

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi .NET

## Utilizando Threads na Prática

Veja todos os detalhes e dicas de como e onde utilizar Threads em suas aplicações



**Ricardo C. Boaro**

(rboaro@aquasoft.com.br)

trabalha com desenvolvimento de sistemas em Delphi há mais de 10 anos e PocketStudio há 3 anos. Atualmente é gerente de informática na Di Hellen Indústria de Cosméticos, e atual como instrutor certificado Borland na Aquasoft Tecnologia da Informação parceira da Borland, em Porto Alegre – RS. Borland Instrutor, Delphi 7, 2007 e Certified.

O foco de nosso artigo são as *Threads*. Mas, o que são *Threads*? Veremos nesse artigo, muito além do que elas são, ou fazem, pois abordaremos a sua utilização prática. Conheceremos o conceito, analisaremos a classe *TThread* e criaremos um exemplo para aplicarmos a teoria. É importante, sempre que possível, abstrairmos os conceitos e funcionalidades das tecnologias, para a realidade, seja ela o desenvolvimento do nosso software, ou do software da empresa onde trabalhamos.

### O que são Threads?

*TThread* é uma classe abstrata e uma classe abstrata é sempre uma superclasse que não possui instâncias. Ela define um modelo implementando métodos abstratos, ou seja, sem implementação para que possamos utilizá-lo da maneira que precisarmos. A classe *TThread*, possibilita a criação de *Threads* separadas, para utilizarmos em nossas aplicações.

Simplificando, *threads* são similares a processos, que são executados em *background*, (“por trás da aplicação”). Podemos utilizar o seguinte exemplo para entendermos melhor:

- Imaginemos que precisamos abrir uma tabela para manipularmos alguns registros, mas não temos outra opção, a não ser abrir toda a tabela. Essa operação de chamar o método *Open* da classe *TDataSet* ou “setar” a propriedade *Active* para *True*, carrega todos os registros da tabela em memória. Tranquilo? Sim se a tabela for pequena, mas imaginemos uma tabela com 1.000.000 de registros, com certeza levará um tempo para que o processador consiga ler essas informações no HD e carregá-las na memória. Esse processo também é conhecido por *overhead*. Enquanto o processo está em andamento nosso sistema “trava” aos olhos do usuário. Nem tente explicar para ele, que você está executando uma operação que exige muito do processa-

dor e que, por isso, o sistema pode ficar lento por uns momentos. Com o passar do tempo, e com o constante crescimento da concorrência no mercado de trabalho, temos mais e mais tarefas para serem executadas.

É completamente inaceitável para uma empresa, depender de um software que “trava” sempre que uma tabela esta sendo aberta, ou seja, carregada para a memória. Essa é uma das situações em que as *Threads* nos ajudam, pois se a utilizarmos para carregar os registros em memória o sistema não vai travar e o usuário pode continuar seu trabalho, sem perceber o que esta por trás (“background”).

## Conhecendo a classe TThread

A classe *TThread* é declarada na *Unit Classes* do Delphi. E ela possui algumas características (“métodos”) que precisam de atenção para que possamos entender perfeitamente. Vejamos algumas delas:

- *Execute*: Método abstrato. Esse método é sempre sobrescrito ao “desenharmos” nossas próprias *threads*. É ele que efetivamente põe a *thread* e execução;
- *Synchronize*: Responsável pelo *Sincronismo*, entre as *thread's* que estiverem sendo executadas, incluindo as *thread's* do sistema operacional;
- *Suspend*: Suspende (“pausa”) temporariamente a execução;
- *Resume*: Reinicia a execução de um processo suspenso;
- *Terminate*: Seta propriedade *Terminated* para *True*;
- *WaitFor*: Aguarda o término da *thread* e retorna o valor da propriedade *ReturnValue*;
- *Priority*: Indica a prioridade da *thread* em relação às outras *thread's* que estão em execução.

