Многопоточное программирование в Delphi   
для начинающих

(редакция 1.1 от 13.07.2020г)

Автор: Логинов Д.С.

Пенза, 2020

Оглавление

[1. Вступление 3](#_Toc45535540)

[1.1 Для чего используется многопоточное программирование в Delphi 4](#_Toc45535541)

[1.2 В чём сложность разработки многопоточных программ в Delphi по сравнению с другими современными языками программирования 4](#_Toc45535542)

[1.3 Поддержка многопоточности в операционных системах и процессорах 6](#_Toc45535543)

[2. Базовый класс многопоточности - TThread 8](#_Toc45535544)

[3. Предотвращаем зависание основного потока (имитация задачи длительного вычисления) 9](#_Toc45535545)

[4. Управление временем жизни потоков 13](#_Toc45535546)

[4.1 Использование свойства TThread.Terminated для выхода из метода Execute 14](#_Toc45535547)

[4.2 Главная форма отвечает за прекращение работы потока и уничтожение объекта потока 15](#_Toc45535548)

[4.3 Главная форма отвечает за уничтожение объекта потока с разовой задачей 17](#_Toc45535549)

[4.4 Главная форма отвечает за уничтожение нескольких долгоживущих потоков 19](#_Toc45535550)

[4.5 Использование списка TObjectList для хранения ссылок на объекты потоков 23](#_Toc45535551)

[4.6 Простой пример организации паузы в работе потока 26](#_Toc45535552)

[4.7 Использование свойства FreeOnTerminate для автоматического уничтожения объекта потока при завершении работы потока 27](#_Toc45535553)

[4.8 Организация корректного завершения работы программы при использовании свойства FreeOnTerminate 28](#_Toc45535554)

[5. Передача информации из дополнительного потока в основной 32](#_Toc45535555)

[5.1 Обращение к визуальным компонентам формы из дополнительного потока – как нельзя делать 32](#_Toc45535556)

[5.2 Использование метода TThread.Synchronize для передачи данных в основной поток 32](#_Toc45535557)

[5.3 Периодическое чтение основным потоком данных, подготовленных в дополнительном потоке 36](#_Toc45535558)

[5.4 Использование функции SendMessage для передачи данных в основной поток 40](#_Toc45535559)

[5.5 Использование функции PostMessage для передачи очереди сообщений в основной поток 44](#_Toc45535560)

[5.6 Использование списка TThreadList для передачи очереди данных в основной поток 46](#_Toc45535561)

[5.7 Использование метода TThread.Synchronize для передачи данных в основной поток – вариант с использованием анонимной процедуры 50](#_Toc45535562)

[5.8 Использование метода TThread.Queue для передачи данных в основной поток 51](#_Toc45535563)

[6. Где можно и где нельзя создавать и уничтожать потоки 57](#_Toc45535564)

[7. Коротко о приоритетах потоков в Windows 58](#_Toc45535565)

# 1. Вступление

В данной работе (условно назовём её «Учебник») речь пойдет о многопоточном программировании в Delphi. Учебник написан для сообщества Delphi-программистов (к этому сообществу следует отнести также Lazarus-программистов). В дальнейшем я буду использовать термин «программист», подразумевая именно Delphi-программиста.

В современных версиях Delphi (например, Delphi 10.3) имеется удобная библиотека многопоточного программирования «Parallel Programming Library» и в последние годы всё больше примеров в интернете даётся именно для PPL. Она хорошо подходит для тех задач, где требуется распараллелить математические вычисления либо выполнить параллельную однотипную обработку информации во множестве файлов на SSD-накопителе. На самом деле, возможности PPL ограничены только фантазией программиста.

Я же постараюсь сделать основной упор на классическом способе многопоточного программирования в Delphi – работе с классом TThread (вернее, с классами-наследниками, которые разрабатывают программисты для решения тех или иных задач). Считаю, что именно классический подход должен оставаться основным (по крайней мере, программисты должны уметь применять этот подход на практике и должны ориентироваться в чужом коде, где применяется классический подход).

К сожалению, в интернете (в открытых источниках) очень мало качественного материала, который поможет начинающему программисту научиться работать с классом TThread. Формально, материала полно, однако большая его часть, вопреки стараниям разработчиков этого материала, не учит многопоточному программированию, а чаще сбивает с толку, в том числе из-за множества ошибочных примеров и советов, которые нельзя использовать для данной темы.

Профессиональные программисты вероятно не найдут в данной работе ничего нового. Каких-либо научных открытий здесь не планируется. В отличие от научных работ, я не планирую увлекаться ссылками на различные известные источники. Вы всегда можете уточнить любой вопрос с помощью поисковой машины.

Местами Вы будете встречать несколько неуклюжую терминологию, объяснения «на пальцах». Сделано так специально, чтобы не раздувать объём материала и не мучать читателя длинными определениями общеизвестных терминов.

Я постараюсь использовать максимально простой стиль изложения материала, надеюсь, начинающие программисты оценят.

## 1.1 Для чего используется многопоточное программирование в Delphi

Я хотел бы выделить две основные причины использования многопоточности в Delphi-программах:

1. Длительные математические вычисления и обработка данных. Примеры: шифрование данных, сжатие данных, анализ данных, конвертация видео или звука, обучение нейронных сетей и многое другое. Для всех этих задач характерна высокая и длительная нагрузка на процессор. Очевидно, что если мы при разработке обычного VCL-приложения попытаемся решить такую задачу без вынесения в отдельный поток, то интерфейс пользователя подвиснет на время, которое требуется для выполнения вычислений.

2. Задачи длительного ожидания завершения ввода-вывода. Примеры: выполнение HTTP-запроса (или REST-запроса) к какому-либо сайту в интернете, обмен данными по протоколу TCP, обмен с устройством по COM-порту, запуск и ожидание работы других программ на компьютере и многое другое. Для всех этих задач характерна низкая нагрузка на процессор. Ресурсы процессора практически не задействуются, однако наша программа после запуска операции будет вынуждена ожидать её завершения. Очевидно, что если мы при разработке обычного VCL-приложения попытаемся решить такую задачу без вынесения в отдельный поток, то интерфейс пользователя подвиснет на время, которое требуется для ожидания завершения операции ввода-вывода.

## 1.2 В чём сложность разработки многопоточных программ в Delphi по сравнению с другими современными языками программирования

В современном программировании принято разделять разрабатываемую систему на бэкенд и фронтэнд. Бэкенд – это сервис, который отвечает за хранение данных, их обработку и взаимодействие с фронтэндом. Фронтэнд – это программа, представляющая интерфейс пользователя. Она, как правило, не хранит и не обрабатывает данные. Её цель – обеспечить для пользователя наиболее удобную работу.

Delphi – это прежде всего инструмент для разработки фронтэнда, поскольку мы создаём в первую очередь программы с графическим интерфейсом пользователя (GUI). Программист фронтэнда должен сделать всё возможное, чтобы пользователю было максимально комфортно работать. Никаких подвисаний интерфейса во фронтэнде быть не должно! Однако этого достичь в Delphi весьма нелегко. Если мы запрашиваем с сервера данные без использования многопоточности, то интерфейс программы подвисает на время обработки запроса. Некоторые сетевые библиотеки позволяют частично обойти данную проблему (например, режим антизаморозки в Overbyte ICS или Indy), однако далеко не всегда данный режим применим.

Основной режим работы в библиотеке Overbyte ICS – асинхронный, при котором весь сетевой обмен выполняется в рамках одного потока, этот поток не блокируется (т.е. не подвисает), однако программист должен самостоятельно реализовать обработчики различных событий (например, обработчик события получения данных). Если протокол обмена с сервером состоит из множества запросов и ответов, то наша программа значительно усложняется, мы можем легко столкнуться с явлением «callback hell» (ад функций обратных вызовов). Конечно, профессиональный программист такого не допустит и сможет реализовать машину состояний (конечный автомат), которая избавит от проблемы «callback hell», но профессионалы, к сожалению, на дороге просто так не валяются. Напомню, статья предназначена в первую очередь для новичков, которые вряд ли смогут легко справиться с этой задачей. К тому же, даже если мы и реализуем асинхронный сетевой обмен данными в рамках одного потока, мы столкнемся с необходимостью обработки принятых данных, необходимостью выполнять запросы к базе данных, работать с файлами (на медленном HDD), а в конечном итоге – к подвисаниям интерфейса пользователя. Знание многопоточного программирования нам помогает справится с этой проблемой.

У некоторых других современных языков программирования (C#, JavaScript, Rust, Kotlin и т.д., их число постоянно растёт) проблема «callback hell» решена на уровне языка: введён встроенный механизм асинхронности: async / await. Смысл в том, что программист на этих языках пишет простой последовательный код (без обработчиков событий), а компилятор автоматически преобразует этот код в асинхронный с использованием машины состояний. Т.е. программист на этих языках теперь не сталкивается с теми сложностями, с которыми сталкиваемся мы при использовании асинхронных сетевых библиотек (кстати, это не только Overbyte ICS, но и ещё десяток других, в том числе коммерческих).

К слову, программист на языке JavaScript, как правило, не сталкивается с многопоточностью. Практически все вещи в JavaScript можно реализовать в контексте одного (главного) потока с использованием возможностей асинхронности. Конечно, асинхронность не спасёт в ситуациях, когда требуется выполнить длительные математические расчёты или обработку данных (интерфейс пользователя будет подвисать на время такой обработки), однако такую работу выполняет, как правило, бэкэнд.

Мы же используем Delphi (на мой взгляд – наиболее мощный инструмент для разработки кроссплатформенных GUI-приложений практически для любой платформы) и вынуждены пользоваться теми возможностями, которые этот инструмент нам предлагает, а встроенной асинхронности у данного инструмента пока нет.

Давайте же перейдём к теме многопоточности!

## 1.3 Поддержка многопоточности в операционных системах и процессорах

Буду говорить своими словами, очень упрощённо. Обязательно почитайте материал на эту тему в Интернете (очень много классных статей на эту тему есть на ресурсе habr.com, например, в статьях, посвящённых языку C#).

Я буду говорить в первую очередь о компьютерах с ОС Windows и процессорах с архитектурой Intel x86 (либо AMD x86\_64), однако аналогичные механизмы есть в различных устройствах, оснащённых ОС Linux, Android, MacOS, iOS и процессорами, на которых работают эти ОС.

Итак, на компьютере одновременно запущены сотни программ (приложений). Мы называем их также «процессы». Процесс (process) – это объект операционной системы. Когда мы запускаем программу, ОС создаёт объект «процесс». При создании процесса ОС создаёт также объект «поток» (thread), для него создаётся стек вызовов (call stack) и код запускается на исполнение. Исполняемый код всегда выполняется в контексте какого-то одного потока. В любом процессе всегда есть минимум один поток. При необходимости, в процессе может быть более одного потока (например, в процессе WEB-сервера могут быть созданы тысячи потоков).

В типовом приложении исполняемый код 99% времени занимается тем, что ожидает поступления каких-либо событий. Как только событие поступило, исполняемый код «просыпается» и выполняет действия по обработке этого события, после чего «засыпает» до тех пор, пока не произойдет следующее событие. Таким образом, на компьютере могут быть созданы одновременно десятки тысяч потоков, которые 99% времени находятся в спящем состоянии и не оказывают значительной нагрузки на процессор.

Важно знать, что в ОС присутствует системный планировщик задач, который отвечает за распределение процессорного времени между потоками. В том случае, если поток выполняет длительную вычислительную задачу (обрабатывает информацию), планировщик задач автоматически выполнит перевод данного потока в спящий режим, как только поток истратит выделенный ему квант времени (предположим, 50 мс). Кстати, в большинстве случаев потоки намного раньше переходят в спящий режим, т.к. выполняют свою задачу раньше, чем успевает закончиться выделенный квант времени, либо переходят в спящий режим в связи ожиданием завершения ввода-вывода.

Как только поток переходит в спящий режим, планировщик задач отыскивает следующий поток, который нуждается в пробуждении, восстанавливает его контекст и переводит поток в рабочий режим.

Каждый раз, когда поток переходит в спящий режим, происходит сохранение его контекста (сохраняется текущее состояние регистров процессорного ядра, на котором поток выполняется). При переводе потока в рабочий режим выполняется обратное действие – восстановление контекста потока (загрузка ранее сохраненных значений в регистры процессорного ядра). Я даю здесь объяснение «на пальцах», а Вы, если интересно, поищите в интернете более подробную информацию.

Кстати, операция переключения контекста по времени далеко не бесплатная, поэтому производители процессоров и разработчики ОС постоянно делают улучшения для повышения скорости данной операции. Если Вы хотите создать эффективный высокопроизводительный WEB-сервер, то будет не лишним подумать над тем, как минимизировать количество переключений контекста при исполнении Вашего кода. Я не буду давать рекомендаций по поводу создания высокопроизводительного WEB-сервера, поскольку задача эта больше подходит для специализированных языков программирования WEB-серверов, например GoLang, а Delphi заточен в первую очередь на разработку фронтэнда, а также корпоративного программного обеспечения.

