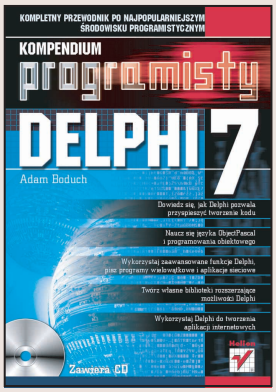
Delphi 7. Kompendium programisty



Autor: Adam Boduch
ISBN: 83-7361-087-1
Format: B5, stron: 686

Spośród wielu dostępnych środowisk programistycznych Delphi wyróżnia się łatwością i szybkością tworzenia aplikacji. Zastosowany język programowania ObjectPascal utrwala dobre nawyki programowania strukturalnego, wzbogacając je o nowe możliwości, jakie niesie za sobą programowanie zorientowane obiektowo. Wygodne środowisko RAD, możliwość łatwego uzyskiwania dostępu do relacyjnych baz danych oraz możliwość tworzenia aplikacji wieloplatformowych, to kolejne zalety Delphi, które przekonały doń rzesze programistów.

Książka „Delphi 7. Kompendium programisty" to — jak wskazuje jej tytuł — kompletny przewodnik po Delphi, obejmujący zarówno opis zintegrowanego środowiska programistycznego, języka ObjectPascal, jak i najważniejszych funkcjonalności zawartych w dołączonych do Delphi bibliotekach. Jest to pozycja dla początkujących adeptów sztuki programistycznej, a także dla tych wszystkich, którzy chcą poszerzyć i wzbogacić swoją wiedzę o tym wygodnym narzędziu.

Książka przedstawia:

- Podstawowe informacje o Delphi
- Język ObjectPascal — jego strukturę i składnię
- Programowanie zorientowane obiektowo
- Interfejs Delphi
- Obsługę komunikatów w Delphi
- Korzystanie z rejestru Windows i plików .ini
- Dostęp do plików
- Tworzenie aplikacji wielowątkowych
- Multimedialne funkcje Delphi
- Tworzenie aplikacji sieciowych
- Pisanie własnych bibliotek DLL
- Wykorzystanie oraz tworzenie własnych bibliotek VCL i CLX
- Dostęp do relacyjnych baz danych
- IntraWeb — Delphi i WWW

Do książki dołączony jest CD-ROM, na którym znajdziesz kody źródłowe wykorzystane w książce. Cennym uzupełnieniem jest także dodatek, przedstawiający przetestowane w praktyce zalecenia dotyczące pisania czytelnego kodu w Delphi.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

8.

Aplikacje wielowątkowe

Słowo *wątek* może mieć różne znaczenie. W świecie programistów może oznaczać możliwość wykonywania wielu czynności naraz. Przykładowo w systemie Windows możemy uruchamiać kilka programów działających jednocześnie — każdy program jest osobnym wątkiem. W tym rozdziale zajmiemy się tworzeniem kilku wątków w ramach jednego procesu.

Wskazówka

Procesem można nazwać każdą aplikację, uruchomioną w danym momencie. Taką też terminologię będę stosował w dalszej części tego rozdziału. Zatem przyjmijmy, że proces to egzemplarz aplikacji uruchomiony w systemie.

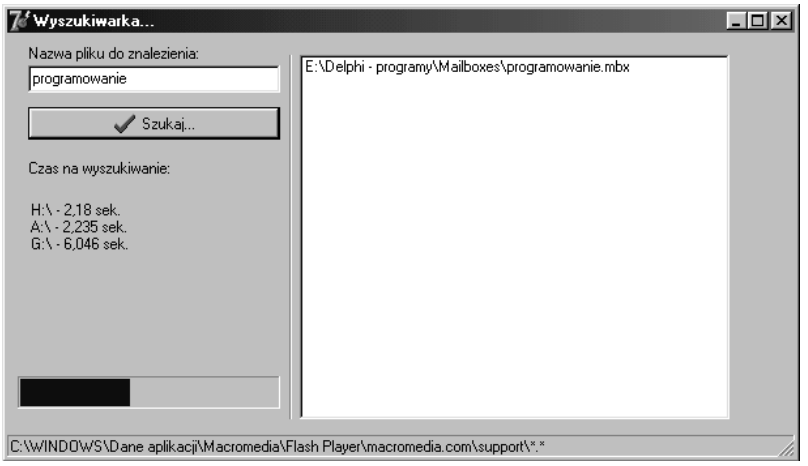
Czym tak naprawdę są wątki?

