

| | |
|---|-----|
| Глава 17. Потоки..... | 402 |
| 17.1 Теория потоков..... | 403 |
| 17.2 Простейший поток | 404 |
| 17.3 Дополнительные возможности потоков. | 408 |
| 17.4 Подробней о синхронизации..... | 409 |



Глава 17. Потоки

Операционная система Windows является многопоточной. Это значит, что она может выполнять несколько задач одновременно. Почему я говорю именно задач, а не программ? Да потому что одна программа может состоять из нескольких независимых блоков кода, которые тоже могут выполняться одновременно. Каждый такой блок называется потоком.

Когда ты запускаешь новое приложение, то для него автоматически создаётся главный поток, в котором и будет выполняться код программы. Но это не значит, что ты ограничен этим потоком. В любой момент ты можешь создать дополнительные потоки, которые будут выполняться параллельно с главным.

Таким образом можно добиться многозадачности внутри самой программы.



17.1 Теория потоков.

Я говорю о потоках и ещё ни слова не сказал о том, зачем же нужно разделять программу на несколько потоков. Я часто в этой книге привожу примеры на основе таких программ, как Word и Excel и сейчас снова пример основанный на работе этих программ. Когда ты запускаешь Word и набираешь текст, то встроенный модуль проверки орфографии автоматически следит за тем, что ты пишешь и подправляет орфографические ошибки. Теперь представь логику проверки. После нажатия кнопки, нужно отобразить на экране нужную букву, затем проверить ближайшие слова на изменения и проверить правильность их написания. После проверки слов, проверять всё предложение на наличие пропущенных запятых или других знаков.

На словах алгоритм описывается достаточно просто. Но попробуй представить себе тот большой труд, который надо проделать после каждого нажатия кнопки. Если бы алгоритм проверки орфографии действительно действовал бы так, то буквы появлялись бы на экране не чаще 1 в пару секунд.

К счастью, проверка орфографии работает отдельным процессом. Ты спокойно набираешь текст, а проверка идёт в отдельном потоке не мешая тебе работать с текстом. При этом практически незаметны задержки, и нет никаких неудобств.

Когда ты пишешь новую программу, то не надо пытаться засунуть все функции в отдельные потоки. Каждый поток накладывает на программу дополнительную сложность и неустойчивость, да и отлаживать потоки намного сложнее.

Какой код нужно помещать в отдельный поток? Вот некоторые пример:

1. Если какие-то функции должны выполняться параллельно основному процессу, то тут деваться некуда и нужно обязательно помещать такие вещи в поток.

2. Если какие-то расчёты идут достаточно долго, то многие считают, что их тоже нужно помещать в поток. Просто когда идут такие расчёты программа блокируется и невозможно нажать кнопку «Отмена» или что-нибудь подобное. Это неправильное утверждение. Поток тут абсолютно необязателен, потому что можно обойтись и без него. Достаточно внутри расчётов поставить вызов *Application.ProcessMessages* и в этом месте выполнение расчётов будет прерываться на некоторое время и программа будет обслуживать другие сообщения, пришедшие от пользователя. Таким образом получится простой эффект многозадачности без использования потока.

3. Код критичен к времени выполнения. Допустим, что твоя программа должна принимать какие-то данные по COM порту. Как только на порт пришли какие-то данные, они должны быть моментально обработаны с минимальной задержкой. Вот такие вещи желательно выносить в отдельный поток, потому что если в момент поступления данных программа занята большими расчётами, то данные могут оказаться необработанными.

Истинную многозадачность можно получить только на многопроцессорных системах, где каждый процессор выполняет свою задачу. В домашних компьютерах в основном ставится только один процессор. Чтобы создать многозадачность на таких процессорах используют псевдомногозадачность. В этом виде один процессор выполняет сразу несколько задач благодаря быстрым переключениям между ними. Например, процессор может выполнять сразу десять задач, при этом каждой из них давать по 10 миллисекунд своего рабочего времени. В этом случае процессор будет через определённые промежутки времени переключаться между задачами и у пользователя будет создаваться впечатление, что они выполняются параллельно. Но это общий вид псевдомногозадачности, реально она реализована по другому.

