

Esta é a sua cópia pessoal da apostila

Olá Diego Garcia! É com grande satisfação que lhe disponibilizamos esta apostila sobre o tema “Controlando a Concorrência em Aplicações Multi-Thread com Delphi”.

Por conta de um compromisso em poupar recursos e contribuir por um mundo melhor não imprimiremos o material e acreditamos que em geral não há necessidade de se fazer isto. Por isto esta é uma cópia pessoal do material.

Esperamos que você compartilhe este material com seus colegas, mas pedimos a gentileza de não compartilhar na grande web. Futuras atualizações serão enviadas diretamente ao seu e-mail, drgarcia1986@gmail.com e por isso solicitamos que mantenha seu cadastro atualizado.

Uma informação relevante é que este material é uma compilação dos artigos do autor Mário Guedes para a Active Delphi bem como postagens no blog eugostododelphi.blogspot.com o que implica que eventualmente algum texto lhe pareça familiar. Outras fontes serão devidamente creditadas ao fim da apostila.

O seu canal de comunicação conosco é através do e-mail aluno@arrayof.com.br.

Sugestões de melhoria serão sempre bem vindas!

Muito obrigado pelo prestígio de sua companhia.

Sobre esta apostila

| |
|---|
| Versão: 001 - Abril/2014 |
| Revisor: José Mário Silva Guedes |
| Dono: Diego Garcia |

Sobre a arrayOF

A arrayOF Consultoria e Treinamento tem por filosofia desenvolver o potencial de seus parceiros ensinando e aprendendo com eles.

Sumário

| | |
|---|----|
| Esta é a sua cópia pessoal da apostila | 1 |
| Sobre esta apostila | 1 |
| Sobre a arrayOF | 1 |
| Introdução | 4 |
| Mas o que é thread? | 4 |
| Mas é realmente ao mesmo tempo? | 5 |
| API do Windows | 5 |
| O Timer não é um thread! | 5 |
| Aplicativo de exemplo | 6 |
| Classe TThread | 8 |
| Construtor Create | 11 |
| Propriedade FreeOnTerminate | 12 |
| Método Execute X Método Start | 12 |
| Evento OnTerminate | 12 |
| Método Terminate() e propriedade Terminated | 13 |
| Método WaitFor() | 13 |
| Controlando a concorrência | 15 |
| Funcionamento básico de um projeto VCL | 15 |
| Handle | 16 |
| Programa alvo | 16 |
| Programa gatilho | 18 |
| Executando os programas | 19 |
| Tornando o programa mais responsivo | 20 |
| Métodos Synchronize() e Queue() | 22 |
| Adequando o sistema | 23 |
| Seção crítica | 26 |
| Problema proposto | 26 |
| Solução sem proteção | 26 |
| Cenário Atual versus Cenário Pretendido | 35 |
| Seção Crítica | 37 |
| Otimizando um thread | 42 |
| Application.ProcessMessages() | 48 |

| | |
|---|----|
| Sleep(1) | 49 |
| TEvent | 49 |
| Instanciando o TEvent | 50 |
| Aguardando a sinalização: WaitFor | 51 |
| Efetuando a sinalização: SetEvent | 52 |
| Finalizando o thread | 52 |
| Cooperação entre threads | 54 |
| TMonitor – Usando como seção crítica | 55 |
| Cenário problemático | 55 |
| Solução proposta | 60 |
| TMonitor – Como sinalizador de evento | 62 |
| Conclusão | 68 |

Introdução

O thread é um recurso do Sistema Operacional, que nos possibilita tornar nossos softwares *mais rápidos, mais responsivos, mais escaláveis e mais eficientes*. Porém, tudo isso não terá valor se não tivermos *controle* e é justamente o foco deste treinamento.

Antes poderíamos até entender que era um recurso avançado, mas tenho convicção de que na verdade é um recurso básico e os equipamentos atuais, com vários núcleos, exige este conhecimento.

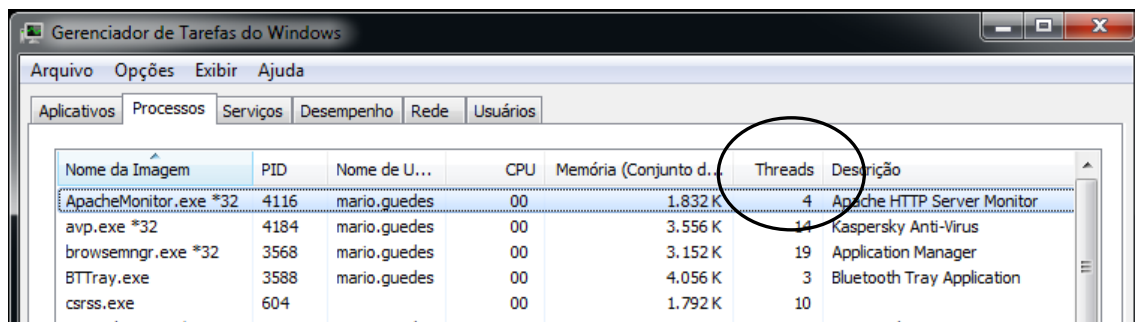
Mas o que é thread?

Os sistemas operacionais modernos podem realizar tarefas simultâneas. Cada tarefa é executada sob um contexto e este contexto é o thread. Por contexto, entenda um espaço bem definido com sua própria pilha, sua própria cópia dos registradores e informações relacionadas.

Muitas vezes, temos consciência apenas do nosso executável, ou seja, o "*.exe". O executável na verdade torna-se um *processo* que por sua vez é um conjunto de threads, todos rodando em um único espaço de endereço podendo então compartilhar os recursos existentes neste espaço como variáveis e objetos.

A tradução literal da palavra thread é "linha". Podemos entender que cada thread é *uma linha de execução paralela*.

No Gerenciador de Tarefas do Windows (CTRL+SHIFT+ESC), na aba "Processos" é possível visualizar os *processos* em execução, bem como a quantidade de *threads* de cada um desses processos. Dependendo da versão do Windows você encontrará a coluna "Segmentos" e em outras a coluna "Threads".



| Nome da Imagem | PID | Nome de U... | CPU | Memória (Conjunto d... | Threads | Descrição |
|-----------------------|------|--------------|-----|------------------------|---------|----------------------------|
| ApacheMonitor.exe *32 | 4116 | mario.guedes | 00 | 1.832 K | 4 | Apache HTTP Server Monitor |
| avp.exe *32 | 4184 | mario.guedes | 00 | 3.556 K | 14 | Kaspersky Anti-Virus |
| browsemgr.exe *32 | 3568 | mario.guedes | 00 | 3.152 K | 19 | Application Manager |
| BTTray.exe | 3588 | mario.guedes | 00 | 4.056 K | 3 | Bluetooth Tray Application |
| csrss.exe | 604 | | 00 | 1.792 K | 10 | |

Graças ao recurso de thread nossos softwares podem, efetivamente, executar mais de uma tarefa a um só tempo. Quando tiramos proveito desta Cópia pessoal de:

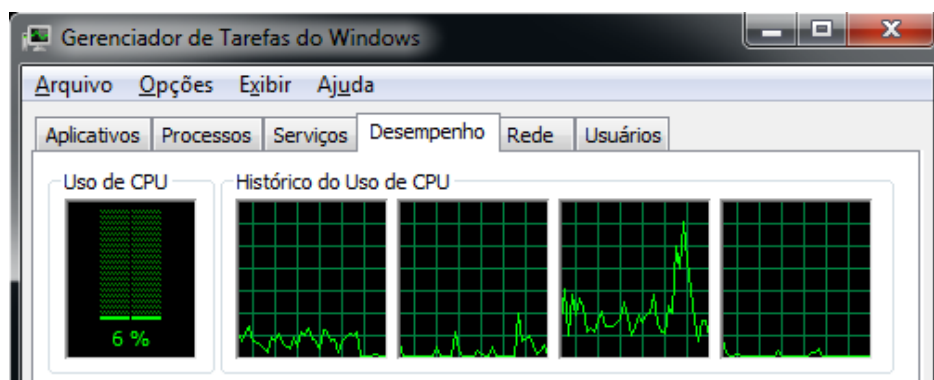
oportunidade estamos aplicando a “*programação paralela*”, também denominada “*programação concorrente*”.

Mas é realmente ao mesmo tempo?

Para que o sistema operacional possa, de fato, executar mais de uma tarefa ao mesmo tempo o equipamento precisa ter mais de um núcleo.

Vale saber que o Windows trabalha no modelo *preemptivo*, tanto para processos quanto para threads. Neste modelo de trabalho o *kernel* - que é núcleo do sistema operacional, tem controle sobre o tempo que é dedicado a cada thread, fazendo um rodízio entre eles e considerando a prioridade de cada um. O oposto deste modelo é o *cooperativo* onde fica a cargo do thread (entenda-se programador) liberar a CPU para o próximo thread.

Ainda no Gerenciador de Tarefas, na aba “Desempenho” pode-se verificar a situação de cada núcleo.



API do Windows

Thread é um recurso do Sistema Operacional, portanto para a criação e manipulação dos threads existem chamadas específicas na API do Windows, como por exemplo: a instrução `CreateThread`. Mas o Delphi nos oferece, entre outras, a classe `TThread` para interagir com os threads, o que nos permite trabalhar em alto nível, nos afastando da complexidade da API do Windows.

Porém, um trabalho mais específico com thread exigirá o estudo desta API.

O Timer não é um thread!

Algo que deve ficar entendido logo de início é que o componente `TTimer` não é um thread. Não se trata de programação paralela.

Na verdade, o Timer também é um recurso do sistema operacional e é uma sinlização que ocorre no tempo determinado na propriedade `Interval` e o código é, invariavelmente, processado pelo thread principal.

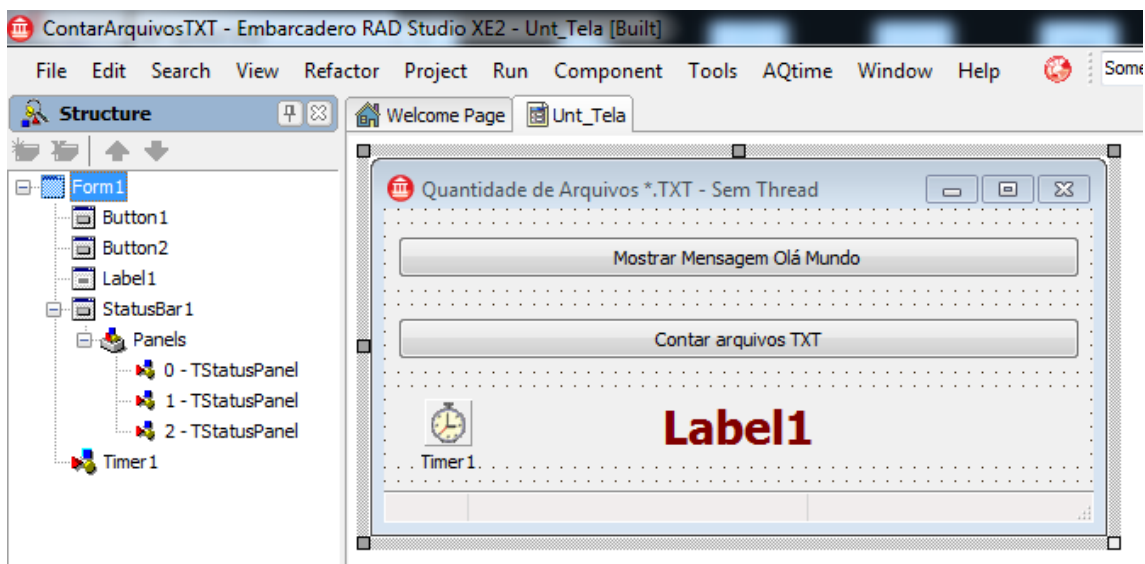
Portanto, não se deve usar o Timer com o propósito de se ter um trabalho paralelo porque isso não acontecerá. A utilização mais correta em muitos casos é usar o Timer para estimular a criação de um thread que efetivamente teria o algoritmo a ser processado de modo paralelo.

Imagine algo como: “verificar de 30 em 30 segundos se existe pedidos novos”.

Um timer pode ser usado para iniciar (a cada 30 segundos) o thread que efetivamente fará a conferência que por ser uma rotina “pesada” pode comprometer a usabilidade do software como um todo. Mas se o próprio timer executar a conferência, daí teremos um problema.

Aplicativo de exemplo

Ao longo da apostila vamos desenvolver diversos aplicativos para validar os conceitos que serão explicados. Como primeiro exemplo, vamos desenvolver um programa simples que verifica a quantidade de arquivos com a extensão “*.txt”.



Vamos ao código do Timer, cujo propósito é informar a hora atual na barra de status:

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
  Self.StatusBar1.Panels[2].Text := TimeToStr(Time);
end;
```

Agora o código do botão “Mostrar Mensagem Olá Mundo”:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Olá Mundo!');
end;
```

Por fim o código do botão “Contar arquivos TXT”. O objetivo do algoritmo é varrer todos os diretórios da unidade C e contar os arquivos com a extensão “*.txt” salvando a listagem dos arquivos localizados.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  slListagem : TStringList;
  iQuantidade: Cardinal;

  procedure _ContarArquivos(ADiretorioAlvo: string);
  var
    rLocalizador: TSearchRec;
    iAchou      : Integer;
  begin
    iAchou := FindFirst(ADiretorioAlvo + '\*.*', faAnyFile, rLocalizador);

    while (iAchou = 0) do
      begin
        if not((rLocalizador.Name = '.') or (rLocalizador.Name = '..')) then
          begin
            if ((rLocalizador.Attr and faDirectory) = faDirectory) then
              begin
                _ContarArquivos(ADiretorioAlvo + '\' + rLocalizador.Name);
              end else begin
                if (AnsiSameText(ExtractFileExt(rLocalizador.Name), '.txt')) then
                  begin
                    slListagem.Add(ADiretorioAlvo + '\' + rLocalizador.Name);
                    Inc(iQuantidade);
                  end;
                end;
              end;
            iAchou := FindNext(rLocalizador)
          end;
        FindClose(rLocalizador);
      end;
    end;

    FindClose(rLocalizador);
  end;

var
  rCronometro : TStopwatch;
begin
  Self.Label1.Caption := '-';
  Self.StatusBar1.Panels[1].Text := EmptyStr;
  Application.ProcessMessages;

  slListagem := TStringList.Create;
  iQuantidade := 0;
  try
    rCronometro := TStopwatch.StartNew;
    _ContarArquivos('c:');
  end;
```

```
Self.Label1.Caption := IntToStr(iQuantidade);  
finally  
  rCronometro.Stop;  
  Self.StatusBar1.Panels[1].Text := Format('Tempo: %d segundos', [rCronometro.Elapsed.Seconds]);  
  slListagem.SaveToFile('.\listagem.txt');  
  slListagem.Free;  
end;  
end;
```

Enquanto este algoritmo estiver em execução não conseguiremos fazer nada como, por exemplo, clicar no primeiro botão, minimizar, maximizar ou até mesmo mover o formulário. E ainda por cima o Timer estará “congelado” durante o processamento.

E é neste ponto que queremos atuar. Esta rotina poderia ser qualquer outra coisa como fechamento do estoque, geração de um relatório, enfim, qualquer processamento mais intenso que afete a usabilidade do nosso software. Vamos melhorar?

Classe TThread

O Delphi nos oferece a classe `TThread`, que encapsula toda a complexidade de se criar e manipular um thread. Ela está presente na unidade `System.Classes`.

O trabalho básico é criar uma classe descendente de `TThread` e reescrever o método `Execute()`. Mas isso é apenas o começo. Devemos nos preocupar com vários detalhes para ter eficiência ao se trabalhar com thread.

Opcionalmente, para se criar o código de um novo thread, podemos ir ao menu principal:

File→New→Other→Delphi Projects→Delphi Files→Thread Object

Mas a criação de um novo thread é tão simples que não há necessidade de ir por este caminho.

Para melhorar o nosso aplicativo de exemplo vamos criar uma nova unidade no projeto e declarar a classe `TContarArquivo` descendente da classe `TThread` e transferir todo o código originalmente feito no `OnClick` do botão para o método `Execute()` da classe com algumas adaptações:

```
unit Unt_Thread;  
  
interface  
  
uses  
  System.Classes;
```



```

type

  TContarArquivo = class(TThread)
  protected
    procedure Execute; override;
  end;

implementation

uses
  System.SysUtils, System.Diagnostics, Unt_Tela, Vcl.Forms;

{ TContarArquivo }

procedure TContarArquivo.Execute;
var
  slListagem : TStringList;
  iQuantidade: Cardinal;

  procedure _ContarArquivos(ADiretorioAlvo: string);
  var
    rLocalizador: TSearchRec;
    iAchou      : Integer;
  begin
    iAchou := FindFirst(ADiretorioAlvo + '\*.*', faAnyFile, rLocalizador);

    while (iAchou = 0) do
    begin
      if not((rLocalizador.Name = '.') or (rLocalizador.Name = '..')) then
      begin
        if (Self.Terminated) then
        begin
          Self.ReturnValue := 0;
          Abort;
        end;

        if ((rLocalizador.Attr and faDirectory) = faDirectory) then
        begin
          _ContarArquivos(ADiretorioAlvo + '\' + rLocalizador.Name);
        end else begin
          if (AnsiSameText(ExtractFileExt(rLocalizador.Name), '.txt')) then
          begin
            slListagem.Add(ADiretorioAlvo + '\' + rLocalizador.Name);
            Inc(iQuantidade);
          end;
        end;
      end;
      iAchou := FindNext(rLocalizador)
    end;

