

Elementos de Programação Multithreading em Delphi



Sequential Programming – Gary Larson

Autor: Constantino Seixas Filho - UFMG

Introdução

Em Delphi existem classes pré definidas que facilitam a criação dos objetos básicos que usaremos neste texto: Threads, Eventos, CriticalSections, etc. Para todos os demais elementos é possível definir componentes que encapsulem um conjunto de entidades básicas e proporcionem as funcionalidades desejadas.

Criação de threads: classe TThread

Para criar uma thread, devemos criar uma classe derivada (descendente) da classe base: TThread. Cada instância desta nova classe será uma thread de execução. Como já foi estudado, as threads compartilham o mesmo espaço de memória do processo que as criou.

Para criar e usar um novo objeto do tipo thread:

- Escolha File / New /Other/Thread Object para criar uma nova Unit que contem um objeto derivado da classe TThread.
- Defina o método Execute do objeto thread inserindo o código que deve ser executado quando a thread **for** executada.

A maior parte dos métodos que acessam um objeto CLX (*Component Libray for cross platform*) e atualizam um formulário, devem ser chamados a partir da thread principal ou usar um objeto de sincronização, como será estudado.

Exemplo 1: Criando Threads

O trecho de programa abaixo cria a classe TprintThread como derivada da classe TThread. A função virtual Execute é então substituída por um código próprio.

Unit E21CreateThreadsMain

```
// E21CreateThreadsMain
//
// Demonstração de criação de threads em Delphi
//
// Autor: Constantino Seixas Filho    Data: 20/09/2001
//
// Comentários: Permite criar três threads que sincronizam a sua execução através
//              da diretiva synchronize.
//              Cada thread recebe um parâmetro dizendo o seu número de ordem.
```

unit E21CreateThreadsMain;

interface

uses

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

type

```
TForm1 = class(TForm)
  btnThread1: TButton;
  btnThread2: TButton;
  btnThread3: TButton;
  lstListBox1: TListBox;
  procedure btnThread1Click(Sender: TObject);
  procedure btnThread2Click(Sender: TObject);
  procedure btnThread3Click(Sender: TObject);
  procedure CloseForm(Sender: TObject; var Action: TCloseAction);
```

private

```
{ Private declarations }
  PT: array[1..3] of TPrintThread; // Cria três threads
```

public

```
{ Public declarations }
```

end;

var

```
Form1: TForm1;
```

implementation

```
{ $R *.dfm }
```

```
procedure TForm1.btnThread1Click(Sender: TObject);
```

```
begin
```

```
  btnThread1.Enabled := False; // Desabilita botão de criação
```

```
  PT[1] := TPrintThread.Create(1); // Cria thread
```

```
end;
```

```
procedure TForm1.btnThread2Click(Sender: TObject);
```

```
begin
```

```
  btnThread2.Enabled := False; // Desabilita botão de criação
```

```
  PT[2] := TPrintThread.Create(2); // Cria thread
```

```
end;
```

```
procedure TForm1.btnThread3Click(Sender: TObject);
```

```
begin
```

```
  btnThread3.Enabled := False; // Desabilita botão de criação
```

```
  PT[3] := TPrintThread.Create(3); // Cria thread
```

```
end;
```

```
procedure TForm1.CloseForm(Sender: TObject; var Action: TCloseAction);
```

```

var index: integer;
begin
  for index:=1 to 3 do
    if Assigned(PT[index]) then PT[2].WaitFor; // Espera threads terminarem
  end;
end.

```

Unit Printth

```

unit Printth;

interface
uses
  Classes, SysUtils;
type
  TprintThread = class (TThread)
  private
    Index: Integer;
  protected
    procedure Execute; override;
    procedure Print;
  public
    constructor Create (Value: Integer);
    // Novo construtor: passa parâmetro na criação
  end;

implementation

uses
  E21CreateThreadsMain, Graphics;

  // Implementa passagem de parâmetro para inicializar a thread

constructor TPrintThread.Create(Value: Integer);
begin
  Index := Value;
  inherited Create(False); // Cria thread: suspenso = Falso
end;

Procedure TPrintThread.Print;
var
  strIndex: string;
begin
  strIndex := IntToStr(Index);
  Form1.lstListBox1.Items.Add(strIndex);
end;

Procedure TPrintThread.Execute;
var

```

```

i: Integer;
begin
  for i:=1 to 6 do
    Synchronize(Print);
end;

end.

```

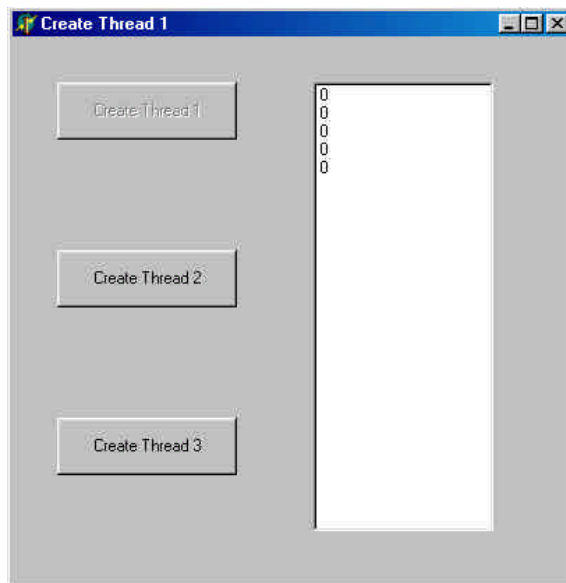


Figura 1 - Tela do exemplo 1

O método Create da classe TThread é utilizado para criar uma thread.

TThread.Create

```

constructor Create(
  CreateSuspended: Boolean // True: a thread é criada no estado suspenso
);

```

TThread.Execute

```

Procedure Execute; virtual; abstract;

```

Fornece um método abstrato que contém o código a ser executado quando a thread é instanciada. Esta função se sobrepõe à função virtual e fornece o código da thread a ser executada. Esta função é responsável por verificar o valor da propriedade Terminated para determinar quando a thread deve terminar.