В 32-х разрядных версиях Windows используется вытесняющая многозадачность (до этого была согласованная). В такой среде ОС разделяет процессорное время между разными приложениями и потоками на основе вытеснения. Разделение происходит в основном благодаря приоритету потока. У каждого потока есть приоритет, по которому определяется его важность. Чем выше приоритет, тем больше процессорного времени

выделяется этому потоку. Потоки с одинаковым приоритетом будут получать одинаковое количество процессорного времени.

У дополнительных потоков приоритет выставляется такой же как и у главного потока программы, но ты его можешь увеличить или уменьшить. Чем выше приоритет потока, тем больше на него отводится процессорного времени.

Снова допустим, что твоя программа должна принимать какие-то данные по СОМ порту и сразу же их обрабатывать. Для этого создаём новый поток в нём реализуем код получения и обработки данных. Теперь достаточно поднять приоритет потока, чтобы на него при необходимости выделялось больше процессорного времени и задача решена. Теперь, как только поступают на СОМ порт новые данные, поток сразу же обработает их, потому что с более высоким приоритетом он получит больше процессорного времени.

В этой книге потоки будут использоваться достаточно часто в главе о программировании звука. Там мы будем создавать отдельный поток, в котором будет выполняться воспроизведение или запись звука, при этом основная программа сможет работать без каких-либо ограничений.

17.2 Простейший поток

Давай попробуем написать простейший поток и в процессе познакомимся с его возможностями и как всё реализовано. На практике этот материал усваивается лучше, поэтому не будем больше тратить время на лишние разговоры и посмотрим на потоки в действии.

Создай новый проект. Поставь на форму компонент *TRichEdit* из палитры Win32 и один компонент *TLabel*. Нам ещё понадобится пару кнопок – одна для запуска потока, другая для его остановки. Посмотри на рисунок 17.2.1, где показана моя форма. У тебя должно получиться нечто похожее.

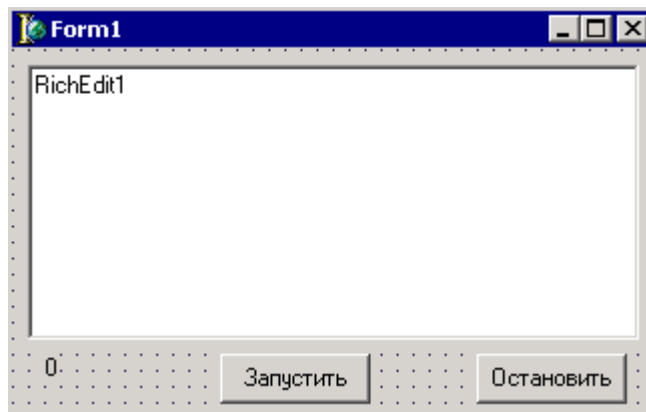


Рис 17.2.1. Главная форма нашей программы.

Теперь создадим модуль для потока. Для этого выбери пункт меню *File->New->Other* для открытия окна создания нового модуля (рисунок 17.2.2). найди в этом окне на закладке *New* пункт *Thread Object*. Выдели его и нажми кнопку "ОК". Появляется окошко, как на рисунке 17.2.3. В этом окне нужно указать имя создаваемого потока. Я назвал свой поток *TCountObj*. Нажимай «ОК» и Delphi создаст модуль-заготовку для нашего будущего потока.

Сохрани весь проект. Главную форму под именем *Main*, а поток под именем *MyThread*.