    FindClose(rLocalizador);
  end;

var
  rCronometro: TStopwatch;
begin
  Form1.Label1.Caption := '-';
  Form1.StatusBar1.Panels[1].Text := EmptyStr;
  Application.ProcessMessages;

  slListagem := TStringList.Create;
  iQuantidade := 0;
  try
    rCronometro := TStopwatch.StartNew;
  
```

```
_ContarArquivos('c:');  
Form1.Label1.Caption := IntToStr(iQuantidade);  
  
Self.ReturnValue := 1;  
finally  
  rCronometro.Stop;  
  Form1.StatusBar1.Panels[1].Text := Format('Tempo: %d segundos',  
[rCronometro.Elapsed.Seconds]);  
  slListagem.SaveToFile('.\listagem.txt');  
  slListagem.Free;  
end;  
end;  
  
end.
```

Voltando ao nosso formulário, vamos à seção `private` da classe `TForm1` e declarar uma variável para o nosso thread, bem como um método que será usado no evento `OnTerminate()` do thread:

```
TForm1 = class(TForm)  
  Button1: TButton;  
  Label1: TLabel;  
  StatusBar1: TStatusBar;  
  Button2: TButton;  
  Timer1: TTimer;  
  procedure Button1Click(Sender: TObject);  
  procedure Button2Click(Sender: TObject);  
  procedure Timer1Timer(Sender: TObject);  
private  
  FContador: TContarArquivo;  
  procedure EventoOnTerminate(Sender: TObject);  
public  
  { Public declarations }  
end;
```

A implementação do código do método `EventoOnTerminate` é o que se segue:

```
procedure TForm1.EventoOnTerminate(Sender: TObject);  
begin  
  Self.FContador := nil;  
  Self.Button2.Enabled := True;  
end;
```

Vamos também codificar o evento `OnClose()` do formulário:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
var  
  iRet: Cardinal;  
begin  
  if (Assigned(Self.FContador)) then  
  begin  
    Self.StatusBar1.Panels[1].Text := 'Esperando ...';  
    Application.ProcessMessages;  
  
    Self.FContador.FreeOnTerminate := False;  
    Self.FContador.Terminate;  
  end;  
end;
```

```

iRet := Self.FContador.WaitFor;

if (iRet = 1) then
begin
  Self.StatusBar1.Panels[1].Text := 'Sucesso !';
end else
begin
  Self.StatusBar1.Panels[1].Text := 'Insucesso !';
end;

Application.ProcessMessages;
Sleep(3000);
end;
end;

```

Por fim, o novo código do botão (Button2):

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  Self.Button2.Enabled := False;
  Self.FContador := TContarArquivo.Create(True);
  Self.FContador.FreeOnTerminate := True;
  Self.FContador.OnTerminate := Self.EventoOnTerminate;
  Self.FContador.Start;
end;

```

Executando o software percebe-se que mesmo que a contagem dos arquivos esteja em andamento, conseguimos interagir com o formulário: clicar no outro botão, mover, maximizar e minimizar o formulário e o Timer não “congela”. E em algum momento temos o resultado da contagem.

Mas alguns cuidados devem ser tomados e iremos, daqui para frente, aprender a proteger os recursos *compartilhados*. O thread, por exemplo, esta manipulando livremente elementos da interface e isto esta propenso a problemas.

Construtor Create

O construtor da classe TThread possui duas versões. Uma sem parâmetro e outra com o parâmetro CreateSuspended .

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  Self.Button2.Enabled := False;
  Self.FContador := TContarArquivo.Create();
end;

procedure TForm1.EventoOnTerminate(Sender: TObject);
begin

```

<no parameters expected>
 CreateSuspended: Boolean
 <no parameters expected>

Na imagem aparece uma terceira opção, mas na verdade refere-se à classe `TObject` da qual a classe `TThread` é imediatamente descendente.

Se a primeira opção (sem parâmetros) for usada, considera-se que o thread será colocado em execução imediatamente. Já a segunda opção nos permite decidir isso. Caso seja necessário preencher alguns atributos da classe antes de efetivamente colocar o thread em execução, é o caso de instanciar o objeto com o parâmetro `CreateSuspended` com o valor `True`, como no nosso exemplo. Com isso o thread ficará suspenso até que a inicializemos com o método `Start()`.

O construtor refere-se à classe e não ao thread propriamente dito. Não devemos confundir a classe `TThread` com o recurso thread. A classe é, simplesmente, um facilitador que o Delphi nos oferece.

Propriedade `FreeOnTerminate`

Observe no código que após instanciar o objeto da nossa classe, mudamos o valor da propriedade `FreeOnTerminate` para `True`. Basicamente estamos instruindo para que o objeto seja *automaticamente* liberado ao fim da execução do método `Execute`. Afinal não sabemos quando isto ocorrerá e perde o sentido esperarmos a execução do thread para a liberação do objeto. Então, podemos usar esta propriedade para eliminar maiores controles e evitar vazamento de memória.

Método `Execute` X Método `Start`

Acredito que você tenha percebido que em nenhum momento invocamos o método `Execute()` de forma explícita. Invocamos o método `Start()`. Este método é que inicializa todos os recursos necessários para se criar o thread e *executa* o método `Execute()`. Se no `OnClick` do botão invocássemos diretamente o método `Execute` este seria executado pelo thread principal e estaríamos no mesmo cenário do primeiro exemplo (sem thread).

Em versões anteriores ao Delphi 2010 era chamado o método `Resume`, porém este foi depreciado.

Evento `OnTerminate`

A classe `TThread` possui um único manipulador de evento que é o `OnTerminate`. Este ocorre imediatamente após a finalização do thread e é

executado pelo thread principal. Neste nosso exemplo usamos para habilitar novamente o `Button2` bem como definir para `nil` a variável `FContador`.

E provavelmente é o melhor uso que se pode fazer dele: Interagir com os controles da VCL ao final do processamento do thread.

Método `Terminate()` e propriedade `Terminated`

A classe `TThread` possui a propriedade `Terminated` e esta deve ser sempre consultada no decorrer do processamento do thread. No nosso thread tem o seguinte trecho de código:

```
if (Self.Terminated) then  
begin  
    Abort;  
end;
```

Colocamos em um ponto estratégico e é isto que deve ser feito. Esta propriedade é simplesmente uma sinalização indicando que foi solicitado o encerramento do thread. Se não verificarmos esta sinalização corremos riscos e esses riscos dependem do que o thread estiver manipulando: arquivos, banco de dados, comunicação TCP/IP enfim, qualquer coisa que deva ser adequadamente finalizada e liberada.

No `OnClose` do formulário nós invocamos o método `Terminate` que justamente tem o propósito de mudar a "sinalização" `Terminated` para `True`.

E é exatamente isso: o método `Terminate` não finaliza o thread por si só. O propósito dela é justamente mudar o valor da propriedade `Terminated` e cabe ao programador verificar esta propriedade continuamente e tomar a melhor decisão possível.

Opcionalmente podemos reescrever os métodos `TerminatedSet()` e `DoTerminate()` presentes na classe `TThread`, o que não foi exemplificado neste artigo, pois estes métodos são invocados pelo método `Terminate`. Com isso podemos criar condições para terminar ou não o thread.

Método `WaitFor()`

Quando for necessário aguardar a efetiva finalização do thread devemos invocar o método `WaitFor()`.

Foi o que fizemos no evento `OnClose` do formulário, pois queríamos ter certeza de só finalizar o processo, ou seja, o nosso software depois que o nosso thread estivesse devidamente finalizado.

Podemos usar o retorno do `WaitFor()` para ter alguma indicação de como o thread foi finalizado: com sucesso, sem sucesso ou com um código de erro que nos faça sentido.

Isso porque a classe `TThread` possui a propriedade privada `ReturnValue`, que pode ser trabalhada no decorrer do thread. No nosso exemplo, usamos o valor (1) para indicar que o processamento foi finalizado com sucesso e o valor (0) para insucesso.

Controlando a concorrência

Uma rodovia não precisa de semáforo, certo? Mas um cruzamento em uma grande cidade precisa, pois do contrário haverá uma colisão. Há um recurso compartilhado neste caso: a *área de intersecção entre as duas vias*. Assim também acontece na programação multi-thread: *concorrência por recursos*.

Funcionamento básico de um projeto VCL

O thread principal de um projeto VCL tem como propósito principal a manipulação dos eventos do Windows. Relembre o código básico de um programa VCL:

```
program Project1;  
  
uses  
  Vcl.Forms,  
  Unit1 in 'Unit1.pas' {Form1};  
  
{$R *.res}  
  
begin  
  Application.Initialize;  
  Application.MainFormOnTaskbar := True;  
  Application.CreateForm(TForm1, Form1);  
  Application.Run;  
end.
```

Atentemos justamente para a última linha: `Application.Run;`

É ela que deixa o nosso software “de pé”. E este “Run” essencialmente é um *loop* infinito que processa as mensagens do Windows enviadas para o nosso software, acionando o código associado a cada mensagem: Movimentação de janela, clique no botão e por aí vai. Este *loop* é encerrado quando o *main form* é fechado.

Em linhas gerais é assim que um programa “for Windows” funciona: processando as mensagens do sistema operacional.

Vamos tomar o `TButton` como exemplo. Esta classe representa um botão e seu principal propósito é executar um comando ao ser acionado por *clique do mouse*. Como esta “mágica” acontece? Vamos analisar.

Handle

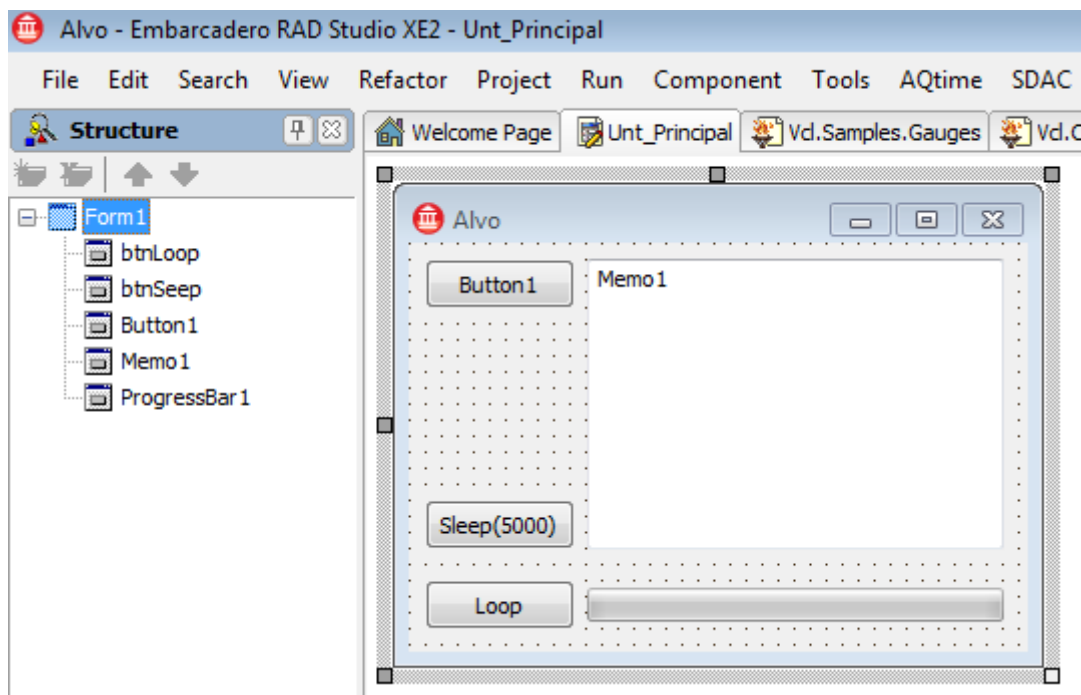
Todo controle visual possui um *handle* que em tradução livre significa: “alça”. Em termos práticos é um ID único e numérico que o Windows atribui a cada controle (visual ou não) ativo no sistema operacional. Desta maneira o Windows consegue referenciar o controle quando necessário. No Delphi, a propriedade `Handle` está presente na classe `TWinControl`, logo, todo elemento visual possui esta propriedade.

Voltando para o exemplo do botão, quando o usuário clica com o mouse na área em que se encontra o botão, o Windows envia uma mensagem para o programa associado a este botão. Então aquele “Run” intercepta esta mensagem e executa o código associado ao `OnClick` do botão. É claro que estamos simplificando bastante a questão até pelo fato de fugir ao escopo do artigo.

Mas vale a pena fazermos um experimento para entender o que de fato acontece com os nossos softwares. Para isso vamos desenvolver dois programas simples onde um acionará um determinado botão do outro.

Programa alvo

O programa alvo será chamado de “Alvo.exe” e tem a seguinte estrutura:



No manipulador de evento `OnCreate` do formulário vamos colocar o valor do `Handle` do `Button1` como `Caption` deste mesmo botão, com o intuito de termos este valor visível:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Self.Button1.Caption := IntToStr(Self.Button1.Handle);  
end;
```

Já no manipulador de evento `OnClick` do `Button1` vamos incrementar uma variável global do tipo `Integer` e escrever no `Memor1`:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Inc(Self.FQuantidade);  
    Self.Memor1.Lines.Insert(0, Format('Botão                acionado!  
[%d]', [Self.FQuantidade]));  
end;
```

Já no `btnLoop`, vamos simular um processamento “pesado” de aproximadamente 15 segundos que será transferido para um thread mais à frente:

```
procedure TForm1.btnLoopClick(Sender: TObject);  
var  
    i      : Integer;  
    rCronus: TStopwatch;  
    iTempo : Integer;  
begin  
    Screen.Cursor := crHourGlass;  
    rCronus := TStopwatch.StartNew;  
  
    Self.ProgressBar1.Max := 15;  
    Self.ProgressBar1.Position := 0;  
    for i := 1 to 15 do  
        begin  
            Sleep(1000);  
            Self.ProgressBar1.StepBy(1);  
        end;  
  
    rCronus.Stop;  
    Screen.Cursor := crDefault;  
  
    iTempo := rCronus.ElapsedMilliseconds div 1000;  
    ShowMessage(Format('Processado em [%d] segundos', [iTempo]));  
end;
```

Por fim, vamos codificar o `btnSleep` que também irá simular um processamento pesado com duração de cinco segundos:

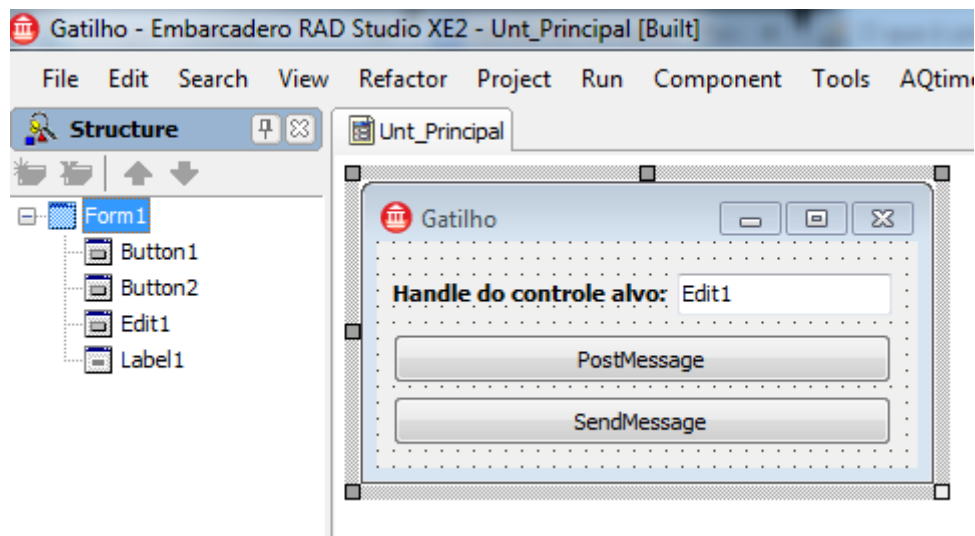
```
procedure TForm1.btnSeepClick(Sender: TObject);  
begin  
    Sleep(5000);  
end;
```

Vamos agora nos concentrar no `Button1`. Executando o software percebemos que o `Caption` do `Button1` muda a cada nova execução do programa. Este é o valor que o Windows atribui ao `handle` deste botão, o que acontece no momento da sua criação. Clicando neste botão, o `Memo1` recebe uma nova linha com o número da execução.

Programa gatilho

Vamos agora desenvolver um programa que terá como propósito “clicar” no `Button1` do programa alvo. Para isso nós iremos enviar uma mensagem de clique para este botão, uma vez que teremos em mãos o `handle` dele.

O programa gatilho será chamado de “Gatilho.exe” e terá a seguinte estrutura:



Existem dois comandos possíveis para se enviar uma mensagem: `PostMessage()` e `SendMessage()`. Salienta-se que não estamos falando de comunicação TCP/IP.

A diferença básica é que a instrução `PostMessage()` coloca a mensagem na pilha e retorna imediatamente. Já o `SendMessage()` só retorna quando o comando é processado. Esses conceitos serão muito importantes, em especial quando falarmos de comunicação entre processos.

