МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и механики
Кафедра технической кибернетики
и автоматического регулирования

Разработка многопоточных приложений

Методические указания к курсу
«Системное и прикладное программное обеспечение»
для студентов 3 курса д/о факультета ПММ

Составители: Рудалев В.Г.,
Крыжановская Ю.А.

Воронеж, 2001

В методических указаниях описываются основы создания многопоточных приложений для операционных систем Windows 95/98/NT/2000. Изложена методика использования стандартных средств интерфейса прикладного программирования API Win32 и специализированных классов среды программирования Delphi.

Методические указания рассчитаны на студентов 3 курса д.о. ф-та ПММ, имеющих начальное представление об архитектуре Win32 и навыки программирования в среде Delphi 4.0 – 5.0.

Рецензент: доцент кафедры МО ЭВМ Ю.Т.Свиридов

Содержание

1. Основные понятия

Поток (Thread, нить выполнения) – исполняемая сущность *процесса*. *Процесс* включает в себя загруженную в оперативную память исполняемую программу, виртуальное адресное пространство, системные и пользовательские ресурсы, выделенные программе, и один или несколько *потоков (нитей)* выполнения. В каждом процессе всегда неявно (незаметно для программиста и независимо от его воли) присутствует один *основной поток приложения,* остальные потоки при необходимости программист пишет сам. Потоки выполняются параллельно в виртуальном адресном пространстве породившего их процесса и разделяют ресурсы этого процесса. Единицей многозадачности в Windows является поток, а не процесс. Поток можно рассматривать как автономно работающую и управляемую часть приложения, а многопоточность приложения – как многозадачность внутри программы. Среди причин, побуждающих создавать именно многопоточные приложения, прежде всего следует выделить следующие.

Повышение надежности программы. Зацикливание основного потока приложения полностью блокирует его работу, при этом приложение может быть завершено лишь при помощи диспетчера задач (Task Manager), что, как правило, сопровождается потерей несохраненных данных. Поэтому «неблагонадежные» вычислительные фрагменты рекомендуется переносить из основного потока в отдельные дополнительные потоки, предусмотрев возможность их досрочного завершения (см. лаб.работы 1,2).

Повышение быстродействия, экономия ресурсов и расширение функциональных возможностей программы. Многопоточность позволяет параллельно выполнять отдельные участки программы на ЭВМ с несколькими процессорами, или выполнять их на одном процессоре «псевдопараллельно», используя механизм вытесняющей многозадачности Windows. Например, различные потоки в Microsoft Word одновременно принимают пользовательский ввод, проверяют орфографию в фоновом режиме и печатают документ. Microsoft Excel строит диаграммы и выполняет математические расчеты в фоновом режиме. Сервер баз данных для ответа на каждый запрос клиента запускает отдельный поток, в противном случае пришлось бы либо запускать отдельную копию сервера, напрасно расходуя ресурсы, либо чрезвычайно усложнять логику его работы. Интерфейс прикладных программ разнообразят анимация, воспроизведение звука и т.п., выполняемые отдельными потоками.

Напомним, что при вытесняющей многозадачности потоки выполняются попеременно, время процессора выделяется потокам *квантами* (около 19 мс). ОС вытесняет поток, когда истечет его квант или когда на очереди поток с большим приоритетом. Приоритеты постоянно пересчитываются, чтобы избежать монополизации процессора одним потоком. С каждым потоком связан *контекст* – структура, содержащая информацию о состоянии потока, в том числе содержимое регистров процессора. (Более подробную информацию можно получить, изучив структуру *TContext* в файле windows.pas из папки ..\borland\delphi4\source\rtl\win.) Вытеснение потока сопровождается сохранением содержимого регистров в контексте, а получение потоком кванта времени - восстановлением регистров из контекста.

К числу типичных проблем, возникающих при использовании потоков, относятся так называемые *гонки и тупики*. Конфликт (collision) *гонок* происходит, когда два потока пытаются изменить одну и ту же области памяти. Если не предпринимать специальных мер синхронизации, результат будет непредсказуемым и зависимым от того, какой из потоков «успеет первым». Например, на счету в банке было 1000$. Первый поток снял 300$ со счета в банке, но остаток записать не успел, так как был вытеснен другим потоком. Второй поток снял деньги 999$ (разумеется, с 1000$), записал остаток 1$, но тут управление вновь получил первый поток и, записав остаток 700$, уничтожил «следы деятельности» второго потока.