Não se deve usar as propriedades e métodos de outros objetos diretamente no corpo do método `Execute`. O uso destes objetos deve ser separado em outro procedimento que deve ser chamado como parâmetro do método `Synchronize` (veja exemplo 1).

`TThread.Suspend`

Suspende a thread temporariamente até que seja emitido um `Resume`.

`TThread.Resume`

A thread reasume sua execução.

`TThread.Terminate`

Seta a propriedade `Terminated` da thread para **True**, indicando que ela deve ser terminada o quanto antes. Para que `Terminate` funcione é necessário que o método `Execute` cheque `Terminated` periodicamente e saia do loop quando esta variável for **True**.

`TThread.WaitFor`

Espera a thread terminar e retorna a propriedade `ReturnValue`. Observe que o método `WaitFor` neste caso não apresenta parâmetros. Quando aplicado a objetos de sincronização este método terá como parâmetro o tempo de `timeout`.

`TThread.Synchronize`

Executa a chamada a um método dentro da thread primária da VCL (*Visual Component Library*).

```
type TThreadMethod = procedure of object;  
procedure Synchronize(Method: TThreadMethod);
```

`Synchronize` causa a chamada especificada por `Method` a ser executada usando a thread primária, evitando conflito de acesso simultâneo a um mesmo componente entre as threads. A execução da thread fica suspensa até que o método seja executado no contexto da thread principal. Outra maneira de se assegurar a exclusão mútua será através do uso de objetos de sincronização ou do *multi-read exclusive-write synchronizer*.

Principais propriedades

Handle	Usado para chamar funções de manipulação de threads type THandle = Integer;
--------	---

	property Handle: THandle;
ThreadID	Identifica a thread no sistema type THandle = Integer; property ThreadID: THandle;
Terminated	Indica que o término da thread foi pedido. O método terminate ativa o flag Terminated.
Suspended	Solicita suspensão da thread.
Priority	Prioridade da thread property Priority: Integer; type TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical); property Priority: TThreadPriority;
ReturnValue	Valor retornado pela thread property ReturnValue: Integer
FreeOnTerminate	Deve ser definido como TRUE para liberar a thread quando terminar. Se for FALSE, a aplicação deve terminar a thread explicitamente.
OnTerminate	Ocorre após o método Execute da thread ter retornado e antes da thread ser destruída. property OnTerminate: TNotifyEvent; O programador deve escrever um handler para o evento OnTerminate após o término da execução da thread. O handler será chamado no contexto da thread principal, o que significa que os métodos e propriedades CLX podem ser chamados livremente.

Alterando a prioridade das threads

Para alterar a prioridade de uma thread ela deve ser criada no estado suspenso, a propriedade prioridade deve ser alterada e a thread deve ser liberada:

Exemplo:

```
MyThread := TMyThread.Create(True);    // Cria a thread em estado suspenso
MyThread.Priority := tpLower; //Muda prioridade para nível abaixo de normal
MyThread.Resume;    // Executa a thread
```

A solução da Exclusão Mútua no WNT

Lock/Unlock (Classe TCanvas)

Pode-se prevenir que duas ou mais threads utilizem o mesmo objeto VCL (*Visual Component Library*), simultaneamente, bloqueando o seu acesso através do método Lock da classe TCanvas e descendentes. O acesso é liberado através do método Unlock.

Outra alternativa é o uso do método synchronize da classe TThread.

Exemplo 2 - Instrução Lock da classe Canvas

E21CreateThreadsMainv4.pas

```
unit E21CreateThreadsMainv4;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls, Printhev4;  
  
type  
  TForm1 = class(TForm)  
    btnThread1: TButton;  
    btnThread2: TButton;  
    btnThread3: TButton;  
    lstListBox1: TListBox;  
    procedure btnThread1Click(Sender: TObject);  
  
    private  
    { Private declarations }  
    PT: array[1..3] of TPrintThread; // Cria três threads  
    public  
    { Public declarations }  
  end;  
  
var  
  Form1: TForm1;  
  
implementation
```



```
{ $R *.dfm }
```

```
procedure TForm1.btnThread1Click(Sender: TObject);  
var Index: Integer;  
begin  
    Index := (Sender as Tbutton).Tag;           // Busca qual button foi acionado  
    (Sender as Tbutton).Enabled := False;     // Desabilita botão de criação  
    PT[Index] := TPrintThread.Create(Index);    // Cria threads  
end;  
  
end.
```

```
Printthv4.pas
```

```
unit Printthv4;  
  
interface  
  
uses  
    Classes, SysUtils;  
  
type  
    TprintThread = class (TThread)  
    private  
        Index: Integer;  
    protected  
        procedure Execute; override;  
    public  
        constructor Create (Value: Integer);  
    end;  
  
implementation  
  
uses  
    E21CreateThreadsMainv4, Graphics;  
  
// Implementa passagem de parâmetro para inicializar a thread  
constructor TPrintThread.Create(Value: Integer);  
begin  
    Index := Value;  
    inherited Create(False);  
end;  
  
Procedure TPrintThread.Execute;  
var  
    i: Integer;  
    strIndex: string;  
begin  
    for i:=1 to 100 do  
        begin
```



```

thread
Var ThreadId: Cardinal // Variável passada por referência para receber o
                        ThreadId.
): Integer;

```

O protótipo da função é como se segue:

```
Function(Parameter: Pointer): Integer;
```

Este exemplo serve apenas para ilustrar esta possibilidade que só deve ser utilizada em casos extremos. Deve-se sempre preferir a criação de threads utilizando a classe TThread.

Exemplo 3: Criação de threads utilizando a diretiva BeginThread

```

// Demonstração do uso de criação de threads em Delphi
//
// Versão 2: Uso da diretiva BeginThread
//
// Autor: Constantino Seixas Filho    Data: 03/06/2002
//
// Comentários: Uma maneira primitiva de criar threads em Delphi é através
//               da diretiva BeginThread. Neste caso o corpo da thread é definido na
//               mesma Unit do programa principal.