Vamos então codificar o `Button1` para executar o `PostMessage()`:

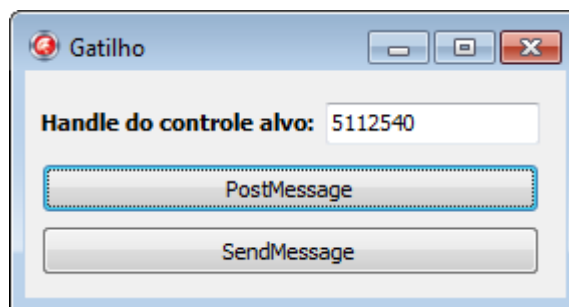
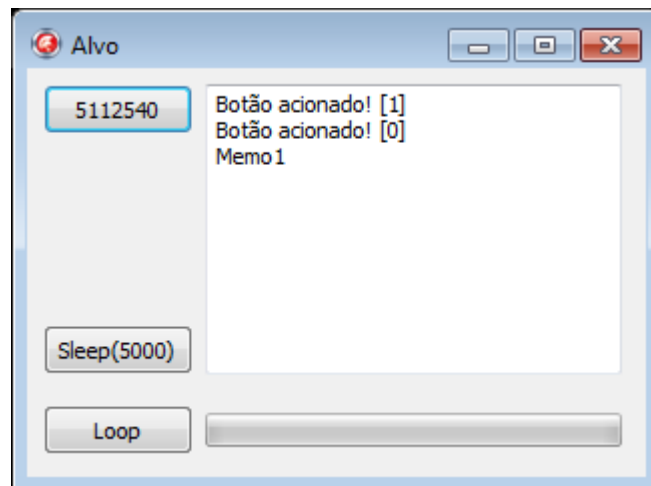
```
procedure TForm1.Button1Click(Sender: TObject);
var
  hBotaoAlvo: NativeUInt;
begin
  hBotaoAlvo := StrToIntDef(Self.Edit1.Text, 0);
  PostMessage(hBotaoAlvo, BM_CLICK, 0, 0);
  ShowMessage('Comando enviado!');
end;
```

E codificar o `Button2` para executar o `SendMessage()`:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  hBotaoAlvo: NativeUInt;
begin
  hBotaoAlvo := StrToIntDef(Self.Edit1.Text, 0);
  SendMessage(hBotaoAlvo, BM_CLICK, 0, 0);
  ShowMessage('Comando enviado!');
end;
```

Executando os programas

Vamos acionar o `Alvo.exe` e o `Gatilho.exe` ao mesmo tempo.



No programa Gatilho.exe basta digitar o handle do Button1 do programa Alvo.exe. No exemplo acima este valor é 5112540. Não há, a princípio, diferenças entre o `PostMessage()` e o `SendMessage()`: ambos sinalizam o Button1 do Alvo.exe, como se houvesse sido dado um clique com o mouse.

Porém, a diferença entre os comandos aparece quando clicamos no `btnLoop` do Alvo.exe. Quando o loop programado neste botão entra em ação o programa como um todo para de processar as mensagens do Windows justamente pelo fato do programa estar ocupado em processar este loop.

Se, no Gatilho.exe, executarmos várias vezes o `PostMessage` ao final do loop teremos, de uma só vez, várias linhas no Memo1. Porém, o `SendMessage` só retornará quando a mensagem finalmente for processada.

Tornando o programa mais responsivo

Vamos melhorar nosso programa Alvo.exe: Perceba que quando clicamos no `btnLoop` ou até mesmo no `btnSleep` a usabilidade do programa fica comprometida uma vez que não conseguimos clicar no Button1 e ter uma resposta imediata.

Na mesma unidade do `TForm1` vamos criar uma classe descendente da `TThread` e transferir para esta o processamento do `btnLoop`:

```
TLoopThread = class(TThread)
private
    FTempoTranscorrido: Integer;
    procedure ZerarProgressBar;
    procedure IncrementarProgressBar;
protected
    procedure Execute; override;
public
    property TempoTranscorrido: Integer read FTempoTranscorrido;
end;
```

Segue o código do método `Execute`:

```
procedure TLoopThread.Execute;
var
    rCronus: TStopwatch;
    i      : Integer;
    iTempo : Integer;
begin
    Self.FTempoTranscorrido := -1;
    rCronus := TStopwatch.StartNew;

    Self.ZerarProgressBar;
```

```
for i := 1 to 15 do
begin
    Sleep(1000);
    Self.IncrementarProgressBar;
end;

rCronus.Stop;
iTempo := (rCronus.ElapsedMilliseconds div 1000);
Self.FTempoTranscorrido := iTempo;
end;
```

Vamos ao código do método `ZerarProgressBar()`:

```
procedure TLoopThread.ZerarProgressBar;
begin
    Form1.ProgressBar1.Max := 15;
    Form1.ProgressBar1.Position := 0;
end;
```

E por fim o código do método `IncrementarProgressBar()`:

```
procedure TLoopThread.IncrementarProgressBar;
begin
    Form1.ProgressBar1.StepBy(1);
end;
```

Com tudo isto o código do `btnLoop` será alterado também, passando a invocar o thread:

```
procedure TForm1.Button2Click(Sender: TObject);
var
    oThread : TLoopThread;
begin
    Self.Button2.Enabled := False;

    oThread := TLoopThread.Create(True);
    oThread.OnTerminate := Self.EventOnTerminate;
    oThread.FreeOnTerminate := True;
    oThread.Start;
end;
```

Para que o manipulador de evento `OnTerminate` funcione adequadamente, declare em `TForm1` o seguinte método:

```
procedure TForm1.EventOnTerminate(Sender: TObject);
var
    iTempo: Integer;
begin
    iTempo := TLoopThread(Sender).TempoTranscorrido;
    ShowMessage(Format('Loop executado em [%d] segundos', [iTempo]));
    Self.btnLoop.Enabled := True;
```

```
end;
```

Executando-se, enfim, o programa Alvo.exe, verificamos que ele está mais responsivo: mesmo com o thread em andamento, conseguimos interagir com o programa. Inclusive o programa Gatilho.exe consegue enviar as mensagens tanto com `PostMessage()` quanto com `SendMessage()`: o retorno é imediato em qualquer cenário pois o thread principal do programa Alvo.exe está livre.

Métodos `Synchronize()` e `Queue()`

O programa Alvo.exe parece muito bom, mas esconde uma armadilha. O thread `TLoopThread` está manipulando diretamente um elemento da VCL: o `ProgressBar1`.

Os componentes da VCL não são, por definição, *thread safe*. Este termo indica que o recurso em questão (no nosso caso o `ProgressBar1`) estará a salvo se duas ou mais threads a manipularem ao um só tempo.

Talvez nunca ocorra um problema. Talvez todo o sistema fique comprometido. Na maioria das vezes não vale a pena arriscar.

Outra questão é a conveniência em se atrasar o processamento do thread por conta de aspectos visuais. Certamente que dar um retorno visual é importante, mas seria o caso de atrasar o processamento por conta disto? Nosso thread deve ter a duração de 15 segundos, mas se durante o processamento o botão `btnSleep` for acionado o programa não processará as mensagens referentes ao `ProgressBar` e por isso o nosso thread sofre um atraso.

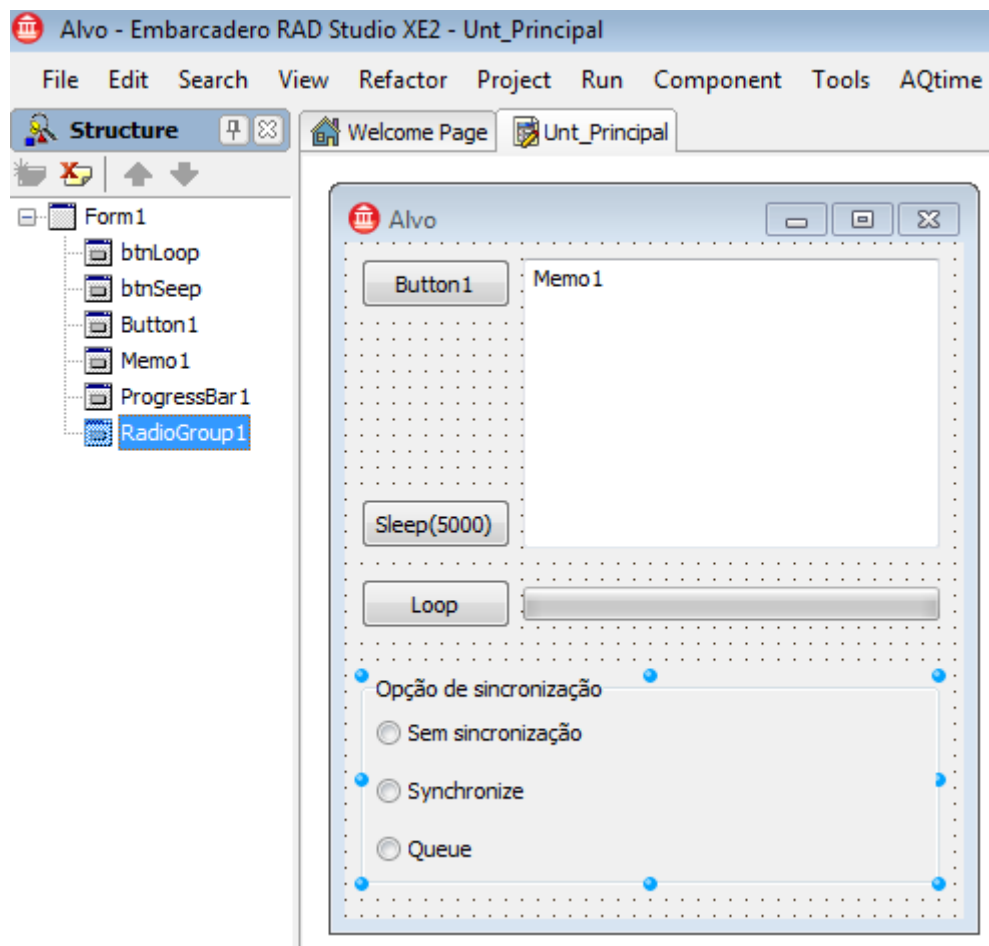
Sempre é aconselhável delegar ao thread principal a manipulação de um elemento VCL, como o `ProgressBar`. Para isso a classe `TThread` nos disponibiliza o método `Synchronize()` bem como o método `Queue()`.

Na literatura em geral vemos muitas referências ao método `Synchronize()`. Isso ocorre pelo fato do `Queue()` ter sido disponibilizado apenas a partir do Delphi 2010. Porém, há diferenças significativas entre essas duas opções.

Ambos os métodos tem funcionamento similar: Passamos um método simples como argumento para o `Synchronize()` ou o `Queue()` e eles delegam ao thread principal a execução deste método. Porém, o `Synchronize()` só retorna quando o método for executado ao passo que o `Queue()` retorna imediatamente.

Adequando o sistema

Vamos adequar o programa Alvo.exe e verificar o impacto de cada decisão. Para isso, vamos colocar um `TRadioGroup` conforme a imagem seguinte:



Em seguida vamos adequar o código do método `Execute()` do thread:

```
procedure TLoopThread.Execute;
var
  rCronus: TStopwatch;
  i       : Integer;
  iTempo : Integer;
begin
  Self.FTempoTranscorrido := -1;
  rCronus := TStopwatch.StartNew;

  case Form1.RadioGroup1.ItemIndex of
    0:
      Self.ZerarProgressBar;
    1:
      Self.Synchronize(Self.ZerarProgressBar);
```

```

2:
  Self.Queue(Self.ZerarProgressBar);
end;

for i := 1 to 15 do
begin
  Sleep(1000);
  case Form1.RadioGroup1.ItemIndex of
    0:
      Self.IncrementarProgressBar;
    1:
      Self.Synchronize(Self.IncrementarProgressBar);
    2:
      Self.Queue(Self.IncrementarProgressBar);
  end;
end;

rCronus.Stop;
iTempo := (rCronus.ElapsedMilliseconds div 1000);
Self.FTempoTranscorrido := iTempo;
end;

```

O tempo esperado para a execução do thread é de 15 segundos. Faça vários testes e para cada opção clique no botão `btnSleep`. Estes são os cenários esperados:

| | Não clicando em <code>btnSleep</code> | Clicando em <code>btnSleep</code> |
|--------------------------|--|--------------------------------------|
| Sem sincronização | 15 segundos | 19~20 segundos |
| Synchronize | 15 segundos | 19~20 segundos |
| Queue | 15 segundos | 15 segundos |

Ou seja, no nosso propósito o `Queue()` se saiu melhor. O `ProgressBar1` até que congelou durante a execução do `btnSleep`, porém o thread continuou em andamento e o `ProgressBar` se recuperou ao final do `btnSleep`.

Importante notar que se ao invés de um `TProgressBar` fosse um `TGauge` os cenários seriam ligeiramente diferentes o que denota que não há uma regra: deve-se estudar caso a caso.

| | Não clicando em <code>btnSleep</code> | Clicando em <code>btnSleep</code> |
|--------------------------|--|--------------------------------------|
| Sem sincronização | 15 segundos | 15 segundos |
| Synchronize | 15 segundos | 19~20 segundos |
| Queue | 15 segundos | 15 segundos |

Especificamente sobre esta diferença entre o `TProgressBar` e o `TGauge` é que o primeiro é incrementado via mensagem do Windows enquanto que o segundo é desenhado diretamente via `Canvas`. O cenário mais conveniente, de qualquer forma, é com a utilização do `Queue`.

Seção crítica

Problema proposto

Vamos desenvolver uma solução de geração de log. Log é o termo usado quando registramos as ocorrências de um sistema, geralmente em arquivo texto. Se algo der errado consulta-se este arquivo para tentar entender o que aconteceu, quando e como.

A princípio a solução parece bem simples. Basta abrir um arquivo, criar uma nova linha e fechar o arquivo. Mas esta é uma operação de I/O pesada pois envolve acesso a disco. Outro problema a ser levado em consideração é que vários threads irão, potencialmente, escrever no mesmo arquivo gerando uma concorrência e eventualmente causar o erro de I/O 32 (que indica que o arquivo esta em uso). Há um recurso crítico neste caso: o *arquivo texto*. Vamos conferir.

Em um novo projeto VCL (GeradorLOG.exe) vamos criar a seguinte estrutura de unidades:

| | |
|--------------------------------|---|
| Unt_Principal.pas | Interface visual do sistema como um todo |
| Unt_DiversasThreads.pas | Threads que serão acionadas e disputarão o recurso de log |
| Unt_FuncaoLog.pas | Solução protegida de geração de log |

Solução sem proteção

Vamos desenvolver a primeira versão sem maiores preocupações para verificarmos os problemas e posteriormente faremos as alterações necessárias para tornar a solução robusta.

Na unidade `Unt_FuncaoLog` vamos desenvolver uma classe que encapsulará toda a complexidade de gerar o arquivo de LOG. Perceba que é uma classe *singleton* e caso este termo ainda lhe seja desconhecido basta, por enquanto, saber que é um padrão de projeto que nos garante que haverá apenas uma instância da classe em todo o sistema. Com isto, em qualquer parte do programa, poderemos simplesmente invocar o método de geração de log sem a preocupação de instanciar ou liberar o objeto.

O código da Listagem 1 contém toda a unidade codificada e comentada.

Listagem 1 – Código da unidade Unt_FuncaoLog

```
unit Unt_FuncaoLog;

interface

type

    /// <summary>
    /// Classe responsável pela geração de LOG
    /// </summary>
    TGeraLog = class
    private
        /// <summary>
        /// Indica o arquivo em que o LOG será gerado, no caso, no mesmo
        /// diretório do executável
        /// </summary>
        const
            C_ARQUIVO = '.\arquivo_log.txt';
        /// <summary>
        /// Instância única do objeto
        /// </summary>
        class var FInstance: TGeraLog;
        /// <summary>
        /// Método responsável por instanciar o objeto singleton
        /// </summary>
        class function GetInstancia: TGeraLog; static;
    public
        /// <summary>
        /// Deleta o arquivo de LOG caso o mesmo exista
        /// </summary>
        procedure AfterConstruction; override;
        /// <summary>
        /// Método que efetivamente gera o LOG
        /// </summary>
        /// <param name="AReferencia">Referência à thread
        /// chamadora</param>
        /// <param name="AData">Date e hora da geração do LOG</param>
        /// <param name="ATextoLog">Texto a ser escrito no arquivo de
        /// log</param>
        procedure GerarLog(const AReferencia: string; const AData:
            TDateTime; const ATextoLog: string);
        /// <summary>
        /// Referência à instância singleton
        /// </summary>
        class property Instancia: TGeraLog read GetInstancia;
    end;

implementation

uses
    System.SysUtils;

{ TGeraLog }

procedure TGeraLog.AfterConstruction;
begin
    inherited;
    DeleteFile(Self.C_ARQUIVO);
```

```
end;

procedure TGeraLog.GerarLog(const AReferencia: string; const AData:
TDateTime; const ATextoLog: string);
var
  _arquivo : TextFile;
  sNovaLinha: string;
begin
  sNovaLinha := Format('%s|%s|%s', [AReferencia, DateTimeToStr(AData),
ATextoLog]);

  AssignFile(_arquivo, Self.C_ARQUIVO);
  if (FileExists(Self.C_ARQUIVO)) then
  begin
    Append(_arquivo);
  end else begin
    Rewrite(_arquivo);
  end;

  Writeln(_arquivo, sNovaLinha);

