尾崎 浩司

株式会社ミガロ.

RAD事業部 営業・営業推進課

[Delphi/400] マルチスレッドを使用したレスポンスタイム向上

略歴
1973 年 8 月 16 日生まれ
1996 年 三重大学工学部卒業
1999 年 10 月 株式会社ミガロ. 入社
1999 年 10 月 システム事業部配属
2013 年 4 月 RAD 事業部配属

現在の仕事内容
ミガロ. 製品の素晴らしさをアピールするためのセミナーやイベントの企画・運営などを主に担当している。

1.はじめに

アプリケーション開発において一般的に重要なのは、仕様通りの動作ができること、画面の使い勝手がよいことなどが挙げられるが、もう一つ重要な要素は、処理レスポンスである。せっかくの便利なアプリケーションであっても、処理レスポンスが悪いとユーザーはなかなか利用してくれない。しかし、大量データの処理や複雑な業務ロジックの実行は、一般的に時間がかかる場合が多い。

本稿では、複雑で処理時間がかかる処理をいかにユーザーが快適に使えるものにするかについて、技術的な解決手法を紹介する。

2.スレッドについて

2-1. スレッドとは

アプリケーションの処理を考える上で、「スレッド」と「プロセス」という概念は非常に重要である。Windows アプリケーションは、通常「プロセス」という単位で処理が行われる。実行中のプロセスは、Windows のタスクマネージャーでも確認することができる。【図 1】

プロセスとは、アプリケーションの実行単位である。つまり、プロセスは、それぞれ固有のメモリ空間をもって実行される独立したアプリケーションとして扱われる。

もう一つの概念に「スレッド」がある。スレッドとは、プロセスの中で、1 つ、あるいは複数動作するプログラムの一連の流れである。プロセスとは違い、スレッドは、1 つのアプリケーション内で、同じメモリ空間を共有して動作する。【図 2】

2-2. シングルスレッドとマルチスレッド

プログラムは、「順次処理」「分岐処理」「繰り返し処理」の組み合わせで構成されており、通常のアプリケーションでは、これらが一つずつ順番に処理されるのが一般的である。このようなアプリケーションの処理を「シングルスレッド」という。

これに対し、複数の処理を並行して行うアプリケーションも作成できる。このようなアプリケーションの処理を「マルチスレッド」といい、プログラムのコードが同時に複数個実行される。【図 3】

アプリケーションをすべてマルチスレッドにすれば、アプリケーションの処理速度が速くなるように思えるかもしれない。しかし実際は、そうならない。なぜならば、CPU は通常、1 度に 1 つの処理しか実行できないからである。マルチスレッドアプリケーションは、確かに複数の処理を同時に実行しているように見えるが、それは CPU が複数の処理を高速に切り替えて実行しているだけである。【図 4】

つまり、複数処理をマルチスレッドにしても全体の処理時間は変わらない。むしろ、CPU を切り替える時間分だけオーバーヘッドがかかるため、遅くなる場合もある。

（ただし、現在のコンピュータで使用される CPU は、マルチコアが主流のた

図1

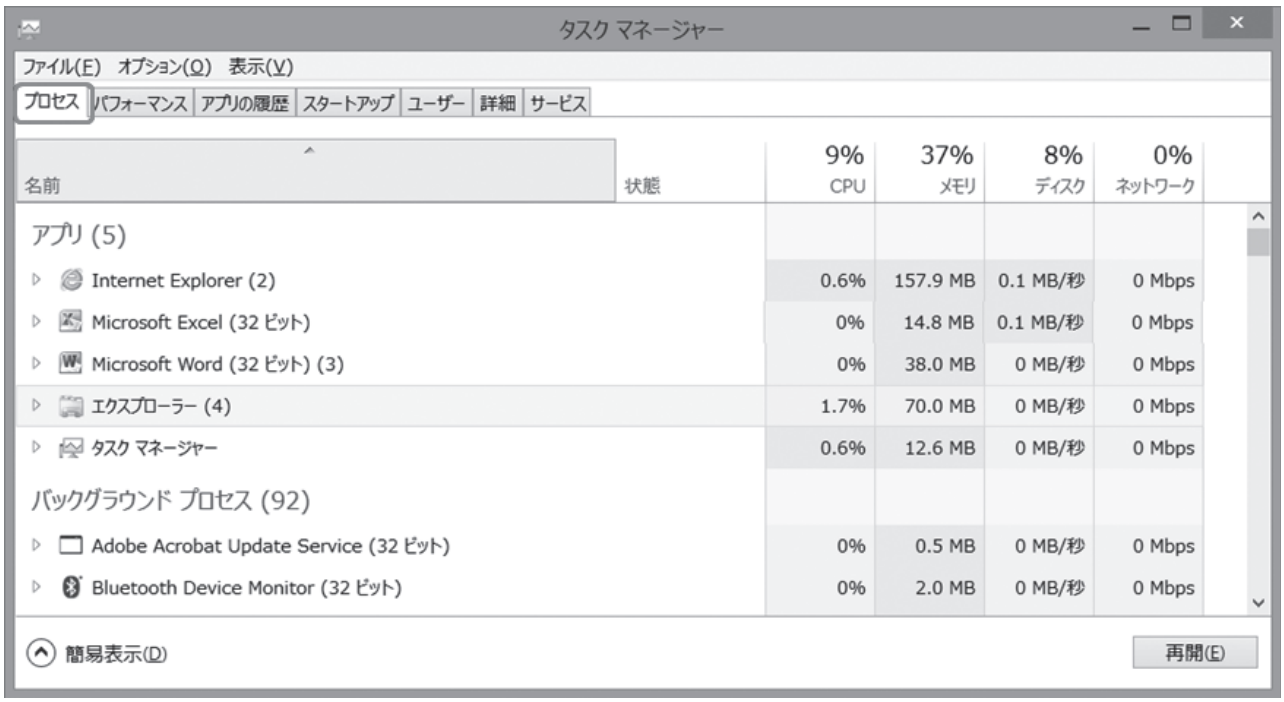
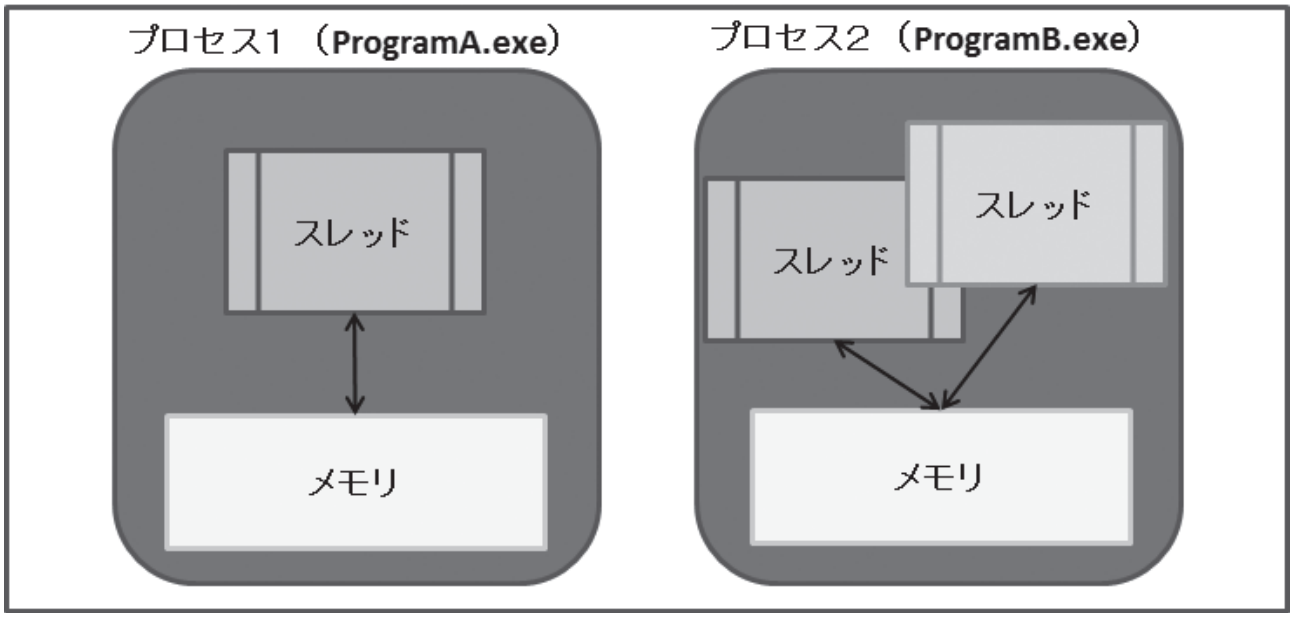