```

```
unit E21BeginThread;
```

```
interface
```

```
uses
```

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

```
type
```

```
//Record para manter a informação de uma thread
```

```
TThreadInfo = record
```

```
    Active      : Boolean;
```

```
    ThreadHandle : integer;
```

```
    ThreadId    : Cardinal;
```

```
end;
```

```
TForm1 = class(TForm)
```

```
    btnThread1: TButton;
```

```
    btnThread2: TButton;
```

```
    btnThread3: TButton;
```

```
    lstListBox1: TListBox;
```

```
    procedure btnThread1Click(Sender: TObject);
```

```
    procedure OnClose(Sender: TObject; var Action: TCloseAction);
```

```

private
  Index: Integer;
public
  ThreadInfo: array[0..2] of TThreadInfo; // Mantém informações da thread
public
  { Public declarations }
end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

Function MyThread(PIndex: Pointer): Integer;
var
  i: Integer;
  strIndex: string;
  Index: Integer;
begin
  Index := PInteger(PIndex)^;
  for i:=1 to 100 do
  begin
    strIndex := IntToStr(Index);
    with Form1.lstListBox1 do
    begin
      Canvas.Lock;
      Items.Add(strIndex);
      Canvas.Unlock;
    end;
    Sleep(10);
  end; // for
  EndThread(Index); // Termina thread
  Result := 0;
end;

procedure TForm1.btnThread1Click(Sender: TObject);
begin
  Index := (Sender as Tbutton).Tag - 1; // Busca qual button foi acionado
  // -1 para índice de 0 a 2.
  (Sender as Tbutton).Enabled := False; // Desabilita botão de criação
  ThreadInfo[Index].ThreadHandle :=
    BeginThread (nil
      0,
      @MyThread,
      @Index,      // nil
      0,
      ThreadInfo[Index].ThreadId);
  if ThreadInfo[Index].ThreadHandle <> 0 //Everything ok
  then ThreadInfo[Index].Active := True;

```

```

end;

procedure TForm1.OnClose(Sender: TObject; var Action: TCloseAction);
var
    Index: Integer;
    ExitCode: DWORD;
begin
    for Index:=0 to 2 do
        begin
            if ThreadInfo[Index].ThreadHandle <> 0
                then begin
                    WaitForSingleObject(ThreadInfo[Index].ThreadHandle, INFINITE);
                    // Espera morte das threads
                    GetExitCodeThread(ThreadInfo[Index].ThreadHandle, ExitCode);
                    CloseHandle(ThreadInfo[Index].ThreadHandle);
                end
            end
        end;
    end.

```

Sincronização entre threads

Mutex

Para usar Mutex em Delphi use o código nativo da API:

```

// Ao criar o formulário:
hMutex := CreateMutex(nil, false, nil);

// Para entrar na seção crítica:
WaitForSingleObject(hMutex, INFINITE);

.....
// Saindo da seção crítica:
ReleaseMutex(hMutex);

// Ao destruir o formulário:
CloseHandle(hMutex);

```

A instrução WaitForSingleObject é semelhante à função correspondente da API Win32, apenas o seu uso em Pascal é que tem sintaxe ligeiramente diferente:

```

if WaitForSingleObject(hMutex, 0) <> wait_TimeOut
    then ***

```

WaitForSingleObject

```
function WaitForSingleObject(
```

```

hHandle: THandle; // handle para um objeto do kernel
dwMilliseconds: DWORD; // tempo máximo que desejamos esperar
): DWORD;

```

Comentários sobre os parâmetros:

hHandle Handle para objeto do kernel que será sinalizado.

dwMilliseconds Tempo de *timeout*. Após este tempo a função retornará, independente da sinalização ter ocorrido.

INFINITE – Não desejamos timeout

0 – Testa se foi sinalizado e retorna

Retorno da função:

Status	Interpretação
WAIT_OBJECT_0	Objeto foi sinalizado
WAIT_TIMEOUT	Ocorreu timeout
WAIT_ABANDONED	Uma thread proprietária de um Mutex terminou (realizou ExitThread) sem liberá-lo. O objeto Mutex será estudado ainda neste capítulo.
WAIT_FAILED	A função falhou

Exemplo 4: Uso de Mutex

E31Mutex.pas

Este problema foi apresentado no capítulo 3. Uma variável global, representada por um *array* de inteiros é acessada por múltiplas threads, as quais a preenchem com valores diferentes. Cada vez que o botão *Exibir Dados* é clicado, os valores correntes dos membros da variável global são exibidos.

```
unit E31Mutex;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, MyThread;
```

```
type
```

```
TForm1 = class(TForm)
  Button1: TButton;
  Button2: TButton;
  Edit1: TEdit;
  Edit2: TEdit;
  Edit3: TEdit;
```

```

Edit4: TEdit;
Edit5: TEdit;
Edit6: TEdit;
Edit7: TEdit;
Edit8: TEdit;
Edit9: TEdit;
Edit10: TEdit;
Label1: TLabel;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Label6: TLabel;
Label7: TLabel;
Label8: TLabel;
Label9: TLabel;
Label10: TLabel;
procedure Button1Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure Button2Click(Sender: TObject);

private
  { Private declarations }
  PT: array[1..5] of TMyThread; // Threads associadas
public
  { Public declarations }
end;

```

```

var
  Form1: TForm1;
  Index: Integer = 0;

```

implementation

```
{ $R *.dfm }
```

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Index < 5 then // Cria até 5 threads
  begin
    Inc(Index); // Index: 1..5
    PT[Index] := TMyThread.Create(Index); // cria thread
  end
  else Button1.Enabled := False; // Desabilita botão de criação
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
var i:Integer;
begin

```