# 2. Базовый класс многопоточности - TThread

TThread – это базовый класс, инкапсулирующий функционал для обеспечения работы параллельного потока. Если мы хотим реализовать код, который должен выполняться в отдельном потоке, то нам необходимо реализовать наследника от класса TThread и, как минимум, реализовать override-метод ***Execute***. Код, находящийся внутри метода ***Execute*** будет исполняться в отдельном потоке.

При создании класса-наследника от TThread мы можем реализовать конструктор, деструктор, объявить собственные поля, методы, свойства и т.д., в зависимости от поставленной перед нами задачи.

В данном разделе я не буду подробно останавливаться на описании возможностей класса TThread. Читайте внимательно следующие разделы, надеюсь, в них Вы найдёте необходимую для Вас информацию.

|  |
| --- |
| Внимание! Дополнительные потоки создаются с помощью вызова соответствующей функции операционной системы, например CreateThread в ОС Windows. Класс TThread является обёрткой, которая добавляет к системному дополнительному потоку объектно-ориентированное измерение, т.е. позволяет создать объект, который привязывается к системному дополнительному потоку. |

|  |
| --- |
| Внимание! Существуют и другие популярные средства для многопоточного программирования в Delphi. К ним стоит отнести:  - Parallel programming library - стандартная библиотека для параллельного программирования в современных версиях Delphi.  - OmniThreadLibrary – популярная библиотека для параллельного программирования (на июль 2020 библиотека поддерживает только Windows, однако в будущем может появиться поддержка других операционных систем). Ссылка: <https://github.com/gabr42/OmniThreadLibrary> |

# 3. Предотвращаем зависание основного потока (имитация задачи длительного вычисления)

Ниже представлен пример кода, в котором имитируется задача длительного вычисления (функция ***DoLongCalculations***). Данная функция запускается двумя путями: 1) из основного потока (см. обработчик кнопки ***btnRunInMainThread***); 2) из параллельного потока (см. обработчик кнопки ***btnRunInParallelThread***). Данный пример Вы можете найти в архиве (папка Ex1).

|  |
| --- |
| Внимание! Все примеры к данному «Учебнику» доступны по ссылке:  <https://github.com/loginov-dmitry/multithread> |

unit Ex1Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls;

type

TMyThread = class(TThread)

public

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

btnRunInMainThread: TButton;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure btnRunInMainThreadClick(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

function DoLongCalculations: Int64;

var

I: Integer;

begin

Result := 0;

// Очень длинный цикл. Имитирует длительные вычисления.

for I := 1 to MaxInt do

Result := Result + Random(1000);

end;

procedure MyShowMessage(Msg: string);

begin

Windows.MessageBox(0, PChar(Msg), '', MB\_OK);

end;

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

begin

// Запускает параллельный поток

TMyThread.Create(False);

end;

{ TMyThread }

procedure TMyThread.Execute;

var

V: Int64;

begin

FreeOnTerminate := True;

V := DoLongCalculations;

MyShowMessage('Результат: ' + IntToStr(V));

end;

procedure TForm1.btnRunInMainThreadClick(Sender: TObject);

var

V: Int64;

begin

V := DoLongCalculations;

MyShowMessage('Результат: ' + IntToStr(V));

end;

end.

Нажмите кнопку ***btnRunInMainThread*** для запуска функции ***DoLongCalculations*** из основного потока. Через несколько секунд появится сообщение с результатом вычислений. Длительность вычислений предсказать сложно, она зависит от характеристик процессора, а также от его текущего состояния. Приблизительное время – от 5 до 20 секунд. Обратите внимание, что после запуска вычислений вы не можете выполнять каких-либо действий с окном (нажимать кнопки, изменять размеры окна и т.д.) до момента окончания вычислений. Таким образом, запуск функции ***DoLongCalculations*** из основного приводит к подвисанию интерфейса пользователя.

А теперь нажмите кнопку ***btnRunInParallelThread*** для запуска функции ***DoLongCalculations*** из дополнительного потока. Вы увидите, что интерфейс пользователя не подвис, а продолжает реагировать на Ваши действия. Через несколько секунд появится сообщение с результатом вычислений. Таким образом, запуск функции ***DoLongCalculations*** из параллельного потока не приводит к подвисанию интерфейса пользователя.

Попробуйте нажать кнопку btnRunInParallelThread два раза друг за другом. Через несколько секунд (скорее всего дольше, чем до этого) появится два сообщения с результатами вычислений. Т.е. параллельно друг другу два дополнительных потока произведут вычисления, а затем выведут результат на экран.

Сложно предположить, сколько времени займут вычисления при выполнении двух параллельных потоков. Это зависит от того, сколько **физических** ядер у процессора, а также от того, на одном или на двух разных **физических** ядрах выполняются потоки. Если наши два потока будут выполняться на двух разных физических ядрах, то время их выполнения будет таким же, как время выполнения одного потока, который мы запускали это этого. Но если они будут выполняться на одном физическом ядре, то время их выполнения будет примерно в 2 или 3 раза больше (т.е. мы не получим реального распараллеливания вычислений, хотя интерфейс пользователя и не будет подвисать).

Я не случайно выделил слово «физических», поскольку в современных процессорах есть понятие «логическое ядро» (можете почитать в Интернете про «Hyper-Threading»). Если оба потока будут работать на двух логических ядрах одного и того же физического ядра, то мы также не получим реального распараллеливания вычислений.

**Замечание по выдаче окна с сообщением из дополнительного потока**

В данном примере для выдачи пользователю сообщения из дополнительного потока используется функция MyShowMessage. В ней производится вызов стандартной WinAPI-функции MessageBox, которая отображает на экране окно с заданным текстовым сообщением. Хотя в Delphi и имеется похожая функция для вывода сообщений – ShowMessage, однако её нельзя вызывать из дополнительного потока, поскольку она не является «потокобезопасной», т.е. при обращении к ней из дополнительного потока могут происходить сбои в работе программы (может и не каждый раз).

Несмотря на то, что в данном примере код параллельного потока выдаёт пользователю окно с сообщением, я не рекомендую так делать в реальных программах. Важно помнить, что визуальное информирование пользователя должно выполняться из основного потока, тогда мы сможем избежать многих проблем.

**Замечания по использованию свойства TThread.FreeOnTerminate**

Обратите внимание, что в примере используется свойство ***FreeOnTerminate*** класса ***TThread***. В начале метода ***TMyThread.Execute*** мы выставили ему значение ***True***. Сделано этого для того, чтобы программа автоматически уничтожила объект ***TMyThread*** (т.е. вызвала деструктор объекта) сразу после завершения работы метода ***Execute***. Поскольку мы не завели отдельной переменной для хранения ссылки на созданный объект ***TMyThread***, то единственный способ уничтожить созданный объект – попросить его самостоятельно себя уничтожить сразу после завершения работы метода ***Execute***. Для этого мы и воспользовались свойством ***FreeOnTerminate***.

Это лишь один из способов управления временем жизни объекта потока. Другие способы будет представлены в следующих разделах.

**Запущенный поток не завершается при выходе из программы**

В данном примере есть следующая проблема: мы можем закрыть программу, не дожидаясь окончания работы дополнительного потока. Хотя для первого примера это не является проблемой, однако в реальных программах такая ситуация является недопустимой. Запомните правило: при выходе из программы Вы должны **корректно** завершать все созданные Вами дополнительные потоки. Можно сказать по-другому: нельзя допустить завершения работы программы, если имеются запущенные Вами дополнительные потоки. Если Вы не будете следовать этому правилу, то Ваша программа с большой вероятностью будет глючить.

# 4. Управление временем жизни потоков

При программировании на языке Delphi программист должен самостоятельно думать о том, каким образом будет уничтожаться тот или иной созданный им объект.

При программировании дополнительных потоков мы должны определиться, каким образом будет завершаться работа метода Execute и каким образом будет уничтожаться созданный нами объект потока. При этом мы должны гарантированно завершить работу потока и уничтожить объект потока при выходе из программы. Если поток выполняет очень длительную важную задачу и его нельзя прерывать, то мы должны заблокировать возможность выхода из программы до момента завершения работы потока.

В зависимости от задачи, которую решает дополнительный поток, код в методе Execute может выполняться однократно либо многократно.

Примеры однократного выполнения кода в методе Execute:

1) Выполнение вычислений при нажатии пользователем кнопки. В этом случае вполне уместно при каждом нажатии пользователем кнопки создавать дополнительный поток и запускать его.

2) Выполнение запроса к базе данных и обработка данных при срабатывании таймера. Здесь следует пояснить, что таймер не должен срабатывать слишком часто, иначе расходы на запуск и уничтожение потока могут оказаться значительными. Также хочу отметить, что при срабатывании таймера не следует обращаться к базе данных из основного потока, лучше это делать из дополнительного потока, разумеется, в отдельном подключении.

3) Выполнение заданной функции или процедуры в контексте дополнительного потока с навешиванием модальной формы ожидания окончания выполнения заданной функции.

4) Можно придумать ещё сотню различных ситуаций, но такой необходимости у нас нет!

Многократное выполнение кода в методе Execute используется при периодическом выполнении какой-либо задачи, например:

1) Раз в секунду выполняется запрос к БД и при наличии нового задания производится обмен с каким-либо устройством.

2) Раз в 15 секунд выполняется синхронизация данных с сервером.

3) 2 раза в секунду выполняется опрос состояния какого-либо устройства (например, фискального регистратора, топливно-раздаточной колонки, системы измерения уровня и т.д.).

4) Можно также придумать ещё сотню различных ситуаций.

|  |
| --- |
| Внимание! Если периодическая задача выполняется редко (например, каждые 10 минут), рекомендуется каждый раз (если это не сложно!) для такой задачи создавать новый поток. Вероятно, это лучше, чем часами удерживать дополнительный поток в спящем состоянии (особенно, если вы разрабатываете 32-разрядное Windows-приложение). |

## 4.1 Использование свойства TThread.Terminated для выхода из метода Execute

В том случае, если поток запущен на длительное время (например, пока пользователь не закроет программу) и периодически выполняет одну и ту же задачу, необходимо предусмотреть механизм завершения работы метода ***Execute*** (работа потока завершается именно после выхода из метода ***Execute***). Код внутри метода ***Execute*** должен периодически проверять флаг (например, переменную логического типа), который программа должна установить для того, чтобы поток знал, что пора завершаться.

Свойство ***TThread.Terminated*** как раз и является тем флагом, который необходимо периодически проверять для завершения работы метода ***Execute***. Ниже представлен пример реализации метода ***Execute***, в котором анализируется значения свойства ***Terminated*** (см. папку Ex2).

procedure TMyLongThread.Execute;

procedure WaitTimeout(ATimeOut: Integer);

begin

Sleep(ATimeOut);

end;

begin

while not Terminated do

begin

DoUsefullTask;

WaitTimeout(10000); // Ожидаем таймаут 10 сек.

end;

end;

В данном примере реализован бесконечный цикл ***while***, критерием выхода из которого является значение свойства ***Terminated***, равное True.

Метод ***DoUsefullTask*** имитирует выполнение полезной работы, которая длится 5 секунд.

После каждой итерации выполнения полезной работы поток должен на какое-то время переходить в спящее состояние. Напомню, что в спящем состоянии поток не создаёт какой-либо нагрузки на процессор. Благодаря этому мы даём возможность работать другим потокам (в том числе в других процессах).

Для перевода потока в спящее состояние мы вызываем вложенную (nested) процедуру WaitTimeout и передаём ей длительность ожидания. В свою очередь в ней происходит вызов стандартной WinAPI-функции Sleep, после чего поток переходит в спящее состояние, длительность которого задана в миллисекундах (в примере – 10 секунд).

|  |
| --- |
| Внимание! Если поток переведён в спящее состояние с помощью функции Sleep, то не существует другого способа выйти из этого состояния кроме истечения указанного временного периода. Попробуйте в примере (Ex2) запустить параллельный поток (с помощью кнопки ***btnRunParallelThread***), а затем закрыть программу. Вы обнаружите, что выход из программы произойдет не сразу, а спустя какое-то время (максимум 15 секунд). На практике вызов функции Sleep для длительных задержек является недопустимым! |

## 4.2 Главная форма отвечает за прекращение работы потока и уничтожение объекта потока

Ниже приведён полный пример из каталога Ex2. В нём, как уже было сказано, код в методе Execute в бесконечном цикле выполняет заданную задачу и засыпает на некоторое время. Работа метода Execute завершается только при выходе из программы:

unit Ex2Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls;

type

TMyLongThread = class(TThread)

private

procedure DoUsefullTask; // Процедура для имитации полезной работы

public

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunParallelThread: TButton;

procedure btnRunParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

private

{ Private declarations }

MyThread: TMyLongThread;

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunParallelThreadClick(Sender: TObject);

begin

// Запускает параллельный поток

if MyThread = nil then

MyThread := TMyLongThread.Create(False)

else

raise Exception.Create('Дополнительный поток уже запущен!');

end;

{ TMyLongThread }

procedure TMyLongThread.DoUsefullTask;

begin

// Реальный поток может выполнять какую угодно полезную работу

// В учебных целях делаем паузу 5 секунд для имитации задержки, которая

// может возникнуть при выполнении полезной работы

Sleep(5000);

end;

procedure TMyLongThread.Execute;

procedure WaitTimeout(ATimeOut: Integer);

begin

Sleep(ATimeOut);

end;

begin

while not Terminated do

begin

DoUsefullTask;

WaitTimeout(10000); // Ожидаем таймаут 10 сек.

end;

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

// При закрытии программы необходимо завершить работу потока

// и уничтожить объект потока MyThread

if MyThread <> nil then

MyThread.Free;

end;

end.