  CloseFile(_arquivo);
end;

class function TGeraLog.GetInstancia: TGeraLog;
begin
  if not (Assigned(FInstance)) then
  begin
    FInstance := TGeraLog.Create;
  end;
  Result := FInstance;
end;

initialization

finalization

TGeraLog.Instancia.Free;

end.
```

Observe no método `GerarLog()` que estamos utilizando as funções nativas da RTL para a escrita em arquivo texto que são muito mais eficientes do que qualquer outra abordagem, como por exemplo abrir o arquivo em um `TStringList`, adicionar a nova linha para daí salvar o arquivo, inteiro, de novo.

Mas o nosso software é multi-thread e a Listagem 2 contém o código da unidade `Unt_DiversasThreads` que possui uma classe derivada de `TThread`. Esta classe não faz nada útil a não ser concorrer pelo recurso de geração de LOG. Portanto ela representa qualquer outro thread candidato a concorrer por este recurso.

Listagem 2 – Código da unidade Unt_DiversasThreads

```
unit Unt_DiversasThreads;

interface

uses
  System.Classes, Unt_FuncaoLog;

type
  /// <summary>
  /// Thread que disputará o recurso de geração de LOG
  /// </summary>
  TDiversaThread = class(TThread)
  private
    FReferencia : string;
    FDescricaoErro: string;
  public
    /// <summary>
    /// Rotina a ser executada pelo thread que eventualmente
    /// gerará uma linha no arquivo de LOG
    /// </summary>
    procedure Execute; override;
    /// <summary>
    /// Referência que será escrito no arquivo
    /// de LOG para sabermos de onde veio a linha
    /// </summary>
    property Referencia: string read FReferencia write FReferencia;
    /// <summary>
    /// Caso ocorra um erro durante a execução do thread
    /// o erro poderá ser consultado nesta propriedade
    /// </summary>
    property DescricaoErro: string read FDescricaoErro;
  end;

implementation

uses
  System.SysUtils;

{ TDiversaThread }

procedure TDiversaThread.Execute;
var
  bGerarLog: Boolean;
begin
  inherited;
  try
    // Loop enquanto o thread não for finalizado
    while not(Self.Terminated) do
    begin
      //Faz com que não haja um consumo elevado de CPU
      Sleep(10);

      //Sorteia um número e verifica se o resto da
      //divisão por dois é zero
      bGerarLog := (Random(1000000) mod 2) = 0;
```

```

    if (bGerarLog) then
    begin
      //Invoca o método de geração de LOG
      TGerarLog.Instancia.GerarLog(Self.FReferencia, Now, 'O rato
      roeu a roupa do Rei de Roma');
    end;
  end;
except
  on E: EInOutError do
  begin
    Self.FDescricaoErro := Format('Erro de I/O #d - %s',
    [E.ErrorCode, SysErrorMessage(E.ErrorCode)]);
  end;
  on E: Exception do
  begin
    Self.FDescricaoErro := Format('( %s) - %s', [E.ClassName,
    E.Message]);
  end;
end;
end;
end.

```

Perceba que estamos dando um tratamento especial à exceção `EInOutError` pois esperamos justamente que dê erros no momento da escrita.

Por fim vamos à unidade principal do sistema, a `Unt_Principal`, que contém o formulário e terá a seguinte estrutura:

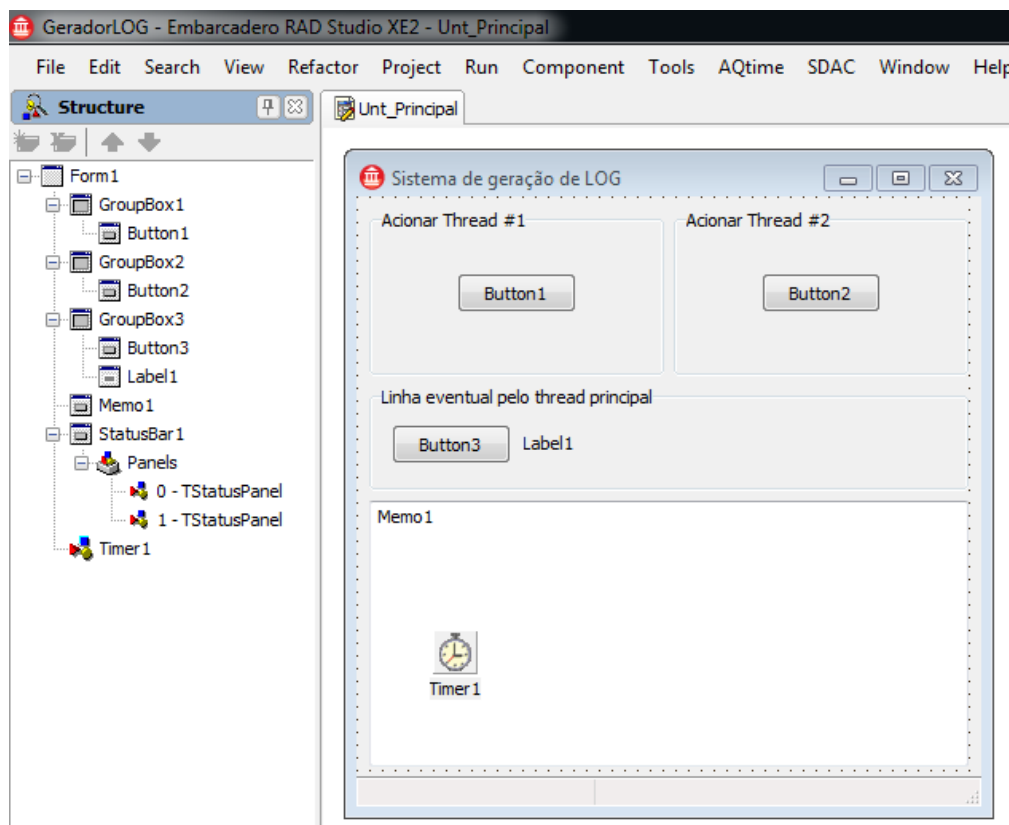


Figura 1 – Proposta de lay-out do aplicativo de estudo

A ideia básica do aplicativo é iniciar dois threads concorrentes (TDiversaThread) e eventualmente concorremos pelo recurso de geração de LOG pelo thread principal. Vamos aos códigos.

Para termos controle sobre os objetos vamos declarar no escopo privado de TForm1 os seguintes elementos:

Listagem 3 – Escopo privado da classe TForm1

```
{...}
private
  /// <summary>
  /// Instância do primeiro objeto
  /// </summary>
  FThread1: TDiversaThread;
  /// <summary>
  /// Instância do segundo objeto
  /// </summary>
  FThread2: TDiversaThread;
  /// <summary>
  /// Método que será associado ao evento OnTerminate de cada thread
  /// </summary>
  procedure EventoFinalizacaoThread(Sender: TObject);
  /// <summary>
  /// Método responsável por alimentar o memo com eventuais erros
  /// </summary>
  procedure AlimentarMemoErro(const AReferencia: string; const
    AErro: string);
```

Listagem 4 – Métodos “EventoFinalizacaoThread” e “AlimentarMemoErro”

```
procedure TForm1.EventoFinalizacaoThread(Sender: TObject);
var
  oThread: TDiversaThread;
begin
  oThread := TDiversaThread(Sender);
  Self.AlimentarMemoErro(oThread.Referencia, oThread.DescricaoErro);
end;

procedure TForm1.AlimentarMemoErro(const AReferencia, AErro: string);
var
  sLinha: string;
begin
  sLinha := Format('%s -> %s', [AReferencia, AErro]);
  Memo1.Lines.Insert(0, sLinha);
end;
```

Quando desenvolvemos em multi-thread corremos riscos maiores de causar vazamento de memória, portanto se faz necessário ativar a flag ReportMemoryLeaksOnShutdown para acompanharmos se está havendo ou

não este problema. Um bom lugar para fazer isso é no evento `OnCreate()` do formulário:

Listagem 5 – Evento `OnCreate()` do `TForm1`

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ReportMemoryLeaksOnShutdown := True;
end;
```

No temporizador vamos indicar na barra de status a hora atual a cada segundo com o propósito de verificar se o sistema esta congelado. É apenas um indicativo visual que nos serve neste momento:

Listagem 6 – Evento `OnTimer()` do `Timer1`

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    StatusBar1.Panels[0].Text := DateTimeToStr(Now);
end;
```

Agora os códigos do `Button1` e `Button2` onde serão criados e acionados dois threads.

Listagem 7 – Eventos `OnClick` de `Button1` e `Button2`

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Self.Button1.Enabled := False;
    Self.FThread1 := TDiversaThread.Create(True);
    Self.FThread1.Referencia := 'THREAD #1';
    Self.FThread1.OnTerminate := Self.EventoFinalizacaoThread;
    Self.FThread1.Start;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Self.Button2.Enabled := False;
    Self.FThread2 := TDiversaThread.Create(True);
    Self.FThread2.Referencia := 'THREAD #2';
    Self.FThread2.OnTerminate := Self.EventoFinalizacaoThread;
    Self.FThread2.Start;
end;
```

Segue agora o código do `Button3`, onde teremos um loop disputando o recurso de log. Perceba que se toma o mesmo cuidado em relação às exceções que podem ser geradas:

Listagem 8 – Evento OnClick de Button3

```
procedure TForm1.Button3Click(Sender: TObject);
var
  i      : Integer;
  rCronus: TStopwatch;
begin
  try
    rCronus := TStopwatch.StartNew;
    for i := 1 to 1000 do
      begin
        TGeralog.Instancia.GerarLog('MAIN THREAD', Now, 'Três pratos de
        trigo para três tigres tristes');
      end;
    rCronus.Stop;

    Self.Label1.Caption := Format('Processado em: %d milisegundos',
    [rCronus.ElapsedMilliseconds]);
  except
    on E: EInOutError do
      begin
        Self.AlimentarMemoErro('MAIN THREAD', Format('Erro de I/O #%d -
        %s', [E.ErrorCode, SysErrorMessage(E.ErrorCode)]));
      end;
    on E: Exception do
      begin
        Self.AlimentarMemoErro('MAIN THREAD', Format('( %s) - %s',
        [E.ClassName, E.Message]));
      end;
    end;
  end;
end;
```

Por fim, no evento OnDestroy do Form1 vamos liberar adequadamente os threads criados:

Listagem 9 – Evento OnDestroy do Form1

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  if Assigned(Self.FThread1) and not(Self.FThread1.Finished) then
    begin
      Self.FThread1.Terminate;
      Self.FThread1.WaitFor;
    end;

  if Assigned(Self.FThread2) and not(Self.FThread2.Finished) then
    begin
      Self.FThread2.Terminate;
      Self.FThread2.WaitFor;
    end;

  if Assigned(Self.FThread1) then
    begin
      Self.FThread1.Free;
    end;

  if Assigned(Self.FThread2) then
```

```
begin  
    Self.FThread2.Free;  
end;  
end;
```

Estamos tirando proveito de uma propriedade da classe `TThread` que é a **Finished**. Ela nos indica se o thread foi finalizado (valor *true*) ou não. Um thread deixa de estar em execução quando sai do método `Execute`.

Acionando o nosso aplicativo conseguimos clicar no `Button3` várias vezes seguidas. Mas recebemos um erro se acionarmos o `Button1` e logo em seguida o `Button3`:

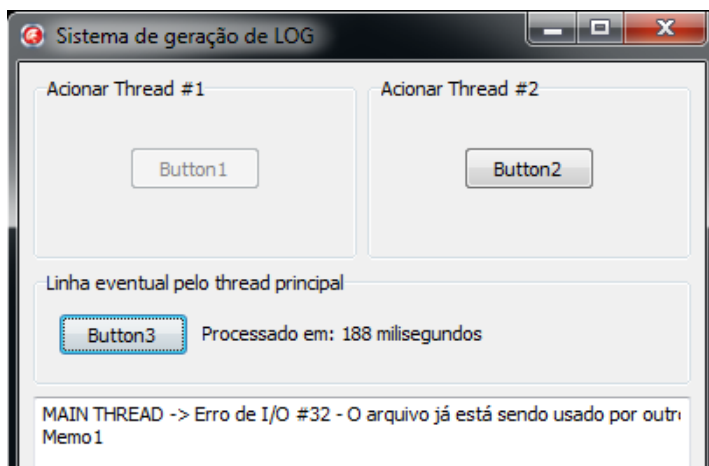


Figura 2 – Evidência do erro de I/O 32

E somente o acaso é que determinará onde e quando ocorrerá o erro. No exemplo acima foi no thread principal. E o pior, pode ser que o erro demore a aparecer por conta de uma conjunção de fatores que, de novo, somente o acaso determinará.

Também não conseguiremos acionar os dois threads ao mesmo tempo. Em algum momento um dos threads irá sofrer a exceção e irá terminar como mostra a Figura 3:

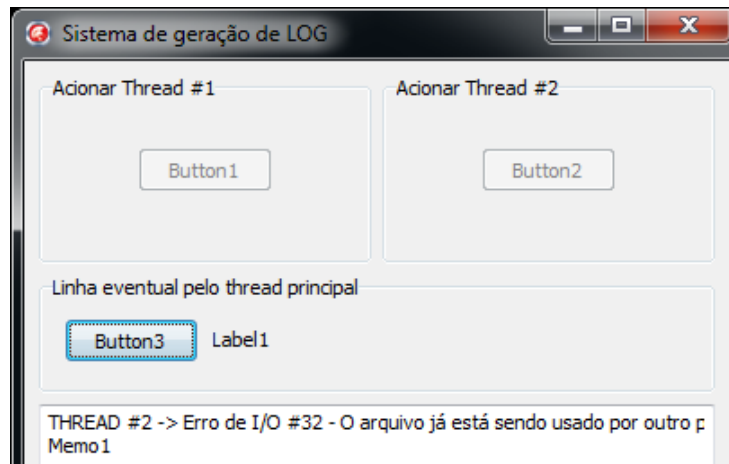


Figura 3 – Evidência do erro de I/O 32

No diretório do executável será criado o “arquivo_log.txt” e poderemos verificar qual thread está efetivamente gerando log. Percebemos então que o nosso aplicativo está inconsistente e não confiável. Precisamos melhorar isso.

Cenário Atual versus Cenário Pretendido

Vamos então lançar mão de um recurso da API do Windows que é a seção crítica. Mas antes de continuarmos cabe-nos uma ótima pergunta: “Porque não usar o *Synchronize* ou até mesmo o *Queue* apresentados na seção anterior?”.

A resposta mais simples é que não estamos lidando com um recurso da VCL. Mas mesmo assim poderíamos usar o *Synchronize()*. Mas se assim o fizéssemos os threads esperariam muito mais tempo que o necessário para gerar o log, atrasando todo o nosso processo, pois entraríamos em uma fila que em nada tem a ver com o nosso problema.

Quando se trata de um recurso que não seja da VCL, no nosso caso o arquivo texto, e temos domínio sobre os threads que irão manipular este recurso, a seção crítica é a melhor alternativa.

Ela cria uma espécie de semáforo, liberando a passagem para um thread e fechando para outros.

Temos o seguinte cenário no momento:

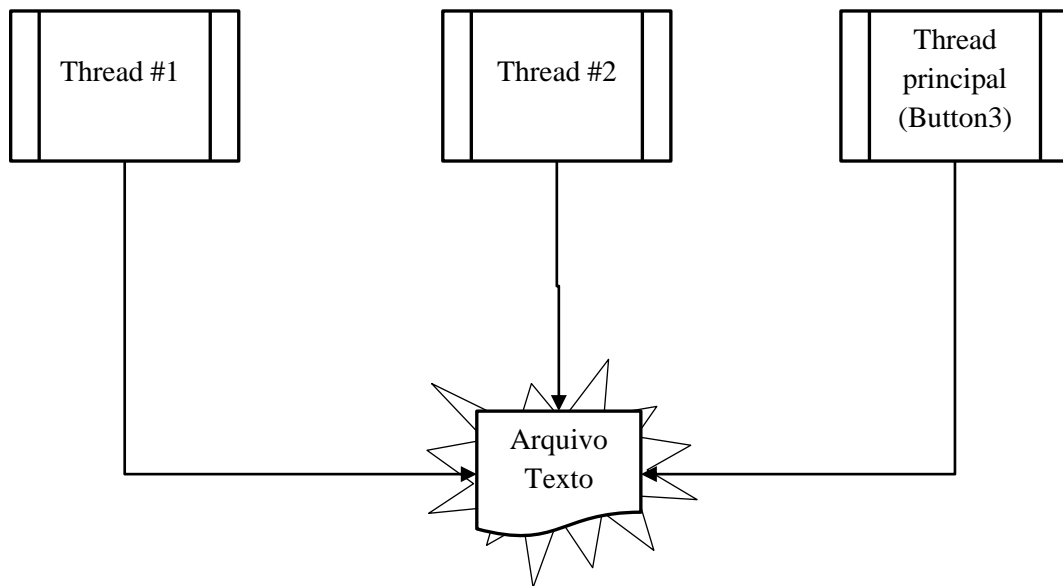


Figura 4 – Cenária sem seção crítica

Com a adoção da seção crítica teremos o cenário a seguir:

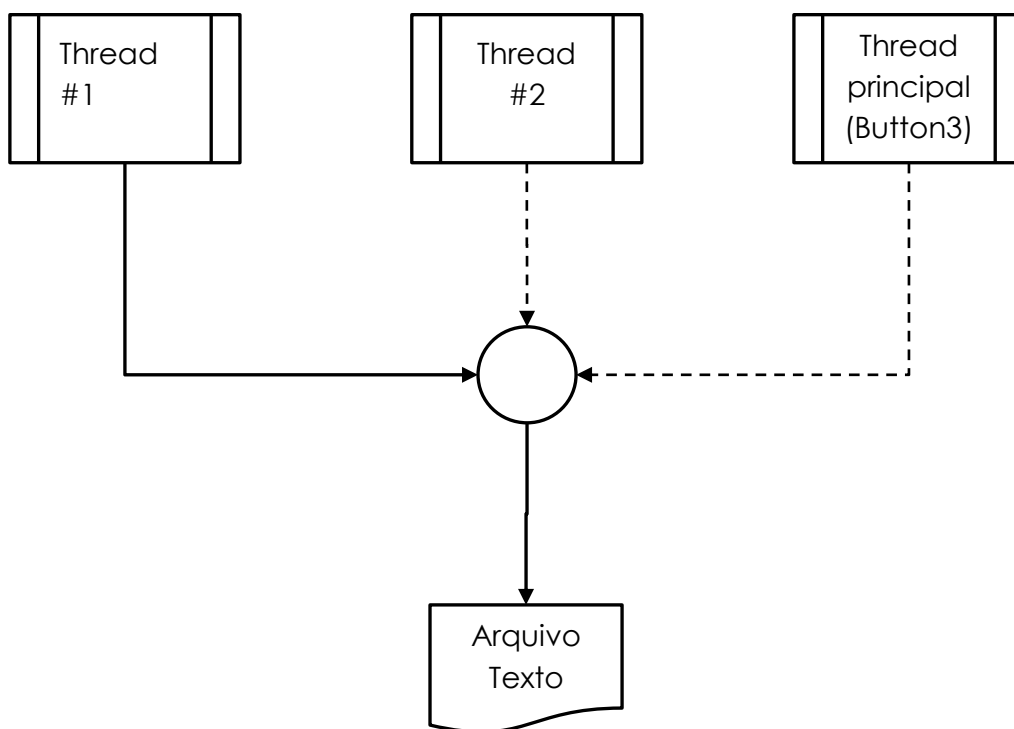


Figura 5 – Cenária com seção crítica

Seção Crítica

Para atingir o cenário desejado vamos à unidade `Unt_FuncaoLog` pois é lá que está o trecho “crítico” do nosso sistema, que é a escrita do log.

Vamos então declarar um novo atributo na classe `TGeraLog` chamado `FCritical` no escopo privado da classe:

Listagem 10 – Declaração do atributo de seção crítica na classe `TGeraLog`