タスク マネージャー
ファイル(F) オプション(O) 表示(V)
プロセス パフォーマンス アプリの履歴 スタートアップ ユーザー 詳細 サービス
名前
状態
9% CPU
37% メモリ
8% ディスク
0% ネットワーク
アプリ (5)
Internet Explorer (2) 0.6% 157.9 MB 0.1 MB/秒 0 Mbps
Microsoft Excel (32 ビット) 0% 14.8 MB 0.1 MB/秒 0 Mbps
Microsoft Word (32 ビット) (3) 0% 38.0 MB 0 MB/秒 0 Mbps
エクスプローラー (4) 1.7% 70.0 MB 0 MB/秒 0 Mbps
タスク マネージャー 0.6% 12.6 MB 0 MB/秒 0 Mbps
バックグラウンド プロセス (92)
Adobe Acrobat Update Service (32 ビット) 0% 0.5 MB 0 MB/秒 0 Mbps
Bluetooth Device Monitor (32 ビット) 0% 2.0 MB 0 MB/秒 0 Mbps
簡易表示(D)
再開(E)

図2


プロセス1 （ProgramA.exe）
スレッド
メモリ
プロセス2 （ProgramB.exe）
スレッド
スレッド
メモリ

め、マルチスレッド化により、CPU の処理が分散される効果は期待できる。）

2-3. マルチスレッドの利点

マルチスレッドの利点について考察してみる。一番大きな利点としては、「重い処理」を実行した時の「レスポンスタイム（応答時間）」の改善である。「レスポンスタイム」とは、処理を実行してから最初の反応が返ってくるまでの時間のことだ。

ここで、シングルスレッドで重い処理を実行するサンプルプログラムを考えてみたい。このプログラムでは、フォーム上に、ビジュアルコンポーネントとして、TButton、TMemo、TStringGrid を配置し、データベースにアクセスするために TSQLConnection、TSQLQuery を配置している。【図 5】

また、「データ取得」ボタン（btnGetData）のクリックイベントは、SQLQuery1 からデータを全件取得し、StringGrid1 に内容を書き出す処理である。【ソース 1】

このプログラムで数万件以上の抽出データを用意して、実行した場合、しばらくの間、画面がすべて固まってしまう（Memo1 に値を入力することもできない）。そして、全件抽出処理が終了して初めて、画面に応答がある。このように処理に時間がかかると、シングルスレッドのアプリケーションは、画面の応答が止まってしまうのである。

次に、このような現象を回避し、レスポンスタイムを改善する方法として、スレッドの活用を考えてみる。考え方としては、重い処理の部分を別のスレッド（サブスレッド）として実行できるようにすればよい。メインスレッドは、サブスレッドが開始したら、そのまま処理を終了する。【図 6】

この方法によって、メインスレッドで処理を実行した後、重い処理はサブスレッドで処理されるため、アプリケーションの画面は応答が止まることなく使用できる。

このようにマルチスレッドの利点は、重い処理で処理時間がかかる時に、レスポンスタイムを格段に向上させられることである。

次節から、具体的な開発方法を紹介する。

3.TThread（スレッドクラス）について

3-1. TThread クラス作成方法

Delphi/400 では、マルチスレッド処理を簡単に実装するために TThread クラスを用意している。プロジェクトに TThread クラスを追加する手順は、次の通りである。

プロジェクトファイルを開いている状態で、［ファイル］→［新規作成］→［その他］を選択し、表示される新規作成ダイアログで［Delphi ファイル］→［スレッドオブジェクト］を選択する。そして、スレッドオブジェクトの新規作成ダイアログで、これから作成するスレッドクラス名を入力して［OK］ボタンを押下する。【図 7】

これで、新しいスレッドクラスを持つユニットが新規に作成できる。【ソース 2】

生成されたスレッドクラスには、Execute メソッドが定義されているので、このメソッドの実装部に、スレッドとして実行したい処理を記述すればよい。たとえば、先ほどのシングルスレッドで記述した【ソース 1】の処理をスレッドクラスに移行すると、【ソース 3】のような実装になる。

3-2. メインスレッドからの呼出し方法

スレッドクラスを作成したら、このスレッドをメインスレッドから呼び出す必要がある。呼び出し方は簡単で単純にスレッドオブジェクトを生成するだけだ。シングルスレッドで記述した【ソース 1】のボタンクリックイベントを、先ほどのスレッドを生成するロジックに変更すればよい。【ソース 4】

変更が完了したら、アプリケーションを実行して確認する。シングルスレッドの場合と異なり、「データ取得」ボタンを押下後、すぐに画面応答ができることがわかる。

（Memo1 に即座に値を入力できる。）

このようにスレッドクラスを用意してメインスレッドからスレッドオブジェクトを生成するだけで、マルチスレッドプログラムが開発できる。

なお、【ソース 3】で作成したプログラムは、スレッドクラス（TGetDataThread）の中で、直接 Form1 を参照していることがわかる。このままでは、同じスレッド処理を別のフォームからも使用したいとなった時に具合が悪い。どうすれば汎用的になるかというと、画面操作に必要な VCL コンポーネントを、スレッドクラスのコンストラクターで受け渡しできるようにすればよい。VCL コンポーネントの受け渡しを加えたソースを【ソース 5】に示す。

【ソース 5】では、メインフォームで使用していた SQLQuery1、StringGrid1 を受け渡しできるように、コンストラクターに 2 つの引数を追加している。受け取った引数をスレッドクラスのプライベート変数に代入し、スレッド内部ではその変数を使って処理を行うようにしている。こうすることで、スレッドクラスはフォームの依存がなくなるため、より独立性の高いプログラムにすることができる。なお、コンストラクターの実装部で、inherited によって、TThread の Create メソッドを呼び出しているが、この時の引数 False は、スレッドが生成後ただちに実行されることを表している。また、FreeOnTerminate プロパティを True に指定しているが、これはスレッド処理が終了した時に、スレッドオブジェクトが自動的に破棄されるようにする設定である。