Обратите внимание на следующие особенности данного примера:

1) не используется свойство ***FreeOnTerminate***, т.е. объект потока **не** уничтожает себя автоматически после завершения метода ***Execute***;

2) при создании объекта потока ссылка на созданный объект запоминается в переменной ***MyThread***, которая является полем класса ***TForm1***;

3) блокируется создание нескольких объектов класса ***TMyLongThread***;

4) завершение работы потока и уничтожение объекта потока выполняется при выходе из программы путём вызова стандартного метода ***MyThread.Free*** (в обработчике ***FormDestroy***). Хотя технически и нет необходимости проверять переменную ***MyThread*** на равенство **nil** перед вызовом метода ***Free***, однако в данном случае проверка на **nil** делает код более наглядным, поскольку программа не обязывает пользователя нажимать кнопку ***btnRunParallelThread***.

Важно понимать хотя бы приблизительно, что происходит при вызове ***MyThread.Free***. А происходит примерно следующее:

1) Производится вызов метода ***TThread.Terminate***, который выставляет свойству ***TThread.Terminated*** значение ***True***.

2) Главный поток (т.е. поток, управляющий интерфейсом пользователя) переводится в спящее состояние до тех пор, пока в дополнительном потоке не произойдет выход из метода ***Execute***.

3) В методе ***Execute*** класса ***TMyLongThread*** рано или поздно произойдёт проверка значения свойства ***Terminated*** и бесконечный цикл ***while*** прервётся, а после этого произойдёт выход из метода ***Execute*** и работа параллельного потока будет завершена.

4) Главный поток вновь возобновит свою работу и окончательно уничтожит объект потока ***MyThread***.

|  |
| --- |
| Совет! Старайтесь и Вы сохранять ссылки на создаваемые потоки и уничтожать объекты потоков при закрытии (или уничтожении) форм, в которых Вы создавали соответствующий поток! |

## 4.3 Главная форма отвечает за уничтожение объекта потока с разовой задачей

Следующий пример (Ex3) интересен тем, что дополнительный поток выполняет разовую задачу длительностью 5 секунд. Обратите внимание, что после завершения работы дополнительного потока объект потока ***MyThread*** всё ещё жив, и жить он будет до тех пор, пока не будет произведён выход из программы. Также в примере добавлена визуализация работы дополнительного потока с помощью модуля ***ProgressViewer.pas***. Я его иногда использую в некоторых рабочих проектах, но лицензия не запрещает его использовать всём желающим. Думаю, что так будет интереснее наблюдать за работой потоков.

unit Ex3Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, ProgressViewer;

type

TMyShortThread = class(TThread)

private

procedure DoUsefullTask; // Процедура для имитации полезной работы

public

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunParallelThread: TButton;

procedure btnRunParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

private

{ Private declarations }

MyThread: TMyShortThread;

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunParallelThreadClick(Sender: TObject);

begin

// Запускает параллельный поток. Если объект потока уже создан,

// то уничтожает его.

if MyThread <> nil then

FreeAndNil(MyThread);

MyThread := TMyShortThread.Create(False);

end;

{ TMyShortThread }

procedure TMyShortThread.DoUsefullTask;

var

AProgress: TProgressViewer;

begin

// Реальный поток может выполнять какую угодно полезную работу

// В учебных целях делаем паузу 5 секунд для имитации задержки, которая

// может возникнуть при выполнении полезной работы

AProgress := TProgressViewer.Create('Выполняется поток TMyShortThread');

Sleep(5000);

AProgress.TerminateProgress;

end;

procedure TMyShortThread.Execute;

begin

DoUsefullTask;

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

// При закрытии программы необходимо завершить работу потока

// и уничтожить объект потока MyThread

if MyThread <> nil then

MyThread.Free;

end;

end.

Хотелось бы отменить ещё некоторые особенности этого примера:

1) При нажатии кнопки ***btnRunParallelThread*** программа уничтожает объект потока, если он был запущен ранее. Предлагаю поэкспериментировать с этой кнопкой – понажимать её несколько раз. Вы обнаружите, что если окно с сообщением «Выполняется поток TMyShortThread» отображается на экране, то нажатие кнопки ***btnRunParallelThread*** приводит к подвисанию интерфейса пользователя до тех пор, пока окно с сообщением не исчезнет. Затем окно с сообщением появится ещё раз, и отсчёт времени начнётся с нуля.

2) При выходе из программы возможны 3 варианта:

- Вариант 1: пользователь не нажимал кнопку ***btnRunParallelThread***, поэтому объект потока не создан и ссылка ***MyThread*** равна ***nil***. В этом случае программа гарантированно закроется без задержки.

- Вариант 2: пользователь нажал кнопку ***btnRunParallelThread*** и сразу, не дожидаясь окончания работы потока (т.е. сообщение «Выполняется поток TMyShortThread» ещё висело на экране), закрыл программу. В этом случае произойдет подвисание интерфейса пользователя (максимум 5 секунд), затем, после завершения работы дополнительного потока, произойдет уничтожение объекта потока, после чего программа закроется.

- Вариант 3: пользователь нажал кнопку ***btnRunParallelThread***, подождал более 5 секунд (сообщение «Выполняется поток TMyShortThread» исчезло) и закрыл программу. В этом случае произойдет вызов метода ***MyThread.Free***, который просто уничтожит объект потока, который к этому моменту уже завершил свою работу, а после этого программа закроется.

## 4.4 Главная форма отвечает за уничтожение нескольких долгоживущих потоков

Мы уже разобрались, как правильно завершить работу одного долгоживущего потока при выходе из программы. А как быть, если таких потоков несколько (например, два или три), а мы хотим максимально ускорить закрытие программы? В этом случае мы должны заранее сообщить каждому потоку о необходимости завершения их работы, а уничтожение объектов потоков выполнить в последнюю очередь. Демонстрация представлена в следующем примере (Ex4):

unit Ex4Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, ProgressViewer;

type

TMyLongThread = class(TThread)

private

FTaskNum: Integer;

procedure DoUsefullTask1; // Первая задача

procedure DoUsefullTask2; // Вторая задача

procedure DoFinalizeTask; // Задача запускается при завершении работы потока

public

constructor Create(TaskNum: Integer);

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunParallelThreads: TButton;

Label1: TLabel;

cbTerminateMode: TComboBox;

procedure btnRunParallelThreadsClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

private

{ Private declarations }

MyThread1: TMyLongThread; // Поток для первой задачи

MyThread2: TMyLongThread; // Поток для второй задачи

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunParallelThreadsClick(Sender: TObject);

begin

// Запускает параллельный поток для задачи 1

if MyThread1 = nil then

MyThread1 := TMyLongThread.Create(1);

// Запускает параллельный поток для задачи 2

if MyThread2 = nil then

MyThread2 := TMyLongThread.Create(2);

end;

{ TMyLongThread }

constructor TMyLongThread.Create(TaskNum: Integer);

begin

inherited Create(False); // Вызываем родительский конструктор

// Запоминаем параметр TaskNum. Он нужен в методе Execute

FTaskNum := TaskNum;

end;

procedure TMyLongThread.DoFinalizeTask;

begin

Sleep(5000); // Данная условная задача занимает 5 секунд

end;

procedure TMyLongThread.DoUsefullTask1;

begin

Sleep(1000); // Данная условная задача занимает 1 секунду

end;

procedure TMyLongThread.DoUsefullTask2;

begin

Sleep(2000); // Данная условная задача занимает 2 секунды

end;

procedure TMyLongThread.Execute;

procedure WaitTimeout(ATimeOut: Integer);

begin

Sleep(ATimeOut);

end;

begin

while True do

begin

if Terminated then

begin

DoFinalizeTask; // Некоторые действия при завершении потока

Exit; // Завершаем работу потока

end else

begin

if FTaskNum = 1 then

DoUsefullTask1 // Запускаем задачу 1

else

DoUsefullTask2; // Запускаем задачу 2

if not Terminated then // Дополнительная проверка не повредит!

WaitTimeout(1000); // Ожидаем таймаут 1 сек

end;

end;

end;

procedure TForm1.FormDestroy(Sender: TObject);

var

AProgress: TProgressViewer;

begin

AProgress := TProgressViewer.Create('Выход из программы');

try

if cbTerminateMode.ItemIndex = 1 then

begin // Выбран режим "Одновременно (быстрее)"

if Assigned(MyThread1) then

MyThread1.Terminate; // Выставляем флаг Terminated

if Assigned(MyThread2) then

MyThread2.Terminate; // Выставляем флаг Terminated

end;

MyThread1.Free;

MyThread2.Free;

finally

AProgress.TerminateProgress;

end;

end;

end.

В этом примере имеются следующие особенности:

1) Поток ***TMyLongThread*** выполняет периодически указанное действие: ***DoUsefullTask1*** либо ***DoUsefullTask2***. Номер действия (TaskNum) указан при создании потока, например:

MyThread1 := TMyLongThread.Create(1);

MyThread2 := TMyLongThread.Create(2);

2) При нажатии кнопки ***btnRunParallelThreads*** создаются два потока: ***MyThread1*** (для решения задачи 1) и ***MyThread2*** (для решения задачи 2). Для обоих потоков используется один и тот же класс ***TMyLongThread***.

3) В классе ***TMyLongThread*** объявлен собственный конструктор ***Create***.

|  |
| --- |
| Важно! Обратите внимание, что при реализации собственного конструктора мы должны обязательно вызвать родительский конструктор, например: ***inherited Create(False)***! |

При вызове конструктора ***TThread.Create*** требуется указать логическое значение (параметр ***CreateSuspended***). Мы указали значение ***False***. Это значит, что мы **не хотим** запускать поток в замороженном (Suspended) состоянии. Наш поток должен запускаться сразу, как только отработает конструктор. Если бы мы создавали поток в замороженном состоянии, то для запуска метода ***Execute*** нам пришлось бы вызвать метод ***TThread.Resume*** либо его современную версию ***TThread.Start***. Такая необходимость возникает редко (по крайней мере, в моих проектах).

|  |
| --- |
| Важно! Вы должны знать, что запуск метода ***Execute*** в дополнительном потоке произойдет только после того, как работа конструктора полностью завершится! |

4) Логика бесконечного цикла ***while*** в методе ***Execute*** изменена. Если флаг ***Terminated*** выставлен в ***True***, то выполняются некоторые действия (метод ***DoFinalizeTask***) и производится выход из метода Execute. Если же флаг ***Terminated*** имеет значение ***False***, то осуществляется вызов метода ***DoUsefullTask1*** либо ***DoUsefullTask2***, в зависимости от переменной ***FTaskNum***. Далее, перед вызовом функции ***WaitTimeout***, производится дополнительная проверка флага ***Terminated***. Если он выставлен, то не нужно тратить лишнее время на ожидание, вместо этого мы должны как можно быстрее завершить работу потока!

5) В программе пользователь может выбрать один из способов завершения работы потоков: «последовательно» или «одновременно». Если выбрать вариант «последовательно» и закрыть программу, то в методе ***TForm1.FormDestroy*** будет произведён сначала вызов ***MyThread1.Free***, а затем вызов ***MyThread2.Free***. В том случае, если пользователь предварительно запустил потоки, мы увидим окно с текстом «Выход из программы», которое будет отображаться примерно 10 секунд.

Если же мы выберем вариант «одновременно» и закроем программу, то окно с текстом «Выход из программы», будет отображаться примерно 5 секунд. Это происходит благодаря тому, что перед вызовом ***MyThread1.Free*** осуществляется уведомление потоков о необходимости завершения их работы путем вызова метода ***TThread.Terminate*** (***MyThread1.Terminate*** и ***MyThread2.Terminate***). При вызове ***MyThread1.Free*** ожидается завершение потока ***MyThread1***, а затем выполняется уничтожение объекта потока. При вызове ***MyThread2.Free*** ожидать завершения второго потока долго не придётся, т.к. завершаться он начал одновременно с первым потоком после вызова ***MyThread2.Terminate***.

|  |
| --- |
| Совет! Старайтесь при выходе из программы заранее вызывать метод ***TThread.Terminate*** для каждого из запущенных потоков! |

|  |
| --- |
| Внимание! Для данного примера перед вызовом метода ***Terminate*** необходимо проверять ссылку на объект потока. Если в ней содержится значение **nil**, то вызывать метод ***Terminate*** нельзя, т.к. это приведёт к ошибке доступа к памяти! Также нельзя обращаться к полям, свойствам и методам объекта, который не создан! Единственный метод, который допускает свой вызов для несозданного объекта: ***TObject.Free***. В нём проверяется значение ссылки на объект и если там **nil**, то деструктор не вызывается. |

## 4.5 Использование списка TObjectList для хранения ссылок на объекты потоков

В том случае, если в Вашем приложении множество потоков, каждый из которых выполняет различные задачи, Вы можете использовать список TObjectList для хранения ссылок на создаваемые объекты потоков. В таком случае Вам не придётся писать код уничтожения каждого потока, достаточно вызвать метод ***Free*** для списка ***TObjectList***.