Zacznijmy od wyjaśnienia, czym tak naprawdę są wątki. Każda aplikacja (proces) działająca w systemie Windows posiada tzw. *wątek główny* (ang. *primary thread*), który może uruchamiać inne wątki poboczne (ang. *secondary threads*). W tym samym czasie może działać kilka wątków pobocznych, które wykonują różne lub te same operacje. Spójrz na rysunek 8.1. Program przedstawiony na tym rysunku dokonuje wyszukiwania wielowątkowego, analizując jednocześnie wszystkie dyski znajdujące się w systemie.

W tym wypadku zadaniem każdego wątku jest wyszukanie plików na osobnym dysku. W rezultacie jeden wątek przypada na każdy dysk, dzięki czemu wyszukiwanie trwa naprawdę szybko.

Wskazówka

Pełny kod źródłowy programu *Wyszukiwanie wielowątkowe* możesz znaleźć na płycie CD-ROM w katalogu *../listingi/8/Wyszukiwarka*.



Rysunek 8.1. *Wyszukiwanie wielowątkowe*

Być może to, co napisałem do tej pory przybliżyło Ci trochę zasadę funkcjonowania wątków. Wyobraź sobie możliwość wykonywania innych czynności *w tle* aplikacji — bez jej jednoczesnego blokowania. Dajesz użytkownikowi możliwość dokonywania zmian w programie, a w tle może działać inny wątek, który wykonywać będzie pozostałe operacje.

Klasa TThread

Podczas tworzenia aplikacji wielowątkowych będziemy korzystali z klasy VCL — `TThread`. Istnieje oczywiście możliwość tworzenia wątków przy wykorzystaniu mechanizmów WinAPI, lecz klasa `TThread` w dużym stopniu zwalnia nas z mozolnego kodowania — jest po prostu łatwiejsza w obsłudze.

Wskazówka

Klasa `TThread` znajduje się w module *Classes.pas*.

Deklaracja klasy TThread

Deklaracja klasy `TThread` znajduje się w pliku *Classes.pas* i przedstawia się w następujący sposób:

```
TThread = class
  private
    FHandle: THandle;
    FThreadID: THandle;
    FTerminated: Boolean;
```

```
    FSuspended: Boolean;
    FFreeOnTerminate: Boolean;
    FFinished: Boolean;
    FReturnValue: Integer;
    FOnTerminate: TNotifyEvent;
    FMethod: TThreadMethod;
    FSynchronizeException: TObject;
    procedure CallOnTerminate;
    function GetPriority: TThreadPriority;
    procedure SetPriority(Value: TThreadPriority);
    procedure SetSuspended(Value: Boolean);
  protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue write FReturnValue;
    property Terminated: Boolean read FTerminated;
  public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
    property Handle: THandle read FHandle;
    property Priority: TThreadPriority read GetPriority write SetPriority;
    property Suspended: Boolean read FSuspended write SetSuspended;
    property ThreadID: THandle read FThreadID;
    property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
  end;
```

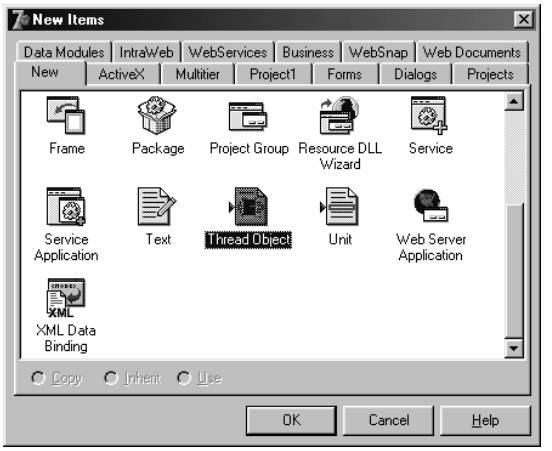
Działanie wątku można wstrzymać lub wznowić dzięki metodom `Suspend` i `Resume`. Rozpoczęcie wątku jest jednak realizowane za pomocą metody `Execute`.

Tworzenie nowej klasy wątku

Jeżeli chcemy utworzyć nowy wątek, jedynym rozwiązaniem jest zadeklarowanie w kodzie programu nowej klasy, dziedziczącej po `TThread`. Klasę tę możemy samodzielnie wpisać bezpośrednio w kod programu lub skorzystać z kreatora Delphi.