メインスレッドの呼出し側は、スレッドに渡したい VCL コンポーネントを指定する。【ソース 6】

これで再度アプリケーションを実行すると、先ほどと同じ動作となることが確認できる。スレッドクラスにおいて、フォームの依存性をなくしたため、たとえば別のフォームで異なる SQLQuery を使用した画面においても、同じスレッドクラスが使用できる。

4.スレッド使用時の留意点

4-1. マルチスレッドアプリケーションの留意点

前節で作成したプログラムは、マルチスレッドアプリケーションであるが、実は 2 つの留意点がある。

アプリケーションを実行させ、「データ取得」ボタンを押下し、サブスレッドが動いている間に、アプリケーションを［×］ボタンで終了すると、実行時エラー

図3

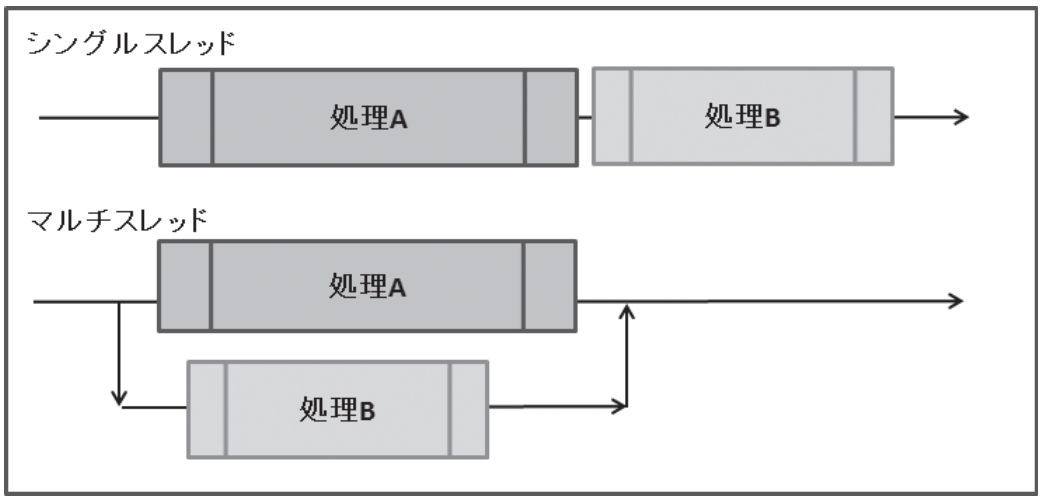

シングルスレッド
処理A
処理B
マルチスレッド
処理A
処理B

図4

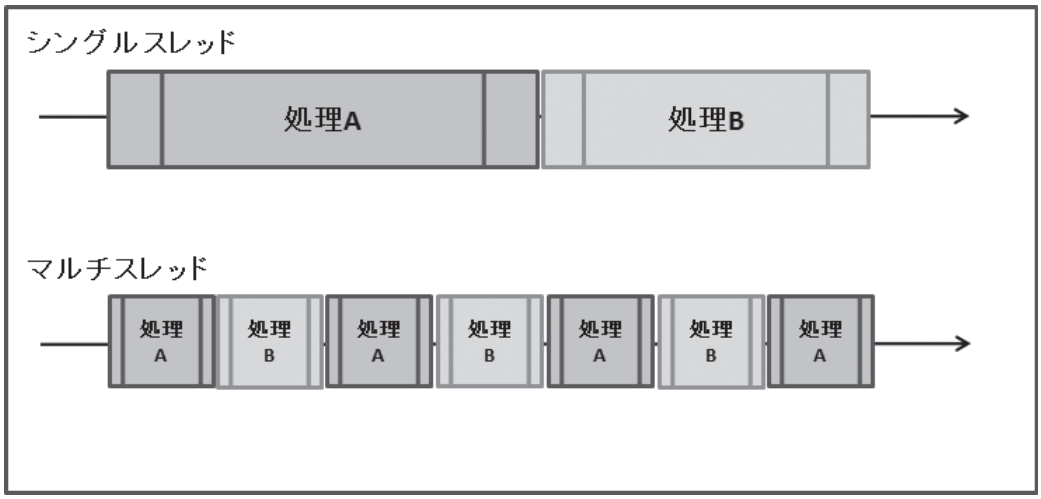
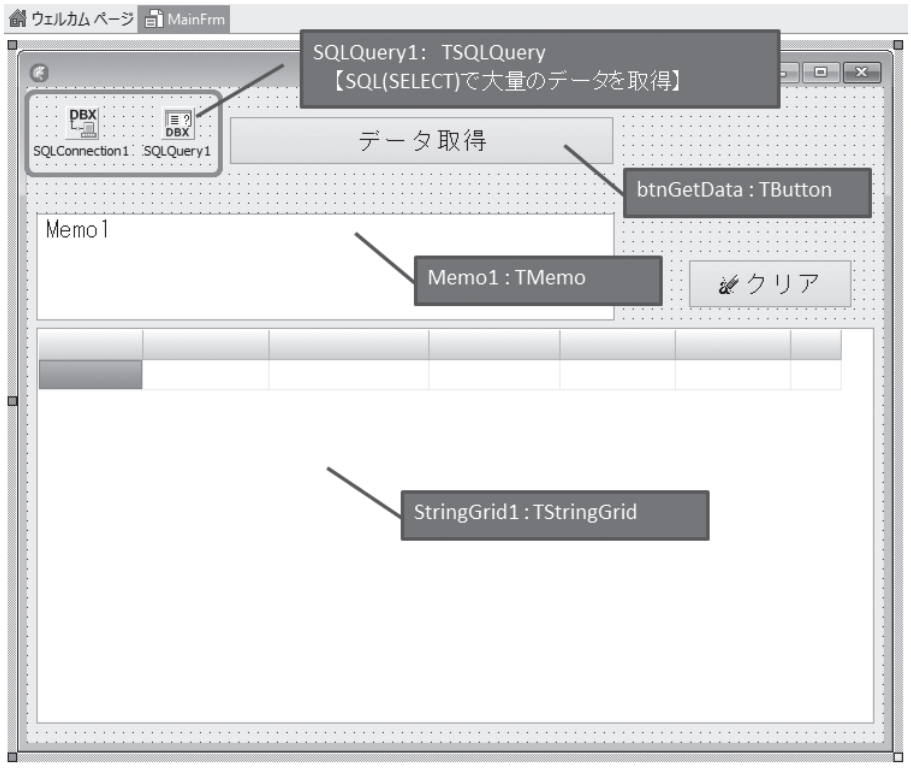

シングルスレッド
処理A
処理B
マルチスレッド
処理 A
処理 B
処理 A
処理 B
処理 A
処理 B
処理 A

図5


ウェルカムページ
MainFrm
SQLQuery1: TSQLQuery
【SQL(SELECT)で大量のデータを取得】
SQLConnection1
SQLQuery1
データ取得
btnGetData : TButton
Memo1
Memo1 : TMemo
クリア
StringGrid1 : TStringGrid