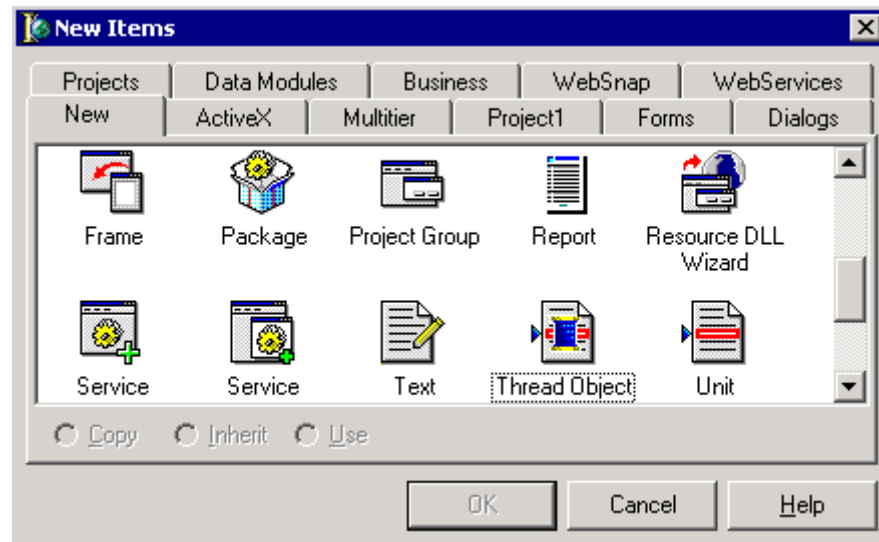


Рис 17.2.2. Создание модуля потока.

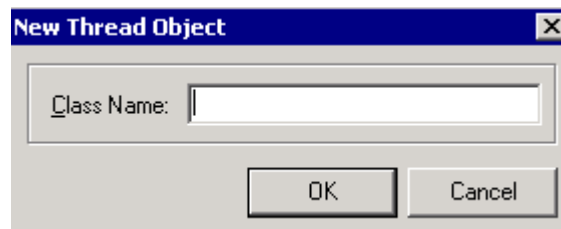


Рис 17.2.3. Задание имени потока.

Теперь посмотрим на код созданного для потока модуля:

```

unit MyThread;

interface

uses
  Classes;

type
  TCountObj = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ Important: Methods and properties of objects in VCL can only be used in a
method called using Synchronize, for example,
}

    Synchronize(UpdateCaption);

and UpdateCaption could look like,

procedure TCountObj.UpdateCaption;
begin

```

```

    Form1.Caption := 'Updated in a thread';
end; }

{ TCountObj }

procedure TCountObj.Execute;
begin
    { Place thread code here }
end;

end.

```

Это новый поток. У объекта есть только одна процедура *Execute*. В любых потоках эта процедура обязана быть переопределена, и в ней должен быть написан собственный код. Это связано с тем, что в объекте *TThread*, эта процедура объявлена как абстрактная (*abstract*) – пустая. Это значит, что процедуре дали имя, выделили место, но её код должен быть написан объектами потомками, т.е. нами.

Метод *Execute* – это и есть заготовка для кода потока. То, что мы напишем здесь будет выполняться параллельно основной задаче. Давай напишем здесь следующий код:

```

procedure TCountObj.Execute;
begin
    index:=1;
    //Запускаем бесконечный счётчик
    while index>0 do
    begin
        Synchronize(UpdateLabel);
        Inc(index);
        if index>100000 then
            index:=0;

        //Если поток остановлен, то выйти.
        if terminated then exit;
    end;
end;

```

Переменную *index* я объявил как *integer* в разделе *private* объекта потока. Там же я объявил процедуру *UpdateLabel*. Эта процедура выглядит так:

```

procedure TCountObj.UpdateLabel;
begin
    Form1.Label1.Caption:=IntToStr(Index);
end;

```

И последнее, что я сделал - подключил главную форму в раздел **uses**, потому что я обращаюсь к ней в коде выше (*Form1.Label1.Caption*) для обновления текста компонента *Label1*.

В методе *Execute* у меня запускается цикл *while*, который будет выполняться, пока переменная *index* больше нуля. Внутри цикла я вызываю метод *Synchronize* (о нём чуть позже) и увеличиваю переменную *index*. Если эта переменная становится больше 100000,

то в *index* присваивается 0 и расчёт начинается с начала. Таким образом цикл будет бесконечно выполнять увеличение переменной *index* от 0 до 100000 и опять сначала.

Самой последней идёт проверка, если свойство *terminated* равно *true*, то выйти из процедуры. Когда мы выйдём, то работа потока закончится, потому что закончится код процедуры *Execute*. Свойство *terminated* станет равной *true* тогда, когда будет вызван метод *Terminate* нашего потока.