Z menu *File* wybierz *New/Other*, co spowoduje otwarcie *Repozytorium* (o *Repozytorium* pisałem w rozdziale 4.). Wystarczy na zakładce *New* wybrać pozycję *Thread Object* (rysunek 8.2).

Po naciśnięciu przycisku *OK* zostaniesz poproszony o wpisanie nazwy klasy w odpowiednim oknie. Wpisz np. `TMojWatek`. Wówczas stworzony zostanie nowy moduł, a w nim deklaracja nowej klasy (patrz listing 8.1).



Rysunek 8.2. *Okno Repozytorium*

Listing 8.1. *Kod źródłowy nowego modułu wygenerowanego przez Delphi*

```
unit Unit2;

interface

uses
  Classes;

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

implementation

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

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TMojWatek.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{ TMojWatek }

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

end.
```

Nowy moduł zawiera klasę `TMojWatek`, w której umieszczona jest jedna metoda (w sekcji `protected`). To właśnie w metodzie `Execute` należy umieścić właściwy kod wątku. Ponadto w module znajduje się ciekawy komentarz, który zostanie przeze mnie omówiony w dalszej części rozdziału.

W każdym bądź razie nie jest konieczne tworzenie nowego modułu dla klasy wątku. Nie jest także konieczne tworzenie samej klasy w taki sposób, w jaki to przedstawiłem. Równie dobrze można zadeklarować klasę samodzielnie.

Wskazówka

Podczas samodzielnego deklarowania klasy dziedziczącej po `TThread` nie wolno zapominać o deklaracji metody `Execute`. Metoda `Execute` musi być umieszczona w sekcji `protected` i opatrzona dyrektywą `override`.

Kilka instancji wątku

W każdej klasie wątku mogą być oczywiście deklarowane metody i właściwości — zupełnie tak samo, jakby to była zwykła klasa. Istnieje także możliwość uruchamiania kilku klas wątku jednocześnie! Powoduje to stworzenie dla każdej klasy osobnej instancji zmiennej i zarezerwowanie osobnego bloku pamięci.

Tworzenie wątku przedstawia się następująco:

```
TMojWatek.Create(False);
```

Po wywołaniu konstruktora klasy uruchamiany jest cały proces (metoda `Execute`), a to za sprawą parametru typu `Boolean` zawartego w konstruktorze. Jeżeli wartość tego parametru to `True`, uruchomienie wątku nastąpi dopiero po wywołaniu metody `Resume`.

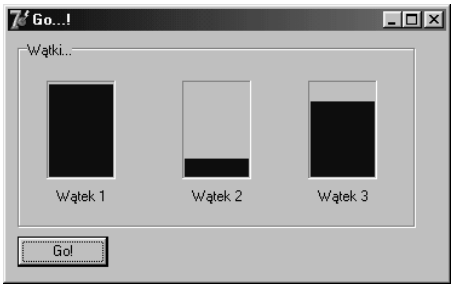
Wskazówka

Nie zaleca się uruchamiania w tym samym czasie dużej ilości wątków w ramach tego samego procesu. Zalecana ilość to 16 wątków w ramach jednego procesu.

Tworzenie klasy

Przedstawię Ci teraz przykładowy program tworzący trzy wątki pochodne, które będą działać jednocześnie. Ich działanie nie spowoduje zablokowania programu — użytkownik będzie mógł przeciągać okno programu, minimalizować go itp.

Przykładowy program będzie banalny i raczej niepraktyczny. Wątek wylosuje jakąś liczbę z zakresu od 0 do 999 i wykona pętlę `for` od liczby 1 do tej wylosowanej wartości. Pętla będzie wykonywana tylko przez jakiś czas —dzięki spowalnianiu (funkcja `Sleep`). Przerwa między kolejnymi iteracjami to 100 milisekund. Program przedstawiony został na rysunku 8.3.



Rysunek 8.3. *Działanie trzech wątków naraz*

Postęp wykonywania pętli przedstawiony jest za pomocą komponentów `TProgressBar`.