Тупики возникают, когда потоки ожидают ресурсы, занимаемые друг другом. Например, поток А держит ресурс 1, ожидая когда поток Б отдаст ему ресурс 2, Но поток Б ресурс не отдает, так как ждет ресурса 1. В результате оба потока бездействуют.

Обе проблемы могут быть решены с помощью средств синхронизации потоков, описываемых ниже.

2. Создание потоков средствами API

Использование библиотеки API Win32 – наиболее мощный и универсальный способ работы с потоками на большинстве языков программирования [6]. Имеющиеся в Delphi специализированные классы являются надстройкой над соответствующими функциями API.

Поток создается с помощью функции API, возвращающей дескриптор потока

```
function CreateThread (
  Attr: Pointer;         // Адрес атрибутов безопасности
  Stack: Dword;            // Размер стека для потока
  Start: Pointer;          // Начальный адрес потока
  par: pointer;            // Аргументы потока
  flag: Dword;             // флаг создания
  var ID: Dword  // Возвращаемый идентификатор потока
): THandle;     // Результат функции – дескриптор потока
```

Параметр *Attr* обычно задается равным *Nil*, что соответствует атрибутам безопасности по умолчанию (более подробно см. раздел Win32 Programmer's Reference в справочной системе Delphi).

Параметр *Stack* задает размер стека для потока. Если он равен нулю, то размер стека совпадает с размером стека основного потока приложения.

Параметр *Start* – основной. Через него передается адрес функции, вызываемой при запуске потока. Эта функция обязана возвращать результат типа *longint*, иметь один параметр типа *Pointer*. При описании функции необходимо указывать атрибут *stdcall* (см.листинг), определяющий стандартный для API способ вызова функции (запись аргументов в стек в порядке справа-налево, очистка стека при завершении работы самой подпрограммой).

Параметр *Par* – указатель на структуру (запись), содержащую аргументы, передаваемые в процедуру потока. Если *Par=Nil*, то аргументы отсутствуют.

Если задать флаг создания *flag*, равный нулю, то поток сразу начнет работу, а если указать константу CREATE_SUSPENSED, то поток начнет работу только после вызова функции

```
function ResumeThread (hThread: THandle): Dword;
```

Здесь *hThread* – дескриптор созданного потока.

Приостановить поток можно с помощью функции

```
function SuspendThread (hThread: THandle): Dword;
```

Чтобы досрочно завершить поток, следует вызвать функцию *Suspend-Thread* , а затем - функцию

```
function CloseHandle(hThread: THandle): Dword;
```

Проиллюстрируем использование API из программ на Delphi следующей лабораторной работой. Простой проект будет выполнять расчет традиционным способом и, для сравнения, с помощью дополнительного потока.

Лабораторная работа № 1

1. Создав новый проект, расположим на форме кнопки с надписями «С потоком», «Без потока», «Приостановить», «Продолжить».

2. Напишем функцию *func* и создадим обработчики событий так, чтобы листинг модуля принял вид:

```
unit Unit1;
interface
uses
     Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

var hThread: THandle; // Дескриптор потока
    ThreadID: DWord;  // Идентификатор потока

{$R *.DFM}

// Эта функция далее будет выполняться внутри потока
function func (p: pointer): longint; stdcall;
var i: integer;
  dc: hDc;//дескриптор контекста графического устройства
  s: string;
begin
  dc:=GetDc(form1.handle);//Получаем контекст формы
  for i:=0 to 10000000 do begin
     s := IntToStr(i); // Выводим в цикле число,
```

```
                        // используя функцию API
      textout(dc,10,10,pchar(s), length(s));
  end;
  ReleaseDC(Form1.handle,dc); //Освобождаем контекст
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  // Вызываем функцию Func в составе потока. Для этого
  // создаем поток и передаем в него адрес функции
  hThread := CreateThread(Nil,0,@Func,Nil,0,THreadID);
  if hThread=0 then ShowMessage('No THread');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  SuspendThread(hThread); // Приостанавливаем поток
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
   ResumeThread(hThread); // Продолжаем поток
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  Func(nil);  // Вызываем функцию обычным способом
end;
end.
```

В результате выполнения окно программы примет вид:

Комментарии

1. При нажатии кнопки «Создать поток» будут выполняться не один, а два потока: основной поток приложения обрабатывает сообщения, адресованные главной форме, дополнительный – выводит числа. Если вызвать функцию *Func(nil)* напрямую (при нажатии кнопки «Без потока»), то все операции будет выполнять основной поток, поэтому программа «зависнет», пока не закончится цикл. В это время окно не реагирует на события, не позволяя себя ни переместить,

ни закрыть. При нажатии кнопки «Создать поток» окно программы будет вести себя обычным, «дружелюбным» образом.