```

for i:=1 to 10 do
    Registro[i]:= 0;
    hMutex := CreateMutex(nil, false, nil);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    CloseHandle (hMutex); // Fecha Handle para Mutex
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    // Coloque a primeira e última instruções em comentário e veja o resultado
    WaitForSingleObject(hMutex, INFINITE);
    Edit1.Text := IntToStr(Registro[1]);
    Edit2.Text := IntToStr(Registro[2]);
    Edit3.Text := IntToStr(Registro[3]);
    Edit4.Text := IntToStr(Registro[4]);
    Edit5.Text := IntToStr(Registro[5]);
    Edit6.Text := IntToStr(Registro[6]);
    Edit7.Text := IntToStr(Registro[7]);
    Edit8.Text := IntToStr(Registro[8]);
    Edit9.Text := IntToStr(Registro[9]);
    Edit10.Text := IntToStr(Registro[10]);
    ReleaseMutex(hMutex);
end;

end.

```

MyThread.pas

```

unit MyThread;

interface

uses
    Windows, Classes, SysUtils;

type
    TMyThread = class (TThread)
    private
        Index: Integer;
    protected
        procedure Execute; override;
    public
        constructor Create (Value: Integer);
    end;

var
    Registro: array [1..10] of Integer;

```



```
hMutex: THandle;
```

implementation

uses

```
Graphics;
```

```
// Implementa passagem de parâmetro para inicializar a thread
```

```
constructor TMyThread.Create(Value: Integer);
```

```
begin
```

```
    Index := Value;
```

```
    inherited Create(False);
```

```
end;
```

```
Procedure TMyThread.Execute;
```

```
var
```

```
    i: Integer;
```

```
begin
```

```
    Randomize();
```

```
    while (TRUE) do
```

```
        begin
```

```
            WaitForSingleObject(hMutex, INFINITE);
```

```
            for i:=1 to 4 do
```

```
                Registro[i]:= Index;
```

```
                Sleep(Random(100));
```

```
            for i:=5 to 7 do
```

```
                Registro[i]:= Index;
```

```
                Sleep(Random(100));
```

```
            for i:=8 to 10 do
```

```
                Registro[i]:= Index;
```

```
                ReleaseMutex(hMutex);
```

```
        end
```

```
end;
```

```
end.
```

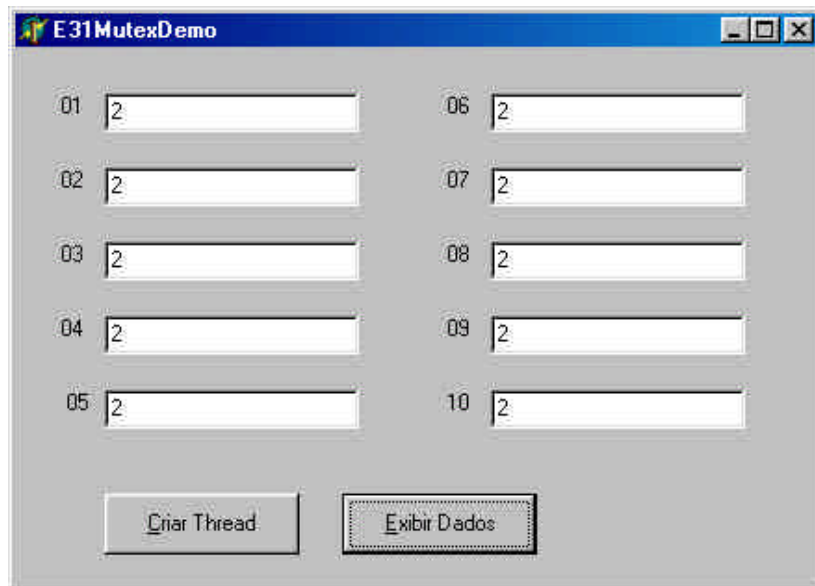


Figura 3 - Demonstrativo: E31MutexDemo

CriticalSections

A primeira forma de uso é empregando as funções nativas da API:

```
var
    Critical1: TRTLCriticalSection;

// Inicializa variável seção crítica
InitializeCriticalSection(Critical1);

// Entra na seção crítica
EnterCriticalSection(Critical1);

*****

// Libera seção crítica
LeaveCriticalSection(Critical1);

// Apaga seção crítica
DeleteCriticalSection(Critical1);
```

A segunda é utilizando a classe TcriticalSection:

```
uses
    SyncObjs;

var
    Critical1: TcriticalSection;

// Inicializa variável seção crítica
Critical1:= TcriticalSection.Create;
// Entra na seção crítica
Critical1.Enter;      // ou Critical1.Acquire

*****

// Libera seção crítica
Critical1.Leave;     // ou Critical1.Release

// Apaga seção crítica
Critical1.Free;
```

Ao proteger um objeto, é importante assegurar que o lock sobre o objeto será liberado em caso de ocorrência de uma exceção.

```
LockX.Acquire;  // bloqueia outras threads
try
    Y := sin(a);
```

```
finally
    LockX.Release;
end;
```

Exemplo 5: Uso de TcriticalSection

O programa que se segue é um trecho da solução para o problema anterior empregando a classe TcriticalSection. A solução completa se encontra no CD.

E31CriticalSection.pas

```
unit E31CriticalSection;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, SyncObjs, E31CSMyThread;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Button2: TButton;
        Edit1: TEdit;
        Edit2: TEdit;
        Edit3: TEdit;
        Edit4: TEdit;
        Edit5: TEdit;
        Edit6: TEdit;
        Edit7: TEdit;
        Edit8: TEdit;
        Edit9: TEdit;
        Edit10: TEdit;
        Label1: TLabel;
        Label2: TLabel;
        Label3: TLabel;
        Label4: TLabel;
        Label5: TLabel;
        Label6: TLabel;
        Label7: TLabel;
        Label8: TLabel;
        Label9: TLabel;
        Label10: TLabel;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure FormDestroy(Sender: TObject);
        procedure Button2Click(Sender: TObject);

private
```

```

    { Private declarations }
    PT: array[1..5] of TMyThread; // Threads associadas
public
    { Public declarations }
end;
var
    Form1: TForm1;
    Index: Integer = 0;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
    if Index < 5 then // Cria até 5 threads
    begin
        Inc(Index); // Index: 1..5
        PT[Index] := TMyThread.Create(Index); // cria thread
    end
    else Button1.Enabled := False; // Desabilita botão de criação
end;