В следующем примере (Ex5) демонстрируется работа со списком ***TObjectList***. Как и в предыдущем примере, в нём реализовано 2 режима завершения работы потоков при выходе из программы: последовательный и одновременный.

Ниже представлен исходный код модуля Ex5Unit:

unit Ex5Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, ProgressViewer, Contnrs, MTUtils;

type

TMyLongThread1 = class(TThread)

private

FUsefullTaskTime: Integer;

public

constructor Create(UsefullTaskTime: Integer);

procedure Execute; override;

end;

TMyLongThread2 = class(TThread)

public

procedure Execute; override;

end;

TMyLongThread3 = class(TThread)

private

FUsefullTaskTime: Integer;

public

constructor Create(UsefullTaskTime: Integer);

procedure Execute; override;

end;

TMyLongThread4 = class(TThread)

public

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunParallelThreads: TButton;

Label1: TLabel;

cbTerminateMode: TComboBox;

procedure btnRunParallelThreadsClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure FormCreate(Sender: TObject);

private

{ Private declarations }

FList: TObjectList; // Потоки для первой и второй задачи

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunParallelThreadsClick(Sender: TObject);

begin

// Запускаем 4 параллельных потока

if FList.Count = 0 then

begin

FList.Add(TMyLongThread1.Create(1000));

FList.Add(TMyLongThread2.Create(False));

FList.Add(TMyLongThread3.Create(2000));

FList.Add(TMyLongThread4.Create(False));

end;

end;

{ TMyLongThread }

constructor TMyLongThread1.Create(UsefullTaskTime: Integer);

begin

inherited Create(False); // Вызываем родительский конструктор

FUsefullTaskTime := UsefullTaskTime;

end;

procedure TMyLongThread1.Execute;

begin

while not Terminated do

begin

EmulateUsefullWork(FUsefullTaskTime);

ThreadWaitTimeout(Self, 60000); // Ожидаем таймаут 60 сек

end;

Sleep(5000); // Оставлено для демонстрации режима "Одновременно"

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

FList := TObjectList.Create;

end;

procedure TForm1.FormDestroy(Sender: TObject);

var

AProgress: TProgressViewer;

I: Integer;

begin

AProgress := TProgressViewer.Create('Выход из программы');

try

if cbTerminateMode.ItemIndex = 1 then

begin // Выбран режим "Одновременно (быстрее)"

// Выставляем флаг Terminated для всех потоков. Можно использовать

// родительский класс TThread для операции приведения типов.

for I := 0 to FList.Count - 1 do

TThread(FList[I]).Terminate;

end;

// При уничтожении списка TObjectList будут уничтожены все объекты потоков

FList.Free;

finally

AProgress.TerminateProgress;

end;

end;

{ TMyLongThread2 }

procedure TMyLongThread2.Execute;

begin

while not Terminated do

begin

EmulateUsefullWork(2000);

ThreadWaitTimeout(Self, 60000); // Ожидаем таймаут 60 сек

end;

Sleep(5000); // Оставлено для демонстрации режима "Одновременно"

end;

{ TMyLongThread3 }

constructor TMyLongThread3.Create(UsefullTaskTime: Integer);

begin

inherited Create(False);

FUsefullTaskTime := UsefullTaskTime;

end;

procedure TMyLongThread3.Execute;

begin

while not Terminated do

begin

EmulateUsefullWork(FUsefullTaskTime);

ThreadWaitTimeout(Self, 60000); // Ожидаем таймаут 60 сек

end;

Sleep(5000); // Оставлено для демонстрации режима "Одновременно"

end;

{ TMyLongThread4 }

procedure TMyLongThread4.Execute;

begin

while not Terminated do

begin

EmulateUsefullWork(1000);

ThreadWaitTimeout(Self, 60000); // Ожидаем таймаут 60 сек

end;

Sleep(5000); // Оставлено для демонстрации режима "Одновременно"

end;

end.

Обратите внимание на следующие особенности данного примера:

1) Реализованы 4 класса-наследника от ***TThread: TMyLongThread1, TMyLongThread2, TMyLongThread3, TMyLongThread4***. Это сделано для того, чтобы показать читателю, что в список ***TObjectList*** можно добавить произвольные объекты потоков и в любом количестве.

2) Список ***FList*** создаётся в конструкторе формы и уничтожается в деструкторе. Класс ***TObjectList*** объявлен в модуле «***Contnrs***» (вечно забываю, какие буквы в этом слове и приходится пользоваться поиском по исходникам Delphi).

3) Для организации паузы в работе потоков используется простая и удобная функция ***ThreadWaitTimeout***. из модуля «MTUtils.pas». Она будет приведена ниже.

4) Для эмуляции полезной работы потоков используется функция ***EmulateUsefullWork*** из модуля «MTUtils.pas». Она состоит из одной строчки: ***Sleep(WorkTime)***.

5) В режиме одновременного завершения работы потоков в методе ***TForm1.FormDestroy*** в цикле выставляется значение ***True*** свойству ***Terminated*** для каждого объекта-потока из списка ***FList***.

6) При создании объектов-потоков (в обработчике ***TForm1.btnRunParallelThreadsClick***) переменные для объектов-потоков не объявляются. Результат, возвращаемый при создании объекта (например, ***TMyLongThread1.Create(1000)***), можно сразу сохранять в списке ***FList***.

## 4.6 Простой пример организации паузы в работе потока

Ниже приведён код функции ***ThreadWaitTimeout*** из модуля «MTUtils.pas»:

procedure ThreadWaitTimeout(AThread: TThread; ATimeout: Integer);

var

p: TTimeInterval;

T: TThreadAccessTerminated;

begin

// Получаем доступ к protected-свойству Terminated

T := TThreadAccessTerminated(AThread);

// Если поток нужно завершить, то сразу выходим из цикла

if T.Terminated then Exit;

p.Start; // Начинаем замер времени

while True do

begin

if T.Terminated or (p.ElapsedMilliseconds >= ATimeout) then

Exit;

// Замораживаем поток примерно на 20 мс

Sleep(ThreadWaitTimeoutSleepTime);

end;

end;

Организация паузы в этой функции основана на циклическом вызове функции Sleep с небольшим таймаутом (20 мс). При этом контролируется свойство ***TThread.Terminated***. Опытные программисты могут использовать более сложные способы организации паузы, например, с использованием объекта синхронизации «Event». Однако на текущем этапе обучения поднимать тему объектов синхронизации считаю преждевременным. Для большинства практических задач Вам будет достаточно функции ThreadWaitTimeout. Задержку в 20 мс считаю оптимальной, она обеспечивает достаточную точность паузы, но при этом не приводит к слишком частым переключениям контекста, т.е. практически не оказывает нагрузки на процессор. При необходимости Вы можете присвоить глобальной переменной ThreadWaitTimeoutSleepTime другое значение, отличное от 20.

Также обратите внимание, что в этой функции для организации замеров промежутков времени используется структура (record) TTimeInterval, объявленная в модуле TimeIntervals.pas. Вы можете свободно использовать модуль TimeIntervals.pas в собственных проектах. Обсуждение данного модуля ведётся на форуме: <https://www.sql.ru/forum/1326896/modul-dlya-udobnogo-zamera-tochnyh-intervalov-vremeni>

## 4.7 Использование свойства FreeOnTerminate для автоматического уничтожения объекта потока при завершении работы потока

Давайте снова рассмотрим первый пример (Ex1). В нём был показан вариант реализации потока с использованием свойства ***FreeOnTerminate***. Как уже было отмечено ранее, при значении свойства ***FreeOnTerminate=True*** объект потока уничтожается автоматически, как только завершается работа потока. Вроде удобно, да? Не требуется заводить переменную для объекта-потока, не нужно писать вызов метода ***Free*** для уничтожения объекта потока. Однако тут не всё так просто! Я бы даже сказал, что у такого подхода больше минусов, чем плюсов. Отмечу такие минусы:

1) Сложно контролировать состояние такого потока, поскольку не объявлена переменная для объекта-потока.

2) Сложно управлять таким потоком (по той же причине).

3) Если же мы решим объявить для такого потока переменную и будем пытаться с нею работать, то рискуем нарваться на ошибку доступа к памяти «Access Violation», поскольку объект потока может быть уничтожен в любой момент времени.

4) Сложно организовать корректный выход из программы. Как уже было сказано ранее, при выходе из программы мы должны гарантировать завершение работы всех запущенных нами потоков. Но это сложно сделать, если у нас нет ссылки на объект потока.

В первом примере было бы лучше использовать список TObjectList для хранения ссылок на созданные объекты потоков. Однако такое усложнение могло бы отпугнуть читателя на этапе знакомства с первым примером.

Вывод такой: я не рекомендую использовать данный режим уничтожения объектов потоков (особенно для новичков). Если Вы по каким-то причинам решили его использовать, то следует проявлять осторожность и предусмотреть механизмы контроля состояния потока и корректного выхода из программы.

## 4.8 Организация корректного завершения работы программы при использовании свойства FreeOnTerminate

В следующем примере (Ex6) демонстрируется один из подходов к организации корректного выхода из программы при использовании свойства FreeOnTerminate. Ниже представлен код модуля Ex6Unit:

unit Ex6Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, ProgressViewer, MTUtils , TimeIntervals;

type

TMyThread = class(TThread)

private

FThreadNum: Integer;

public

procedure Execute; override;

constructor Create;

destructor Destroy; override;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

ThreadCount: Integer;

StopThreadsFlag: Boolean;

implementation

{$R \*.dfm}

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

begin

// Запускаем параллельный поток

TMyThread.Create;

end;

procedure TForm1.FormDestroy(Sender: TObject);

var

pv: TProgressViewer;

begin

// Выставляем флаг StopThreadsFlag, чтобы все потоки завершились

StopThreadsFlag := True;

// Задерживаем выход из программы, пока не будут завершены все потоки

if ThreadCount > 0 then

begin

pv := TProgressViewer.Create('Ожидаем завершение потоков');

while ThreadCount > 0 do

Sleep(10);

pv.TerminateProgress;

end;

end;

{ TMyThread }

constructor TMyThread.Create;

begin

inherited Create(False);

// Увеличиваем глобальную переменную ThreadCount на 1 и запоминаем

// полученное значение

FThreadNum := InterlockedIncrement(ThreadCount);

end;

destructor TMyThread.Destroy;

begin

inherited;

// Уменьшаем глобальную переменную ThreadCount на 1

InterlockedDecrement(ThreadCount);

end;

procedure TMyThread.Execute;

var

ti: TTimeInterval;

begin

FreeOnTerminate := True;

// Организуем паузу 10 секунд. При этом каждые 20 мс

// проверяем флаг StopThreadsFlag

ti.Start;

while ti.ElapsedSeconds < 10 do

begin

// Заканчиваем ожидание, если выставлен флаг StopThreadsFlag

if StopThreadsFlag then Break;

Sleep(20);

end;

ThreadShowMessageFmt('Работа потока #%d завершена!', [FThreadNum]);

end;

end.

Особенности данного примера:

1. Введены две глобальные переменные: ThreadCount (количество запущенных потоков) и StopThreadsFlag (флаг завершения потоков).

2. Для увеличения переменной ThreadCount на единицу используется потокобезопасная функция InterlockedIncrement. В принципе, в этом примере мы могли бы использовать операцию инкремента Inc (либо ThreadCount := ThreadCount + 1), однако, операция Inc не является потокобезопасной и при одновременном вызове Inc(ThreadCount) из нескольких потоков мы могли бы получить некорректное значение (например, 9 вместо 10).

3. Каждому запущенному потоку присваивается порядковый номер FThreadNum. Для этого используется результат, который возвращает функция InterlockedIncrement. Благодаря функции InterlockedIncrement гарантируется, что каждый поток получит свой уникальный порядковый номер. Если бы мы использовали следующий код для увеличения ThreadCount и запоминания FThreadNum:

ThreadCount := ThreadCount + 1; // либо Inc(ThreadCount)

FThreadNum := ThreadCount;

то столкнулись бы со следующими проблемами (с вероятностью примерно 1%):

а) при одновременном вызове ThreadCount := ThreadCount + 1 из нескольких потоков мы можем получить некорректное значение (например, 9 вместо 10). Тут есть очень простое объяснение. Допустим, 2 потока начали выполнять команду инкремента одновременно, в этом момент в переменной ThreadCount хранилось значение 8. Далее оба потока одновременно увеличили это значение на 1, получилось значение 9. Функция InterlockedIncrement такого не допустит!

б) даже если инкремент ThreadCount был выполнен корректно, существует риск того, что в поле FThreadNum нескольких потоков попадёт одно и то же значение. Причина этого может заключаться в том, что планировщик задач Windows может в любой момент приостановить работу потока, например, непосредственно перед инструкцией FThreadNum := ThreadCount. После того, как планировщик возобновит работу потока, в переменной ThreadCount может оказаться совсем другое значение. Функция InterlockedIncrement такого не допустит!

4. Для уменьшения переменной ThreadCount на единицу используется потокобезопасная функция InterlockedDecrement в методе-деструкторе TMyThread.Destroy.