Kod klasy

Deklaracja klasy jest dość prosta — wykorzystujemy jedną metodę, konstruktor oraz dwie właściwości:

```
TGoThread = class(TThread)
private
  FV : Integer; // wylosowana liczba
  FCounter : Integer; // numer wątku
protected
  procedure Execute; override;
public
  constructor Create(Counter : Integer);
end;
```

Deklarowanie konstruktora przez programistę nie jest konieczne, lecz ja stworzyłem go ze względu na konieczność przekazania do klasy pewnego parametru, jakim jest numer wątku:

```
constructor TGoThread.Create(Counter: Integer);
begin
  inherited Create(False); // wywołanie wątku
  FCounter := Counter; // przypisanie wartości do zmiennej
end;
```

Na początku w konstruktorze wywołujemy konstruktor klasy bazowej. Następnie zmiennej `FCounter` przypisujemy wartość, która została podana wraz z parametrem konstruktora.

Oto, jak wygląda główna procedura — `Execute`:

```
procedure TGoThread.Execute;
var
  i : Integer;
begin
  FreeOnTerminate := True; // zwolnij po zakończeniu wątku

  Randomize;
  FV := Random(1000);
  { odnalezienie komponentu na formularzu }
  TProgressBar(MainForm.FindComponent('ProgressBar' + InttoStr(FCounter))).Max := FV;
```

```
  for i := 0 to FV do
  begin
    Sleep(10);
    TProgressBar(MainForm.FindComponent('ProgressBar' + IntToStr(FCounter))).Position := i;
  end;
end;
```

Zwróć uwagę na przypisanie do właściwości `FreeOnTerminate` wartości `True`. Spowoduje to zwolnienie klasy po zakończeniu działania wątku.

Kolejne instrukcje są już ściśle związane z działaniem owego wątku. Ciekawą konstrukcją jest:

```
TProgressBar(MainForm.FindComponent('ProgressBar' + InttoStr(FCounter))).Max := FV;
```

Taki zapis umożliwia znalezienie na formularzu komponentu bez znajomości jego nazwy. Wystarczy jedynie podać nazwę komponentu w parametrze funkcji `FindComponent`. Kompletny kod źródłowy modułu znajduje się w listingu 8.2.

Listing 8.2. *Kod źródłowy modułu*

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;

type
  TMainForm = class(TForm)
    gbHome: TGroupBox;
    ProgressBar1: TProgressBar;
    ProgressBar2: TProgressBar;
    ProgressBar3: TProgressBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    btnGo: TButton;
    procedure btnGoClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  TGoThread = class(TThread)
  private
    FV : Integer; // wylosowana liczba
    FCounter : Integer; // numer wątku
  protected
    procedure Execute; override;
  public
    constructor Create(Counter : Integer);
  end;

var
  MainForm: TMainForm;
implementation
```

```
{$R *.dfm}

procedure TMainForm.btnGoClick(Sender: TObject);
begin
{ utworzenie trzech wątków }
  TGoThread.Create(1);
  TGoThread.Create(2);
  TGoThread.Create(3);
end;

{ TGoThread }

constructor TGoThread.Create(Counter: Integer);
begin
  inherited Create(False); // wywołanie wątku
  FCounter := Counter; // przypisanie wartości do zmiennej
end;

procedure TGoThread.Execute;
var
  i : Integer;
begin
  FreeOnTerminate := True; // zwolnij po zakończeniu wątku

  Randomize;
  FV := Random(1000);
  { odnalezienie komponentu na formularzu }
  TProgressBar(MainForm.FindComponent('ProgressBar' + InttoStr(FCounter))).Max := FV;

  for i := 0 to FV do
  begin
    Sleep(10);
    TProgressBar(MainForm.FindComponent('ProgressBar' + IntToStr(FCounter))).Position := i;
  end;
end;

end.
```

Wznawianie i wstrzymywanie wątków

Klasa `TThread` posiada metody, dzięki którym możemy wznowić lub zatrzymać wykonywanie danego wątku. Zadanie wstrzymywania i wznawiania wykonywania danego wątku realizuje metoda `Suspend` i `Resume`.