## Criando uma aplicação

A aplicação que criaremos será simples, mas suficiente para entendermos onde e quando devemos utilizar *Thread's*. Vamos utilizar o Delphi 7 para criarmos nosso exemplo. A idéia inicial que proponho é criarmos uma aplicação que precisa acessar uma tabela para poder alterar os dados da mesma. A base de dados que utilizarei em nosso

### Listagem 1. Código para gravação de registros

```
procedure TfrmThreads.btnGravarClick(Sender: TObject);
var
  I : Integer;
begin
  { Passo o valor máximo de registros para a propriedade Max do ProgressBar }
  ProgressBar1.Max:= 1000000;

  { Chamo o método Open do DataSet }
  TblClientes.Open;

  { Mostro uma mensagem com a quantidade de registros existente na tabela de clientes }
  ShowMessage(IntToStr(TblClientes.RecordCount));

  { Desabilito o controle visual chamando o método DisableControls. Esse método é muito importante
  para melhorar a performance da aplicação, deve ser utilizando sempre que necessário
  percorrer os registros de um DataSet }
  TblClientes.DisableControls;

  { Passo os valor para o laço }
  for i:= 0 to 1000000 do
  begin
    { Chamo o método insert do DataSet para colocar a tabela em modo de inserção }
    TblClientes.Insert;

    { Passo um valor para o Field Nome da tabela de clientes }
    TblClientes.Nome.AsString := 'Sou o cliente: ' +
      IntToStr(i);

    { Chamo o método Post para gravar os registros }
    TblClientes.post;

    { Atualizo o progresso do ProgressBar }
    ProgressBar1.Position := i;
  end;

  { No final do método mostro uma mensagem informando que os registros foram gravados com sucesso }
  ShowMessage('Todos os clientes gravados com sucesso!');
end;
```

exemplo, é o SQL Server Express, que é a versão *free* do SQL Server. Nele vamos criar uma tabela e salvá-la com o nome de Clientes. Após isso, vamos preenchê-la com 1.000.000 (um milhão) de registros. Isso será mais do que suficiente, para demonstrarmos o tempo que o sistema levará, para abrir a tabela, sempre que necessário.

Não vou entrar em detalhes quanto a criação da base de dados e da tabela, por não ser o foco do artigo. Não é necessário criar a tabela no SQL Server. Você pode utilizar qualquer banco de dados que estiver habituado a trabalhar, tais como Firebird, MySQL. O método que criaremos para preenchê-la funciona da mesma forma com qualquer base de dados. A tabela deve possuir apenas um campo inteiro e outro *Varchar*(50).

Salvamos a tabela com o nome de *Clientes*, feito isso, vamos codificar o botão para preencher os registros como proposto. A interface do sistema pode ser simples, como na **Figura 1**.

Os componentes necessários para criar a interface da aplicação de exemplo são:

- *ProgressBar*: para que possamos acompanhar o preenchimento da tabela;

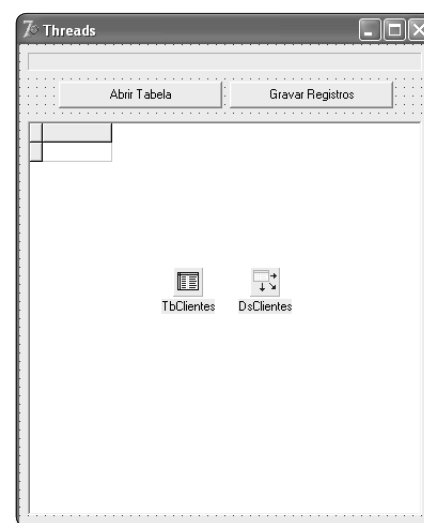


Figura 1. Janela de exemplo

- *DbGrid*: para podermos visualizar e alterar os dados;
- *ADOTable*(“TblClientes”): Como utilizei o SQL Server para criar a tabela, utilizei um *ADOTable* para fazer a conexão com a base e manipular a tabela;
- 2 *Buttons*: um para abrir a tabela, e outro para gravar os registros no evento *OnClick*.