5. Для остановки потоков используется глобальная переменная StopThreadsFlag. Ей присваивается значение True в методе TForm1.FormDestroy. Переменная StopThreadsFlag проверяется в цикле в методе TMyThread.Execute. Цикл прерывается после того, как в переменной StopThreadsFlag окажется значение True. Перед выходом из метода TMyThread.Execute выдаётся сообщение с текстом «Работа потока #%d завершена!». Для вывода сообщения используется функция ThreadShowMessageFmt, объявленная в модуле MTUtils. Также обратите внимание, что деструктор будет вызван после выхода из метода TMyThread.Execute!

6. Для организации цикла ожидания 10 секунд в методе TMyThread.Execute используется тип TTimeInterval, объявленный в модуле TimeIntervals. Обратите внимание, что поддержка методов в записях появилась в версии Delphi 2007, поэтому в более ранних версиях Delphi примеры не скомпилируются.

7. При выходе из приложения в методе TForm1.FormDestroy выставляется флаг «StopThreadsFlag := True», а затем выполняется цикл ожидания нулевого значения в переменной ThreadCount. Данная переменная проверяется каждые 10 мс (пауза в работе основного потока организована с помощью Sleep(10)). Выход из программы произойдет после того, как пользователь нажмёт кнопку «ОК», во всех сообщениях, открытых из дополнительных потоков.

8. Кнопка btnRunInParallelThread может быть нажата произвольное количество раз.

# 5. Передача информации из дополнительного потока в основной

При разработке многопоточных приложений в Delphi очень актуальной является задача передачи информации, подготовленной в дополнительном потоке, в главный поток. Это весьма сложная тема, требующая от Delphi-программиста повышенного внимания, т.к. при недостаточных знаниях у программиста есть очень высокий шанс сделать программу глюченной и нестабильной.

## 5.1 Обращение к визуальным компонентам формы из дополнительного потока – как нельзя делать

|  |
| --- |
| Внимание! Не допускается обращение к визуальным компонентам из дополнительного потока! Т.е. внутри метода Execute мы не можем просто так взять, и изменить что-либо на форме. Например, следующие инструкции внутри метода Execute **недопустимы**:  Form1.Label1 := 'Привет';  Form1.Show;  Form1.ShowModal;  Form1.Button1.Visible := True;  Form1.Memo1.Lines.Add('Привет');  и т.д. |

Запомните, что интерфейс пользователя обслуживается основным (главным) потоком! Любые изменения, влияющие на интерфейс пользователя должны всегда выполняться в контексте основного потока! Это не является уникальной особенностью Delphi. Для других языков программирования, в которых имеется поддержка разработки визуальных интерфейсов, действует такое же правило!

## 5.2 Использование метода TThread.Synchronize для передачи данных в основной поток

В Delphi есть механизм, позволяющий дополнительному потоку выполнить запуск процедуры (метода) в контексте основного потока. Для этого предназначен метод TThread.Synchronize. Суть работы данного метода состоит в следующем:

1. Метод Synchronize запоминает в списке SyncList (глобальная переменная типа TList) переданный ему метод (метод представляет собой структуру-запись TMethod, в которой есть всего 2 поля: адрес функции и указатель на объект). Например, если мы выполним такой код: TThread.Synchronize (nil, Form1.Show), то в списке будет сохранён указатель на объект (т.е. Form1) и адрес процедуры TForm.Show.

2. Основной поток периодически проверяет наличие элементов в списке SyncList. Если в списке обнаружен элемент, то производится запуск метода, хранящегося в списке.

3. Метод TThread.Synchronize, вызванный в дополнительном потоке, вернёт управление после того, как основной поток завершит вызов переданного ему метода.

|  |
| --- |
| Внимание! Обратите внимание, что в метод TThread.Synchronize можно передавать только методы-процедуры без параметров! Если нужны параметры, то Вы должны завести в своём классе потока дополнительные поля, присвоить им значения до вызова Syncronize, а затем использовать эти значения при вызове метода, который вы передали в Syncronize, в контексте основного потока. |

|  |
| --- |
| Совет! В современных версиях Delphi в метод Syncronize можно передавать анонимную процедуру. В этом случае намного реже возникает необходимость в объявлении дополнительных полей в классе потока. |

|  |
| --- |
| Внимание! Каждый вызов TThread.Synchronize несёт в себе значительные накладные расходы, поэтому старайтесь использовать его как можно реже. В идеале, старайтесь обойтись без него. |

|  |
| --- |
| Внимание! Нельзя уничтожать дополнительные потоки из метода, который Вы передаёте в TThread.Synchronize! Это приводит к зависанию программы. |

Ниже представлен пример (Ex7), в котором дополнительный поток выполняет длительные вычисления (суммирует ряд чисел от 1 до MaxValue) и периодически вызывает метод Synchronize для отображения прогресса на главной форме.

unit Ex7Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, MTUtils, ComCtrls;

type

TMyThread = class(TThread)

private

FResult: Int64;

FCurrValue: Integer;

procedure SetProgressParams;

procedure SetProgressCurrValue;

public

MaxValue: Integer;

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

ProgressBar1: TProgressBar;

Label1: TLabel;

labResult: TLabel;

edMaxValue: TEdit;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

private

{ Private declarations }

FMyThread: TMyThread;

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

begin

// Уничтожаем запущенный поток

if Assigned(FMyThread) then

FreeAndNil(FMyThread);

// Создаём поток в спящем состоянии

FMyThread := TMyThread.Create(True);

// Запоминаем длину ряда в поле MaxValue

FMyThread.MaxValue := StrToIntDef(edMaxValue.Text, 0);

// Пробуждаем поток для выполнения вычислений

FMyThread.Resume;

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

FMyThread.Free;

end;

{ TMyThread }

procedure TMyThread.Execute;

var

Res: Int64;

CurrVal: Integer;

begin

// Выставляем параметры компонента ProgressBar1

Synchronize(SetProgressParams);

// Выполняем некоторые вычисления

Res := 0;

CurrVal := 0;

while CurrVal < MaxValue do

begin

if Terminated then Break;

Inc(CurrVal);

Res := Res + CurrVal;

if CurrVal mod 10000 = 0 then

begin // Обновление прогресса выполняется только 1 раз из 10000

FCurrValue := CurrVal;

FResult := Res;

Synchronize(SetProgressCurrValue);

end;

end;

// Обновляем прогресс в конце вычислений

FCurrValue := CurrVal;

FResult := Res;

Synchronize(SetProgressCurrValue);

end;

procedure TMyThread.SetProgressCurrValue;

begin

Form1.ProgressBar1.Position := FCurrValue;

Form1.labResult.Caption := IntToStr(FResult);

end;

procedure TMyThread.SetProgressParams;

begin

Form1.ProgressBar1.Max := MaxValue;

Form1.ProgressBar1.Position := 0;

Form1.labResult.Caption := '0';

end;

end.

Особенности данного примера:

1. Объект дополнительного потока создаётся с параметром True (FMyThread := TMyThread.Create(True)). Это означает, что метод Execute не будет вызван до тех пор, пока программа не вызовет метод Resume (либо Start в современных версиях Delphi). Таким образом, Вы можете создавать поток в замороженном состоянии, а запускать его позже, когда Вам будет удобно.

2. Запуск дополнительного потока осуществляется вызовом FMyThread.Resume.

|  |
| --- |
| Внимание! В объявлении класса TThread рядом с методом Resume находится метод Suspend, предназначенный для принудительного перевода потока в спящее состояние. Вы не должны его использовать! Многие начинающие программисты пытаются управлять состоянием потока с помощью метода Suspend, однако, как правило, ни к чему хорошему это не приводит, кроме глюков в программе. |

3. Установка параметров компонента ProgressBar1, а также визуализация прогресса осуществляется вызовом метода Synchronize: Synchronize(SetProgressParams) и Synchronize(SetProgressCurrValue).

|  |
| --- |
| Внимание! Не забывайте вызывать метод Synchronize! Нет ничего проще, чем забыть это сделать (например, вызвать SetProgressCurrValue вместо Synchronize(SetProgressCurrValue))! Delphi Вам не подскажет, зато при работе программы будут возникать сбои. |

4. Перед вызовом Synchronize(SetProgressCurrValue) выполняется установка значений полям FCurrValue и FResult, поскольку они в дальнейшем используются в методе SetProgressCurrValue.

5. Вызов Synchronize(SetProgressCurrValue) выполняется каждую десятетысячную итерацию цикла (см. условие «if CurrVal mod 10000 = 0 then…»). Если вызывать Synchronize чаще, то скорость вычислений замедлится, а нагрузка на главный поток программы возрастёт.

|  |
| --- |
| Обратите внимание, что для выхода из программы не обязательно дожидаться окончания работы потока. Вызов FMyThread.Free в методе TForm1.FormDestroy не приводит к зависанию программы, несмотря на то, что в методе TMyThread.Execute последней строкой осуществляется вызов метода Synchronize. Взаимной блокировки главного и дополнительного потока не происходит, т.к. в деструкторе объекта потока вызывается код, который предотвращает взаимную блокировку. |

## 5.3 Периодическое чтение основным потоком данных, подготовленных в дополнительном потоке

В следующем примере (Ex8) задача визуализации текущего состояния вычислений, выполняющихся в дополнительном потоке, решается без использования метода Synchronize. На главной форме находится таймер, который через определённые промежутки времени (каждые 100 мс) считывает из объекта-потока FMyThread свойства CalcResult, CurrValue и ThreadStateInfo и отображает их значения на главной форме.

На мой взгляд, такой код становится проще и логичнее, чем при использовании метода Synchronize. Дополнительный поток занимается своим делом (выполняет вычисления) и не мешает главному потоку. Главный поток занимается своим делом – обслуживает интерфейс пользователя. Даже если главный поток подвиснет, что часто происходит при работе с базами данных, это никак не повлияет на работу дополнительного потока.

Вы не всегда сможете обойтись без Synchronize, но если задача позволяет и качество кода не ухудшится, то старайтесь сделать это. Но имейте ввиду, что сложные структуры, такие как строки, динамические массивы, объекты, варианты, и т.п. придётся (вероятно) защитить от одновременного доступа из разных потоков. Одной из целей данного примера является показать, как защитить строку (string).

Ниже исходный код примера Ex8:

unit Ex8Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, MTUtils, ComCtrls, ExtCtrls, SyncObjs;

type

TMyThread = class(TThread)

private

FMaxValue: Integer;

FResult: Int64;

FCurrValue: Integer;

// Информация о текущем состоянии потока

FThreadStateInfo: string;

function GetThreadStateInfo: string;

procedure SetThreadStateInfo(const Value: string);

public

constructor Create(MaxValue: Integer);

procedure Execute; override;

property CalcResult: Int64 read FResult;

property CurrValue: Integer read FCurrValue;

// Свойство для доступа к строке FThreadStateInfo с помощью

// потокозащищенных методов GetThreadStateInfo и SetThreadStateInfo

property ThreadStateInfo: string read GetThreadStateInfo

write SetThreadStateInfo;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

ProgressBar1: TProgressBar;

Label1: TLabel;

labResult: TLabel;

edMaxValue: TEdit;

Timer1: TTimer;

Label2: TLabel;

labThreadStateInfo: TLabel;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure Timer1Timer(Sender: TObject);

private

{ Private declarations }

FMyThread: TMyThread;

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

var

MaxValue: Integer;

begin

// Уничтожаем запущенный поток

if Assigned(FMyThread) then

FreeAndNil(FMyThread);

MaxValue := StrToInt(edMaxValue.Text);

ProgressBar1.Max := MaxValue;

ProgressBar1.Position := 0;

labResult.Caption := '0';

labThreadStateInfo.Caption := '???';

FMyThread := TMyThread.Create(MaxValue);

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

FreeAndNil(FMyThread);

end;

procedure TForm1.Timer1Timer(Sender: TObject);

begin

if Assigned(FMyThread) then

begin

ProgressBar1.Position := FMyThread.CurrValue;

labResult.Caption := IntToStr(FMyThread.CalcResult);

labThreadStateInfo.Caption := FMyThread.ThreadStateInfo;

end;

end;

{ TMyThread }

constructor TMyThread.Create(MaxValue: Integer);

begin

FMaxValue := MaxValue;

inherited Create(False);

end;

procedure TMyThread.Execute;

begin

ThreadStateInfo := 'Start';

while FCurrValue < FMaxValue do

begin

if Terminated then Break;

Inc(FCurrValue);

FResult := FResult + FCurrValue;

ThreadStateInfo := Format('Progress: %f%%',

[FCurrValue / FMaxValue \* 100]);

end;

ThreadStateInfo := 'Complete';

end;

function TMyThread.GetThreadStateInfo: string;

begin

// Защищаем строку с помощью критической секции. Если её убрать,

// то в главном потоке периодически будет возникать ошибка

// "Invalid pointer operation" либо "Out of memory"

StringProtectSection.Enter; // Входим в режим защиты

Result := FThreadStateInfo;

StringProtectSection.Leave; // Выходим из режима защиты

end;

procedure TMyThread.SetThreadStateInfo(const Value: string);

begin

StringProtectSection.Enter; // Входим в режим защиты

FThreadStateInfo := Value;

StringProtectSection.Leave; // Выходим из режима защиты

end;

end.