procedure TForm1.FormCreate(Sender: TObject);
var i:Integer;
begin
    for i:=1 to 10 do
        Registro[i]:= 0;
        MyCS := TCriticalSection.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    MyCS.Free; // Fecha Handle para Mutex
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    // Coloque a primeira e última instruções em comentário e veja o resultado
    MyCS.Enter;
    Edit1.Text := IntToStr(Registro[1]);
    Edit2.Text := IntToStr(Registro[2]);
    Edit3.Text := IntToStr(Registro[3]);
    Edit4.Text := IntToStr(Registro[4]);
    Edit5.Text := IntToStr(Registro[5]);
    Edit6.Text := IntToStr(Registro[6]);
    Edit7.Text := IntToStr(Registro[7]);
    Edit8.Text := IntToStr(Registro[8]);
    Edit9.Text := IntToStr(Registro[9]);
    Edit10.Text := IntToStr(Registro[10]);

```

```

    MyCS.Leave;
end;

end.

```

Incremento e decremento com exclusão mútua

Também em Delphi, as funções para incremento e decremento de uma variável podem ser acessadas diretamente:

```

LONG InterlockedIncrement(
    LPLONG lpAddend // Endereço da variável a ser incrementada
);

```

```

LONG InterlockedDecrement(
    LPLONG lpAddend // Endereço da variável a ser decrementada
);

```

Eventos

Para usar eventos em Delphi use os seguintes métodos da classe TEvent:

TEvent.Create

```

constructor Create(
    EventAttributes: PSecurityAttributes // Especifica direitos de acesso e herança do objeto
                                        // Evento. Use nil para atributos default. Se o objeto
                                        // já existir apenas bInheritHandle o campo é usado.
                                        type
                                        PSecurityAttributes = ^TSecurityAttributes;
                                        TSecurityAttributes = record
                                        nLength: DWORD;
                                        lpSecurityDescriptor: Pointer;
                                        bInheritHandle: BOOL;
                                        end;
    ManualReset: Boolean // True: Evento com reset manual. Só é desligado
                        // através da instrução ResetEvent();
    InitialState: Boolean // True: o Evento é criado no estado sinalizado.
    const Name: string // Nome do Evento até 260 caracteres
);

```

Em C usa-se a função `Create` para criar um objeto e `Open` para abrir um objeto nomeado já existente. Em Delphi, `Create` é usado tanto para criar como para abrir um objeto nomeado, por exemplo um `TEvent`.

`TEvent.SetEvent`

Ativa um evento.

`TEvent.ResetEvent`

Reseta um evento.

`TEvent.WaitFor`

Espera até que o evento se torna sinalizado.

```
function WaitFor(Timeout: DWORD):TWaitResult;
```

Retorno:

```
type TwaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);
```

```
wrSignaled // O objeto foi sinalizado
```

```
wrTimeout // O tempo de timeout expirou sem que o objeto fosse  
sinalizado.
```

```
wrAbandoned // O objeto foi destruído antes que o tempo de timeout  
expirasse.
```

```
wrError // Um erro aconteceu durante a espera. LastError fornecerá a  
identidade do erro.
```

`TEvent.Free`

Apaga objeto da classe `TEvent`.

Exemplo 6: Uso da classe `TEvent`

`E41Event.pas`

Este programa ilustra o uso das função de sincronização através de eventos. Através de uma interface amigável é possível criar eventos de diversos tipos, ativar e desativar os eventos, e examinar o comportamento das threads que esperam por estes eventos.

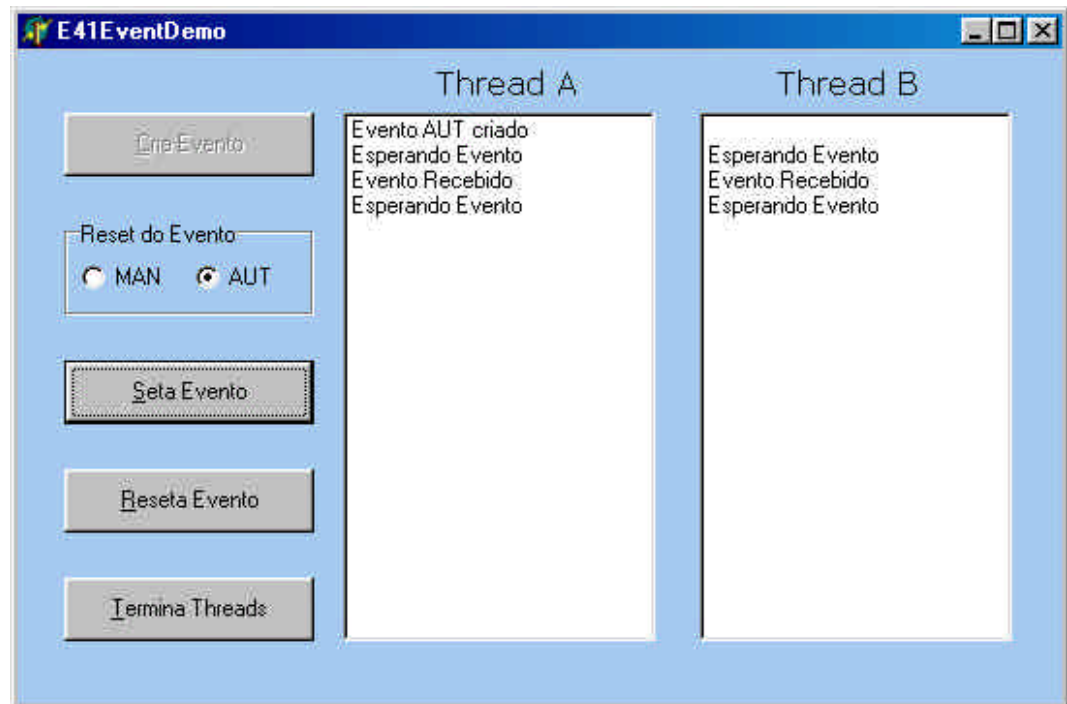


Figura 4 - Demonstração do uso da classe TEvent

Escolha se você deseja criar um evento Manual ou Automático e clique em Cria Evento. O evento será criado juntamente com duas threads que ficarão aguardando eventos. Clique em Seta Evento para causar um evento e em Reseta Evento para resetar um evento manual. Para criar um evento de outro tipo, antes será necessário terminar as threads anteriores através de Termina Threads. Como o sincronismo para término das threads é feito através da instrução Terminate, as threads devem realizar um polling na propriedade Terminated para sair de seu loop de espera. Como elas estarão bloqueadas à espera de um evento e não testando a propriedade Terminated, necessita-se clicar mais duas vezes em SetaEvento para que as threads realmente terminem. No próximo exemplo nós eliminaremos esta limitação.