```
var
  MojWatek : TMojWatek;
begin
  MojWatek := TMojWatek.Create(True);
  MojWatek.Resume; // uruchomienie wątku
  MojWatek.Suspend; // wstrzymanie
end;
```

O tym, czy wątek jest w danym momencie uruchomiony, informuje właściwość `Suspended`. Przyjmuje ona wartość `True`, jeżeli wątek jest wstrzymany, natomiast w przeciwnym wypadku — `False`.

Priorytet wątku

Wątkom można nadawać różne priorytety, zależnie od „ważności" zadania, jakie dany wątek wykonuje. Nadając operacji wyższy priorytet, uzyskujesz pewność, że procesor przydzieli czas wykonania właśnie naszemu wątkowi.

Priorytet nadaje się wątkom poprzez właściwość `Priority`, wykorzystując takie oto wartości: `tpIdle`, `tpLowest`, `tpLower`, `tpNormal`, `tpHigher`, `tpHighest`, `tpTimeCritical`. Najniższym priorytetem jest `tpIdle` — taki wątek jest wykonywany wtedy, gdy żaden inny proces nie wymaga użycia procesora (np. wygaszacze ekranu). Natomiast priorytet `tpTimeCritical` otrzymują procesy, które wymagają użycia procesora w trybie natychmiastowym.

```
MojWatek.Priority := lpHigher; // nadanie wyższego priorytetu
```

Wskazówka

Nie należy zbytnio przesadzać z nadawaniem wątkom priorytetów. Zalecane jest zachowanie priorytetu normalnego (`tpNormal`). Nadanie wątkowi zbyt wysokiego priorytetu może spowodować nieprawidłowe działanie pozostałych programów uruchomionych w tym samym czasie.

Synchronizacja

Należy rozważyć jeszcze jedną sytuację, a mianowicie uruchamianie kilku wątków w tym samym czasie. Jeżeli owe wątki modyfikują właściwości lub dokonują jakichkolwiek innych zmian w bibliotece VCL, może dojść do kolizji. Dotyczy to np. przypadku, gdy owe wątki muszą pobierać jakieś wartości z komponentów i jednocześnie je modyfikować. W tym celu zalecane jest użycie metody `Synchronize` klasy `TThread`.

```
procedure TGoThread.Execute;
begin
  Synchronize(SetProprties);
end;
```

W ten sposób wątek wywołuje metodę `Synchronize`, w której podana została nazwa procedury do wykonania — `SetProprties`. Dzięki temu masz pewność, że spośród kilku uruchomionych w danym momencie funkcji tylko jedna będzie wykonywana w danym czasie i tylko ona będzie mogła dokonywać zmian w bibliotece VCL.

Treść komentarza

Na początku rozdziału chcąc stworzyć nowy wątek, użyłeś *Repozytorium*. W module, który został utworzony przez Delphi, widniał taki komentarz:

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

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TMojWatek.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }
```

Oto jego tłumaczenie:

Ważne! Metody i właściwości obiektów VCL mogą być użyte jedynie w metodzie wywoływanej za pomocą `Synchronize`.

Pamiętasz jeszcze program, który prezentowałem Ci kilka stron wcześniej (trzy wątki modyfikujące właściwość `Position` komponentu `TProgressBar`)? W listingu 8.3 zaprezentowany jest program wykorzystujący metody `Synchronize`, dzięki której w jednym momencie dostęp do komponentów VCL ma tylko jeden wątek.

Listing 8.3. *Dostęp do VCL ma tylko jeden wątek*

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;

type
  TMainForm = class(TForm)
    gbHome: TGroupBox;
    ProgressBar1: TProgressBar;
    ProgressBar2: TProgressBar;
    ProgressBar3: TProgressBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    btnGo: TButton;
    procedure btnGoClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
  TGoThread = class(TThread)
  private
    FV : Integer; // wylosowana liczba
    FCounter : Integer; // numer wątku
    procedure SetProprties;
  protected
    procedure Execute; override;
  public
    constructor Create(Counter : Integer);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.dfm}

procedure TMainForm.btnGoClick(Sender: TObject);
begin
{ utworzenie trzech wątków }
  TGoThread.Create(1);
  TGoThread.Create(2);
  TGoThread.Create(3);
end;

{ TGoThread }

constructor TGoThread.Create(Counter: Integer);
begin
  inherited Create(False); // wywołanie wątku
  FCounter := Counter; // przypisanie wartości do zmiennej
end;

procedure TGoThread.Execute;
begin
  Synchronize(SetProprties);
end;

procedure TGoThread.SetProprties;
var
  i : Integer;
begin
  FreeOnTerminate := True; // zwolnij po zakończeniu wątku