Особенности данного примера:

1. Введено дополнительное строковое поле FThreadStateInfo (информация о текущем состоянии потока). Доступ к этому полю осуществляется с помощью потокозащищённого свойства ThreadStateInfo, использующего методы GetThreadStateInfo и SetThreadStateInfo для чтения и записи строки FThreadStateInfo. Перед тем, как записать либо прочитать строку, выполняется вход в режим защиты с помощью кода StringProtectSection.Enter. После того, как работа со строкой завершилась, выполняется выход из режима защиты с помощью кода StringProtectSection.Leave.

|  |
| --- |
| Внимание! После каждого входа в режим защиты обязательно должен осуществляться выход из режима защиты. |

|  |
| --- |
| Совет. Защищать строки (и другие структуры и объекты) имеет смысл при наличии двух условий:  а) предполагается, что будет одновременный доступ к строке (структуре, объекту, массиву) из разных потоков;  б) строка (структура, объект, массив) не является «константной», т.е. она может меняться в зависимости от логики программы (в таких случаях используют ещё термин «mutable» или «мутабельный»). |

|  |
| --- |
| Внимание! Если строка (структура, объект, массив) является «константной», т.е. если значение присваивается лишь один раз и больше не меняется, то нет смысла её защищать! |

2. Глобальная переменная StringProtectSection (класс TCriticalSection) объявлена в модуле MTUtils, а соответствующий ей объект создаётся в секции initialization.

3. Непосредственный доступ к полям класса TMyThread (FResult, FCurrValue и FThreadStateInfo) допускается только из методов класса TMyThread. Сторонний код должен использовать соответствующие свойства: CalcResult, CurrValue, ThreadStateInfo. Свойства CalcResult и CurrValue обеспечивают доступ к полям FResult и FCurrValue только для чтения.

|  |
| --- |
| Что будет, если закомментировать строки StringProtectSection.Enter и StringProtectSection.Leave? У меня в этом случае в программе иногда возникают следующие ошибки: "Invalid pointer operation" либо "Out of memory". Особенно легко ошибки воспроизвести, если запустить одновременно 4 экземпляра программы. Я не могу объяснить, из-за чего выдаётся "Out of memory", а с ошибкой "Invalid pointer operation" объяснение может быть таким: при каждом присвоении новой строки переменной FThreadStateInfo происходит две вещи:  а) выделяется новая область памяти под новую строку (с помощью функции менеджера памяти GetMem);  б) освобождается память, занятая старой строкой (с помощью функции менеджера памяти FreeMem). Память будет освобождена при условии, если старая строка нигде больше не используется, т.е. у неё нулевой счётчик ссылок.  Главный поток при чтении незащищенной строки FThreadStateInfo может попасть в такую ситуацию, при которой:  а) память под старую строку уже освобождена с помощью FreeMem;  б) в переменной FThreadStateInfo всё ещё хранится указатель на старую строку. При попытке обращения к строке по недействительному указателю возникает ошибка «Invalid pointer operation».  Применение критической секции избавляет нас от этой проблемы. |

## 5.4 Использование функции SendMessage для передачи данных в основной поток

Мы уже рассмотрели возможность использования метода Synchronize для вызова указанного метода-процедуры в контексте основного потока. Также в Delphi существует альтернативный механизм передачи данных в основной поток, основанный на передаче и обработке оконных сообщений. Суть этого механизма состоит в том, что дополнительный поток вызывает стандартную Windows-функцию SendMessage (по ней я не планирую давать подробную информацию, поэтому читайте официальную документацию), а в классе формы (например, TForm1) реализуется метод обработки соответствующего оконного сообщения (этот метод вызывается в контексте основного потока). Вызов функции SendMessage является блокирующим, поэтому работа дополнительного потока будет приостановлена в точке вызова функции SendMessage до тех пор, пока не завершится работа метода обработки оконного сообщения.

Функция SendMessage объявлена следующим образом:

function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;

При вызове данной функции мы должны передать Handle компонента, в котором реализован обработчик сообщения (параметр hWnd), код сообщения (параметр Msg), а также, если Вам это необходимо, параметры (wParam, lParam).

Параметры wParam и lParam являются целочисленными (32 или 64 бита в зависимости от разрядности разрабатываемого приложения), поэтому с их помощью Вы можете передать а) целое число; б) ссылку на любой объект любого класса; в) указатель на любой блок выделенной памяти.

Метод обработки оконного сообщения объявляется с указанием ключевого слова message и кода сообщения. Пример объявления:

type

TForm1 = class(TForm)

private

procedure UMProcessMyMessage(var Msg: TMessage); message WM\_USER + 1;

public

end;

|  |
| --- |
| Внимание! Если Вы вводите свои (нестандартные коды сообщений), то необходимо использовать коды не менее WM\_USER, например: WM\_USER + 1, WM\_USER + 2, WM\_USER + 3 и т.д. |

Рекомендуется заводить константы для нестандартных оконных сообщения (иначе Вы быстро забудете, что у Вас означает WM\_USER + 1). Пример объявления такой константы:

const

UM\_PROCESS\_MY\_MESSAGE = WM\_USER + 1;

Префикс «UM\_» является сокращением от «User message».

Ниже представлен исходный код модуля из примера Ex9:

unit Ex9Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, MTUtils, ComCtrls;

const

UM\_PROGRESS\_INIT = WM\_USER + 1;

UM\_PROGRESS\_CHANGE = WM\_USER + 2;

type

TProgressData = class

CurrValue: Integer;

CalcResult: Int64;

ThreadStateInfo: string;

end;

TMyThread = class(TThread)

private

ProgressData: TProgressData;

FFormHandle: THandle;

FMaxValue: Integer;

public

procedure Execute; override;

constructor Create(AMaxValue: Integer; AFormHandle: THandle);

destructor Destroy; override;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

ProgressBar1: TProgressBar;

Label1: TLabel;

labResult: TLabel;

edMaxValue: TEdit;

Label2: TLabel;

labThreadStateInfo: TLabel;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

private

{ Private declarations }

FMyThread: TMyThread;

procedure UMProgressInit(var Msg: TMessage); message UM\_PROGRESS\_INIT;

procedure UMProgressChange(var Msg:TMessage); message UM\_PROGRESS\_CHANGE;

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

begin

// Уничтожаем запущенный поток

if Assigned(FMyThread) then

FreeAndNil(FMyThread);

// Создаём и запускаем новый поток

FMyThread := TMyThread.Create(StrToInt(edMaxValue.Text), Handle);

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

FMyThread.Free;

end;

procedure TForm1.UMProgressChange(var Msg: TMessage);

var

ProgressData: TProgressData;

begin

ProgressData := TProgressData(Msg.WParam);

ProgressBar1.Position := ProgressData.CurrValue;

labResult.Caption := IntToStr(ProgressData.CalcResult);

labThreadStateInfo.Caption := ProgressData.ThreadStateInfo;

end;

procedure TForm1.UMProgressInit(var Msg: TMessage);

var

MaxValue: Integer;

begin

MaxValue := Msg.WParam;

ProgressBar1.Max := MaxValue;

ProgressBar1.Position := 0;

labResult.Caption := '0';

labThreadStateInfo.Caption := 'Start';

end;

{ TMyThread }

constructor TMyThread.Create(AMaxValue: Integer; AFormHandle: THandle);

begin

inherited Create(False);

FMaxValue := AMaxValue;

FFormHandle := AFormHandle;

ProgressData := TProgressData.Create;

end;

destructor TMyThread.Destroy;

begin

//ProgressData.Free; - НЕЛЬЗЯ ТУТ!

inherited;

ProgressData.Free;

end;

procedure TMyThread.Execute;

var

CurrVal: Integer;

begin

// Выставляем параметры компонента ProgressBar1

SendMessage(FFormHandle, UM\_PROGRESS\_INIT, FMaxValue, 0);

ThreadWaitTimeout(Self, 1000); // Просто пауза 1 сек.

CurrVal := 0;

// Выполняем некоторые вычисления

while CurrVal < FMaxValue do

begin

if Terminated then Break;

Inc(CurrVal);

ProgressData.CurrValue := CurrVal;

ProgressData.CalcResult := ProgressData.CalcResult + CurrVal;

ProgressData.ThreadStateInfo := Format('Progress: %f%%',

[CurrVal / FMaxValue \* 100]);

// Обновление прогресса выполняется только 1 раз из 10000

if CurrVal mod 10000 = 0 then

SendMessage(FFormHandle, UM\_PROGRESS\_CHANGE, WPARAM(ProgressData), 0);

end;

// Обновляем прогресс в конце вычислений

SendMessage(FFormHandle, UM\_PROGRESS\_CHANGE, WPARAM(ProgressData), 0);

end;

end.

В данном примере используется два способа передачи данных в основной поток с использованием функции SendMessage. В первом способе (код UM\_PROGRESS\_INIT) целочисленное значение FMaxValue передаётся в качестве параметра wParam. Во втором способе (код UM\_PROGRESS\_CHANGE) передаётся ссылка на объект класса TProgressData в качестве параметра wParam. Для передачи ссылки на объект в качестве параметра wParam необходимо выполнять приведение типа к WPARAM следующим образом: WPARAM(ProgressData). С lParam ситуация аналогична (тип LPARAM).

Объект ProgressData создаётся в конструкторе TMyThread.Create, а уничтожается в деструкторе TMyThread.Destroy. Для использования объекта в методе TForm1.UMProgressChange требуется использовать приведение типа к TProgressData: ProgressData := TProgressData(Msg.WParam).

|  |
| --- |
| Обратите внимание, что уничтожать объект ProgressData в деструкторе TMyThread.Destroy следует после того, как был вызван inherited. Если уничтожать объект до вызова inherited, то при уничтожении объекта-потока (FMyThread.Free) будет возникать ошибка Access violation в том случае, если Вы попробуете закрыть программу, не дожидаясь окончания вычислений. Это объясняется следующим образом:  1) при вызове FMyThread.Free происходит вызов деструктора TMyThread.Destroy, в котором вызывается inherited;  2) при вызове inherited срабатывает родительский деструктор TThread. Destroy, в котором выставляется флаг Terminated;  3) внутри метода TMyThread.Execute всё ещё продолжается выполняться код и проверяется флаг Terminated. Объект ProgressData к этому моменту должен существовать, иначе при обращении к нему возникнет Access Violation. |

Обратите внимание, что при уничтожении объекта-потока (FMyThread.Free) главный поток не зависает, несмотря на то, что последней командой в методе TMyThread.Execute является SendMessage. Это происходит благодаря тому, что в деструкторе потока есть код, который обеспечивает предотвращение взаимной блокировки.

|  |
| --- |
| Внимание! Функцией SendMessage Вы можете пользоваться только при разработке программы под Windows. Данная функция может быть недоступна при программировании под другие операционные системы! Универсальным (кроссплатформенным) способом передачи данных в основной поток является Synchronize. |

## 5.5 Использование функции PostMessage для передачи очереди сообщений в основной поток

Функция PostMessage очень похожа на функцию SendMessage и также входит в состав Windows API. Однако существует принципиальная разница: функция PostMessage не является блокирующей, т.е. она вернёт управление сразу же после вызова. Из-за такой особенности Вы не всегда сможете использовать функцию PostMessage в тех же сценариях, что и SendMessage или Synchronize. При использовании функции PostMessage следует соблюдать осторожность, поскольку Вы можете столкнуться со следующими проблемами, которые не могут возникнуть при использовании SendMessage:

1) переполнение оконной очереди сообщений из-за слишком частого вызова функции PostMessage (т.е. главный поток попросту не сможет обрабатывать сообщения из оконной очереди с той же скоростью, с которой сообщения добавляются в оконную очередь).

2) задержка визуализации актуальной информации, поскольку обработка сообщений из оконной очереди выполняется по принципу FIFO. Допустим, дополнительный поток решил передать в основной поток 100 сообщений с помощью PostMessage, а главный поток при обработке каждого оконного сообщения выполняет запрос к базе данных, длительность которого составляет 1 секунду. Это приведёт к тому, что актуальная информация будет показана пользователю лишь спустя 100 секунд.

3) обращение к недействительному указателю и, как следствие, ошибка «Invalid pointer operation» либо «Access violation». Такую ошибку легко получить, если дополнительный поток выполнит создание объекта, затем он вызовет функцию PostMessage (передаст в неё ссылку на объект), а затем сразу же уничтожит созданный ранее объект. Ошибка возникнет в основном потоке, как только в нём произойдет вызов обработчика оконного сообщения и он попытается обратиться к объекту по недействительному указателю.