2. Для вывода строки текста вместо «напрашивавшихся» визуальных средств Delphi использовалась функция API *TextOut*. Дело в том, что подавляющее большинство визуальных компонентов Delphi стабильно работают только внутри главного потока приложения. Исключением являются компоненты для доступа к базам данных и графические классы, работающие с *TCanvas* (с последними необходимо использовать блокировки холста *Lock* и *UnLock* [4]). Если же без компонентов Delphi внутри дополнительных потоков обойтись нельзя, то для создания таких потоков рекомендуется использовать специализированный класс *TThread*, входящий в библиотеку Delphi.

3. Класс TThread

Класс *TThread* инкапсулирует функции API программирования потоков. Его основными преимуществами являются удобство использования, свойственное всем классам-надстройкам, и наличие специального метода Synchronize для корректного использования внутри потоков визуальных компонентов VCL Delphi.

Свойства и методы класса *TThread*

`constructor Create (CreateSuspensed: Boolean);`	Создает поток. Если аргумент имеет значение *False*, созданный поток немедленно начинает выполнение (управление передается методу *Execute*) . Если *True* – поток ожидает вызова метода *Resume*
`destructor Destroy: override;`	Завершает поток и освобождает все ресурсы, им занятые. Вызывается автоматически при завершении метода *Execute*
`procedure Resume;`	Возобновляет поток после приостановки
`procedure Suspend;`	Приостанавливает поток
`property Suspended: Boolean;`	При записи *True/False* приостанавливает/возобновляет поток. При чтении показывает, не приостановлен ли поток
`procedure Terminate;`	Устанавливает свойство *Terminated* в *True*. При использовании этого метода для завершения потока метод *Execute* должен включать в себя проверку свойства *Terminated* (см. лаб. раб. №2).
`property Terminated: Boolean;`	Показывает True, если ранее был вызван метод *Terminate*.
`function WaitFor: integer;`	Приостанавливает текущий поток до завершения заданного потока и возвращает код завершения. Например, внутри потока *T1* вызов *code:=T2.WaitFor* приостанавливает *T1* до завершения *T2*
`property Handle: THandle;`	Дескриптор потока

property ThreadID: THandle;	Идентификатор потока
property Priority: TThreadPriority;	Приоритет потока
procedure Synchronize (Method: TThreadMethod);	Метод используется для обращения к компонентам VCL внутри потока. Указанный в качестве аргумента метод, содержащий вызовы VCL, включается в главный поток приложения.
procedure Execute; virtual; abstract;	Главный метод класса. Обязательно переопределяется, после чего должен содержать код потока
property ReturnValue: integer;	Код завершения потока. По умолчанию – ноль. Другие значения могут быть присвоены внутри потока по усмотрению программиста
property OnTerminate: TNotifyEvent;	Событие, происходящее после завершения метода *Execute*, но перед *Destroy*

Примечания

1. Необходимыми являются только методы *Create* и *Execute*. Последний метод является абстрактным. Поэтому нельзя создать экземпляр *TThread*, необходимо предварительно обязательно создать потомка класса *TThread* и перекрыть в нем метод *Execute*, добавив в него необходимую функциональность.

2. Приоритет потока может принимать следующие значения:

tpIdle	Поток получает квант времени только тогда, когда система находится в состоянии простоя
tpLowest	Приоритет на два пункта ниже нормального
tpLower	Приоритет на один пункта ниже нормального
tpNormal	Нормальный приоритет
tpHigher	На один пункт выше нормы
tpHighest	На два пункта выше нормы
tpTimeCritical	Максимальный приоритет

Приоритеты **tpHighest** и **tpTimeCritical** следует назначать с осторожностью, чтобы не нарушить работу приложения.

Лабораторная работа № 2

Работа демонстрирует методику создания и завершения потока с помощью методов класса *TThread*.