```
/// <summary>
/// Seção crítica para o acesso ao arquivo de log
/// </summary>
FCritical: TCriticalSection;
```

A seção crítica está encapsulada na classe `TCriticalSection` e esta se encontra na unidade `System.SyncObjs`.

Vamos então achar o melhor lugar para instanciar e liberar este objeto. No nosso exemplo faremos nos métodos `AfterConstruction` e `BeforeDestruction` da classe `TGeraLog`:

Listagem 11 – Declaração dos métodos “`AfterConstruction`” e “`BeforeDestruction`” da classe `TGeraLog`

```
{...}
public
  /// <summary>
  /// Deleta o arquivo de LOG caso o mesmo exista e
  /// cria a seção crítica
  /// </summary>
  procedure AfterConstruction; override;
  /// <summary>
  /// Libera a seção crítica
  /// </summary>
  procedure BeforeDestruction; override;
```

Listagem 12 – Implementação dos métodos “`AfterConstruction`” e “`BeforeDestruction`”

```
procedure TGeraLog.AfterConstruction;
begin
  inherited;
  DeleteFile(Self.C_ARQUIVO);
  Self.FCritical := TCriticalSection.Create;
end;

procedure TGeraLog.BeforeDestruction;
```

```
begin
  inherited;
  Self.FCritical.Free;
end;
```

Por fim, vamos modificar o método GerarLog(). Quando usamos a seção crítica temos que escolher cirurgicamente o local em que a colocaremos para que não haja riscos de colisão entre threads e ao mesmo tempo não atrase mais que o necessário o fluxo de trabalho desses mesmos threads. Também temos que garantir de que o recurso não será acessado sem passar pela seção crítica.

Listagem 13 – Adaptação do método GerarLog para o uso da seção crítica

```
procedure TGenerLog.GerarLog(const AReferencia: string; const AData:
TDateTime; const ATextoLog: string);
var
  _arquivo : TextFile;
  sNovaLinha: string;
begin
  sNovaLinha := Format('%s|%s|%s', [AReferencia, DateTimeToStr(AData),
ATextoLog]);

  //Entra na seção crítica
  Self.FCritical.Enter;
  try
    AssignFile(_arquivo, Self.C_ARQUIVO);
    if (FileExists(Self.C_ARQUIVO)) then
    begin
      Append(_arquivo);
    end else begin
      Rewrite(_arquivo);
    end;

    Writeln(_arquivo, sNovaLinha);

    CloseFile(_arquivo);
  finally
    //Sai da seção crítica
    Self.FCritical.Release;
  end;
end;
```

Basicamente invocamos o método Enter() da TCriticalSection para entrar na seção crítica. Se algum outro thread já estiver nessa seção crítica então a execução ficará parada nesta linha até que a seção crítica seja liberada com o comando Release.

Sugiro sempre colocar o Release no bloco finally, pois se algo der errado todos os outros threads ficarão indefinidamente parados aguardando uma liberação que nunca acontecerá comprometendo todo o sistema.

Com as mudanças feitas podemos testar novamente o nosso aplicativo e verificar que conseguimos iniciar os dois threads bem como clicar no Button3 sem nenhum problema:

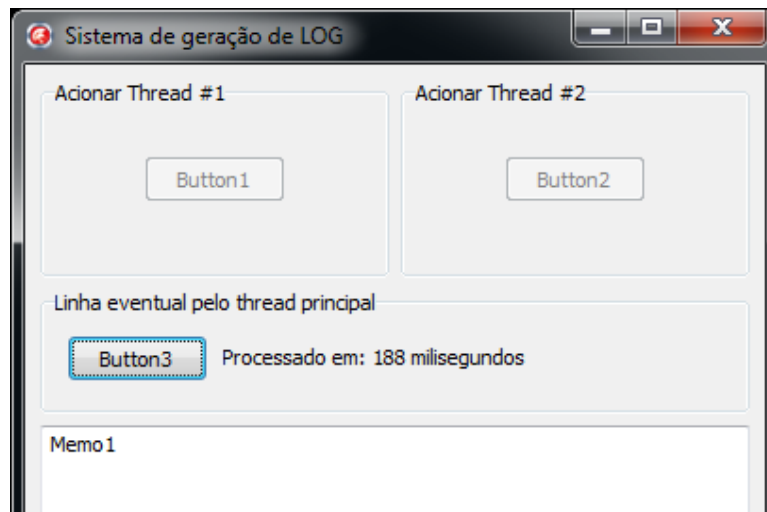


Figura 6 – Evidência de que os dois threads estão em andamento

E, verificando o arquivo de log, percebemos que os dois threads se alternam na escrita do arquivo:

Listagem 14 – Evidência de que os dois threads estão em andamento

```
THREAD #1|12/02/2013 18:05:09|O rato roeu a roupa do Rei de Roma
THREAD #2|12/02/2013 18:05:09|O rato roeu a roupa do Rei de Roma
THREAD #1|12/02/2013 18:05:09|O rato roeu a roupa do Rei de Roma
THREAD #2|12/02/2013 18:05:09|O rato roeu a roupa do Rei de Roma
THREAD #2|12/02/2013 18:05:09|O rato roeu a roupa do Rei de Roma
...
```

Portanto nosso objetivo foi atingido: *fizemos vários threads concorrer por um recurso sem colisão.*

A seção crítica é muito simples de usar, basicamente usa-se o `Enter` e o `Release`, e resolve a maioria das situações de conflito. Existe alguns outros métodos na `TCriticalSection` mas todos eles são apelidos para esses dois primeiros.

Mas vale a pena destacar um que é o `TryEnter()`. Ele tenta entrar na seção crítica e se não conseguir ele retorna imediatamente com `false`. É útil nas situações em que não é conveniente aguardar a liberação da seção crítica.

Vamos criar um novo método na classe `TGeraLog` que ao invés de usar o método `Enter()` da `TCriticalSection` usará o `TryEnter()`:

Listagem 15 – Declaração do método TryGerarLog

```
/// <summary>
/// Método que TENTA gerar uma nova linha de LOG
/// </summary>
/// <param name="AReferencia">Referência à thread
/// chamadora</param>
/// <param name="AData">Date e hora da geração do LOG</param>
/// <param name="ATextoLog">Texto a ser escrito no
/// arquivo de log</param>
function TryGerarLog(const AReferencia: string; const AData:
TDateTime; const ATextoLog: string): Boolean;
```

Listagem 16 – Implementação do método TryGerarLog

```
function TGerarLog.TryGerarLog(const AReferencia: string; const AData:
TDateTime; const ATextoLog: string): Boolean;
var
  _arquivo : TextFile;
  sNovaLinha: string;
begin
  sNovaLinha := Format('%s|%s|%s', [AReferencia, DateTimeToStr(AData),
ATextoLog]);

  // Tenta entrar na seção crítica
  Result := Self.FCritical.TryEnter;
  if (Result) then
  begin
    try
      AssignFile(_arquivo, Self.C_ARQUIVO);
      if (FileExists(Self.C_ARQUIVO)) then
      begin
        Append(_arquivo);
      end else begin
        Rewrite(_arquivo);
      end;

      Writeln(_arquivo, sNovaLinha);

      CloseFile(_arquivo);
    finally
      // Sai da seção crítica
      Self.FCritical.Release;
    end;
  end;
end;
```

Por fim, voltando à nossa interface gráfica, vamos colocar mais um botão (Button4) para acionar este novo método que acabamos de criar:

Listagem 17 – Código do Button4

```
procedure TForm1.Button4Click(Sender: TObject);
var
  bRet: Boolean;
```



```
begin
  bRet := TGerLog.Instancia.TryGerarLog('Button4', Now, 'Testando o
  TryEnter ...');
  ShowMessage(Format('Sucesso? [%s]', [BoolToStr(bRet, True)]));
end;
```

Colocando o sistema em funcionamento e acionando os dois threds eventualmente nos retorna um `False` quando clicamos no `Button4`. É um exemplo didático mais ilustra bem a utilização do `TryEnter`.

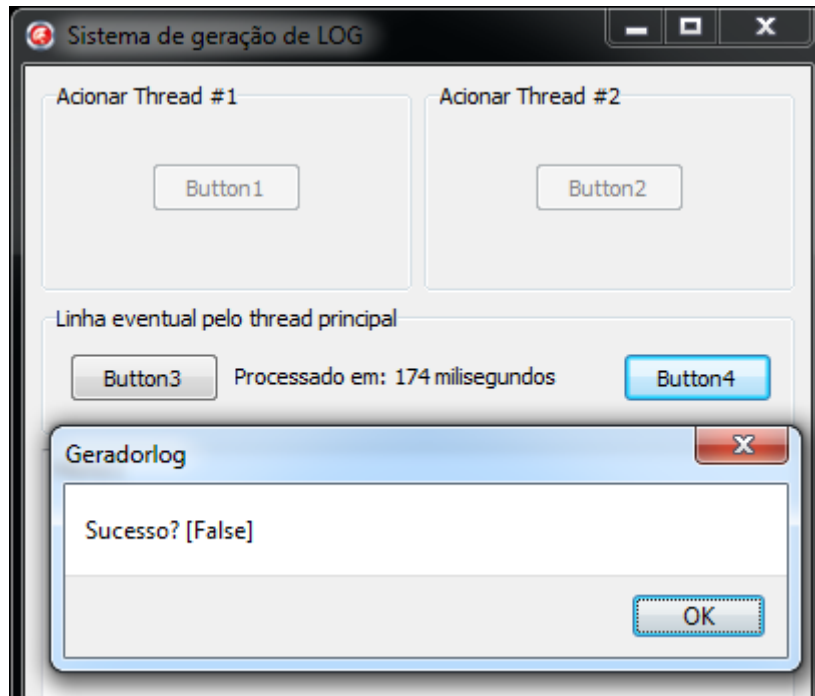


Figura 7 - Momento em que houve uma falha no `TryEnter`

Otimizando um thread

É muito importante se preocupar com a otimização dos threads, em especial em softwares “24 X 7” (*vinte e quatro horas, sete dias da semana*). Esta preocupação com certeza demonstra profissionalismo da nossa parte. São muitos os ganhos, que vão do desempenho do software à economia de recursos que tem, inclusive, um impacto financeiro positivo.

Percebe-se que boa parte dos algoritmos thread consiste em um loop infinito, como no exemplo:

```
procedure TExemploThread.Execute;  
begin  
  inherited;  
  while not (Self.Terminated) do  
  begin  
    // ...  
  end;  
end;
```

E obviamente não tem nada errado com isto. Porém, muitas vezes o propósito principal do thread esta condicionado à alguma situação: ou à presença de um arquivo, ou à quantidade de itens em uma pilha, ou a um clique de um botão e por ai vai. Mas dependendo de como elaborarmos o algoritmo todo o sistema operacional será prejudicado.

Isto porque este loop infinito tomará o tempo dos outros threads e processos e, o que é muito interessante notar, contribuirá para o gasto de energia elétrica. Isto mesmo: podemos tornar o nossos sistemas ecologicamente corretos tomando alguns cuidados. E obviamente torná-los mais eficientes. Você já abordou o seu código por este ponto de vista?

Criaremos um thread que possui uma fila interna de objetos que serão descartados. Em outras palavras, criaremos um “garbage collector” e delegaremos a este thread a liberação dos objetos.

Em um novo projeto VCL vamos criar duas units:

| | | |
|--------------------------|----------|--|
| Unt_Principal.pas | VCL Form | Interface visual do sistema como um todo |
| Unt_Coletor.pas | Unit | Thread com a fila de objetos a serem liberados |

Indo para a unit `Unt_Coletor` vamos codificar o nosso thread como a listagem que segue. Ela está bem comentada e basicamente usa todos os elementos que vimos até aqui. A novidade é o `TObjectQueue` que,

basicamente, é uma fila de objetos no estilo PEPS (Primeiro que Entra, Primeiro que Sai) – ou, se preferir, FIFO (First In, First Out). É uma classe do tipo *Singleton*, por isso dê uma atenção especial aos métodos de classe bem como às seções *initialization* e *finalization* da unit.

```
unit Unt_Coletor;  
  
interface  
  
uses  
    System.Classes, System.Generics.Collections, System.SyncObjs;  
  
type  
  
    /// <summary>  
    /// Classe singleton com o propósito de liberar objetos  
    /// </summary>  
    TColetorDeLixo = class(TThread)  
    private  
        /// <summary>  
        /// Instância única desta classe  
        /// </summary>  
        class var FColetorDeLixo: TColetorDeLixo;  
        /// <summary>  
        /// Liberador da instância única desta classe  
        /// </summary>  
        class procedure ReleaseInstance;  
        /// <summary>  
        /// Instanciador da classe  
        /// </summary>  
        class function GetInstance: TColetorDeLixo; static;  
    private  
        /// <summary>  
        /// Enfilerador dos objetos (FIFO) que serão liberados  
        /// </summary>  
        FFila: TObjectQueue<TObject>;  
        /// <summary>  
        /// Seção crítica para a fila de objetos  
        /// </summary>  
        FSecaoCritica: TCriticalSection;  
        /// <summary>  
        /// Quantidade de objetos liberados  
        /// </summary>  
        FQuantidadeLiberada: NativeUInt;  
        /// <summary>  
        /// Retorna a quantidade de objetos ainda a serem liberados  
        /// </summary>  
        function GetQuantidadeFila: NativeUInt;  
        /// <summary>  
        /// Processamento efetivo da fila de objetos  
        /// </summary>  
        procedure ProcessarFila;  
    public  
        /// <summary>  
        /// Aloca os recursos necessários para o funcionamento da classe  
        /// </summary>  
        procedure AfterConstruction; override;  
        /// <summary>
```

```
/// Desaloca os recursos
/// </summary>
procedure BeforeDestruction; override;
/// <summary>
/// Coloca um objeto na pilha, será invocada pelos outros threads
/// </summary>
procedure ColocarNaPilha(AObjeto: TObject);
/// <summary>
/// Rotina que será, efetivamente executado pelo thread
/// </summary>
procedure Execute; override;
/// <summary>
/// Exposição da instância única desta classe
/// </summary>
class property ColetorDeLixo: TColetorDeLixo read GetInstance;
/// <summary>
/// Indica a quantidade de objetos na fila
/// </summary>
property QuantidadeFila: NativeUInt read GetQuantidadeFila;
/// <summary>
/// Indica a quantidade de objetos já liberado
/// </summary>
property QuantidadeLiberada: NativeUInt read FQuantidadeLiberada;
end;

implementation

uses
  System.SysUtils;

{ TExemploThread }

procedure TColetorDeLixo.AfterConstruction;
begin
  inherited;
  Self.FQuantidadeLiberada := 0;
  Self.FSecaoCritica := TCriticalSection.Create;
  Self.FFila := TObjectQueue<TObject>.Create(True);
end;

procedure TColetorDeLixo.BeforeDestruction;
begin
  inherited;
  Self.FSecaoCritica.Free;
  Self.FFila.Free;
end;

procedure TColetorDeLixo.ColocarNaPilha(AObjeto: TObject);
begin
  Self.FSecaoCritica.Enter;
  try
    Self.FFila.Enqueue(AObjeto);
  finally
    Self.FSecaoCritica.Release;
  end;
end;

procedure TColetorDeLixo.Execute;
var
  iQuantidade : NativeUInt;
```

```
begin
  inherited;
  while not(Self.Terminated) do
  begin
    iQuantidade := Self.FFila.Count;
    if (iQuantidade = 0) then
    begin
      Continue;
    end;