Пример использования функции PostMessage для передачи данных в очередь сообщений основного потока находится в папке «Ex10». К сожалению, объём кода в примере получился большим, поэтому считаю, что нет необходимости приводить здесь весь пример целиком. Ниже приведён код метода TMyThread.Execute, а также код обработчика сообщения UMProgressChange:

procedure TMyThread.Execute;

var

CurrVal: Integer;

CalcResult: Int64;

ThreadStateInfo: string;

ProgressData: TProgressData;

begin

CurrVal := 0;

CalcResult := 0;

// Выполняем некоторые вычисления

while CurrVal < FMaxValue do

begin

if Terminated then Break;

Inc(CurrVal);

CalcResult := CalcResult + CurrVal;

ThreadStateInfo := Format('Progress: %f%%', [CurrVal / FMaxValue \* 100]);

// Обновление прогресса выполняется только 1 раз из 10000

if CurrVal mod 10000 = 0 then

begin

// Создаём объект ProgressData непосредственно перед PostMessage

ProgressData:=TProgressData.Create(CurrVal,CalcResult,ThreadStateInfo);

PostMessage(FFormHandle, UM\_PROGRESS\_CHANGE, WPARAM(ProgressData), 0);

Inc(PostMessageCount);

end;

end;

// Обновляем прогресс в конце вычислений

ProgressData := TProgressData.Create(CurrVal, CalcResult, ThreadStateInfo);

PostMessage(FFormHandle, UM\_PROGRESS\_CHANGE, WPARAM(ProgressData), 0);

Inc(PostMessageCount);

// Этот флаг необходим, чтобы главный поток мог убедиться, что

// доп. поток отработал корректно до последнего

EndWork := True;

end;

procedure TForm1.UMProgressChange(var Msg: TMessage);

var

ProgressData: TProgressData;

begin

ProgressData := TProgressData(Msg.WParam);

ProgressBar1.Position := ProgressData.CurrValue;

labResult.Caption := IntToStr(ProgressData.CalcResult);

labThreadStateInfo.Caption := ProgressData.ThreadStateInfo;

Inc(FPostMsgProcessCount);

// Здесь необходимо уничтожить объект TProgressData:

ProgressData.Free;

end;

Обратите внимание на то, что объект ProgressData создаётся в методе TMyThread.Execute, а уничтожается в методе UMProgressChange.

В окне программы отображается количество вызовов функции PostMessage и количество вызовов обработчика UMProgressChange. В данном примере вызов функции PostMessage выполняется 1 раз из 10000.

|  |
| --- |
| Если производить вызов 1 раз из 100 (или чаще), то будет происходить переполнение оконной очереди сообщений и обработчик UMProgressChange будет срабатывать меньшее количество раз по сравнению с количеством вызовов функции PostMessage. Для нашего примера это плохо тем, что будет происходить утечка памяти, поскольку перед каждым вызовом функции PostMessage выполняется создание объекта ProgressData. Если обработчик UMProgressChange будет срабатывать реже, чем PostMessage, то не все объекты ProgressData будут уничтожены. |

## 5.6 Использование списка TThreadList для передачи очереди данных в основной поток

При разработке многопоточных программ очень часто возникает необходимость организовать передачу очереди данных между потоками. Для организации такой очереди можно использовать массивы или списки. Очень важно, чтобы массив / список был защищён от одновременного доступа из нескольких потоков. В простейшем случае мы можем защитить массив / список с помощью критической секции. В Delphi имеются специализированные классы, позволяющие организовать очередь сообщений для обмена данными между потоками. Одним из таких классов является TThreadList – потокобезопасный список. По сути, это обычный TList, в котором используется критическая секция. Он не допускает одновременного доступа к списку из разных потоков.

В примере Ex11 будет решена такая задача: несколько дополнительных потоков будут генерировать текстовую информацию (события), а главный поток будет отображать дату, время, номер потока и текст события в компоненте TListBox.

Ниже представлен листинг кода примера Ex11:

unit Ex11Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, MTUtils, ComCtrls, ExtCtrls, StrUtils, Contnrs;

type

TLogMessage = class

ThreadId: Integer;

EventTime: TDateTime;

EventText: string;

constructor Create(AThreadId: Integer; AEventTime: TDateTime;

AEventText: string);

end;

TMyThread = class(TThread)

private

procedure LogEvent(s: string);

protected

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

Timer1: TTimer;

Button1: TButton;

ListBox1: TListBox;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure Timer1Timer(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure Button1Click(Sender: TObject);

private

{ Private declarations }

FList: TObjectList;

public

{ Public declarations }

end;

var

Form1: TForm1;

ThreadCounter: Integer;

EventList: TThreadList;

implementation

{$R \*.dfm}

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

begin

// Создаём и запускаем новый поток

FList.Add(TMyThread.Create(False));

end;

procedure TForm1.Button1Click(Sender: TObject);

begin

FList.Clear;

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

FList := TObjectList.Create;

EventList := TThreadList.Create;;

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

FList.Free;

// В списке могут остаться элементы. Это не страшно, поскольку

// выполняется выход из программы

EventList.Free;

end;

procedure TForm1.Timer1Timer(Sender: TObject);

var

L, TmpList: TList;

I: Integer;

m: TLogMessage;

Cnt: Integer;

begin

// Определяем, есть ли элементы в списке EventList

L := EventList.LockList;

Cnt := L.Count;

EventList.UnlockList;

if Cnt = 0 then Exit;

TmpList := TList.Create;

try

L := EventList.LockList;

try

// Переносим все элементы во временный список

TmpList.Assign(L);

// Очищаем список EventList

L.Clear;

finally

// Как можно быстрее снимаем блокировку списка

EventList.UnlockList;

end;

// Дальше обрабатываем элементы из временного списка

for I := 0 to TmpList.Count - 1 do

begin

m := TLogMessage(TmpList[I]);

if ListBox1.Count < 50000 then

ListBox1.Items.Add(Format('%s [T:%d] - %s',

[FormatDateTime('hh:nn:ss.zzz', m.EventTime), m.ThreadId, m.EventText]));

// Здесь необходимо уничтожить объект TLogMessage

m.Free;

end;

ListBox1.ItemIndex := ListBox1.Count - 1;

finally

TmpList.Free;

end;

end;

{ TMyThread }

procedure TMyThread.Execute;

var

I: Integer;

begin

I := 0;

LogEvent('Thread start');

while not Terminated do

begin

Inc(I);

LogEvent('Event #' + IntToStr(I));

ThreadWaitTimeout(Self, 1000);

end;

LogEvent('Thread stop');

end;

procedure TMyThread.LogEvent(s: string);

var

L: TList;

m: TLogMessage;

begin

m := TLogMessage.Create(GetCurrentThreadId, Now, s);

L := EventList.LockList;

L.Add(m);

EventList.UnlockList;

end;

{ TLogMessage }

constructor TLogMessage.Create(AThreadId: Integer; AEventTime: TDateTime;

AEventText: string);

begin

ThreadId := AThreadId;

EventTime := AEventTime;

EventText := AEventText;

end;

end.

В данном примере используется таймер TTimer, который срабатывает каждые 100 мс. В обработчике таймера TForm1.Timer1Timer выполняются следующие действия:

1) проверяется количество элементов в списке EventList;

2) если в списке EventList есть элементы, то они копируются во временный список TmpList, а из списка EventList все элементы удаляются;

3) элементы из списка TmpList выводятся в список ListBox1.

Обратите внимание, как следует пользоваться объектом класса TThreadList. Для начала необходимо выполнить блокировку списка с помощью метода LockList и запомнить ссылку на объект класса TList: L := EventList.LockList. Далее с полученной ссылкой вы можете производить любые действия (добавление, удаление, очистка элементов списка TList). После того, как Вы закончили работать со списком, необходимо его разблокировать. Для этого следует вызвать метод UnlockList.

|  |
| --- |
| Внимание! Не следует устанавливать блокировку на длительное время! Старайтесь проектировать свой код таким образом, чтобы длительность блокировки была как можно меньше. |

Обратите внимание, что в обработчике TForm1.Timer1Timer вызов методов LockList / UnlockList выполняется дважды. Сначала мы блокируем список для доступа к количеству элементов (свойство TList.Count). При этом мы не используем конструкцию try..finally. Далее мы блокируем список для копирования элементов списка во временный список TmpList и в этом случае мы используем конструкцию try..finally.

|  |
| --- |
| Внимание! Решение о том, нужно ли использовать конструкцию try..finally в каждом конкретном случае, зависит от того, какова вероятность возникновения ошибки (exception) между LockList и UnlockList. Если код очень простой и ошибка возникнуть не может (например, при обращении к свойству Count), то использовать конструкцию try..finally нет необходимости. В противном случае рекомендуется подстраховаться и использовать конструкцию try..finally, чтобы гарантировать, что метод UnlockList будет вызван при любой ситуации. Также рекомендуется использовать try..finally для улучшения визуального восприятия структуры кода в случае значительного количества строк кода между LockList и UnlockList. В некоторых задачах (например, при математических вычислениях) следует по возможности избегать использования конструкций try..finally и try..except, т.к. они могут нести значительные накладные расходы. |

В данном примере используется функция GetCurrentThreadId из состава Windows API. Данная функция возвращает идентификатор того потока, который вызвал эту функцию. Идентификатор каждого потока в операционной системе является уникальным, вне зависимости от того, какой процесс его создал.

## 5.7 Использование метода TThread.Synchronize для передачи данных в основной поток – вариант с использованием анонимной процедуры

В современных версиях Delphi можно передавать анонимную процедуру при вызове метода TThread.Synchronize.

Анонимная процедура / функция, как следует из названия, не имеет имени и может использоваться в качестве параметра при вызове другой процедуры/функции. Механизм работы анонимных функций довольно сложный и у меня нет возможности его здесь описывать (в этом нет ничего страшного, поскольку в интернете есть много информации на данную тему). Хочу отметить, что анонимные функции давно являются нормой в современных языках программирования. Термин «анонимная функция» используется в Delphi (на мой взгляд, это не очень удачное название, поскольку очень часто объявление такой функции присваивается переменной, а переменная имеет вполне конкретное имя). В других языках программирования данный механизм может называться иначе, например: делегат, замыкание, лямбда. В независимости от того, как этот механизм называется в том или ином языке, существует одна очень важная особенность: анонимная функция «видит» все переменные, которые являются внешними по отношению к ней. В случае передачи анонимной процедуры в метод Synchronize, её вызов будет произведён в контексте главного потока, однако внутри анонимной процедуры мы вполне можем использовать локальные переменные, которые были объявлены в методе Execute.

Ниже представлен простейший пример вызова метода Synchronize с анонимной процедурой из метода Execute:

CurTime := Now; // Запоминаем время ДО вызова Synchronize

Synchronize(

procedure

begin

Form1.labLabLastThreadTime.Caption :=

'Последний поток был запущен: ' + DateTimeToStr(CurTime);

end);

## 5.8 Использование метода TThread.Queue для передачи данных в основной поток

В современных версиях Delphi имеется ещё один метод, позволяющий произвести вызов заданной процедуры в контексте основного потока: TThread.Queue. Данный метод имеет параметры, аналогичные методу TThread.Synchronize. Основное отличие от метода Synchronize заключается в том, что метод Synchronize является блокирующим, а метод Queue блокирующим не является. Также как и функция PostMessage, метод Queue возвращает управление немедленно. Метод Queue, как и метод Synchronize, может принимать в качестве параметра:

а) метод (процедуру);

б) анонимную процедуру;

в) обычную процедуру (в этом случае компилятор создаёт анонимную процедуру, из которой производится вызов обычной процедуры).

Суть работы метода состоит TThread.Queue в следующем:

1. Метод Queue добавляет в конец списка SyncList (глобальная переменная типа TList) информацию, необходимую для вызова переданной ей процедуры.

2. Основной поток периодически проверяет наличие элементов в списке SyncList. Если в списке обнаружен элемент, то производится запуск процедуры, хранящейся в списке.

3. Метод TThread.Queue, вызванный в дополнительном потоке, вернёт управление немедленно, т.е. он не будет дожидаться, когда основной поток выполнит запуск заданной процедуры.

С появлением метода TThread.Queue многие Delphi-программисты стали гораздо реже использовать старые способы взаимодействия с основным потоком (Synchronize, SendMessage, PostMessage). Вероятная причина этого заключается в том, что дополнительный поток не блокируется при вызове Queue, и никак не зависит от того, насколько сильно в данный момент загружен главный поток программы. Метод TThread.Queue можно использовать в большинстве тех случаев, когда раньше приходилось использовать метод Synchronize. Если Вы используете метод Queue, то должны учитывать, что при вызове заданной процедуры в контексте основного потока дополнительный поток продолжает выполняться и существует риск одновременного обращения к объекту / структуре / массиву / строке из главного и из дополнительного потока. Мы помним, что в некоторых случаях мы должны защищать сложные объекты от одновременного доступа, например, с помощью критической секции, либо с помощью механизма TMonitor.