が発生してしまう。【図8】

なぜエラーが起こるかというと、問題はサブスレッドの処理にある。Delphi/400アプリケーションでは、VCLコンポーネント（ビジュアルコンポーネント）をサブスレッドの中で直接操作することができない。つまり、VCLコンポーネントは必ずメインスレッド側で操作する必要がある。サブスレッド側でVCLコンポーネントを使用したい場合には、いったんメインスレッドを一時停止させ、サブスレッドのVCLコンポーネント操作をメインスレッド側に割り込ませる必要がある。これが、マルチスレッドアプリケーションを構築する際の1つ目の留意点である。【図9】

2つ目の留意点は、スレッド内の「繰り返し処理」実行時に、いつでも処理が中断できるようにスレッドの終了確認を行う必要があるということだ。つまり、繰り返し処理の中で、適宜スレッドの終了通知が出ているかどうかを確認し、スレッド外部から終了通知が出されたら、いつでもスレッド処理を終了できるような処理にする必要がある。

次節では、これらの留意点に関する具体的な解決手順を紹介したい。

4-2. Synchronizeの使用方法

1つ目の留意点であるVCLコンポーネントの操作だが、これは、Synchronizeメソッドの使用により対応可能である。Synchronizeメソッドとは、サブスレッド処理側から、メインスレッド側に制御を移し、VCLコンポーネントの操作等を行う特定の手続きを実行させるものである。具体的には、VCLコンポーネントの操作を行う手続き（procedure）をスレッドクラスに作成し、Executeメソッドの中で、Synchronizeメソッド経由して作成した手続きを呼び出せばよい。

先ほどの【ソース5】を改良し、Executeメソッドの中にSynchronizeメソッドを追加したものを【ソース7】に示す。

Synchronizeメソッド経由でVCLコンポーネントを操作する手続きを呼び出すと、その手続きが実行されている間、元のメインスレッド側処理は待機状態になる。

この仕組みでアプリケーションを実行すると、先ほどと同じように「データ取得」ボタンを押下し、サブスレッドが動いている間に、アプリケーションを［×］ボタンで終了しても、正しくアプリケーションを終了できる。

このようにSynchronizeメソッドを使用することで、サブスレッド側からVCLコンポーネントを操作できるが、このVCL操作の手続きで時間がかかってはいけないので注意が必要である。その間メインスレッドの停止状態が続くため、【ソース8】に示すようなプログラムは作成してはならない。

4-3. スレッド中断方法

2つ目の留意点であるスレッドの終了確認だが、これは、「繰り返し処理」における条件において、Terminatedプロパティをチェックすればよい。これを入れることにより、処理を中断させたい時に安全にスレッドを終了できる。Executeメソッドの中にTerminatedプロパティのチェックを入れたものを【ソース9】に示す。

このようなスレッド処理にしておくと、重い処理の実行時に、途中で中断するようなことも行えるようになる。たとえば、今回のサンプルプログラムに処理の中断機能を追加してみる。まずフォーム上に「中止」ボタン（btnAbort）を配置する。【図10】

次にスレッドオブジェクトを扱う変数を宣言部に追加し、「データ取得」ボタンのクリックイベントにて、スレッド生成時に変数に代入するように変更する。そうしておくことで、別のイベントでスレッドオブジェクトの操作が可能になる。

「中止」ボタンのクリックイベントでは、スレッドオブジェクト変数に対し、Terminateメソッドを実行するだけでよい。改良したプログラムを【ソース10】に示す。

このプログラムを実行すると、「データ取得」ボタン押下し、サブスレッドが動いている間に、「中止」ボタンを押下すると、すぐにスレッドが中断されることがわかる。

このようにマルチスレッドアプリケーションとして実装すると、中断処理も容易に実装できるので、時間がかかる処理を開発する際には非常に有効である。

5.CreateAnonymousThreadを使用したスレッド

5-1. シンプルなスレッドの利用

前節までが、TThreadクラスの使用方法である。Delphi/400では、このTThreadクラスを使用することでマルチスレッドアプリケーションが作成できるが、スレッド実行したい処理ごとにスレッドクラスを作成しなければならないため、実装に手間がかかる。汎用的なスレッドクラスであればこの形がよいが、たとえばある画面の一部だけスレッドを使用したい場合にも、都度スレッドクラスを生成するのは少々面倒である。

実は、Delphi/400 XE以降であれば、もっとシンプルにマルチスレッドアプリケーションを作成できる。

Delphi/400 XE以降では、CreateAnonymousThreadメソッドが用意されているため、このメソッドを使用すると、メインスレッドの中に直接、サブスレッドを無名メソッドとして記述できる。【図11】

本稿の初めに作成したシングルスレッドアプリケーションの【ソース1】を基にCreateAnonymousThreadメソッドを使用するように改良したプログラムが【ソース11】である。

【ソース1】と【ソース11】を比べると、CreateAnonymousThreadで処理を括っている部分以外、ほとんど変わらない。

このようにマルチスレッドアプリケーションをシングルスレッドアプリケーション同様に一つの手続きに集約できるため、単純なプログラムとして記述することができる。

ただし、このプログラムも考慮点はある。サブスレッド中で直接VCLコンポーネントを操作しているため、TThreadクラスの場合と同様、VCLコンポーネントはメインスレッドで操作しなければならない。

5-2. Synchronizeの使用方法

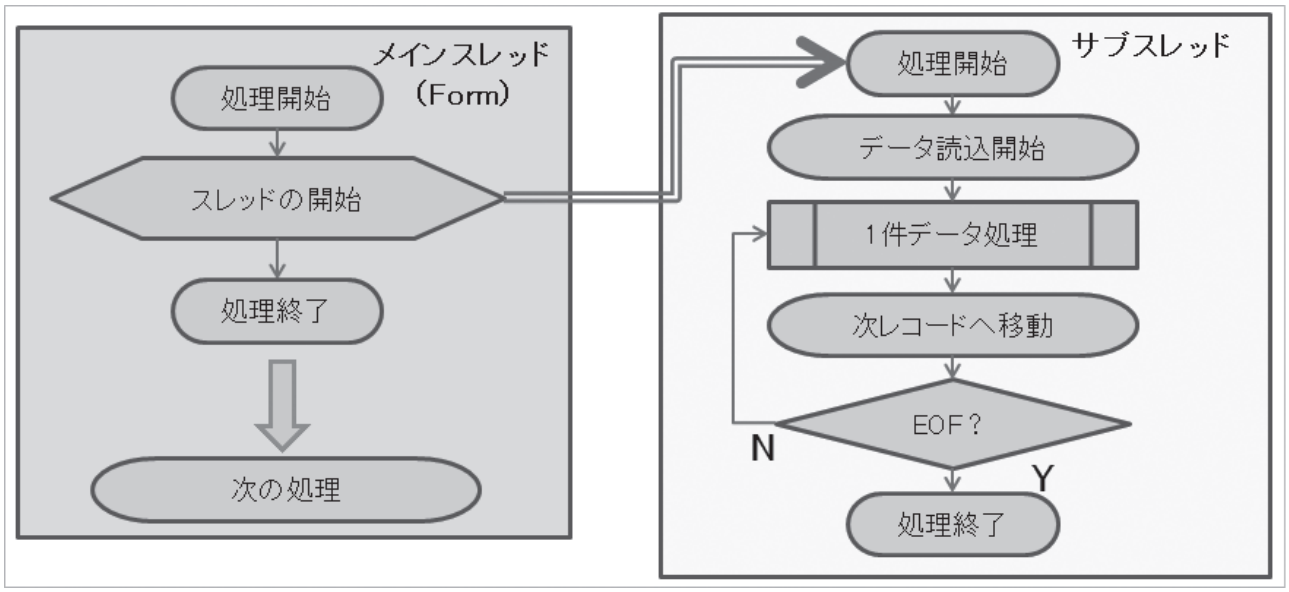
実は、Synchronizeメソッドも無名メソッドを使用することで、よりシンプルに記述できる。【図12】