    Self.ProcessarFila;
  end;
end;

class function TColetorDeLixo.GetInstance: TColetorDeLixo;
begin
  if not(Assigned(FColetorDeLixo)) then
  begin
    FColetorDeLixo := TColetorDeLixo.Create(True);
    FColetorDeLixo.Start;
  end;
  Result := FColetorDeLixo;
end;

function TColetorDeLixo.GetQuantidadeFila: NativeUInt;
begin
  Result := Self.FFila.Count;
end;

procedure TColetorDeLixo.ProcessarFila;
var
  i          : Integer;
  oTemp      : TObject;
begin
  Self.FSecaoCritica.Enter;
  try
    for i := 1 to Self.FFila.Count do
    begin
      oTemp := Self.FFila.Extract;

      oTemp.Free;
      Inc(Self.FQuantidadeLiberada);
    end;
  finally
    Self.FSecaoCritica.Release;
  end;

  // Demora artificial para verificarmos as quantidades
  Sleep(250);
end;

class procedure TColetorDeLixo.ReleaseInstance;
begin
  if (Assigned(FColetorDeLixo)) then
  begin
    FColetorDeLixo.Terminate;
    FColetorDeLixo.WaitFor;
    FColetorDeLixo.Free;
    FColetorDeLixo := nil;
  end;
end;
```

```

end;

initialization

finalization

TColetorDeLixo.ReleaseInstance;

end.

```

Esta classe está bem funcional e para verificarmos se está tudo correto vamos desenvolver a nossa interface, que será bem simples, conforme a figura 1:

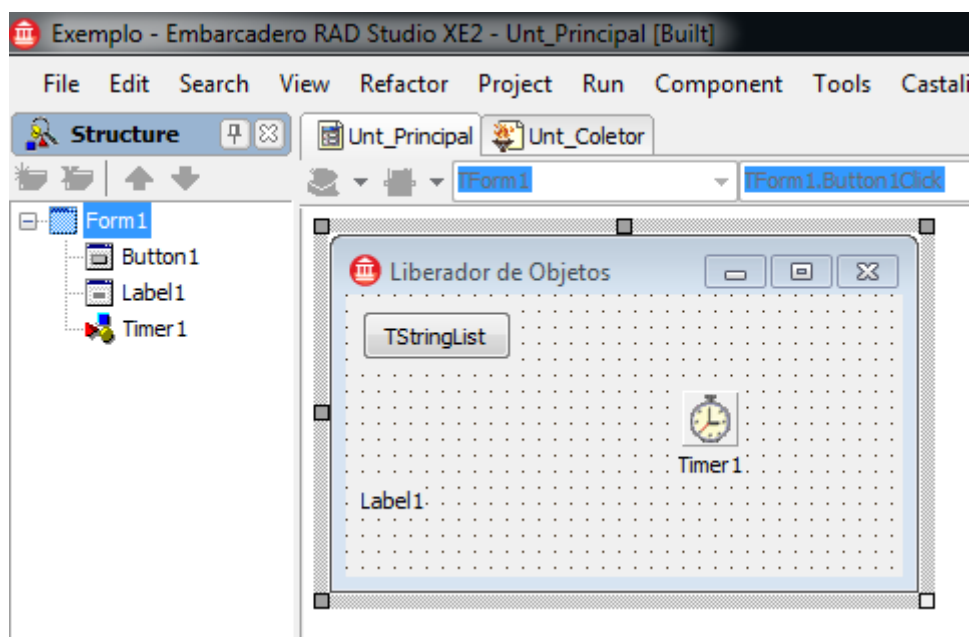


Figura 1 – Disposição dos elementos da interface

Se a classe `TColetorDeLixo` estiver fazendo o papel dela não teremos vazamentos de memória. A forma mais simples de verificar isso é ativando o relatório de Memory Leaks, o que será feito no evento `OnCreate` do formulário:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  ReportMemoryLeaksOnShutdown := True;
end;

```

Segue o código do `Button1`, cujo propósito é instanciar um objeto qualquer e colocar na fila de liberação. Para evidenciar a “eficiência” do coletor de lixo vamos alocar 50.000 objetos, a uma só vez, da classe `TStringList`. Mas perceba que a nossa classe aceita qualquer tipo de objeto:

```

procedure TForm1.Button1Click(Sender: TObject);
var
  oTemp: TStringList;
  i      : Word;
begin
  for i := 1 to 50000 do
  begin
    oTemp := TStringList.Create;

    { ... Algo útil ... }

    Self.Caption := Format('Objeto de #%d',[i]);
    Application.ProcessMessages;

    TColetorDeLixo.ColetorDeLixo.ColocarNaPilha(oTemp);
  end;
end;

```

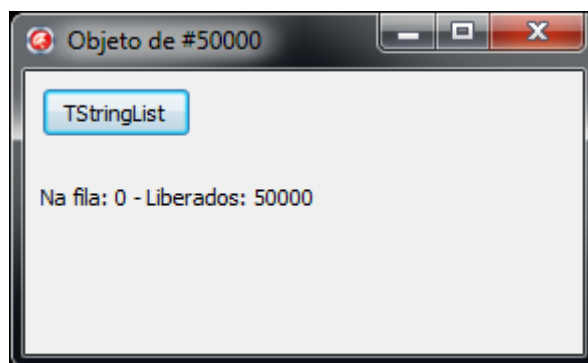
Para acompanharmos a evolução do processo vamos mudar a propriedade Interval do Timer1 para 50 milissegundos e codificar o evento OnTimer da seguinte maneira:

```

procedure TForm1.Timer1Timer(Sender: TObject);
const
  C_QUANT = 'Na fila: %d - Liberados: %d';
begin
  with TColetorDeLixo.ColetorDeLixo do
  begin
    Self.Label1.Caption := Format(C_QUANT, [QuantidadeFila,
    QuantidadeLiberada]);
  end;
end;

```

Ao executarmos a rotina teremos algo parecido com a figura seguinte:



Atingimos o nosso objetivo! Estamos delegando a um thread a liberação de objetos. Ela esta bem projetada, inclusive aplicou-se a seção crítica em um elemento que sofre concorrência: *A fila de objetos*.

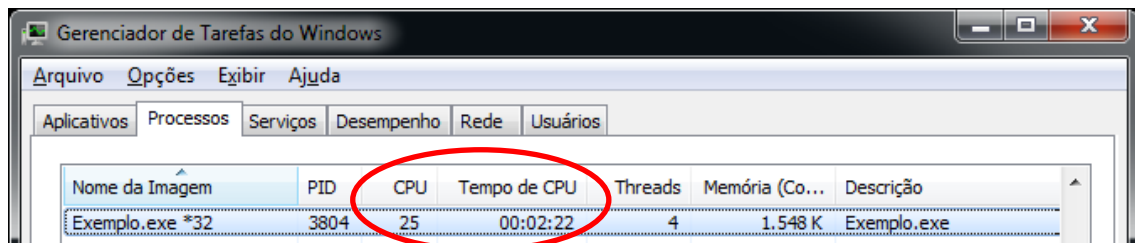
Mas se olharmos a solução mais de perto, vamos perceber que ela não esta otimizada, especialmente neste trecho:

```

{...}
while not(Self.Terminated) do
begin
  iQuantidade := Self.FFila.Count;
  if (iQuantidade = 0) then
  begin
    Continue;
  end;
{...}

```

Este **if..then** esta ocorrendo milhares de vezes por segundo. Se passarmos horas sem clicar no Button1 todo este processamento redundou em desperdício de recursos. Para evidenciar este problema, acione o “Gerenciador de Tarefas do Windows” (CTRL + SHIFT + ESC) e exiba a coluna “Tempo de CPU”. Esta informação consiste no tempo total que o processo tomou da CPU desde o seu início:



Na máquina em que o exemplo foi desenvolvido havia quatro núcleos. Por conta disto o “Exemplo.exe” está usando 25% de toda a capacidade de processamento do equipamento. Isso equivale a um núcleo inteiro! Provavelmente a conta de luz virá um pouco mais alta este mês. Perceba que o “Tempo de CPU” incrementa constantemente.

Devemos, então, tomar alguma medida para diminuir o uso da CPU e deixar o programa mais eficiente. Ciclos “infinitos” tendem a exigir muito da CPU e é isto que esta acontecendo. Vamos às alternativas:

Application.ProcessMessages()

Relembrando o início da apostila, falamos um pouco de como um programa “for Windows” funciona: processando mensagens do sistema operacional.

Independentemente do nosso exemplo atual, vale mencionar que muitas vezes quando o nosso software não responde visualmente a algumas situações podemos lançar mão da instrução `Application.ProcessMessages()`. Na prática isso fará com que o nosso software “pare” o que está fazendo e processe todas as mensagens do Windows que estão represadas.

Isso é muito útil no thread principal, em especial para atualização visual dos controles VCL. Experimente voltar ao código do `Button1` e comentar a linha que contém `Application.ProcessMessages()`. O `Label1` não se atualizará com a mesma frequência.

Mas esta instrução não é útil dentro de um thread, pois causa um atraso desnecessário para a maioria das situações. Afinal, entraríamos em uma longa fila e o thread ficará parado enquanto isso. Além de ser um contrassenso, pois tira o benefício do thread, não resolve o nosso problema atual. Portanto, neste caso, não é uma boa opção.

Sleep(1)

Uma alternativa é colocar um `Sleep(1)` em um ponto estratégico. Isso congela o thread por um milissegundo, baixando drasticamente o uso de CPU, de 25% baixa para 0%.

No caso o código ficaria assim:

```
{...}
while not(Self.Terminated) do
begin
  iQuantidade := Self.FFila.Count;
  if (iQuantidade = 0) then
  begin
    Sleep(1); {← Ponto estratégico}
    Continue;
  end;
{...}
```

Isso resolveu parte do problema. Visualmente não percebemos mais um consumo excessivo de CPU. Se olharmos o “Gerenciador de Tarefas do Windows” os indicadores estarão zerados.

Mas o problema ainda existe. O thread continua passando várias vezes por aquele trecho. Obviamente que o `Sleep()` contribuiu. Talvez se aumentarmos o tempo, para 10 ou até mesmo 100 milissegundos, fiquemos mais confortáveis.

Porém, dependendo da importância do algoritmo, nós criaremos um problema uma vez que haverá um atraso na execução. Considerando um `Sleep(100)`, se entrar um objeto na fila de descarte agora, só daqui a 100 milissegundos é que iremos processar esse objeto. E isto não parece correto.

TEvent

Enfim, a solução ideal para o problema levantado: o `TEvent` é uma classe disponibilizada na unit `System.SyncObjs` e tem o propósito de sinalizar que um “evento” ocorreu. Assim como diversos recursos relacionados aos threads, a classe `TEvent` na verdade é uma camada para a chamada correspondente à API do Windows.

Esta classe é útil para sinalização entre threads e também entre processos. Basicamente um thread (ou mais, se for o caso) aguarda uma sinalização para seguir em frente com o algoritmo.

Instanciando o TEvent

Deve-se tomar um cuidado especial no momento de se instanciar um `TEvent`. Isto porque em um dos parâmetros do construtor é solicitado um nome para este sinalizador.

Este nome deve ser único em todo o sistema operacional. Se dois softwares distintos criarem sinalizadores com o mesmo nome, ambos serão sinalizados. Nenhum problema, obviamente, se for isso o desejado. Mas do contrário os programas se mostrarão instáveis.

Na nossa classe `TColetorDeLixo` vamos declarar um atributo do tipo `TEvent` no escopo privado:

```
private
{...}
/// <summary>
/// Sinalizador para o thread
/// </summary>
FSinalizador: TEvent;
{...}
```

E no método `AfterConstruction` da classe `TColetorDeLixo` vamos instanciar o `TEvent`:

```
{...}
Self.FSinalizador := TEvent.Create(nil, False, True, '_sinalizador');
{...}
```

Aqui cabem algumas explicações importantes sobre os parâmetros do construtor:

| | |
|------------------------|--|
| EventAttributes | Trata-se de um ponteiro para a estrutura <code>_SECURITY_ATTRIBUTES</code> com informações adicionais de |
|------------------------|--|

| | |
|---------------------|---|
| | segurança. O valor nil é adequado para a maioria das necessidades. |
| ManualReset | Indica se o estado do sinalizador será modificado para desligado automaticamente após a sinalização (valor <i>False</i>) ou manualmente, ou seja, pelo programador com o método <code>ResetEvent</code> (valor <i>True</i>) |
| InitialState | Indica o estado inicial do sinalizador. O valor <i>True</i> indicará ligado e o <i>False</i> desligado. |
| Name | É o nome associado ao sinalizador em questão. Como dito anteriormente deve ser um nome único em todo o sistema operacional. Há também a opção de se deixar o nome em branco para se garantir a exclusividade do sinalizador. |
| UseCOMWait | Define o método de espera pela sinalização, sendo o valor <i>False</i> o valor padrão e adequado para a maioria das necessidades. |

Aguardando a sinalização: **WaitFor**

Agora vamos efetivamente usar o sinalizador. No método `Execute` do nosso thread, vamos readequar o código conforme a listagem que segue:

```

procedure TColetorDeLixo.Execute;
var
  eWait: TWaitResult;
begin
  inherited;
  while not(Self.Terminated) do
  begin
    eWait := Self.FSinalizador.WaitFor(INFINITE);

    case eWait of
      wrSignaled:
        begin
          Self.ProcessarFila;
        end;
      wrTimeout:
        ;
      wrAbandoned:
        ;
      wrError:
        ;
      wrIOCompletion:
        ;
    end;
  end;
end;

```

Continuamos com a estrutura **while..do**, mas ao invés de um ciclo insano e infinito, estamos aguardando uma sinalização através do método `WaitFor()`.

Este método pede como argumento um tempo de espera em milissegundos. A não ser que seja realmente necessário definir um tempo específico, a constante `INFINITE` é suficiente. E apesar do nome sugestivo, o valor equivale a 4.294.967.295 milissegundos equivalendo a 49 dias de espera.

Quando a sinalização ocorre o método `WaitFor()` retorna o valor `wrSignaled`, sendo o momento mais adequado para se efetuar o processamento. Quando o tempo de espera chega ao limite retorna-se o valor `wrTimeout`. Esses são os dois retornos mais relevantes neste contexto.

Efetuando a sinalização: `SetEvent`

A primeira parte do problema foi resolvida. Agora precisamos fazer com que este evento seja efetivamente sinalizado. De acordo com o nosso propósito, o melhor momento para isto é na inclusão de um novo objeto na fila de descarte, o que é feito no método `ColocarNaPilha()`.

Então o código do método ficaria assim:

```
procedure TColetorDeLixo.ColocarNaPilha(AObjeto: TObject);
begin
  Self.FSecaoCritica.Enter;
  try
    Self.FFila.Enqueue(AObjeto);
    Self.FSinalizador.SetEvent; { ← Sinalização }
  finally
    Self.FSecaoCritica.Release;
  end;
end;
```

Com isso, nosso thread ficou otimizado e só consome recursos quando efetivamente precisa.

Colocando o software para funcionar não notamos qualquer diferença no comportamento e os indicadores ficam zerados o tempo todo, exceto quando o `Button1` é pressionado, o que é esperado.

Finalizando o thread

Por fim sobrou um último problema: o thread não finaliza no encerramento do software. Isso ocorre por causa do `WaitFor()` dentro do método `Execute`. Precisamos sinalizar este evento em algum momento para que o thread finalize adequadamente.

Existe um método virtual na classe `TThread` que é acionado após a chamado do método `Terminate()`, nos possibilitando então, reagir a este

cenário. Este método é o `TerminatedSet()` e nos basta reescrevê-lo na nossa classe:

```
protected
  /// <summary>
  ///   Aciona o evento para evitar deixar o programa zumbi
  /// </summary>
  procedure TerminatedSet; override;
```

```
procedure TColetorDeLixo.TerminatedSet;
begin
  inherited;
  Self.FSinalizador.SetEvent;
end;
```

Cooperação entre threads

Há um detalhe bastante importante nos exemplos que desenvolvemos até aqui: criamos explicitamente os threads, ou seja, classes funcionais descendentes de `TThread`. Mas há casos especiais em que você desenvolverá um determinado algoritmo sem a consciência de que ele será executado por um thread. Um exemplo imediato é quando desenvolvemos uma classe a ser disponibilizada por um servidor **DataSnap**.

Aos não iniciados: DataSnap é o framework nativo ao Delphi que nos permite desenvolver soluções em *n* camadas. Por fugir ao escopo do artigo não nos aprofundaremos nesta vertente. Basta apenas entender como funciona, grosso modo, o mecanismo do DataSnap e então perceber um problema em potencial.

No DataSnap você codifica as suas *classes remotas* e as disponibiliza no framework. O DataSnap, por sua vez, se utiliza do Indy para a servidão TCP/IP e cada cliente conectado equivale a um thread no servidor. Neste processo, então, os *métodos remotos* são invocados e o resultado do processamento enviado ao cliente solicitante.