Теперь о магической функции *Synchronize*. В качестве параметра ей передаётся процедура *UpdateLabel*, которая производит вывод в главную форму. Для чего нужно вставлять процедуру вывода на экран в *Synchronize*? Библиотека VCL имеет один недостаток - она не защищена от потоков. Все пользовательские компоненты разрабатывались так, что к ним может получить доступ только один поток. Если главная форма и поток попробуют одновременно вывести что-нибудь в одну и ту же область экрана или компонент, то программа рухнет как башни близнецы. Поэтому весь вывод на форму нужно выделять в отдельную процедуру и вызывать эту процедуру с помощью *Synchronize*.

Если процедура вызвана в методе *Synchronize*, то выполнение основной программы и потока морозиться и к компонентам окна получает доступ только объект, вызвавший метод *Synchronize*. Этот процесс незаметен для пользователя.

Так что если тебе нужно вывести какие-то данные из потока на экран главного окна, то делай это в отдельной процедуре и вызывай её с помощью метода *Synchronize*.

Всё, наш поток готов. Возвращаемся к главной форме. В раздел **uses** (самый первый, который идёт после **interface**) я добавил модуль потока *MyThread*. Почему именно в этот раздел, а не тот, что расположен ниже? Это связано с тем, что в разделе **private** мне нужно объявить переменную имеющую тип нашего объекта. Если добавить имя модуля во второй раздел **uses**, то он находится ниже той части кода, где нам нужно написать объявление. Именно поэтому добавлять модуль *MyThread* нужно в первый раздел **uses**.

В разделе **private** я объявил переменную *co* типа *TCountObj* (объект моего потока). По нажатию кнопки "Запустить" я написал такой код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  co:=TCountObj.Create(true);
  co.Resume;
  co.Priority:=tpLower;
end;
```

В первой строке я создаю поток *co*. В качестве параметра может быть *true* или *false*. Если *false*, то поток сразу начинает выполнение, иначе поток создаётся, но не запускается. Если поток создан не запущенным, то для запуска нужно использовать метод *Resume*, что я делаю во второй строке.


В третьей строке я устанавливаю приоритет потока поменьше, чтобы он не мешал работе основному потоку и выполнялся в фоне. Если установить приоритет повыше, то основной поток начнёт притормаживать, потому что у них будут одинаковые приоритеты.

По нажатию кнопки "Остановить" я написал:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  co.Terminate;
end;
```

Здесь я останавливаю выполнение потока с помощью вызова метода *Terminate* объекта потока. После вызова этого метода свойство *terminated* станет равной *true* и выполнение процедуры *Execute* закончиться.

Попробуй запустить эту программу, запустить поток (нажатием кнопки "Запустить") и набирать текст в *RichEdit*. Текст будет набираться без проблем, и в это время в компоненте *TLabel* будет работать счётчик. Если бы ты запустил счётчик без отдельного потока, то ты бы не смог набирать текст в *RichEdit*, потому что все ресурсы программы (основного потока) уходили бы на работу счётчика.

 На компакт диске, в директории \Примеры\Глава 17\Thread1 ты можешь увидеть пример этой программы.

17.3 Дополнительные возможности потоков.

Теперь я хочу дать краткий обзор ещё некоторых свойств и методов объекта потока, и сделать несколько замечаний по работе с потоком. В принципе, у объекта не так уж и много свойств и методов, и большую часть возможностей мы уже рассмотрели, но всё же я обязан сделать ещё несколько замечаний.

Suspend - приостанавливает поток. Для вызова просто напиши *so.Suspend*. Чтобы возобновить работу с этой же точки нужно вызвать *Resume*.

Priority- устанавливает приоритет потока. Например *Priority:=tpIdle*;

tpIdle - поток будет работать только когда процессор бездельничает.

tpLowest - самый слабый приоритет

tpLower - слабый приоритет

tpNormal - нормальный

tpHigher - высокий

tpHighest - самый высокий

tpTimeCritical - критичный (не советую использовать, потому что может грохнуть систему).

Suspended - если этот параметр *true*, то поток находится в паузе.

Terminated - если *true*, то поток должен быть остановлен, иначе поток должен продолжать работу.

Terminate – остановить выполнение потока.

FreeOnTerminate – если это свойство равно *true*, то по завершении выполнения процедуры *Execute* поток самоуничтожится. Советую использовать этот параметр, чтобы быть уверенным в том, что поток корректно удален из памяти.