```
unit E41Event;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, ExtCtrls, SyncObjs, ProcessEvent, CheckError;
```

```
type
```

```
TE41EvDemoForm = class(TForm)  
  ButtonCreate: TButton;  
  ButtonSet: TButton;  
  ButtonReset: TButton;  
  ListBox1: TListBox;  
  ListBox2: TListBox;
```



```

Label1: TLabel;
Label2: TLabel;
RadioGroup1: TRadioGroup;
ButtonTermina: TButton;
procedure ButtonCreateClick(Sender: TObject);
procedure ButtonSetClick(Sender: TObject);
procedure ButtonResetClick(Sender: TObject);
procedure ButtonTerminateClick(Sender: TObject);
procedure FormCreate(Sender: TObject);
private
  { Private declarations }
  PT: array[1..2] of TProcessEvent; // Cria duas threads
public
  { Public declarations }
  procedure PrintEvent(Index: Integer; Str:string);
end;

var
  E41EvDemoForm: TE41EvDemoForm;
  EventMain: TEvent;
  ManAutState: BOOLEAN;
  EventoCriado: Boolean; // Indica que o evento já foi criado
  TTerminated: array[1..2] of Boolean = (True, True);

```

implementation

```
{ $R *.dfm }
```

```

procedure TE41EvDemoForm.PrintEvent(Index: Integer; Str:string);
begin
  if Index = 1
  then with E41EvDemoForm.ListBox1 do
    begin
      Canvas.Lock;
      Items.Add(Str);
      Canvas.Unlock;
    end
  else with E41EvDemoForm.ListBox2 do
    begin
      Canvas.Lock;
      Items.Add(Str);
      Canvas.Unlock;
    end;
  end; // PrintEvent

```

```

procedure TE41EvDemoForm.ButtonCreateClick(Sender: TObject);
//var
// EventMain2: TEvent; // Só para teste de CheckForError
begin

```

```

if ((TTerminated[1] = True) and (TTerminated[2] = True))
then begin
    if RadioGroup1.ItemIndex = 0
        then ManAutState := TRUE
        else ManAutState := FALSE;
    if (EventMain <> nil) then EventMain.Free;
    EventMain := TEvent.Create(nil, ManAutState, FALSE, 'EventoMain');
    // Para forçar erro e testar CheckForError
    // EventMain2 := TEvent.Create(nil, ManAutState, FALSE, 'EventoMain');
    CheckForError;
    EventoCriado := True;
    ButtonCreate.Enabled := False; // Desabilita botão
    PT[1]:= TProcessEvent.Create(1); // Passa ListBox a ser usado para status
    PT[2]:= TProcessEvent.Create(2);
    TTerminated[1]:= False;
    TTerminated[2]:= False;
    if ManAutState = TRUE
        then begin
            PrintEvent(1, 'Evento MAN criado');
            PrintEvent(2, "");
        end
        else begin
            PrintEvent(1, 'Evento AUT criado');
            PrintEvent(2, "");
        end
    end
else begin
    PrintEvent(1, 'Cause Eventos p/ terminar threads');
    PrintEvent(2, "");
end
end; // ButtonCreateClick

```

```

procedure TE41EvDemoForm.ButtonSetClick(Sender: TObject);
begin
    if (EventoCriado)
        then EventMain.SetEvent;
end;

```

```

procedure TE41EvDemoForm.ButtonResetClick(Sender: TObject);
begin
    if (EventoCriado)
        then EventMain.ResetEvent;
end;

```

```

procedure TE41EvDemoForm.ButtonTerminateClick(Sender: TObject);
begin
    if (EventoCriado)
        then begin // Solicita término das threads
            PT[1].Terminate;
        end
end;

```

```

        PT[2].Terminate;
        ButtonCreate.Enabled := True; // Habilita botão
    end
end;

procedure TE41EvDemoForm.FormCreate(Sender: TObject);
begin
    RadioGroup1.ItemIndex:=1; // Reset Automático
end;

end.

```

ProcessEven.pas

```

unit ProcessEvent;

interface

uses
    Classes, SyncObjs;

type
    TProcessEvent = class(TThread)
    private
        { Private declarations }
        Index: Integer; // Índice da thread criada
    protected
        procedure Execute; override;
    public
        constructor Create(Value: Integer); // Novo construtor: passa parâmetro na
        criação
    end;

implementation

uses
    E41Event, Windows;

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

    Synchronize(UpdateCaption);

and UpdateCaption could look like,

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

```

```

{ TProcessEvent }

// Implementa passagem de parâmetro para inicializar a thread
constructor TProcessEvent.Create(Value: Integer);
begin
  Index := Value;
  inherited Create(False);
end;

procedure TProcessEvent.Execute;
var
  Status: TWaitResult;
begin
  while (not Terminated) do begin
    E41EvDemoForm.PrintEvent(Index, 'Esperando Evento');
    Status := EventMain.WaitFor(INFINITE);
    // Substitua esta instrução por um Sleep(2000) e efetue o reset manualmente
    // if (ManAutState = TRUE) // Evento Manual: Reset
    // then EventMain.ResetEvent;
    E41EvDemoForm.PrintEvent(Index, 'Evento Recebido');
    if (ManAutState = TRUE) then Sleep(2000);
  end;
  TTerminated[Index]:= True;
end; // TProcessEvent.Execute

end.

```

Aproveitamos para criar a versão Delphi do procedimento CheckForError, muito útil para diagnosticar problemas em nossas aplicações.

CheckError.pas