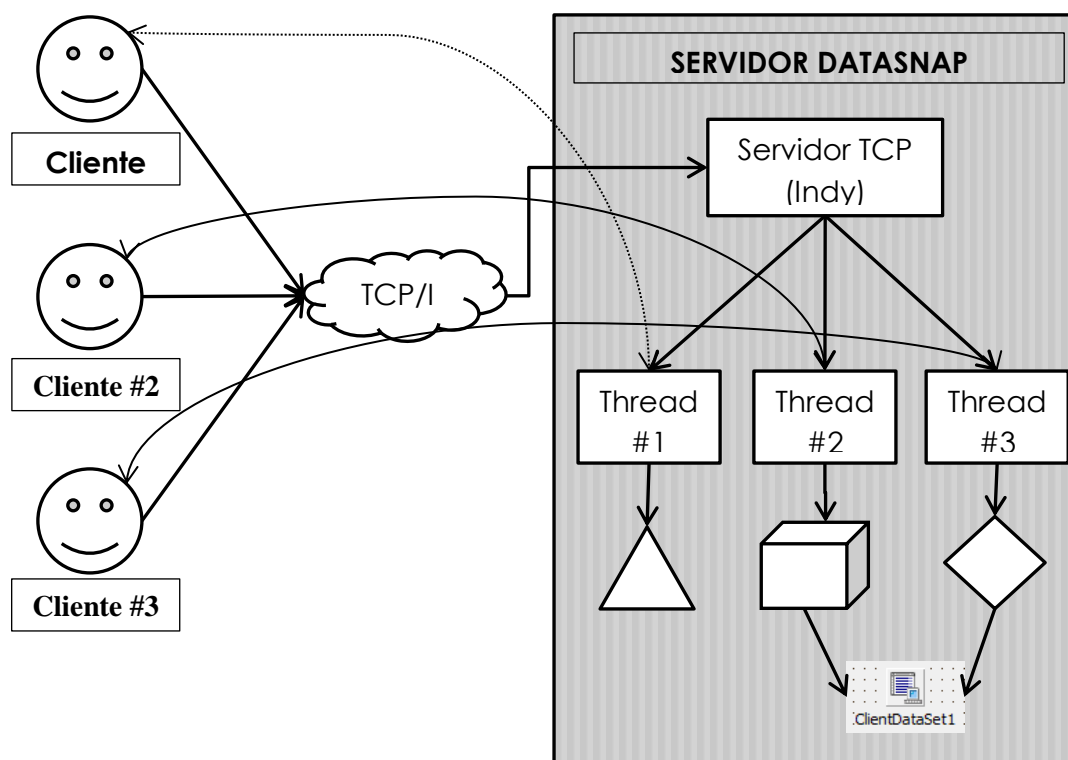


Figura 1 – Esquema simplificado do funcionamento de um servidor DataSnap

Na Figura 1 é mostrado um diagrama onde temos três clientes simultâneos em um servidor DataSnap e consequentemente três threads. Esses threads estão executando três classes remotas distintas sendo que duas delas estão disputando um recurso em comum: o `ClientDataSet1`. Imagine este `ClientDataSet` como uma tabela em memória que possui dados de interesse geral e você decidiu coloca-lo em um `DataModule` central com a intenção de qualquer um usar quando for necessário. Sem as devidas proteções um thread irá atrapalhar o outro, pois poderá deslocar o cursor do `ClientDataSet` para um registro diferente no exato momento da leitura, gerando inconsistências.

A dificuldade que se destaca em todo o cenário descrito até aqui é justamente o de onde declarar uma seção crítica? No `DataModule` provavelmente. Mas não parece muito prático.

TMonitor – Usando como seção crítica

No nosso primeiro exemplo, simularemos uma situação problemática parecida com o cenário descrito anteriormente e em seguida aplicaremos a solução proposta.

Cenário problemático

Criaremos um `ClientDataSet` em memória com a lista dos estados da federação. Ele terá apenas dois campos: `SIGLA` e `NOME`. Em seguida faremos dois threads disputarem este recurso nos apontando um erro de sincronia quando ocorrer. Crie um novo projeto com o seguinte layout:

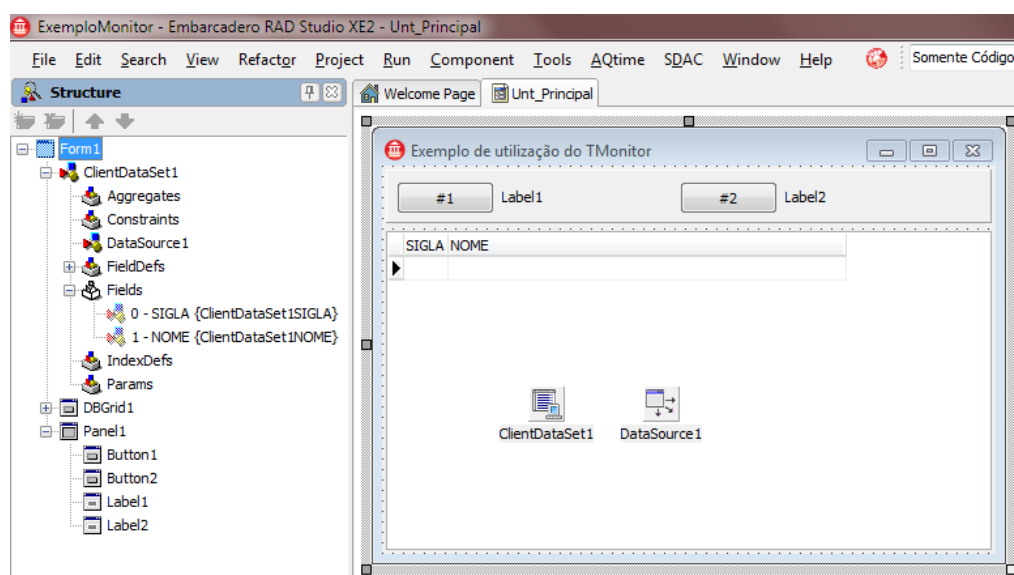


Figura 2 - Proposta de layout de tela

No `ClientDataSet` siga as etapas abaixo para criar e popular a tabela:

- Dê um duplo clique no `ClientDataSet1`;
- Clique com o botão direito do "Fields Editor" e escolha a opção "Add Fields ..."
- Adicione dois campos conforme tabela abaixo:

| Nome | Tipo | Tamanho |
|--------------|----------|---------|
| SIGLA | ftString | 2 |
| NOME | ftString | 50 |

- Clique com o botão direito no `ClientDataSet1` e escolha a opção "Create DataSet"

Vamos agora popular esta tabela com algumas informações. No evento `OnCreate` do formulário vamos codificar conforme a Listagem 1:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with Self.ClientDataSet1 do
  begin
    Open;
    AppendRecord(['AC', 'Acre']);
    AppendRecord(['AL', 'Alagoas']);
    AppendRecord(['AP', 'Amapá']);
    AppendRecord(['AM', 'Amazonas']);
    AppendRecord(['BA', 'Bahia']);
    AppendRecord(['CE', 'Ceará']);
    AppendRecord(['DF', 'Distrito Federal']);
    AppendRecord(['ES', 'Espírito Santo']);
    AppendRecord(['GO', 'Goiás']);
    AppendRecord(['MA', 'Maranhão']);
    AppendRecord(['MT', 'Mato Grosso']);
    AppendRecord(['MS', 'Mato Grosso do Sul']);
    AppendRecord(['MG', 'Minas Gerais']);
    AppendRecord(['PA', 'Pará']);
    AppendRecord(['PB', 'Paraíba']);
    AppendRecord(['PR', 'Paraná']);
    AppendRecord(['PE', 'Pernambuco']);
    AppendRecord(['PI', 'Piauí']);
    AppendRecord(['RJ', 'Rio de Janeiro']);
    AppendRecord(['RN', 'Rio Grande do Norte']);
    AppendRecord(['RS', 'Rio Grande do Sul']);
    AppendRecord(['RO', 'Rondônia']);
    AppendRecord(['RR', 'Roraima']);
    AppendRecord(['SC', 'Santa Catarina']);
    AppendRecord(['SP', 'São Paulo']);
    AppendRecord(['SE', 'Sergipe']);
    AppendRecord(['TO', 'Tocantins']);
  end;
end;
```

Listagem 1 - Populando o `ClientDataSet` com a lista de estados da federação

Com isso teremos a listagem dos estados da federação sempre que o sistema entrar em execução.

Vamos criar um thread que irá localizar um registro pela sigla, dar uma pausa e depois conferir se o registro atual é o que ele espera. Com isso estaremos simulando um processamento ligeiramente demorado que precisa das informações do registro.

Na mesma unit do formulário, declare a seguinte classe na seção type:

```
TLocalizaUF = class(TThread)
private
  FConteudoLabel: string;
  FLabelError: TLabel;
  procedure EscreverLabel;
protected
  procedure Execute; override;
public
  constructor Create(ALabelError: TLabel);
end;
```

Listagem 2 - Escopo do thread

Codifique o construtor conforme a listagem a seguir:

```
constructor TLocalizaUF.Create(ALabelError: TLabel);
begin
  inherited Create(True); //Cria o thread suspenso
  Self.FLabelError := ALabelError;
end;
```

Listagem 3 - Construtor da classe TLocalizaUF

Em seguida vem o código do método `EscreverLabel()` que esta tendo um tratamento especial uma vez que manipulará elementos da VCL e portanto é conveniente que o faça de forma sincronizada com o thread principal:

```
procedure TLocalizaUF.EscreverLabel;
begin
  Self.FLabelError.Caption := Self.FConteudoLabel;
end;
```

Listagem 4 - Método EscreverLabel da classe TLocalizaUF

Agora o método execute do nosso thread. A ideia básica é, aleatória e continuamente, localizar alguns estados pela sigla e interromper o processo em caso de erro, mostrando este erro no label correspondente.

```
procedure TLocalizaUF.Execute;
const
  C_MAX = 5;
  C_UF: array [0..C_MAX-1] of string = ('SP', 'BA', 'RJ', 'SE', 'AM');
var
  iPos      : Byte;
  sUF       : string;
```

```
iQuantidade: NativeUInt;  
begin  
  inherited;  
  iQuantidade := 0;  
  while not(Self.Terminated) do  
  begin  
    iPos := Random(C_MAX);  
    sUF := C_UF[iPos];  
    if (Form1.ClientDataSet1.Locate('SIGLA', sUF, [])) then  
    begin  
      Sleep(100);  
      if (sUF <> Form1.ClientDataSet1.SIGLA.AsString) then  
      begin  
        Self.FConteudoLabel := 'Deu erro!!!';  
        Self.Queue(Self.EscreverLabel);  
        Abort;  
      end;  
      Inc(iQuantidade);  
      Self.FConteudoLabel := Format('Passou [%d] vezes',  
[iQuantidade]);  
      Self.Queue(Self.EscreverLabel);  
    end;  
  end;  
end;;
```

Listagem 5 - Método Execute da classe TLocalizaUF

Pronto: este código tem tudo para dar errado.

De volta ao nosso formulário vamos codificar a inicialização de dois threads e observar o que acontece. Para isso, declare no escopo `private` da classe `TForm1` dois atributos para controlarmos as instâncias dos nossos threads:

```
{...}  
private  
  FThread1: TLocalizaUF;  
  FThread2: TLocalizaUF;  
{...}
```

Listagem 6 - Declaração dos atributos na classe TForm1

Em cada um dos botões iniciaremos os threads, associando o `label` correspondente:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Self.FThread1 := TLocalizaUF.Create(Self.Label1);  
  Self.FThread1.Start;  
  Self.Button1.Enabled := False;  
end;  
  
procedure TForm1.Button2Click(Sender: TObject);  
begin  
  Self.FThread2 := TLocalizaUF.Create(Self.Label2);  
  Self.FThread2.Start;  
  Self.Button2.Enabled := False;  
end;
```

Listagem 7 - Inicialização dos dois threads

Por fim tomaremos um cuidado muito importante na finalização do programa, pois se não o fizermos poderemos ter problemas dos mais variados uma vez que os threads poderão estar em execução. Este é um ponto sensível a ser considerado.

Perceba que, da forma como codificamos na Listagem 5, o thread será finalizado em caso de erro, pois é dado a instrução `Abort()`. Porém caso não ocorra o erro ele estará em execução portanto temos que finalizar este thread antes do encerramento do programa.

Para contemplarmos as duas situações utilizaremos a propriedade `Finished` da classe `TThread`. Não confunda com a propriedade `Terminated` que é uma *flag* que indica que solicitamos o término do thread. Já o `Finished` fica *verdadeiro* no exato momento em que o método `Execute` sai de contexto.

Codifique o manipulador de evento `OnClose()` do `TForm1` conforme a Listagem 8:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if Assigned(Self.FThread1) then
  begin
    if not (Self.FThread1.Finished) then
    begin
      Self.FThread1.Terminate;
      Self.FThread1.WaitFor;
    end;
    Self.FThread1.Free;
  end;

  if Assigned(Self.FThread2) then
  begin
    if not (Self.FThread2.Finished) then
    begin
      Self.FThread2.Terminate;
      Self.FThread2.WaitFor;
    end;
    Self.FThread2.Free;
  end;
end;
```

Listagem 8 - Finalizando os threads

Colocando o sistema em execução percebemos que tudo funciona bem se clicarmos apenas no primeiro botão. Mas a coisa muda de figura quase que imediatamente ao clicarmos no segundo:

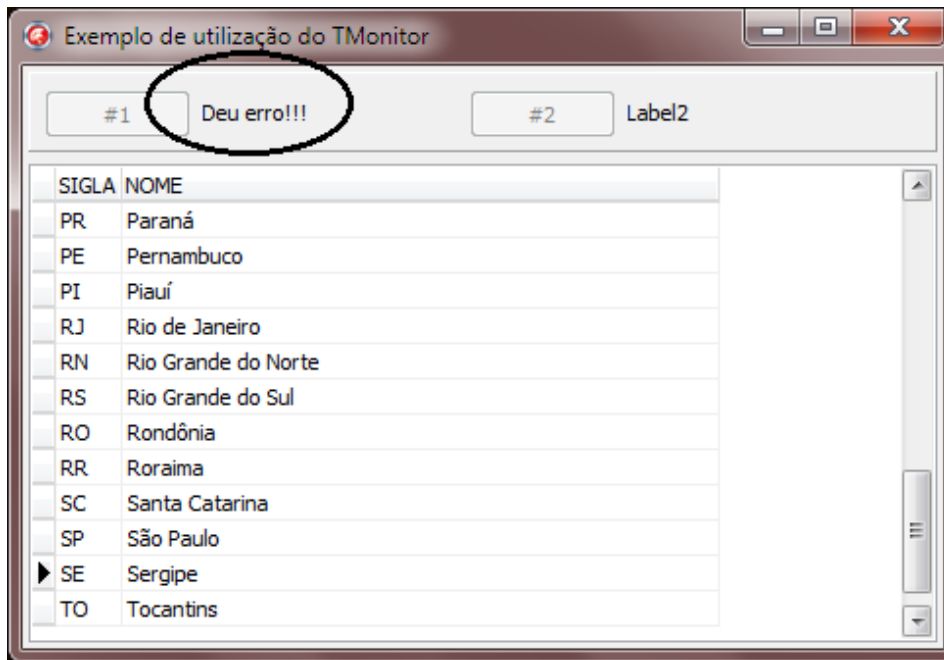


Figura 3 - Exemplo de uma colisão entre os dois threads

Solução proposta

Vamos melhorar esta situação. O `TMonitor` que queremos usar encontra-se na unit `System` e o funcionamento dele é simples: Utilizamos um objeto como referência a um bloqueio. Essa é uma abordagem bastante interessante, pois o recurso "protegido" fica intrínseco ao bloqueio, deixando o código mais claro.

Modificaremos o método `Execute()` do thread passando a utilizar o `TMonitor`. Como desenvolvemos o thread na mesma unit do formulário, é necessário colocar o nome totalmente qualificado para resolver a ambiguidade com o `TMonitor` da unit `Vcl.Forms`, que representa o monitor (tela) do seu equipamento.