Faremos um laço para preencher a tabela e o botão de abrir tabela, deve chamar o método *Open* da classe *TDataSet* ou *setar* a propriedade *Active* para *True*. Na **Listagem 1** segue o código do botão gravar registros. Todo o código encontra-se comentado para melhor entendimento.

Após montar o exemplo, execute-o e clique em *Gravar Registros*. Executado esse procedimento, temos 1.000.000 de registros gravados na tabela de clientes. Agora poderemos iniciar os testes. O primeiro teste que faremos será abrir a tabela e marcar o tempo em segundos, que o sistema levará para carregar todos os registros para a memória e liberar a tela da aplicação. Notaremos que a tela ficará travada, até que o *Open* seja finalizado. Antes de executarmos a aplicação, vamos codificar o evento *BeforeOpen* e *AfterOpen* do componente *Table*. Os mesmos serão executados antes e depois da tabela processar o *Open*, respectivamente. A **Listagem 2** mostra o código de ambos eventos.

**Nota:** Declare as variáveis *HoraFim* e *HoraIni*, ambas do tipo *TDateTime*, na seção *public* do formulário.

#### Listagem 2. Código dos eventos BeforeOpen e AfterOpen

```
procedure TForm1.Tb1ClientesAfterOpen(DataSet: TDataSet);
begin
    ShowMessage(TimeToStr(Now));
    HoraFim := Now;
end;

procedure TForm1.Tb1ClientesBeforeOpen(DataSet: TDataSet);
var
begin
    ShowMessage(TimeToStr(Now));
    HoraIni := Now;
end;
```

#### Listagem 3. Código do botão Abrir Tabela

```
procedure TThreads.btnAbrirTabelaClick(Sender: TObject);
var
    TempoTotal : TDateTime;
begin
    { Chamo o método Open do DataSet }
    Tb1Clientes.Open;

    { Calculo o tempo utilizado }
    TempoTotal := HoraFim - HoraIni;

    { Mostro uma mensagem com o tempo total que o sistema levou para executar o Open }
    ShowMessage('Tempo total em segundos para abrir a tabela: ' + ' - ' + 
        TimeToStr(TempoTotal));
end;
```

No botão *Abrir Tabela* adicione o código da **Listagem 3**. O código é bem simples, apenas invocamos o método *Open* do objeto *AdoTable*, subtraímos a *HoraFim* da *HoraIni* e então mostramos uma mensagem com o tempo em segundos que a tabela levou para ser aberta. (**Figura 2**)

Perceba, pelas imagens, que em minha máquina o tempo para abertura da tabela foi de 51 segundos. Isso significa que foram 51 segundos com o sistema “travado”. É claro que esse valor pode variar ainda mais dependendo da máquina onde se executa o procedimento, dos processos que estão sendo executados, tráfego na rede e uma série de outras situações corriqueiras em uma rede.

Agora imaginemos um usuário com pilhas de dados a serem alterados, esperar pacientemente até que a tabela esteja disponível para alteração. No mínimo ele já terá clicado em *Fechar* ou *CTRL+ALT+DEL*, pensando que o sistema está travado. Nesse exemplo apresentamos uma situação problemática, agora cabe a nós resolvermos. Uma das soluções para essa situação é utilizarmos uma *Thread*, e dispararmos o método *Open* da tabela dentro dessa *Thread*.