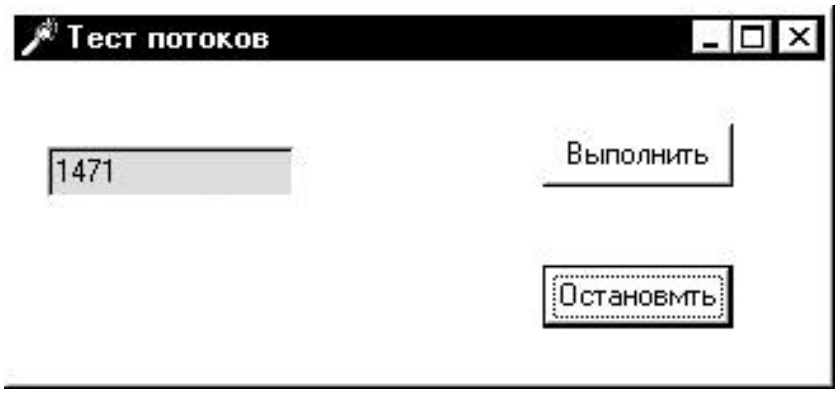
Организуйте на форме две кнопки и одну строку редактирования, запишите объявление класса *TSimpleThread* и переопределите его метод *Execute*, как это

показано на листинге. В форме определите метод OutMessage. Внешний вид работающего приложения показан на рисунке.

```
unit ThrdUnit;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, ComCtrls, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
type    // Объявление класса потока-потомка
  TSimpleThread = class(TThread)
    procedure Execute; override;
  public // Здесь удобно добавить любые поля и методы,
   //которые будут  использоваться локально, внутри потока
    Count: Integer; // счетчик цикла
    procedure OutMessage;
  end;

procedure TSimpleThread.Execute;
begin
  // Содержание потока
 while not Terminated do begin//Поток будет выполняться,
                // пока не будет нажата кнопка Button2
    Count := Count+1;
    Synchronize(OutMessage); // Безопасный вывод
  end;
end;

procedure TSimpleThread.OutMessage;
begin
  Form1.Edit1.Text:=IntToStr(count); // Обращение к VCL
end;

var Thread1: TSimpleThread; // Экземпляр класса «поток»

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  Thread1:=TSimpleThread.Create(False);//Создание потока
  Thread1.Priority := tpLowest;        // Приоритет
  Thread1.Count := 0;    // Начальное значение счетчика
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
   Thread1.Terminate; // Завершаем бесконечный поток
end;

end.
```

Результат выполнения программы:



Комментарии

1. Аргументом метода *Synchronize* является имя *метода (но не процедуры!),* который обращается к объектам VCL. Последний метод должен быть оформлен в виде процедуры без аргументов. Например, выше для вывода в строку редактирования *Edit1* был определен метод потока

```
procedure TSimpleThread.OutMessage;
begin
  Form1.Edit1.Text := IntToStr(count);
end;
```

а внутри метода *Execute* потока происходил вызов

```
Synchronize(OutMessage);
```

Метод *Synchronize* здесь необходим, чтобы во время записи в объект *Edit1* заблокировать главный поток приложения: работающий главный поток позволяет компонентам, расположенным в окне, обмениваться сообщениями; при этом может наступить ситуация гонок, например, одновременное присваивание

```
Form1.Edit1.Text := IntToStr(count);
```

и редактирование *Edit1* другим потоком или пользователем, что в данном простом примере не имеет особого значения, но в принципе *недопустимо*. Чтобы лучше понять происходящее, измените метод, добавив задержку на 1 секунду:

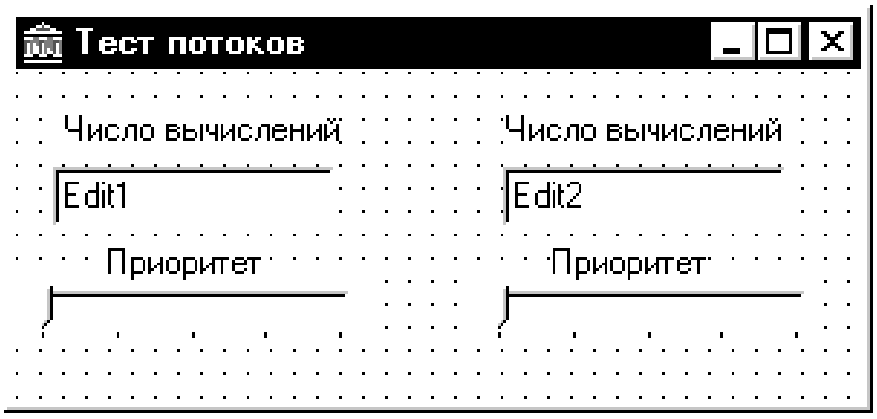
```
procedure TSimpleThread.OutMessage;
begin
  Sleep(1000);
  Form1.Edit1.Text := IntToStr(count);
end;
```

После запуска программы обнаружится, что в течение одной секунды между приращениями счетчика невозможно ни отредактировать строку, ни переместить окно: метод *OutMessage* выполняется в главном потоке и блокирует его остальные действия, как это было в лаб. работе 1 при нажатии кнопки «Без потока». (Временная задержка 1 сек. нужна лишь, чтобы визуально заметить данный эффект). Очевидно, метод *OutMessage* не рекомендуется загромождать вычислениями, так как это сильно замедлит работу главного потока.