```
procedure TLocalizaUF.Execute;
const
  C_MAX = 5;
  C_UF: array [0..C_MAX-1] of string = ('SP', 'BA', 'RJ', 'SE', 'AM');
var
  iPos      : Byte;
  sUF       : string;
  iQuantidade: NativeUInt;
begin
  inherited;
  iQuantidade := 0;
  while not(Self.Terminated) do
  begin
    iPos := Random(C_MAX);
    sUF := C_UF[iPos];
    System.TMonitor.Enter(Form1.ClientDataSet1); { ⬅ }
    try
      if (Form1.ClientDataSet1.Locate('SIGLA', sUF, [])) then
      begin
        Sleep(100);
        if (sUF <> Form1.ClientDataSet1.SIGLA.AsString) then
        begin
          Self.FConteudoLabel := 'Deu erro!!!';
          Self.Queue(Self.EscreverLabel);
          Abort;
        end;
        Inc(iQuantidade);
        Self.FConteudoLabel := Format('Passou [%d] vezes',
[iQuantidade]);
        Self.Queue(Self.EscreverLabel);
      end;
    finally
      System.TMonitor.Exit(Form1.ClientDataSet1); { ➡ }
    end;
  end;
end;
```

Listagem 9 - Correção do método Execute do thread

O funcionamento é similar ao `TCriticalSection`: Utiliza-se o método `Enter` para entrar na seção crítica porém podendo definir um tempo de espera e o `Exit()` para liberar a seção crítica. Note que há uma sobrecarga neste método e um deles é uma função que indica se entrou ou não na seção crítica.

E temos também o `TryEnter` que retorna imediatamente um valor booleano indicando se entrou (*true*) ou não (*false*) na seção crítica – o que é útil em algumas situações em que não é conveniente ficar parados esperando.

Ao entrar na seção crítica é sempre interessante seguir com uma seção `try..finally` para garantir que saíamos dela mesmo em caso de exceção. E isso vale para qualquer outra técnica de bloqueio. Com as correções feitas agora nosso programa funciona bem.

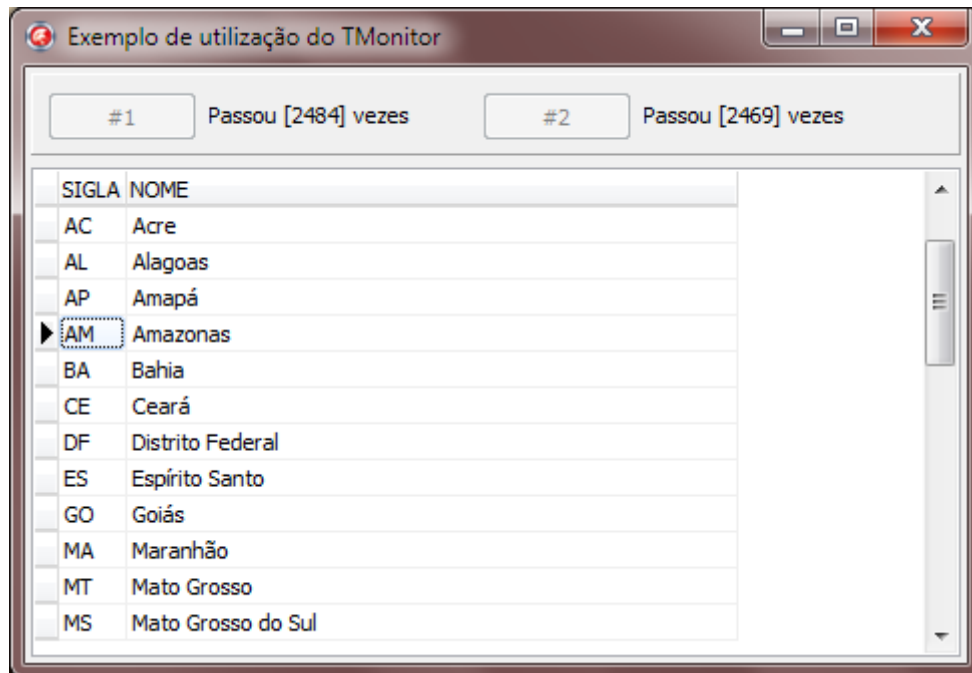


Figura 4 - Evidência de que não houve conflito

Como abrimos mão de vários cuidados, até mesmo por se tratar de um exemplo didático, um problema não resolvido é a possibilidade de clicarmos no DBGrid durante a execução do teste. Isso faz com que o thread acuse erro uma vez que este DBGrid desconhece a nossa estratégia de bloqueio. Isso evidencia o fato de que se algo não obedecer à sinalização proposta os resultados serão imprevisíveis.

TMonitor – Como sinalizador de evento

Agora vamos a um exemplo mais complexo que tira um real proveito do verdadeiro diferencial do TMonitor: **Notificação**.

No artigo anterior falamos sobre o TEvent que sinaliza o thread de que algo aconteceu. No exemplo que desenvolvemos naquele artigo criamos uma fila de objetos para descarte: Ao colocar um objeto na fila notificávamos o thread para que ele fizesse o trabalho dele.

Porém se tivermos vários threads interessados neste objeto o que teríamos que fazer? Colocar uma cópia deste objeto em cada thread parece ser a solução mais imediata. Mas e se tivéssemos que garantir que apenas um thread processe o objeto a um só tempo? Seria algo bem complicado de se resolver.

O `TMonitor` supre esta necessidade pois nos permite criar uma fila de threads interessados em um determinado objeto o que é uma abordagem diferente do `TEvent`. Os métodos do `TMonitor` em questão são `Wait`, `Pulse` e `PulseAll`. Desenvolveremos um novo aplicativo onde teremos um thread “*produtor*” e três threads “*consumidores*”. A Figura 5 possui o layout proposto ao nosso teste de conceito.

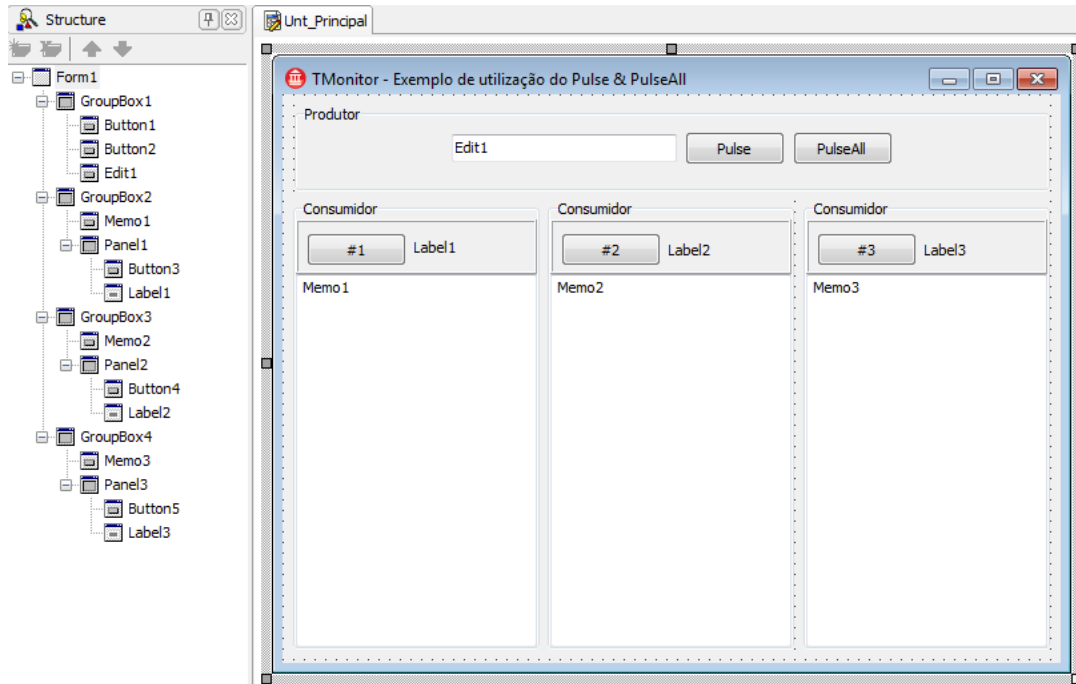


Figura 5 - Layout do aplicativo de teste de conceito

A proposta do nosso aplicativo é notificarmos os três threads consumidores de que há um novo texto no `Edit1`. O propósito desses threads, por sua vez, é ler o conteúdo do `Edit1` e replicar no `Memo` correspondente. Não queremos que façam ao mesmo tempo uma vez que estamos entendendo que o `Edit1` é um recurso crítico. A Listagem 10 possui o escopo da nossa classe thread que, novamente por conveniência, colocaremos na mesma unit do formulário:

```
TPreencheMemo = class(TThread)
private
  FBufferTexto: string;
  FMemo       : TMemo;
  FLabel      : TLabel;
  procedure EscreverMemo;
  procedure EscreverLabel;
protected
  procedure Execute; override;
public
  constructor Create(AMemo: TMemo; ALabel: TLabel);
end;
```

Listagem 10 - Escopo da classe TPreencheMemo

A Listagem 11 possui todo o código da classe TPreencheMemo que tem o mesmo perfil do exemplo anterior.

```
{ TPreencheMemo }

constructor TPreencheMemo.Create(AMemo: TMemo; ALabel: TLabel);
begin
    inherited Create(True);
    Self.FMemo := AMemo;
    Self.FLabel := ALabel;
end;

procedure TPreencheMemo.EscreverLabel;
begin
    Self.FLabel.Caption := Self.FBufferTexto;
end;

procedure TPreencheMemo.EscreverMemo;
var
    sLinha      : string;
    sHoraAtual: string;
begin
    sHoraAtual := TimeToStr(Time());
    sLinha := Format('%s - %s', [sHoraAtual, Self.FBufferTexto]);
    Self.FMemo.Lines.Insert(0, sLinha);
end;

procedure TPreencheMemo.Execute;
var
    bRet: Boolean;
begin
    inherited;
    while not(Self.Terminated) do
    begin

        Self.FBufferTexto := 'Enter ...';
        Self.Synchronize(Self.EscreverLabel);
        System.TMonitor.Enter(Form1.Edit1);

        try

            Self.FBufferTexto := 'Wait ...';
            Self.Synchronize(Self.EscreverLabel);
            bRet := System.TMonitor.Wait(Form1.Edit1, INFINITE);
            if (bRet) then
            begin
                Self.FBufferTexto := Form1.Edit1.Text;
                Self.Synchronize(Self.EscreverMemo);
            end;

        finally

            Self.FBufferTexto := 'Exit ...';
            Self.Synchronize(Self.EscreverLabel);
            System.TMonitor.Exit(Form1.Edit1);

        end;
    end;
end;
```



```
end;
```

Listagem 11 - Codificação da classe TPreencheMemo

Concentremo-nos no método `Execute` da classe. Percebemos três fases bem definidas: `Enter()` → `Wait()` → `Exit()`. Todos eles fazem referência ao mesmo objeto que no caso é o `Form1.Edit1`.

Vamos codificar a inicialização dos threads consumidores. No escopo `private` da classe `TForm1` declare um array de 3 elementos para controlarmos os nossos threads:

```
TForm1 = class(TForm)
{...}
private
    FConsumidores: array [1 .. 3] of TPreencheMemo;
{...}
```

Listagem 12 - Declaração do array para controlar os threads consumidores

Em cada um dos três botões “consumidores” instancie e inicie o thread correspondente:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    Self.FConsumidores[1] := TPreencheMemo.Create(Self.Memo1,
Self.Label1);
    Self.FConsumidores[1].Start;
    Self.Button3.Enabled := False;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    Self.FConsumidores[2] := TPreencheMemo.Create(Self.Memo2,
Self.Label2);
    Self.FConsumidores[2].Start;
    Self.Button4.Enabled := False;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    Self.FConsumidores[3] := TPreencheMemo.Create(Self.Memo3,
Self.Label3);
    Self.FConsumidores[3].Start;
    Self.Button5.Enabled := False;
end;
```

Listagem 13 - Inicialização dos threads consumidores

Em seguida vamos codificar a finalização do programa como um todo. No evento `OnClose` do formulário codifique como na Listagem 14. Perceba que está em duas etapas. Na primeira etapa estamos, após algumas checagens, solicitando o término dos threads. Mas os threads estarão travados no `Wait()` e é por isso que estamos dando o comando `PulseAll()`. Isto notificará todos os threads, que concluem o processamento e daí tomam conhecimento da finalização. Por fim liberamos as instâncias alocadas.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  oThread: TPreencheMemo;
begin
  for oThread in Self.FConsumidores do
  begin
    if (Assigned(oThread)) then
    begin
      if not(oThread.Finished) then
      begin
        oThread.Terminate;
      end;
    end;
  end;

  System.TMonitor.PulseAll(Self.Edit1); {←}

  for oThread in Self.FConsumidores do
  begin
    if (Assigned(oThread)) then
    begin
      if not(oThread.Finished) then
      begin
        oThread.WaitFor;
      end;
      oThread.Free;
    end;
  end;
end;
```

Listagem 14 - Finalização dos threads consumidores junto com a finalização do programa

Finalmente os dois últimos botões: Pulse e PulseAll :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  System.TMonitor.Pulse(Self.Edit1);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  System.TMonitor.PulseAll(Self.Edit1);
end;
```

Listagem 15 - Botões Pulse e PulseAll

Colocando o programa em execução e iniciando os threads percebemos imediatamente que todos ficam na fase de espera – Wait() :

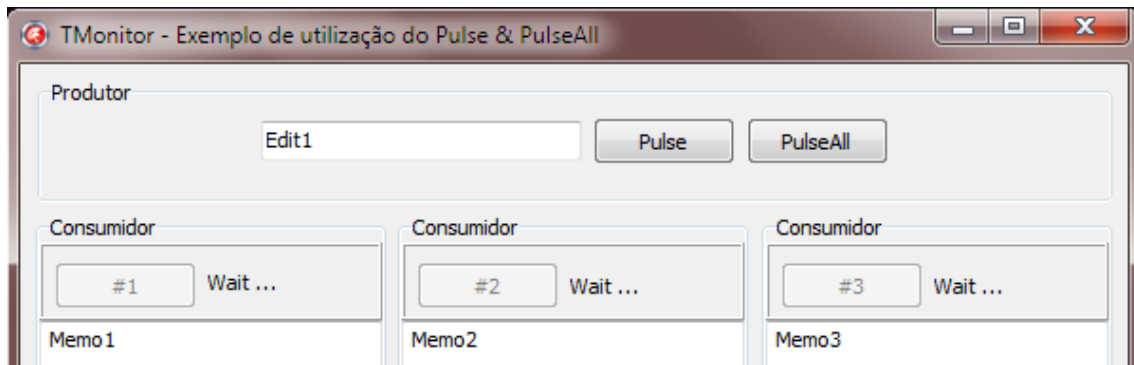


Figura 6 - Evidência de que os threads estão na fase de espera (Wait)

Ao clicarmos no botão "Pulse" seguidamente percebemos que os Memos vão recebendo o texto "Edit1" na *mesma ordem em que os threads foram inicializados*:

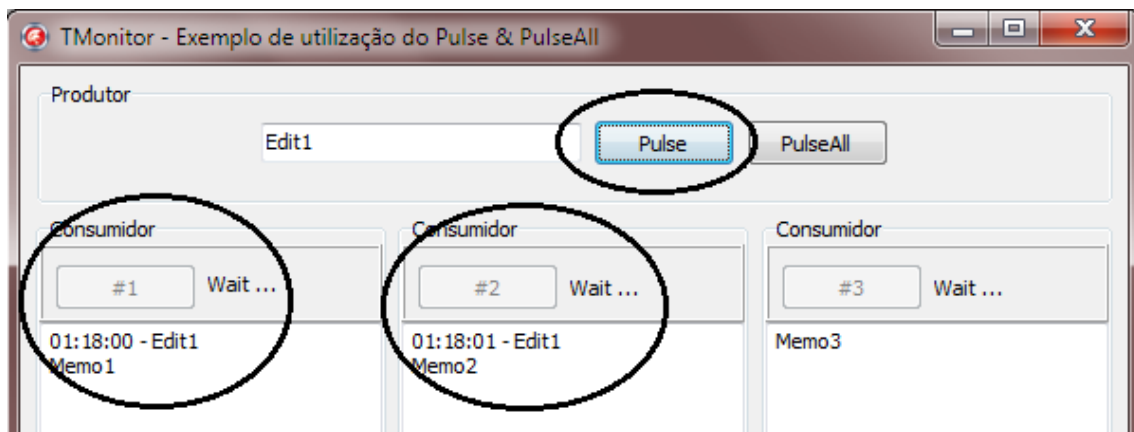


Figura 7 - Comando Pulse

O "PulseAll" , por fim, sinaliza todos os threads consecutivamente. Cada vez que um thread passar pela fase Exit já abre espaço para o próximo atuar:

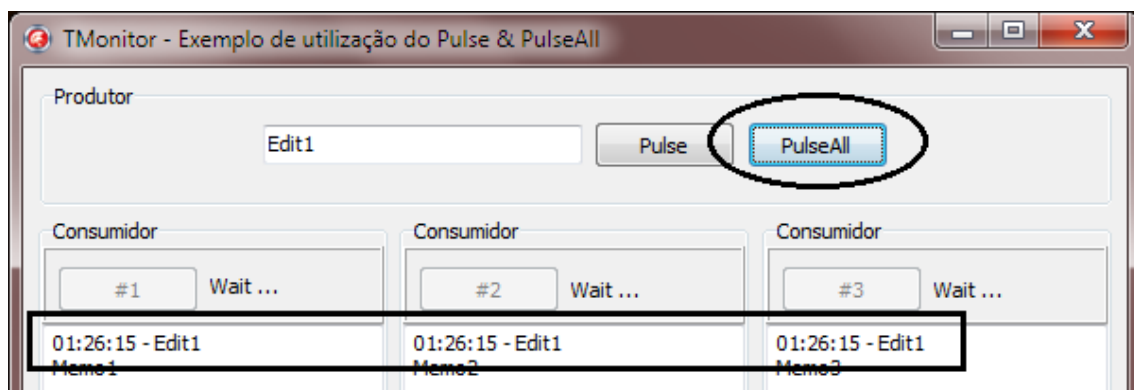


Figura 8 - Comando PulseAll

Conclusão

O `TMonitor` é uma estratégia já presente em outras plataformas de desenvolvimento e passou a existir nativamente no Delphi 2009. De fato se mostra bastante conveniente quanto à mecânica de utilização. Em pesquisas sobre o assunto percebe-se alguns questionamentos quanto a desempenho e é algo a ser considerado. Pode-se utilizar o `TMonitor` com grande sucesso em uma solução que envolva mensagens *assíncronas* a um servidor. O desafio neste tipo de solução é enviar a solicitação e processar a resposta, geralmente demandando bastante código e controles. Com o `TMonitor` as coisas ficaram mais simples.

Mostra-se também ser a melhor abordagem em novos softwares multiplataforma uma vez que é uma solução independente do S.O.