## Criando uma Thread

Para criarmos uma *Thread* acesse o menu *File|New|Other|New|Thread Object*. Clicando em *Ok*, será criada uma nova *Unit*. Digite o nome da classe em *Class Name* (“*TBackground*”) e salve a *Unit* como “*uBackground.pas*”. Declare o método *AbreTabela* na seção *Private* como segue:

```
...
private
    { Private declarations }
    procedure AbreTabela;
protected
    procedure Execute; override;
end;
...
```

Em seguida pressione *CTRL + SHIFT + C* para que o Delphi nos crie o cabeçalho do método. Encontre sua implementação na seção *Implementation* da *Unit* e digite o código a seguir:

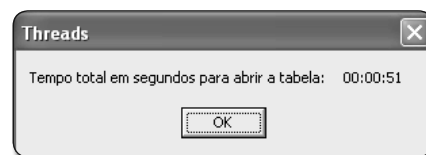
```
Form1.Tb1Clientes.Open;
```

Na sequência sobrescreva o método *Execute* com o apenas invocando o método *Synchronize* como segue:

```
Synchronize(AbreTabela);
```

Como vimos na explicação teórica sobre *Thread*, devemos utilizá-la criando uma classe descendente de *Thread* e sobrescrever o método *Execute*. Todo esse trabalho de criar a estrutura da classe com a declaração do método *Execute* foi feito pelo Delphi ao adicionamos o *Thread Object* na aplicação. O próximo passo, foi criarmos um método chamado *AbreTabela*, que executará o método *Open* da *Table*. Como *Tb1Clientes* está na *Unit* principal, devemos adicioná-la na *Uses* de *uBackground*, ou seja, pressione *Alt + F11* e inclua *uPrincipal* ao *Uses*.

O método *Synchronize* é utilizado para executar o método *AbreTabela*. Este, é responsável por fazer sincronismo entre as *Threads* da VCL e de nossa aplicação. Essa é a maneira correta de chamarmos os métodos dentro do método *Execute* da *Thread*.



**Figura 2.** Tempo para abrir a tabela

Voltando ao formulário principal de nossa aplicação, faremos as alterações necessárias para utilizarmos a *Thread*. O primeiro passo é adicionarmos o *uBackground* na *Uses* do formulário. Feito isso, ao invés de chamar-mos o método *Open*, no evento *OnClick* do botão de *Abrir Tabela*, vamos executar a nossa *Thread* criando uma instância dela. Modifique o código do botão *Abrir Tabela* pelo código que vemos a seguir:

```
TBackground.Create(False);
```

O esquema é bem simples, basta criarmos a *Thread* e passarmos como parâmetro *False*. Esse parâmetro indica que a *Thread* deve ser criada e executada imediatamente. Testando a aplicação podemos perceber que mesmo sendo uma tabela com centenas de registros a aplicação não trava e podemos continuar interagindo com o sistema.

## Monitorando a Execução da Thread

Pode ser necessário monitorarmos a execução da *Thread* no momento em que ela é instanciada e executada. Para isso devemos colocar um *BreakPoint* no evento *OnClick* do botão *Abrir Tabela* e acessarmos o menu *View\Debug Windows>Threads*. Uma janela semelhante a **Figura 3** será mostrada. Nela temos o *ID*, ou seja, o identificador de todas as *Thread's* que estiverem em execução no momento. O *State* ("estado") da mesma e também a localização dela em memória.

**Nota:** É importante considerarmos, que as *Threads* são executadas diretamente no processador, o que pode comprometer o processamento caso abusarmos de mais do uso das mesmas. Atualmente, os computadores possuem processadores com grande capacidade de processamento, mas devemos ficar atentos caso nossa aplicação precise rodar em máquinas mais lentas.

## Conclusão

Nesse artigo, conhecemos e criamos um exemplo prático de utilização de *Threads*, uma poderoso recurso disponível no Delphi. Não devemos nos esquecer que ele é muito útil, e também pode ser utilizado para gerar grandes relatórios que envolvam *Joins* em várias

tabelas, ou ainda atualizarmos o estoque de produtos após a digitação e ou impressão de vários pedidos. As situações para o uso de *Thread* podem ser as mais variadas possíveis. Conhecendo e entendendo os conceitos, basta utilizarmos a imaginação para melhorarmos a performance da nossa aplicação. Bons códigos e até o próximo artigo. ●

### Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

[www.devmedia.com.br/clubedelphi/feedback](http://www.devmedia.com.br/clubedelphi/feedback)



Figura 3. Threads em execução na aplicação