2. Чтобы создать поток, удобно воспользоваться имеющимся в Delphi шаблоном: выбрать пункт меню File | New | Thread Object и ввести имя класса потока-потомка *TThread*, например, *TFirstThread*. После этого появится автоматически сгенерированный модуль с объявлением класса *TFirstThread*, который следует должным образом видоизменить и подключить к вызывающему модулю обычной директивой *uses*. *Упражнение:* переделайте лабораторную работу с использованием шаблонов.

Лабораторная работа №3 (самостоятельная)

Доработайте предыдущий проект, сделав два независимых аналогичных потока *Thread1, Thread2*, выводящих результаты в отдельные строки редактирования. Здесь можно описать два класса, но поскольку два потока идентичны, можно ограничиться созданием двух экземпляров одного класса *TSimpleThread*. Для ступенчатой регулировки приоритетов используйте два компонента *TTrackBar* (см. рисунок). Проект должен показывать изменение скорости работы в зависимости от установленного приоритета. Образцы решения задачи см. в [4,5].



4. Синхронизация потоков

Как уже упоминалось, при доступе потоков к общим ресурсам (глобальным данным, одному и тому же файлу, DLL, коммуникационному порту и т.п.) возникают коллизии. Предотвратить коллизии можно с помощью различных средств синхронизации. Чаще всего для этой цели используются: *критические секции, исключающие семафоры (мьютексы), обычные семафоры, события*. Все упомянутые механизмы основаны на блокировке доступа к ресурсу, пока этот ресурс занят каким-либо потоком.

Рассмотрим более подробно механизм критической секции, ограничившись его реализацией на базе функций API Win32 (как обычно, в Delphi имеются аналогичные классы-надстройки). Участок кода, содержащий потенциально опасные обращения к ресурсу, помечается как «критическая секция». Перед входом в критическую секцию проверяется глобальная запись типа *TRTLCriticalSection* (структуру записи см. в файле ..\borland\delphi4\source\rtl\win\windows.pas), в которой отмечено, выполняется ли критическая секция в данный момент времени каким-либо другим потоком. Если секция не занята, поток допускается к работе.

В начале критической секции необходимо вызвать подпрограмму API

```
procedure EnterCriticalSection (sect:TRTLCriticalSection);
```

которая проверяет состояние записи *sect: TRTLCriticalSection* и, если в записи секция помечена как занятая другим потоком, приостанавливает текущий поток, пока критическая секция не освободится.

В конце критической секции вызывается подпрограмма

```
procedure LeaveCriticalSection (sect:TRTLCriticalSection);
```

которая делает пометку в записи, что критическая секция свободна, и другой поток в соответствии с приоритетом получает квант времени и входит в критическую секцию.

Предварительно перед использованием необходимо проинициализировать запись *sect: TRTLCriticalSection* с помощью процедуры

```
procedure InitializeCriticalSection(Sect);
```

а при завершении использования – удалить:

```
procedure DeleteCriticalSection(Sect1);
```

В рассматриваемом ниже примере первая операция выполняется в обработчике события создания формы OnCreate, а вторая – в обработчике события удаления формы из памяти OnDestroy.

Лабораторная работа № 4. Синхронизация доступа к глобальной переменной с помощью критической секции

Рассмотрим пример, когда два потока обращаются к одной и той же глобальной переменной GlobalData, увеличивают и сразу уменьшают ее значение

```
GlobalData := GlobalData+3;
GlobalData := GlobalData-3;
```

а результаты выводят в соответствующие компоненты *TMemo*. Для создание программы выполните следующее.

1. Разместите на форме два компонента *Memo1* и *Memo2*, две метки с надписями «Первый поток:» и «Второй поток:», кнопку *Button1* «Выполнить», выключатель *cbSect: TCheckBox* с подписью «Использовать критическую секцию». Чтобы можно было сравнивать результаты, полученные с использованием синхронизации и без синхронизации, в обработчике события *OnClick* для выключателя запишите

```
CritSect := cbSect.Checked;
```

2. Инициализацию и удаление критической секции запишите, соответственно, в обработчиках событий *OnCreate* и *OnDestroy* для формы.