```

unit CheckError;

interface

uses
  Windows, SysUtils, Forms, Dialogs;

procedure CheckForError;

implementation

procedure CheckForError;
var
  ErrorCode: DWORD;
  LineNumber: Integer; // Ainda não sei como obter
  FileName: string;
  ErrorMessage: string;
  OutMessage: string;

```

```

begin
  ErrorCode := GetLastError(); //FORMAT_MESSAGE_ALLOCATE_BUFFER
  if (ErrorCode <> 0) then
    begin
      LineNumber := 0; // Get Line Number: descobrir propriedade
      // Nome da aplicação (Application.Title) ou do executável (ExeName)
      // e não nome do arquivo
      FileName := Application.ExeName;

      SetLength(ErrMsg, 256);
      FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM,
        nil,
        ErrorCode,
        LOCALE_USER_DEFAULT, //LANG_NEUTRAL,
        PChar(ErrMsg), // Buffer de Mensagem
        Length(ErrMsg),
        nil);
      SetLength(ErrMsg, StrLen(PChar(ErrMsg)));
      while (Length(ErrMsg) > 0) and
        (ErrMsg[Length(ErrMsg)] in [#10, #13]) do
        SetLength(ErrMsg, Length(ErrMsg) - 1);

      // Show Message é uma chamada simplificada de MessageDlg
      // ShowMessage('Erro: ' + IntToStr(ErrorCode) + ': ' + ErrMsg);
      OutMessage := Format(
        'Erro: %4d'#13 +
        'Aplicação: %20s'#13 +
        'Linha: %3d'#13 +
        'Descr: %30s',
        [ErrorCode,
        FileName,
        LineNumber,
        ErrMsg]);
      MessageDlg(OutMessage, mtCustom, [mbOk], 0);
    end
  end; // CheckForError

end.

```

O programa E41Event2 utiliza um evento especial para notificar as threads aguardando evento que elas devem terminar. As threads espera por um dos dois eventos através da instrução WaitForMultipleObjects da API. Este programa é muito mais eficiente que o anterior.

Trecho de ProcessEvent2

```

constructor TProcessEvent.Create(Value: Integer);
begin
  Index := Value;

```

```

inherited Create(False);
end;

procedure TProcessEvent.Execute;
var
  Handles: array[0..1] of THandle;
  Return: DWORD;
  Abort: Boolean;
begin
  Abort := False;
  Handles[0] := EventMain.Handle;
  Handles[1] := EventKillThreads.Handle;
  repeat
    E41EvDemoForm.PrintEvent(Index, 'Esperando Evento');
    Return := WaitForMultipleObjects(2, @Handles, False, INFINITE);
    Case (Return) of
      WAIT_OBJECT_0: // Recebeu Evento
        begin
          E41EvDemoForm.PrintEvent(Index, 'Evento Recebido');
          // Substitua esta instrução por um Sleep(2000) e
          // efetue o reset manualmente
          // if (ManAutState = TRUE) // Evento Manual: Reset
          // then EventMain.ResetEvent;
          if (ManAutState = TRUE) then Sleep(2000);
        end;
      WAIT_OBJECT_0 + 1: // Pedido de morte das threads
        begin
          E41EvDemoForm.PrintEvent(Index, 'Thread Terminando');
          Abort := True;
          TTerminated[Index]:= True;
        end
    end; // case
  until Abort;
end; // TProcessEvent.Execute

end.

```

Semáforos

Neste exemplo, o problema da corrida de fórmula 1 é resolvido utilizando semáforos. Cada Listbox representa um box de fórmula 1 com diversos carros, gerados aleatoriamente, querendo entrar na pista. Apenas um carro de cada equipe pode entrar na pista de cada vez. No máximo quatro carros podem estar simultaneamente na pista.

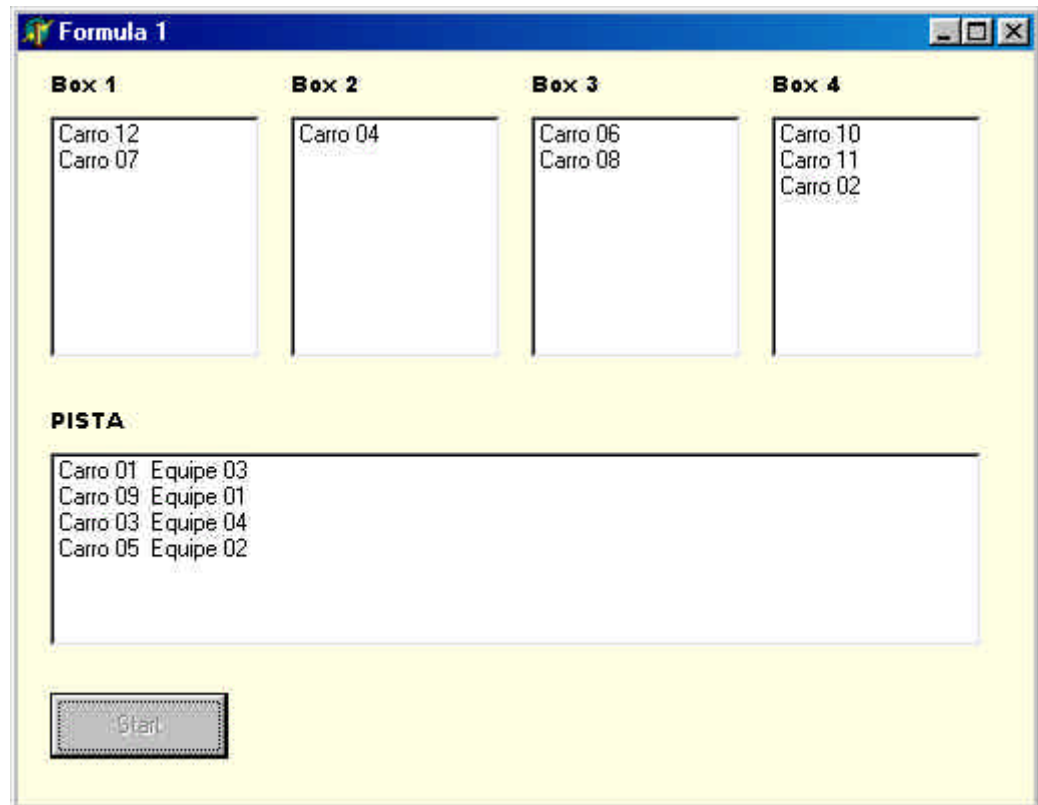


Figura 5: Demonstração do uso de semáforos

Exemplo 7: Uso de semáforos: E41Semaphore.pas