ソース1

```
procedure TForm1.btnGetDataClick(Sender: TObject);
var
  i, iRow: Integer;
begin
  //初期化
  iRow := 0;

  SQLQuery1.Active := True;
  try
    //繰り返し
    while (not SQLQuery1.Eof) do
    begin
      Inc(iRow);   //カウントアップ
      StringGrid1.RowCount := iRow + 1;
      //グリッドにデータを書き出す
      for i := 0 to SQLQuery1.FieldCount - 1 do
        StringGrid1.Cells[i, iRow] := SQLQuery1.Fields[i].Text;
      SQLQuery1.Next;
    end;
  finally
    SQLQuery1.Active := False;
  end;
end;
```

図6



この方法を使用すると、安全なマルチスレッド処理を一つのサブルーチンにまとめられる。【ソース 11】を改良したプログラムを【ソース 12】に示す。

このように安全なマルチスレッドアプリケーションを一つのイベントの中にまとめて書けるため、TThread クラスを別途作成しなくとも、簡単にアプリケーションのマルチスレッド化が可能となる。

開発時の使い分けとしては、汎用的なスレッド処理や、実行中の中断処理などを含むスレッドは、TThread クラスを使用し、特定の場面だけで使用するスレッドは、CreateAnonymousThread メソッドで対応することが望ましい。

6.まとめ

本稿では、マルチスレッドを使用してレスポンスタイムを向上させる手法として、TThread クラスおよび CreateAnonymousThread メソッドについて説明してきた。

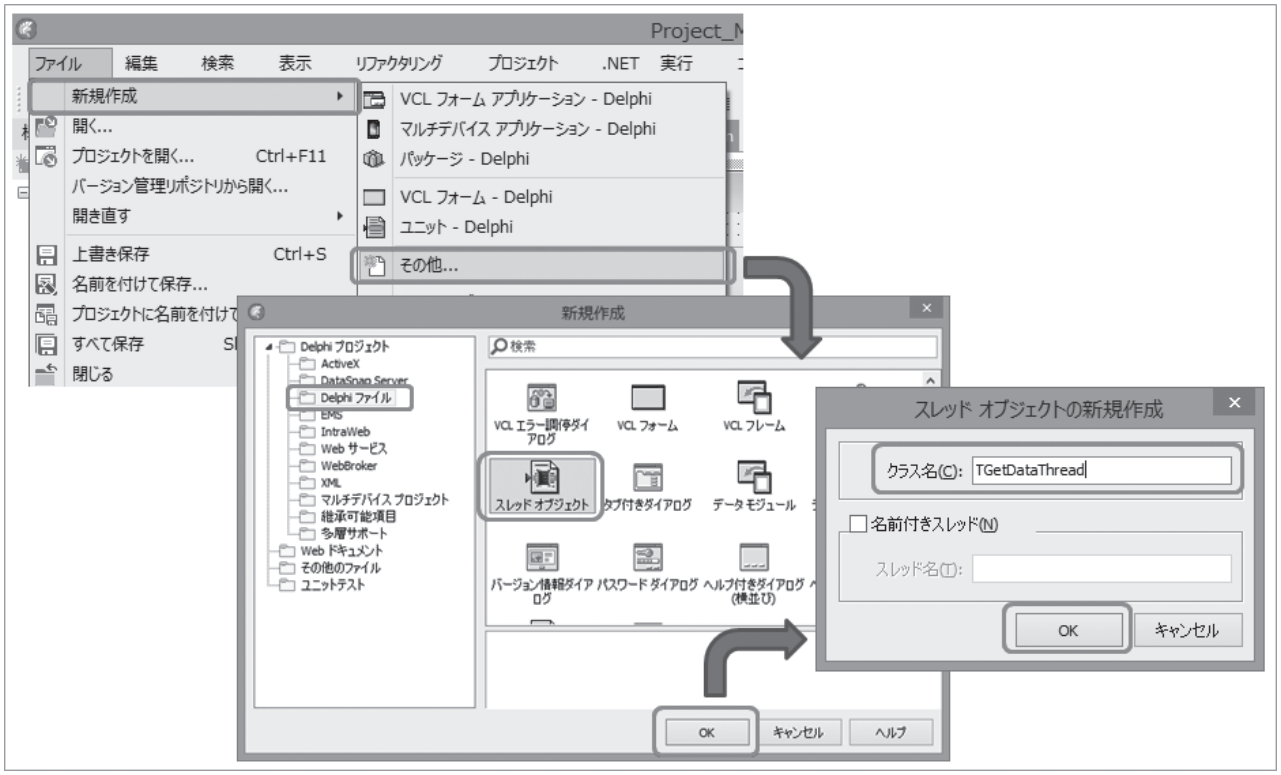
冒頭でも述べた通り、スレッドを使用しても、全体の処理時間が短縮するわけではないが、応答時間が早いと、ユーザーはアプリケーションを快適に使用できる。

またスレッド処理を開発する場合、メインスレッドとサブスレッド間で VCL コンポーネント操作の競合や、Synchronize メソッド内の処理時間など、考慮点がいくつかある。

しかし、ユーザーにとって使いやすいアプリケーションを実現するという意味で、マルチスレッド開発は非常に有用であり、手間をかける価値が十分にある技術と言える。

M

図7



ソース2

```
unit Unit1;

interface

uses
  System.Classes;

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

implementation


{ TGetDataThread }

procedure TGetDataThread.Execute;
begin
  { スレッドとして実行したいコードをここに記述してください }
end;

end.
```

ソース3

```
unit ThreadUnit;

interface

uses
  System.Classes;

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

implementation

uses MainFrm;  //---- メインフォームを参照

{ TGetDataThread }

procedure TGetDataThread.Execute;
var
  i, iRow: Integer;
begin
  //初期化
  iRow := 0;

  //メインフォームを使用
  with Form1 do
  begin
    SQLQuery1.Active := True;
    try
      //繰り返し
      while (not SQLQuery1.Eof) do
      begin
        Inc(iRow);   //カウントアップ
        StringGrid1.RowCount := iRow + 1;
        //グリッドにデータを書き出す
        for i := 0 to SQLQuery1.FieldCount - 1 do
          StringGrid1.Cells[i, iRow] := SQLQuery1.Fields[i].Text;
        SQLQuery1.Next;
      end;
    finally
      SQLQuery1.Active := False;
    end;
  end;
end;

end.
```