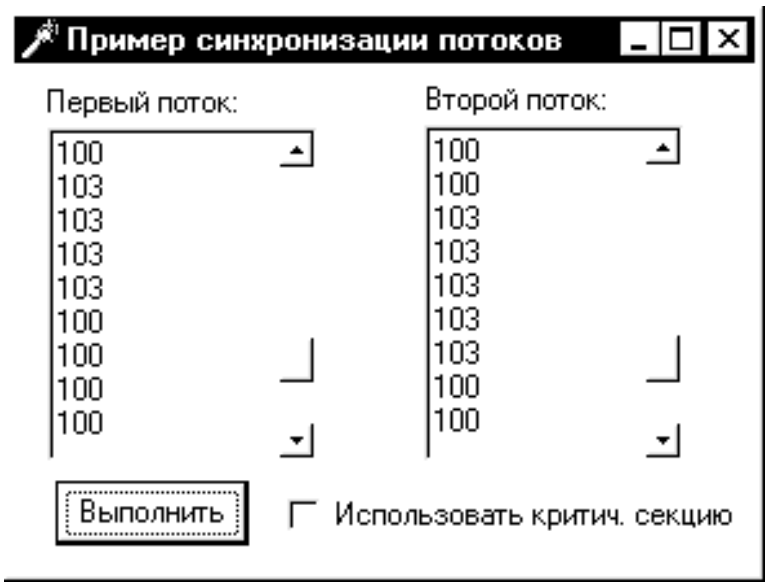
3. Так оба потока идентичны, нерационально создавать два отдельных класса. Поэтому опишите один класс *TMyThread*, переопределите его методы и создайте два экземпляра этого класса, как показано на листинге.

```
unit ThrdUnit;
interface
uses
      Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, ComCtrls, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    cbSect: TCheckBox;
    Label1: TLabel;
    Label2: TLabel;
    Memo1: TMemo;
    Memo2: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure cbSectClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

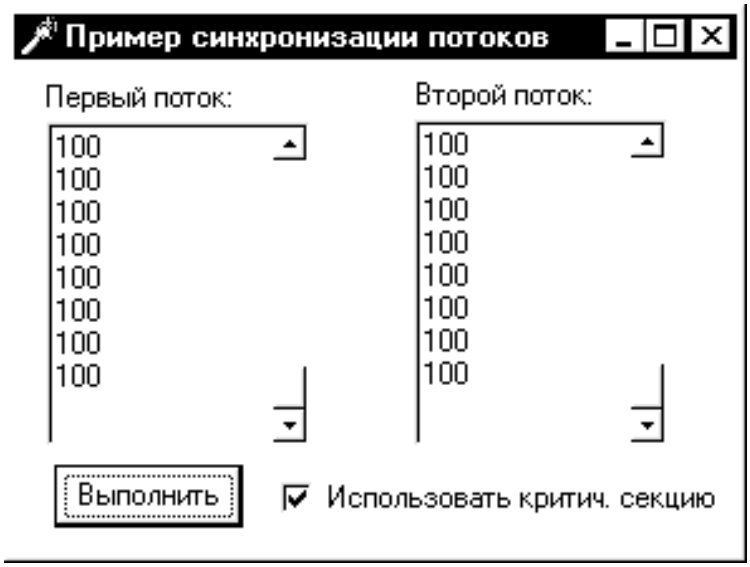
```
var
  Form1: TForm1;

type
  TMyThread = class(TThread)
  private
    { Private declarations }
    mem: TMemo;
   // В переменную mem при вызове конструктора потока будет
   // передаваться конкретный экземпляр TMemo
  protected
    constructor Create (M: TMemo);// Переопределение
    procedure Execute; override;
    procedure OutMessage;
  end;

implementation

{$R *.DFM}

var CritSect: boolean; // Флаг «включать-не включать»
    Sect1: TRTLCriticalSection; //Критическая секция
    GlobalData: integer; // Глобальная переменная

constructor TMyThread.Create(m: TMemo);
begin                       //Переопределяем конструктор
  Inherited Create(False);//Вызываем унаследованный
  Mem := m;               // Передаем, куда записывать числа
end;

procedure TMyThread.OutMessage;
begin // Добавление  строки в редактор
  Mem.Lines.add(IntToStr(GlobalData));
end;

procedure TMyThread.Execute;
var j: integer;
begin
  for j:=1 to 50 do begin
    // Если механизм критической секции используется
    if (CritSect) then EnterCriticalSection(Sect1);
    // Вход в критическую секцию
    // Изменение глобальной переменной
    GlobalData := GlobalData+3;
    Sleep(3); // Задержка на 3 мс для увеличения
           // вероятности рассинхронизации
    GlobalData := GlobalData-3;
    Synchronize(OutMessage);
    // Выход из критической секции
    if (CritSect) then LeaveCriticalSection(Sect1);
  end;
end;
```