  Randomize;
  FV := Random(1000);
  { odnalezienie komponentu na formularzu }
  TProgressBar(MainForm.FindComponent('ProgressBar' + InttoStr(FCounter))).Max := FV;

  for i := 0 to FV do
  begin
    Sleep(10);
    TProgressBar(MainForm.FindComponent('ProgressBar' + IntToStr(FCounter))).Position := i;
  end;
end;

end.
```

Zdarzenia klasy TThread

Chciałem w tym miejscu wspomnieć jeszcze o zdarzeniach, a właściwie o jednym zdarzeniu, znajdującym się w klasie `TThread`. To zdarzenie to `OnTerminate`, które może się przydać, jeżeli chcemy przechwycić zakończenie działania wątku.

Najlepiej przypisać do zdarzenia odpowiednią procedurę w momencie utworzenia klasy, czyli w konstruktorze `Create`.

```
constructor TMainThread.Create;
begin
  inherited Create(False);
  OnTerminate := MyTerminate;  // przypisanie procedury do zdarzenia
end;

procedure TMainThread.Execute;
begin
  { kod wątku }
end;

procedure TMainThread.MyTerminate(Sender: TObject);
begin
  { kod zdarzenia }
end;
```

Wskazówka

Użycie metody `DoTerminate` klasy `TThread` powoduje wywołanie zdarzenia `OnTerminate`.

Przykład: wyszukiwanie wielowątkowe

Na rysunku 8.1 przedstawione zostało działanie wielowątkowej wyszukiwarki. Co prawda cały kod źródłowy umieszczony jest na dołączonej do książki płycie CD-ROM, lecz warto omówić jego działanie. Zaprezentuję więc proces tworzenia takiego programu krok po kroku.

Jak to działa?

Sam proces wyszukiwania opisany został w poprzednim rozdziale. Nasz program będzie się różnił tym, że zadaniem każdego wątku będzie wyszukanie plików na innej partycji, co w konsekwencji potrwa krócej niż w sytuacji, gdyby miałoby to być realizowane w ramach jednego wątku.

Wyszukiwanie

Procedura wyszukiwania jest podobna do tej, którą prezentowałem w poprzednim rozdziale. Poniżej przedstawiona procedura jest *rekurencyjna*, czyli — jak zapewne pamiętasz — realizuje przeszukiwanie również w podkatalogach.

```
procedure Search(StartDir : String);
var
  SR, DR  : TSearchRec;
  Found, FoundFile : Integer;

  { ta procedura sprawdza, czy na końcu zmiennej znajduje się znak \ - jeżeli
    tak, nic nie jest wykonywane; jeżeli tego znaku brak, zostaje on dodany... }
  function IsDir(Value : String) : String;
  begin
    if Value[Length(Value)] <> '\' then  // jeżeli na końcu znajdziesz znak
      Result := Value + '\' else Result := Value; // dodaj go... w przeciwnym wypadku nie
      ➥wykonuj nic
  end;

begin
  Found := FindFirst(IsDir(StartDir) + '*.*', faDirectory, DR); // następuje pobieranie
  ➥katalogów z podanej lokalizacji
  while Found = 0 do // pętelka
  begin
    if ((DR.Attr and faDirectory) = faDirectory) and  // sprawdza, czy pozycja jest katalogiem
       ((DR.Name <> '.') and (DR.Name <> '..')) then
    begin
      MainForm.StatusBar.SimpleText := IsDir(StartDir) + DR.Name + '\*.*'; //
      ➥na komponencie wyświetl aktualnie przeszukiwany katalog
      if Pos(FFileName, DR.Name) > 0 then // sprawdź, czy w nazwie jest szukany ciąg znaków
        MainForm.lbResults.Items.Add(IsDir(StartDir) + DR.Name);

    { pobierz na razie wszystkie pliki z danego katalogu - potem je przeanalizujemy }
      FoundFile := FindFirst(IsDir(StartDir) + DR.Name + '\*.*', faAnyFile, SR);
      while FoundFile = 0 do
      begin
        if ((SR.Name <> '.') and (SR.Name <> '..')) then //
          if Pos(FFileName, SR.Name) > 0 then  // następuje sprawdzenie, czy plik nie
          ➥zawiera części szukanego ciągu
            MainForm.lbResults.Items.Add(IsDir(StartDir) + DR.Name + '\' + SR.Name);

        FoundFile := FindNext(SR); // kontynuuj przeszukiwanie
      end;
      FindClose(SR);  // zakończ