Давай посмотрим, как будет выглядеть наш метод *Execute* из предыдущего примера с использованием свойства *FreeOnTerminate*:

```
procedure TCountObj.Execute;
begin
  FreeOnTerminate:=true;
  index:=1;
  //Запускаем бесконечный счётчик
  while index>0 do
  begin
    Synchronize(UpdateLabel);
    Inc(index);
    if index>100000 then
```



```
index:=0;  
  
//Если поток остановлен, то выйти.  
if terminated then exit;  
end;  
end;
```

В принципе, этой информации тебе будет достаточно для написания собственных потоков. У объекта *TThread* есть ещё несколько свойств и методов, но они не так важны и лично я ими никогда ещё не пользовался (не было такой задачи, где бы можно было их применить). Поэтому я не буду тратить место в книге на описания оставшихся возможностей, а лучше дам несколько советов.

Объекты потоков создаются как полноценные объекты. В основной программе мы создаём в памяти отдельный экземпляр потока и потом работаем с ним. Ты можешь создавать по несколько экземпляров одного потока и они будут работать одновременно абсолютно не мешая друг другу. Представим пример программы, копирующей файлы. Ты можешь создать поток, который будет копировать файлы из одного места в другое. В основной программе можно создать два экземпляра таких потоков и каждому из них задать копирования разных файлов в разные места. Оба потока будут копировать свои файлы абсолютно не мешая друг другу.

17.4 Подробней о синхронизации.

В предыдущем примере я использовал процедуру *UpdateLabel*, в которой на главную форму выводиться значение переменной *index*. Если бы мы программировали главное окно, то вполне логичным было бы создать переменную *index* локальной для процедуры *Execute*, а её значение передавать в *UpdateLabel* в качестве параметра. В потоках с этим проблема. Чтобы передать какие-то значения в процедуру, которая должна вызываться методом *Synchronize* нужно пользоваться переменными объекта. Даже не советую пробовать передавать параметры в процедуры которые вызываются методом *Synchronize*.

Но использование синхронизации – не единственный способ обновления параметров окна. Мы можем использовать для этого событийную модель Windows. Каждый раз, когда надо обновить содержимое текста мы можем посылать окну сообщение *SendMessage* с указанием значения, которое надо установить. Главное окно будет получать это сообщение и компонент сам изменит заголовок. В этом случае мы не обращаемся к главному окну из потока, а только отправляем сообщение, поэтому никаких проблем не будет.

Итак, функция *SendMessage* имеет следующие параметры:


1. Указатель на окно (компонент) которому нужно послать сообщение.
2. Тип сообщения.
3. Первый параметр.
4. Второй параметр.

Судя по функции, нам нужен компонент, у которого есть свойство *Handle*. В предыдущем примере у нас был *TLabel*, у которого нет такого свойства. У значит он нам не подходит. Замени этот компонент на *TEdit*. Теперь перейдём в поток. Тут в разделе **uses** нужно добавить два модуля: *windows* (здесь объявлена сама функция) и *messages* (здесь находятся все типы сообщений Windows).

Теперь удаляй из потока процедуру *UpdateLabel*, больше она не нужна, потому что мы не будем использовать метод *Synchronize*. Ну и наконец, подкорректируем наш метод *Execute*:

```
procedure TCountObj.Execute;
begin
  index:=1;
  while index>0 do
  begin
    SendMessage(Form1.Edit1.Handle, WM_SETTEXT, 0,
      Integer(PChar(IntToStr(index))));
    Inc(index);
    if index>100000 then
      index:=0;
    if terminated then exit;
  end;
end;
```

Как видишь, теперь у нас вместо метода *Synchronize* генерируется событие на обновления компонента *TEdit*. В качестве второго параметра я указываю тип сообщения *WM_SETTEXT* – обновить информацию. Третий параметр равен нулю. В последнем параметре нужно указать значение, которое нужно установить. Вот тут есть небольшая сложность. У нас значение представлено в виде целого числа, но нужно превратить его в *PChar*. Для этого я сначала конвертирую переменную *index* в строку (*IntToStr*), потом привожу его к типу *PChar* и тут же указываю размер *Integer*. Сложно? Зато не надо ничего синхронизировать.

 На компакт диске, в директории \Примеры\Глава 17\Thread2 ты можешь увидеть пример этой программы.