent(HANDLE hEvent );//после возврата управления функцией //событие остается в занятом состоянии



Для GUI-приложения:

MsgWaitForMultipleObjects!!! – без сообщений!!!



##### Именованное событие

Синхронизация потоков посредством именованного объекта-события.

#### Событие с автоматическим сбросом

Позволяет уведомить об окончании операции один поток. Функции ожидания переводят объект в занятое состояние.