      Search(IsDir(StartDir) + DR.Name); // tutaj następuje rekurencja
    end;
    Found := FindNext(DR); // kontynuuj
  end;
  FindClose(DR);
end;
```

W powyższej procedurze zagnieżdżona jest kolejna — `IsDir`. Sprawdza ona, czy na końcu ścieżki znajduje się znak *backslash* (`\`). Jeżeli go nie ma, dodawany jest ten znak, gdyż wymagany jest on do prawidłowego działania funkcji *rekurencyjnej*.

Znalezienie konkretnego pliku jest kwalifikowane za pomocą funkcji `Pos`. Jeżeli dany plik lub katalog zawiera szukany ciąg znaków (a sprawdza to funkcja `Pos`), następuje wyświetlenie ścieżki w komponencie `TListBox`.

Obliczanie czasu przeszukiwania

Do obliczenia czasu potrzebnego na przeszukanie konkretnej partycji skorzystamy z funkcji `GetTickCount`. Funkcja ta zwraca ilość milisekund, jakie upłynęły od czasu uruchomienia systemu. Wystarczy więc pobrać wartość początkową przed wywołaniem wątku oraz wartość końcową po zakończeniu wykonywania operacji — np. przy zakończeniu wątku:

```
destructor TSearchThread.Destroy;
begin
  Stop := GetTickCount;  // pobierz czas zakończenia
  Total := Stop - Start;   // odejmij czas startu od czasu zakończenia
  Total := Total / 1000; // podziel przez 1000, aby uzyskać liczbę sekund

  { wyświetl na komponencie czas wyszukiwania na danym dysku }
  MainForm.lbEnd.Items.Add(FDrive + ':\ - ' + CurrToStr(Total) + ' sek.');
  inherited;
end;
```

Zmienna `Start` jest uprzednio pobraną wartością, określającą czas rozpoczęcia wątku. Finalną wartość `Total` należy podzielić przez 1 000, aby uzyskać liczbę sekund.

Kod źródłowy modułu

Pełny kod źródłowy modułu znajduje się w listingu 8.4.

Listing 8.4. *Kod źródłowy modułu*

```
{
   Copyright (c) 2002 by Adam Boduch
}

unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, Buttons, ComCtrls;

type
  TMainForm = class(TForm)
    leFileName: TLabeledEdit;
```

```
    btnFind: TBitBtn;
    Bevel1: TBevel;
    lbResults: TListBox;
    StatusBar: TStatusBar;
    lbEnd: TListBox;
    Label1: TLabel;
    ProgressBar: TProgressBar;
    procedure btnFindClick(Sender: TObject);
  end;

  TSearchThread = class(TThread)
  private
    Start, Stop : Integer; // wartości te przechowują czas rozpoczęcia i zakończenia
    ➥działania wątku
    Total : Currency;  // wartość całkowitego czasu przeszukania
    FFileName : String; // nazwa pliku do odnalezienia
    FDrive : Char; // dysk, na którym odbędzie się szukanie
    procedure MyOnTerminate(Sender: TObject); // obsługa zdarzenia OnTerminate
  public
    constructor Create(const FileName : String; Drive : Char); // konstruktor dla klasy
    destructor Destroy; override; // destruktor dla klasy
    procedure SearchInDrive;  // procedura poszukiwawcza
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;
  SearchThread : TSearchThread;

implementation

{$R *.dfm}

constructor TSearchThread.Create(Const FileName : String; Drive : Char);
begin
  inherited Create(False); // wywołanie konstruktora klasy bazowej
  FreeOnTerminate := True;  // zwolnij przy zakończeniu
  OnTerminate := MyOnTerminate;  // przypisz procedurę zdarzenia
  FFileName := FileName;  // nazwa pliku do znalezienia
  FDrive := Drive; // dysk
  Start := GetTickCount;  // pobierz czas startu (w milisekundach)
end;

destructor TSearchThread.Destroy;
begin
  Stop := GetTickCount;  // pobierz czas zakończenia
  Total := Stop - Start;   // odejmij czas startu od czasu zakończenia
  Total := Total / 1000; // podziel przez 1000, aby uzyskać liczbę sekund

  { wyświetl na komponencie czas wyszukiwania na danym dysku }
  MainForm.lbEnd.Items.Add(FDrive + ':\ - ' + CurrToStr(Total) + ' sek.');
  inherited;
end;

procedure TSearchThread.SearchInDrive;
```

```
procedure Search(StartDir : String);
var
  SR, DR  : TSearchRec;
  Found, FoundFile : Integer;