フォーム側のSQLQuery1
StringGrid1を操作

ソース4

```
implementation

{$R *.dfm}

uses ThreadUnit;  //スレッドユニットを追加


procedure TForm1.btnGetDataClick(Sender: TObject);
begin
  //スレッドの生成
  TGetDataThread.Create;
end;
```

ソース5

```
unit ThreadUnit;

interface

uses
  System.Classes, vcl.Grids, Data.SqlExpr;

type
  TGetDataThread = class(TThread)
  private
    { Private 宣言 }
    FStringGrid: TStringGrid;
    FQuery: TSQLQuery;
  protected
    procedure Execute; override;
  public
    constructor Create(AStringGrid: TStringGrid; AQuery: TSQLQuery); virtual;
  end;

implementation

{ TGetDataThread }

constructor TGetDataThread.Create(AStringGrid: TStringGrid; AQuery: TSQLQuery);
begin
  FStringGrid := AStringGrid;
  FQuery := AQuery;

  inherited Create(False);      //False指定でスレッド生成時、即実行
  FreeOnTerminate := True;      //スレッド終了時にオブジェクト破棄
end;

procedure TGetDataThread.Execute;
var
  i, iRow: Integer;
begin
  //初期化
  iRow := 0;

  FQuery.Active := True;
  try
    //繰り返し
    while (not FQuery.Eof) do
    begin
      Inc(iRow);   //カウントアップ
      FStringGrid.RowCount := iRow + 1;
      //グリッドにデータを書き出す
      for i := 0 to FQuery.FieldCount - 1 do
        FStringGrid.Cells[i, iRow] := FQuery.Fields[i].Text;
      FQuery.Next;
    end;
  finally
    FQuery.Active := False;
  end;
end;

end.
```

TStringGrid、TSQLQueryを使用する為にユニットを追加

スレッド側で使用するVCLコンポーネントを変数として定義

メインフォーム側から、VCLコンポーネントがセットできるよう、コンストラクターの引数を追加

メインフォームに依存しないロジックに修正

ソース6

```
procedure TForm1.btnGetDataClick(Sender: TObject);
begin
  //スレッド生成
  TGetDataThread.Create(StringGrid1, SQLQuery1);
end;
```

図8

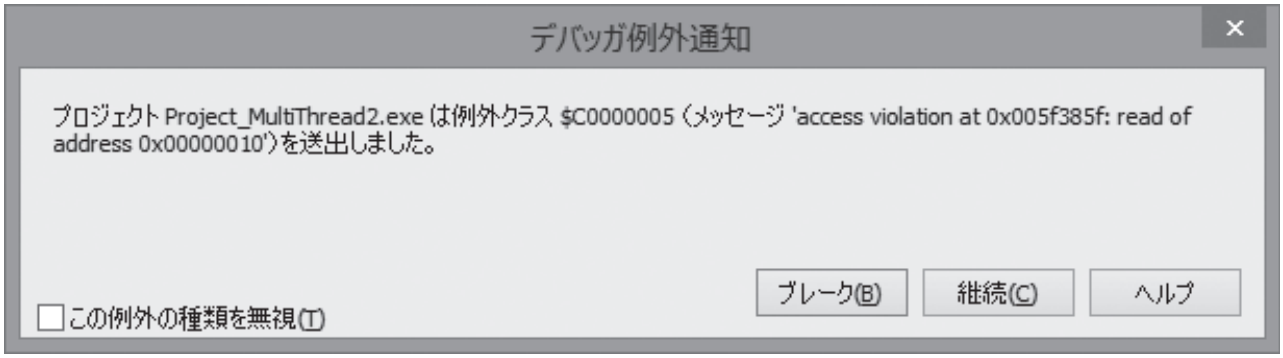
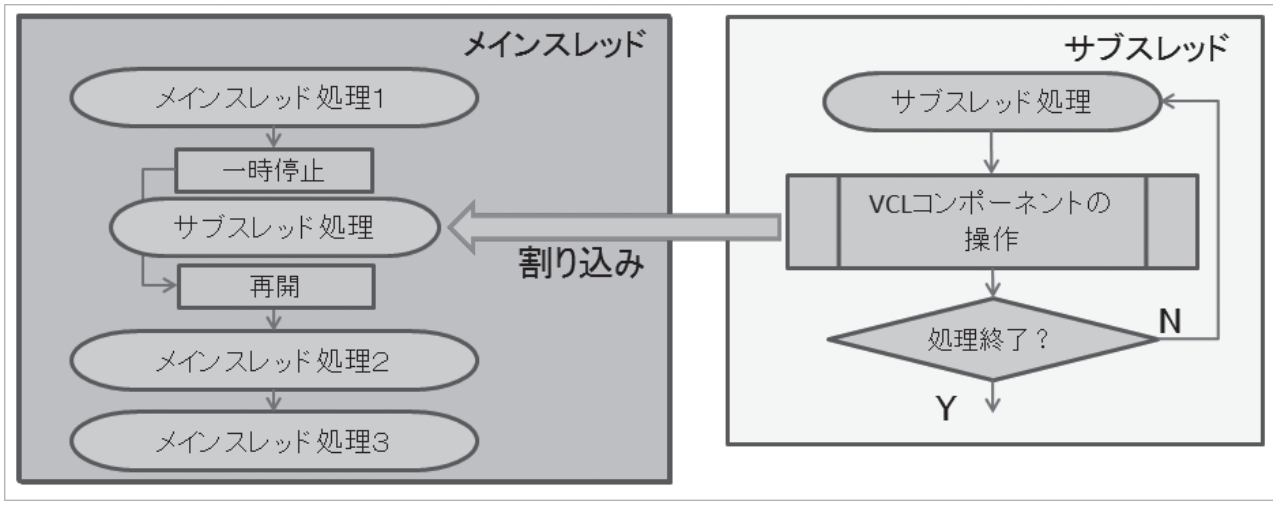


図9



ソース7

```
unit ThreadUnit;

interface

uses
  System.Classes, vcl.Grids, Data.SqlExpr;

type
  TGetDataThread = class(TThread)
  private
    { Private 宣言 }
    FStringGrid: TStringGrid;
    FQuery: TSQLQuery;
    FRow: Integer;          //現在処理行を保持するグローバル変数
  protected
    procedure Execute; override;
    procedure VCLDraw;              //VCL操作を行う手続き
  public
    constructor Create(AStringGrid: TStringGrid; AQuery: TSQLQuery); virtual;
  end;

implementation

{ TGetDataThread }

procedure TGetDataThread.Execute;
begin
  //初期化
  FRow := 0;

  FQuery.Active := True;
  try
    //繰り返し
    while (not FQuery.Eof) do
    begin
      Inc(FRow);   //カウントアップ

      Synchronize(VCLDraw); //メインスレッドを待機させて処理実行

      FQuery.Next;
    end;
  finally
    FQuery.Active := False;
  end;
end;

procedure TGetDataThread.VCLDraw;
var
  i: Integer;
begin
  FStringGrid.RowCount := FRow + 1;
  //グリッドにデータを書き出す
  for i := 0 to FQuery.FieldCount - 1 do
    FStringGrid.Cells[i, FRow] := FQuery.Fields[i].Text;
end;

end.
```