Пример Ex12 демонстрирует применение метода TThread.Queue для журналлирования событий, происходящих в дополнительном потоке.

unit Ex12Unit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, MTUtils, ComCtrls, ExtCtrls, StrUtils, Contnrs,

Generics.Collections;

type

TMyThread = class(TThread)

private

procedure LogEvent(EventText: string);

protected

procedure Execute; override;

end;

TForm1 = class(TForm)

btnRunInParallelThread: TButton;

Button1: TButton;

ListBox1: TListBox;

labLabLastThreadTime: TLabel;

btnClearListBox: TButton;

procedure btnRunInParallelThreadClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure Button1Click(Sender: TObject);

procedure btnClearListBoxClick(Sender: TObject);

private

{ Private declarations }

FList: TObjectList<TMyThread>;

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

procedure TForm1.btnClearListBoxClick(Sender: TObject);

begin

ListBox1.Clear;

end;

procedure TForm1.btnRunInParallelThreadClick(Sender: TObject);

begin

// Создаём и запускаем новый поток

FList.Add(TMyThread.Create(False));

end;

procedure TForm1.Button1Click(Sender: TObject);

var

t: TMyThread;

begin

for t in FList do

t.Terminate; // Сообщаем потокам о необходимости завершаться

FList.Clear; // Уничтожаем потоки

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

FList := TObjectList<TMyThread>.Create;

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

FList.Free;

end;

{ TMyThread }

procedure TMyThread.Execute;

var

I: Integer;

CurTime: TDateTime;

begin

CurTime := Now; // Запоминаем время ДО вызова Queue

Queue(

procedure

begin

Form1.labLabLastThreadTime.Caption :=

'Последний поток был запущен: ' + DateTimeToStr(CurTime);

end);

LogEvent('Thread start');

I := 0;

while not Terminated do

begin

Inc(I);

LogEvent('Event #' + IntToStr(I));

ThreadWaitTimeout(Self, 500);

end;

LogEvent('Thread stop');

end;

procedure TMyThread.LogEvent(EventText: string);

var

ThreadId: Cardinal;

EventTime: TDateTime;

begin

// Запоминаем ID потока и текущее время ДО вызова Queue

ThreadId := GetCurrentThreadId;

EventTime := Now;

TThread.Queue(nil,

procedure

begin

Form1.ListBox1.Items.Add(Format('%s [T:%d] - %s',

[FormatDateTime('hh:nn:ss.zzz', EventTime), ThreadId, EventText]));

Form1.ListBox1.ItemIndex := Form1.ListBox1.Count - 1;

end);

end;

end.

Обратите внимание на следующие особенности данного примера:

1. В методе TMyThread.LogEvent при вызове метода Queue используется следующая форма: TThread.Queue(nil, АнонимнаяПроцедура). Благодаря тому, что используется значение **nil**, анонимная процедура **не ассоциируется** с объектом дополнительного потока. Благодаря этому **гарантируется**, что основной поток обработает все вызовы Queue с параметром **nil**. Даже если дополнительный поток 1000 раз подряд вызовет метод Queue, а главный поток вызовет метод Free для объекта-потока, не успев обработать все вызовы Queue, объект потока будет уничтожен, но главный поток всё равно обработает все вызовы.

|  |
| --- |
| Следует быть предельно внимательным при использовании такой формы вызова метода Queue. Дело в том, что если в анонимной процедуре будут обращения к полям / свойствам / методам объекта-потока, то в ситуации досрочного уничтожения объекта потока произойдёт ошибка доступа к памяти (например, «Invalid pointer operation», «Access violation» либо «Out of memory») при вызове процедуры в контексте основного потока. |

2. В анонимной процедуре, которая является параметром метода Queue в методе TMyThread.LogEvent, отсутствуют обращения к полям / свойствам / методам объекта-потока. Реализация данной анонимной процедуры никак не привязана к состоянию дополнительного потока. В ней не произойдет ошибки, даже если объект-поток будет уничтожен раньше времени.

3. В методе TMyThread.LogEvent все данные, которые используются в анонимной процедуре, формируются **заранее**. В анонимной процедуре используются переменные EventTime, ThreadId, EventText. Переменные EventTime и ThreadId объявляются в методе TMyThread.LogEvent и им присваиваются значения **до** вызова Queue. Переменная EventText является параметром метода LogEvent, т.е. её значение также **сформировано** до вызова Queue.

|  |
| --- |
| К сожалению, в метод Queue невозможно передать дополнительные параметры, которые в дальнейшем могли бы использоваться при вызове анонимной процедуры. Но благодаря тому, что переменные EventTime, ThreadId, EventText объявлены в методе LogEvent и их значения сформированы до вызова Queue, то мы можем их рассматривать как параметры вызова анонимной процедуры. При каждом вызове метода Queue (внутри метода LogEvent) переменные EventTime, ThreadId, EventText будут «захватываться», а затем использоваться при вызове анонимной процедуры в основном потоке. |

4. Метод Queue вызывается **из отдельного** метода TMyThread.LogEvent. Это очень важно! Если мы хотим эмулировать передачу параметров в анонимную функцию, то вызов метода Queue лучше выполнять из отдельного метода. Например:

|  |
| --- |
| procedure TMyThread.CallMyFuncInMainThread(param1: Integer; param2: string);  begin  TThread.Queue(nil,  procedure  var  s: string;  begin  s := Format('param1=%d; param2=%s', [param1, param2]);  Form1.ListBox1.Items.Add(s);  end);  end;  Вызов из Execute:  I := 111;  sVal := 'Text 1';  CallMyFuncInMainThread(I, sVal);  I := 222;  sVal := 'Text 2';  CallMyFuncInMainThread(I, sVal); |

Благодаря этому, при передаче анонимной процедуры в метод Queue происходит захват параметров и переменных отдельного метода. Если бы мы в методе Execute выполнили 2 раза подряд вызов Queue вместо CallMyFuncInMainThread, например:

|  |
| --- |
| I := 111;  sVal := 'Text 1';  TThread.Queue(nil,  procedure  var  s: string;  begin  s := Format('param1=%d; param2=%s', [I, sVal]);  Form1.ListBox1.Items.Add(s);  end);  I := 222;  sVal := 'Text 2';  TThread.Queue(nil,  procedure  var  s: string;  begin  s := Format('param1=%d; param2=%s', [I, sVal]);  Form1.ListBox1.Items.Add(s);  end); |

то для обоих вызовов анонимной процедуры использовались бы одни и те же значения захваченных переменных I и sVal. В этом случае в ListBox1 будет добавлено 2 одинаковых строки:

param1=222; param2=Text 2

param1=222; param2=Text 2

Конечно, если после первого вызова Queue поставить паузу (например, Sleep(100)), то главный поток скорее всего успеет выполнить вызов первой анонимной процедуры с правильными значениями 111 и «Text 1».

5. В методе Execute присутствует вызов метода Queue, в который передаётся анонимная процедура, которая записывает в компонент labLabLastThreadTime время запуска дополнительного потока. Обратите внимание, что здесь используется форма вызова без указания **nil**. Это означает, что анонимная процедура будет ассоциирована с объектом потока, что в свою очередь означает, что при уничтожении потока необработанный главным потоком вызов Queue (вернее, информация, необходимая для вызова процедуры), будет удалена из очереди отложенных вызовов. В данном случае нет никаких рисков по следующим причинам:

а) метод Queue вызывается в начале метода Execute, поток не завершится сразу же после данного вызова Queue (пользователь просто не успеет так быстро нажать кнопку «Остановить потоки»), значит главный поток успеет обработать данный вызов Queue.

б) даже если главный поток не успеет выполнить отложенный вызов процедуры, а дополнительный поток по каким-то причинам будет уничтожен, необходимая для вызова отложенной процедуры информация попросту будет удалена из очереди. В данном случае это не приведёт к негативным последствиям, поскольку процедура не выполняет никакой ответственной работы.

|  |
| --- |
| Внимание! Будьте внимательны при использовании формы вызова метода Queue без указания **nil**! В этом случае вызов отложенной процедуры может не произойти (если дополнительный поток был уничтожен до того, как главный поток произвёл вызов процедуры). В некоторый случаях это может приводить к некорректной работе программы (если программа написана таким образом, что вызов отложенной процедуры является обязательным), либо к утечке памяти (если отложенная процедура должна освободить память, выделенную перед вызовом метода Queue). |

6. В методе Execute организована пауза 500 мс между вызовами метода LogEvent в цикле.

|  |
| --- |
| Внимание! Не следует вызывать метод Queue слишком часто. Если дополнительный поток будет вызывать метод Queue слишком часто, то главный поток программы не будет успевать обрабатывать элементы очереди SyncList, в результате чего размер списка отложенных вызовов SyncList будет увеличиваться, а интерфейс пользователя будет выглядеть зависшим. |

7. В данном примере для хранения ссылок на объекты-потоки используется список, объявленный следующим образом: FList: TObjectList<TMyThread>. Это позволяет избавиться от необходимости использовать операцию приведения типа (как при использовании обычного TObjectList).

Для исследования работы метода Queue вы можете использовать пример Ex12Full. Он наглядно демонстрирует проблему обработки отложенных вызовов при уничтожении дополнительных потоков. Его особенности следующие:

1. Понижается приоритет дополнительного потока с помощью кода Priority := tpLowest. Это означает, что потоку будет отводиться гораздо меньше процессорного времени, чем обычно. Это необходимо для того, чтобы код, который выполняется в конце метода Execute (цикл от 1 до 100) занимал процессор на минимальное количество времени. По этим же причинам в данном цикле дополнительно вызывается Sleep(0). Благодаря этому главный поток начнёт обработку очереди отложенных вызовов раньше, чем завершится цикл, т.е. что-нибудь из этого цикла скорее всего попадёт в ListBox.

2. Переменной ThreadWaitTimeoutSleepTime, которая используется в процедуре ThreadWaitTimeout, выставляется значение «1». Это означает, что процедура ThreadWaitTimeout завершит свою работу сразу же, как только объекту потока будет произведён вызов метода Terminate. Это важно в том случае, если Вы запустили несколько потоков. Благодаря этому у всех потоков цикл от 1 до 100 будет выполняться практически одновременно.

3. В анонимной процедуре, которая реализована в методе TMyThread.LogEvent, вызывается Sleep(5). Это необходимо для того, чтобы главный поток не мог сразу (за свой квант времени) обработать все сообщения из очереди.

5. Реализовано 2 способа вызова метода TThread.Queue в методе TMyThread.LogEvent: 1) ассоциировать анонимную процедуру с объектом потока; 2) не ассоциировать. Если выбран способ 1, то при уничтожении потоков в список ListBox попадут не все строки от 1 до 100. Это означает, что при уничтожении объекта-потока произошло удаление всех отложенных вызовов (т.е. записей из списка SyncList), которые ассоциированы с объектом потока, но ещё не обработаны главным потоком. Если выбран способ 2, то в ListBox попадут все строки от 1 до 100.

# 6. Где можно и где нельзя создавать и уничтожать потоки

Данные замечания актуальны при разработке программ под Windows. Существует ряд мест, где не следует создавать либо уничтожать потоки.

1. Создавать потоки не следует в секции initialization (как в EXE, так и в DLL). По моему опыту создание дополнительных потоков в секции initialization EXE-модуля (не важно, в каком pas-файле) приводит к снижению стабильности приложения. Грубо говора, один раз из 100 возникает сбой при запуске приложения.

2. Не следует уничтожать потоки в секции finalization DLL-библиотеки. Если Вы попытаетесь это сделать, то, скорее всего программа зависнет (это происходит из-за особенностей выгрузки DLL-библиотек в ОС Windows).

Если Вы разрабатываете DLL-библиотеку, в которой используются дополнительные потоки, то рекомендую реализовать в ней две экспортные функции – одну для создания необходимых объектов в библиотеке (например, CreateLibrary), вторая для уничтожения созданных объектов (например, DestroyLibrary). В этом случае код создания потоков можно разместить в CreateLibrary (либо создавать их позже), а код уничтожения объектов потоков разместить в DestroyLibrary.

# 7. Коротко о приоритетах потоков в Windows

При разработке многопоточной программы для Windows существует возможность назначать потоку относительный приоритет (свойство TThread.Priority). По умолчанию используется приоритет tpNormal. Тип TThreadPriority объявлен следующим образом:

|  |
| --- |
| {$IFDEF MSWINDOWS}  TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest,  tpTimeCritical) platform;  {$ENDIF MSWINDOWS} |

|  |
| --- |
| Обычно программисту нет смысла изменять приоритеты потоков. Приоритет потока никак не влияет на скорость исполнения программного кода. Приоритет потока никак не влияет на размер кванта времени. Относительный приоритет потока работает только в рамках процесса и никак не влияет на выделение квантов времени потокам, которые работают в других процессах.  Изменять приоритет потока не имеет никакого смысла, если выполняется задача, в которой основное время уходит на ожидание какого-либо события. |

Если в Вашей программе работают 2 потока, у одного приоритет tpNormal, а у второго tpLower и оба потока привязаны к одному и тому же ядру процессора, то первому потоку гораздо чаще будет предоставляться процессорное время (например, 98%). С другой стороны, если у этих же потоков не будет привязки к одному и тому же ядру, то они получат одинаковое процессорное время. Скорее всего, оба варианта – это не то, чего Вы хотите добиться при изменении относительного приоритета потока.

Существует 2 крайних уровня приоритетов: tpIdle и tpTimeCritical. Поток с приоритетом tpIdle получит процессорное время только в том случае, если нет других активных потоков с более высоким приоритетом. Поток с приоритетом tpTimeCritical будет забирать себе всё процессорное время, т.е. потоки с более низким приоритетом не получат квант времени, если выполняется поток с приоритетом tpTimeCritical. Это верно для одноядерного процессора. Если процессор многоядерный (сейчас это обычная ситуация), то, скорее всего, будут выполняться одновременно и поток с приоритетом tpIdle и поток с приоритетом tpTimeCritical.

|  |
| --- |
| Заготовка для таблицы Внимание! |

|  |
| --- |
| Совет / рекомендация |