  { ta procedura sprawdza, czy na końcu zmiennej znajduje się znak \ - jeżeli
    tak, nic nie jest wykonywane; jeżeli tego znaku brak, zostaje on dodany... }
  function IsDir(Value : String) : String;
  begin
    if Value[Length(Value)] <> '\' then  // jeżeli na końcu znajdziesz znak
      Result := Value + '\' else Result := Value; // dodaj go... w przeciwnym wypadku nie
      ➥wykonuj nic
  end;

begin
  Found := FindFirst(IsDir(StartDir) + '*.*', faDirectory, DR); // następuje pobieranie
  ➥katalogów z podanej lokalizacji
  while Found = 0 do // pętelka
  begin
    if ((DR.Attr and faDirectory) = faDirectory) and  // sprawdza, czy pozycja jest katalogiem
       ((DR.Name <> '.') and (DR.Name <> '..')) then
    begin
      MainForm.StatusBar.SimpleText := IsDir(StartDir) + DR.Name + '\*.*'; // na
      ➥komponencie wyświetl aktualnie przeszukiwany katalog
      if Pos(FFileName, DR.Name) > 0 then // sprawdź, czy w nazwie jest szukany ciąg znaków
        MainForm.lbResults.Items.Add(IsDir(StartDir) + DR.Name);

    { pobierz na razie wszystkie pliki z danego katalogu - potem je przeanalizujemy }
      FoundFile := FindFirst(IsDir(StartDir) + DR.Name + '\*.*', faAnyFile, SR);
      while FoundFile = 0 do
      begin
        if ((SR.Name <> '.') and (SR.Name <> '..')) then //
          if Pos(FFileName, SR.Name) > 0 then  // następuje sprawdzenie, czy plik nie
          ➥zawiera części szukanego ciągu
            MainForm.lbResults.Items.Add(IsDir(StartDir) + DR.Name + '\' + SR.Name);

        FoundFile := FindNext(SR); // kontynuuj przeszukiwanie
      end;
      FindClose(SR);  // zakończ

      Search(IsDir(StartDir) + DR.Name); // tutaj następuje rekurencja
    end;
    Found := FindNext(DR); // kontynuuj
  end;
  FindClose(DR);
end;

begin
  Search(FDrive + ':\');  // rozpocznij wyszukiwanie na danym dysku
end;

procedure TSearchThread.Execute;
begin
  (SearchInDrive); // wywołaj procedurę...
end;
```

```
procedure TSearchThread.MyOnTerminate(Sender: TObject);
begin
{ podczas kończenia wyszukiwania wyświetl na komponencie ilość odnalezionych pozycji }
  MainForm.ProgressBar.Position := MainForm.ProgressBar.Position + 1;
  MainForm.StatusBar.SimpleText := 'Znaleziono: ' +
IntToStr(MainForm.lbResults.Items.Count) + ' plików...';
end;


procedure TMainForm.btnFindClick(Sender: TObject);
var
  i : Integer;
  DriveType : Integer;
begin
  lbResults.Clear; // wyczyść komponent
  lbEnd.Clear; // wyczyść komponent
  ProgressBar.Max := 0; // ustaw wartość maksymalną na 0
  ProgressBar.Position := 0; // pozycja na 0
  for I := Ord('A') to Ord('Z') do // pętla po wszystkich dyskach
  begin
    DriveType := GetDriveType(PChar(chr(i) + ':\')); // pobierz informacje o dysku

    if not (DriveType = 0) and not (DriveType = 1) then // jeżeli dysk istnieje
    begin
      ProgressBar.Max := ProgressBar.Max + 1; // zwiększ właściwość maks. o jeden
      SearchThread := TSearchThread.Create(leFileName.Text, Chr(i)); // wywołaj wątek
      ➥z literą dysku jako początkowy parametr
    end;
  end;
end;

end.
```

Podsumowanie

W niniejszym rozdziale przedstawiłem Ci zasadę działania wątków. Myślę, że po dokładniejszym zapoznaniu się z tym zagadnieniem nie wygląda ona tak strasznie, tym bardziej, że nie jesteśmy zmuszeni do korzystania z funkcji WinAPI, ale używamy wygodnej klasy VCL. Na pewno nieraz będziesz w swojej aplikacji wykorzystywał wątki…