Executeメソッドと、VCLDrawメソッドの両方で行番号変数を使用する為、グローバル変数として定義

VCLコンポーネントを操作する処理を手続きとして宣言

Synchronizeメソッドを経由してVCLDraw手続きを実行

VCLDraw手続き実行中、メインスレッドは一時停止となる。

VCLコンポーネント(StringGrid)を操作する処理を記述

ソース8

```
procedure TGetDataThread.Execute;
begin
  //初期化
  FRow := 0;
  Synchronize(VCLDraw); //メインスレッドを待機させて処理実行

end;

procedure TGetDataThread.VCLDraw;
var
  i: Integer;
begin
  FQuery.Active := True;
  try
    //繰り返し
    while (not FQuery.Eof) do
    begin
      Inc(FRow);   //カウントアップ
      FStringGrid.RowCount := FRow + 1;
      //グリッドにデータを書き出す
      for i := 0 to FQuery.FieldCount - 1 do
        FStringGrid.Cells[i, FRow] := FQuery.Fields[i].Text;

      FQuery.Next;
    end;
  finally
    FQuery.Active := False;
  end;
end;

end.
```

繰り返しが行われる為、メインスレッドが待機状態のままとなってしまう。

ソース9

```
procedure TGetDataThread.Execute;
begin
  //初期化
  FRow := 0;

  FQuery.Active := True;
  try
    //繰り返し
    while (not FQuery.Eof) and (not Terminated) do
    begin
      Inc(FRow);   //カウントアップ

      Synchronize(VCLDraw); //メインスレッドを待機させて処理実行

      FQuery.Next;
    end;
  finally
    FQuery.Active := False;
  end;
end;
```

終了通知(Terminated)がFalseの場合のみ、処理を継続する

図10

ソース10

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, DBXDynalink, FMTBcd, DB,
  SqlExpr, StdCtrls, Buttons, Grids, Mask, DBClient, ThreadUnit;

type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    btnGetData: TBitBtn;
    SQLConnection1: TSQLConnection;
    SQLQuery1: TSQLQuery;
    btnClear: TBitBtn;
    Memo1: TMemo;
    btnAbort: TBitBtn;
    procedure btnGetDataClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure btnAbortClick(Sender: TObject);
  private
    { Private 宣言 }
    GetDataThread: TGetDataThread; // スレッドオブジェクト変数
  public
    { Public 宣言 }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.btnGetDataClick(Sender: TObject);
begin
  //スレッドの生成
  GetDataThread := TGetDataThread.Create(StringGrid1, SQLQuery1);
end;