```
var Thread1, Thread2: TMyThread; //Экземпляры потоков

procedure TForm1.Button1Click(Sender: TObject);
begin
  GlobalData := 100;
  Memo1.Lines.Clear;
  Memo2.Lines.Clear;
  Thread1 := TMyThread.Create(Memo1);// Старт потока 1
  Thread2 := TMyThread.Create(Memo2);// Старт потока 2
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  InitializeCriticalSection(Sect1);
  CritSect := false;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  DeleteCriticalSection(Sect1);
end;

procedure TForm1.cbSectClick(Sender: TObject);
begin
  CritSect := cbSect.Checked;
end;

end.
```

Очевидный, казалось бы, результат работы потоков (*GlobalData=100*) искажается коллизией «гонок». Окно программы без использования синхронизации имеет вид:



Использование критической секции решает проблему:



В заключение кратко охарактеризуем некоторые другие методы синхронизации.

Семафоры регулируют число потоков, которые могут использовать ресурс. *Исключающий семафор (mutex, мьютекс)* допускает к ресурсу только один поток, *обычные семафоры* допускают конечное число потоков. Семафоры и мьютексы во многом похожи на критические секции, методика их использования практически полностью совпадает с методикой, рассмотренной выше. Преимуществом по сравнению с критическими секциями является, прежде всего, глобальность действия, т.е. способность блокировать потоки, запущенные всеми активными приложениями. Кроме того, при обращении к семафорам и мьютексам можно указывать максимальное время ожидания доступа к ресурсу, что позволяет предотвращать ситуацию «тупиков».

Мьютекс создается функцией

```
function CreateMutex(
  Attr: Pointer;   // указатель на атрибуты безопасности
  flag: Boolean;   // флаг создания
  lpName: PChar    // указатель на имя мьютекса
 ): THandle;       // дескриптор мьютекса
```

Стандартный вариант создания:

```
hMutex := CreateMutex(Nil, False, Nil);
```

Мьютексы могут находиться в сигнальном (если им не владеет ни один поток, т.е. ресурс свободен) или несигнальном состоянии (поток владеет мьютексом). Чтобы поток завладел мьютексом (т.е. получил доступ к ресурсу), необходимо вызвать функцию ожидания

```
function WaitForSingleObject(
  hHandle: THandle;// дескриптор мьютекса
  dwMilliseconds: DWord // время ожидания в мс
): DWord;
```

Параметр *dwMilliseconds* может принимать значение *Infinite*, что означает бесконечный интервал ожидания.

Функция завершает работу, если мьютекс становится сигнальным или истекло время ожидания. После этого поток, находившийся в состоянии ожидания мьютекса, получает к нему доступ, а мьютекс, получивший нового владельца, перестает сигнализировать. Мьютекс можно сравнить с «эстафетной палочкой», которую бегун (поток) молча держит, проходя дистанцию, и кричит «Держи!» (сигнализирует), завершив дистанцию.

После того, как поток выполнил участок кода, содержащий совместно используемый ресурс, необходимо освободить мьютекс от потока-владельца с помощью функции API

```
function ReleaseMutex(
  hHandle: THandle;// дескриптор мьютекса
): Boolean;
```

При завершении работы мьютекс удаляется из памяти с помощью известной Вам функции *CloseHandle*, в которую передается дескриптор мьютекса.

Семафор создается функцией

```
function CreateSemaphore (
  Attr: Pointer;   // указатель на атрибуты безопасности
  InitialCount: longint; // Начальное число потоков,
                         // допущенных к объекту
  MaxCount: longint; // Максимальное число допускаемых
                     // потоков
  lpName: PChar  // указатель на имя семафора
 ): THandle;     // дескриптор семафора
```

Семафор сигнализирует, если *InitialCount* больше нуля, и не подает сигнала, если *InitialCount* равен нулю.

Для доступа к объекту используется функция ожидания **WaitForSingleObject** или **WaitForMultipleObjects** [6].

При завершении функции ожидания значение счетчика *InitialCount* соответственно уменьшается и поток получает доступ к ресурсу.

При завершении доступа к ресурсу необходимо вызвать функцию, увеличивающую значение счетчика семафора:

```
function ReleaseSemaphore(
  hHandle: THandle;// дескриптор семафора
  ReleaseCount: longint // приращение счетчика
  LP: Pointer  // как правило, nil
): Boolean;
```

Более подробная информация приведена в [6].

Упражнение: Переделайте лабораторную работу №4, заменив критические секции а) мьютексами, б) семафорами.

Задания для самостоятельной работы

1. Создать два потока, один из которых заполняет Memo случайными числами до остановки процесса пользователем, а второй осуществляет вычисление очередного числа Фибоначчи после нажатия пользователем кнопки. При превышении допустимой границы вычисления целого числа изменить надпись на кнопке и перейти к вычислению квадратного корня.

2. Создать два потока, один из которых выводит в Memo 200 строк, заполняя их соответствующими номеру строки числами, а второй вычисляет значение функции в точке. Выбор функции реализовать с помощью ComboBox, а значение аргумента вводить в редакторе. Для заполнения Memo предусмотреть индикатор процесса.

3. Вычислить приближенное значение определенного интеграла с помощью метода прямоугольника, метода трапеций, метода Симпсона. Выбор функции реализовать с помощью ComboBox, а границы отрезка задавать в редакторе. Предусмотреть возможность графического отображения.

4. Вычислить приближенное значение определенного интеграла с помощью метода трапеций. Разные потоки осуществляют вычисления с различным шагом. Выбор функции реализовать с помощью ComboBox, а границы отрезка задавать в редакторе. Предусмотреть возможность графического отображения.

5. Реализовать решение задачи Коши для обыкновенных дифференциальных уравнений методами Эйлера и Рунге-Кутта. Предусмотреть индикатор процесса в виде круговой диаграммы.

6. Решить нелинейное уравнение с одним неизвестным методами половинного деления, методом хорд, методом Ньютона. Предусмотреть возможность графического отображения.

7. Пользователь задает 2 функции. Реализовать потоки для построения графиков функций по задаваемому числу точек с возможностью задания приоритета потока. Точки пересечения выделить другим цветом.

8. Задан массив целых положительных чисел. Осуществить сортировку массива методами пузырька, выборки и методом быстрой сортировки. Для каждого метода предусмотреть возможность графического отображения: каждому числу массива поставить в соответствие линию пропорциональной длины; при сортировке линии также меняются местами.

9. Задается функция. В нескольких окнах построить графики функции по задаваемому числу точек с возможностью задания приоритета потока.

10. Описать движение шарика в прямоугольной области с отражением от границ под случайным углом. Предусмотреть возможность запуска нескольких шариков.

11. Реализовать алгоритм нахождения кратчайшего пути в графе. Создавать потоки при выходе из рассматриваемой вершины нескольких дуг.

12. Зашифровать текст по формуле $y_i=x_i+k_i \bmod n$, а также методом перестановки с заданием двух ключей.

13. Приблизительно вычислить значение интеграла

$$\int_{-0,82}^{0,976} [1/(1+t)+(1+t)^{0,146} - e^t - cos(t)]\,dt$$

как сумму соответствующих степенных рядов, задавая ограничения на величину n члена ряда для каждого потока отдельно.

14. Изобразить на форме несколько фигур, вращающихся с различными скоростями.

15. Рассчитать сигналы на выходе авторегрессионных моделей различных порядков, изображая графики на различных PaintBox.

16. Рассчитать реакции ФНЧ Баттерворта n-го порядка на белый шум.

Литература

1. Кастер Х. Основы Windows NT и NTFS. – М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd», 1996. – 440 с.

2. Кинг А. Windows 95 изнутри. – СПб: Питер, 1995. –512 с.

3. Калверт Ч. Delphi 4. Самоучитель. – К.: Изд-во «Диасофт»,1999. – 192с.

4. Дарахвелидзе П.Г., Марков Е.П. Delphi 4. – СПб.: БХВ-Санкт-Петербург, 1999. – 816 с.

5. Александровский А.Д. Delphi 5.0. Разработка корпоративных приложений. – М.: ДМК, 2000. – 512 с.

6. Бартеньев О.В. Visual Fortran: новые возможности. - М.: «Диалог – МИФИ», 1999. - 304 с.

7. Введение в Delphi / Сост. :Рудалев В.Г., Кремер А.И.; Воронеж. гос. ун-т. - Воронеж, 2000. – 36 с.

Составители: Рудалев Валерий Геннадьевич,
Крыжановская Юлиана Александровна

Редактор Тихомирова О.А.