procedure TForm1.btnAbortClick(Sender: TObject);
begin
  if Assigned(GetDataThread) then
    GetDataThread.Terminate; //スレッド中断
end;
```

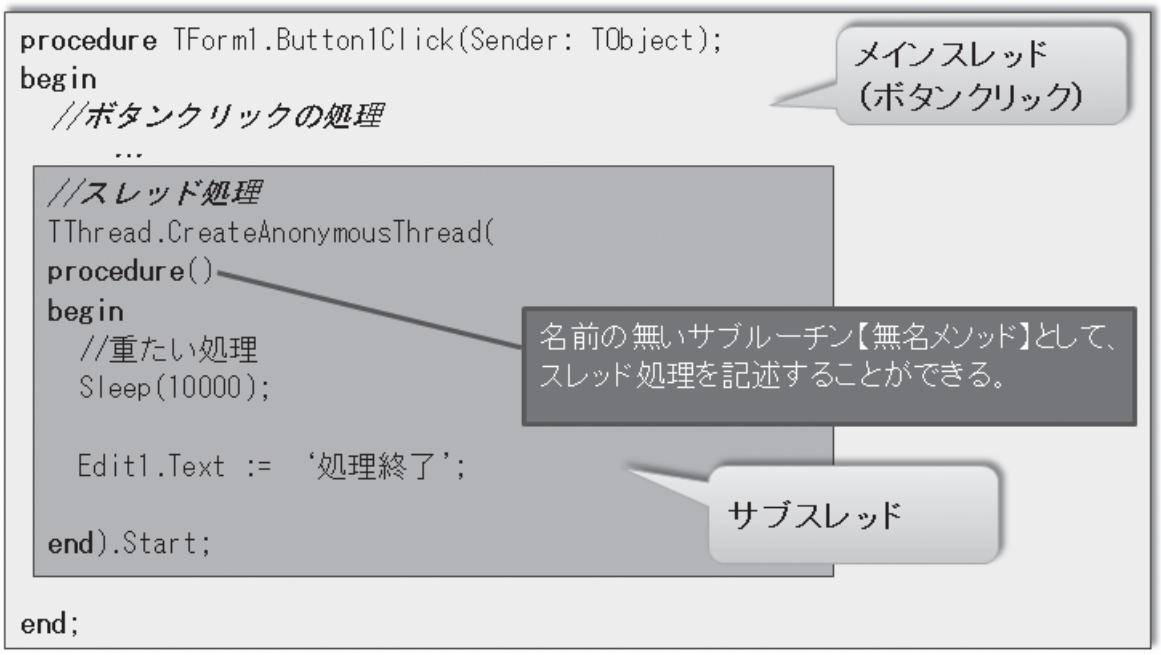
宣言部にスレッドユニットの参照を追加

スレッドオブジェクト変数をグローバル変数として追加

生成したスレッドオブジェクトを変数に代入

スレッドに終了を通知

図11


procedure TForm1.Button1Click(Sender: TObject);
begin
//ボタンクリックの処理
...
//スレッド処理
TThread.CreateAnonymousThread(
procedure()
begin
//重たい処理
Sleep(10000);
Edit1.Text := '処理終了';
end).Start;
end;
メインスレッド
(ボタンクリック)
名前の無いサブルーチン【無名メソッド】として、
スレッド処理を記述することができる。
サブスレッド

ソース11

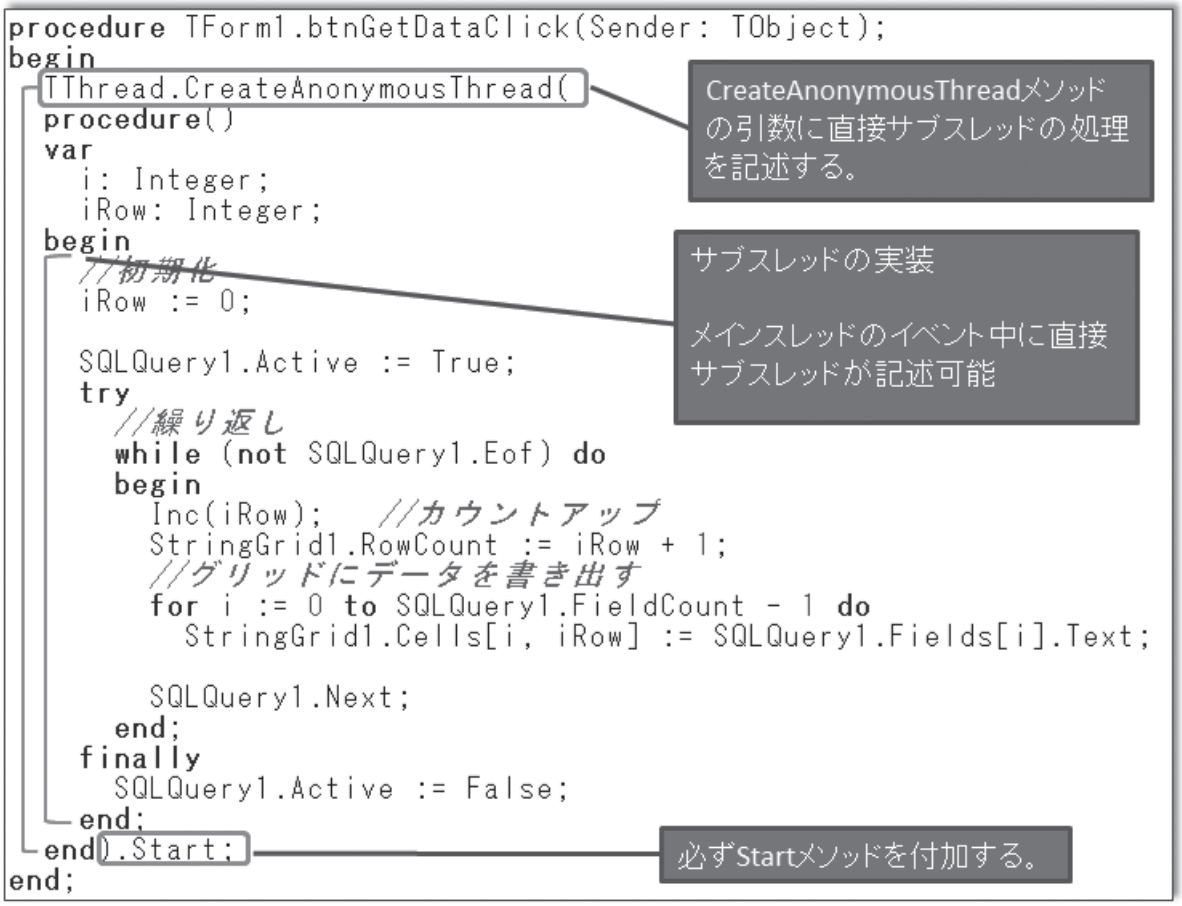
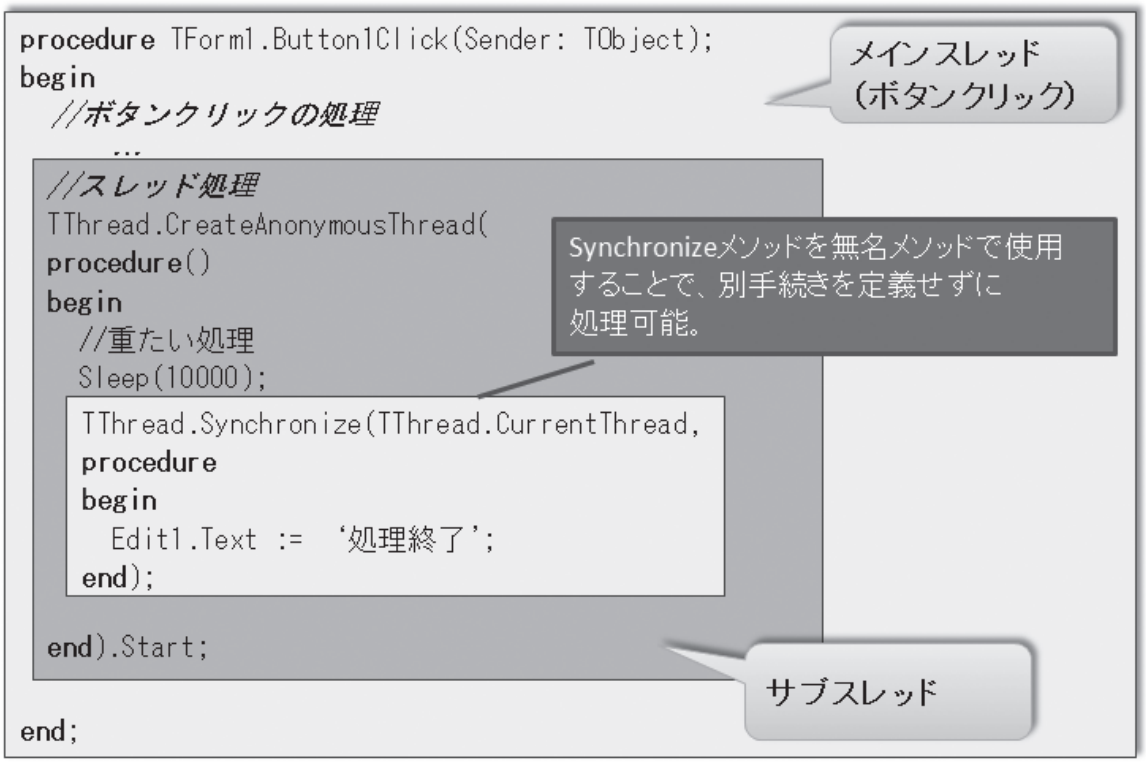

procedure TForm1.btnGetDataClick(Sender: TObject);
begin
TThread.CreateAnonymousThread(
procedure()
var
i: Integer;
iRow: Integer;
begin
//初期化
iRow := 0;
SQLQuery1.Active := True;
try
//繰り返し
while (not SQLQuery1.Eof) do
begin
Inc(iRow); //カウントアップ
StringGrid1.RowCount := iRow + 1;
//グリッドにデータを書き出す
for i := 0 to SQLQuery1.FieldCount - 1 do
StringGrid1.Cells[i, iRow] := SQLQuery1.Fields[i].Text;
SQLQuery1.Next;
end;
finally
SQLQuery1.Active := False;
end;
end).Start;
end;
CreateAnonymousThreadメソッド
の引数に直接サブスレッドの処理
を記述する。
サブスレッドの実装
メインスレッドのイベント中に直接
サブスレッドが記述可能
必ずStartメソッドを付加する。

図12



ソース12

```
procedure TForm1.btnGetDataClick(Sender: TObject);
begin
  TThread.CreateAnonymousThread(
  procedure()
  var
    iRow: Integer;
  begin
    //初期化
    iRow := 0;

    SQLQuery1.Active := True;
    try
      //繰り返し
      while (not SQLQuery1.Eof) do
      begin
        Inc(iRow);   //カウントアップ

        //メインスレッドを待機させて処理実行
        TThread.Synchronize(TThread.CurrentThread,
        procedure
        var
          i: Integer;
        begin
          StringGrid1.RowCount := iRow + 1;
          //グリッドにデータを書き出す
          for i := 0 to SQLQuery1.FieldCount - 1 do
            StringGrid1.Cells[i, iRow] := SQLQuery1.Fields[i].Text;
        end);

        SQLQuery1.Next;
      end;
    finally
      SQLQuery1.Active := False;
    end;
  end).Start;
end;
```

Synchronizeメソッドの引数にVCLコンポーネントを操作する処理を記述する。

VCLコンポーネントの操作をイベント中に直接記述可能