```
unit E41Semaphore;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, Car, CheckError;
```

```
type
```

```
TTreinoF1 = class(TForm)
```

```
  ListBox5: TListBox;
```

```
  ListBox1: TListBox;
```

```
  ListBox2: TListBox;
```

```
  ListBox3: TListBox;
```

```
  ListBox4: TListBox;
```

```
  Label1: TLabel;
```

```
  Label2: TLabel;
```

```
  Label3: TLabel;
```

```
  Label4: TLabel;
```

```
  Label5: TLabel;
```

```
  ButtonStart: TButton;
```

```

procedure FormCreate(Sender: TObject);
procedure ButtonStartClick(Sender: TObject);
private
    // Deve ser um membro privado do form para ser apagado quando o programa termina.
    MasterThread: TMaster;

public

end;

const
    MaxCarrosPista = 4;
    MaxCars = 12;
    NumBoxes = 4;

var
    TreinoF1: TTreinoF1;
    hBox: array[1..NumBoxes] of THandle;
    ListBoxes: array[1..5] of TListBox;
    hPista: THandle;

implementation

{$R *.dfm}

procedure TTreinoF1.FormCreate(Sender: TObject);
var i: Integer;
begin
    ListBoxes[1]:= ListBox1;
    ListBoxes[2]:= ListBox2;
    ListBoxes[3]:= ListBox3;
    ListBoxes[4]:= ListBox4;
    ListBoxes[5]:= ListBox5;

    Randomize;
    // Cria Semáforo para contar carros na pista
    hPista := CreateSemaphore(nil, MaxCarrosPista, MaxCarrosPista,
        'SemContCarrosPista');
    CheckForError;
    // Cria Mutexes para simular funcionamento dos boxes
    for i:=1 to NumBoxes do
        begin
            hBox[i]:= CreateMutex(nil, False, nil);
            CheckForError;
        end;
end; // Form Create

procedure TTreinoF1.ButtonStartClick(Sender: TObject);
begin
    SetLastError(0); // Limpa erro se porventura existente

```



```

if Assigned(MasterThread) then // Se o handle já existe: libera
    MasterThread.Free;
    MasterThread:=TMaster.Create(False);
    CheckForError;
end;

end.

```

C a r . p a s

```

unit Car;

interface

uses
    Classes, Windows, SysUtils, Dialogs, CheckError;

type
    TMaster = class(TThread)
    private
    protected
        procedure Execute; override;
    public
    end;

type
    TCar = class(TThread)
    protected
        Index: Integer;
        Equipe: Integer;
        Car: Integer;
        procedure Execute; override;
    public
        constructor Create (Value: Integer);
        procedure PrintEvent(Janela: Integer; Str:string);
        procedure ClearEvent(Janela:Integer; Car: Integer; Equipe: Integer);
    end;

implementation

uses E41Semaphore;

{ TCar }

procedure TCar.PrintEvent(Janela: Integer; Str:string);
begin
    case (Janela) of
        1..5: with ListBoxes[Janela] do
            begin
                Canvas.Lock;
                Items.Add(Str);
            end;
    end;

```

```

        Canvas.Unlock;
    end;
    else MessageDlg('PrintEvent: Index '+ IntToStr(Janela) + 'Desconhecido',
        mtCustom, [mbOk], 0);
    end; // case
end; // PrintEvent

procedure TCar.ClearEvent(Janela:Integer; Car: Integer; Equipe: Integer);
var
    Texto: string;
    i: Integer;
    Index: Integer;
begin
    Index := -1; // Flag não encontrado
    case (Janela) of
        1..4: Texto := Format('Carro %2.2d', [Car]);
        5: Texto := Format('Carro %2.2d Equipe %2.2d', [Car, Equipe]);
    else MessageDlg('ClearEvent: Index '+ IntToStr(Janela) + 'Desconhecido',
        mtCustom, [mbOk], 0);
    end; // case

    with ListBoxes[Janela] do
        begin
            Canvas.Lock;
            for i:=0 to (Items.count - 1) do
                if Items[i] = Texto then
                    begin
                        Index:= i;
                        break;
                    end; // if
                if (index > -1) then Items.Delete(Index);
                Canvas.Unlock;
            end;
        end; // ClearEvent
procedure TMaster.Execute;
var Cars: array[1..MaxCars] of Tcar;
    Equipe: Integer;
    i: Integer;
begin
    // Cria Threads
    for i:=1 to MaxCars do
        begin
            Equipe := 1 + Random(MaxCarrosPista); // Equipe: 1 a 4
            Cars[i]:=Tcar.Create(i + 256 * Equipe);
            CheckForError;
        end; // for
        TreinoF1.ButtonStart.Enabled := False;
        // Espera threads terminarem
        // Não funciona se colocada na GUI thread porque quem atualiza o ListBox é a thread principal
    for i:=1 to MaxCars do

```

```

begin
    Cars[i].WaitFor;
    Cars[i].Free;
end; // for
    TreinoF1.ButtonStart.Enabled := True;
end; // Tmaster.execute

// Implementa passagem de parâmetro para inicializar a thread
constructor TCar.Create(Value: Integer);
begin
    Index := Value;
    Equipe := Value div 256;
    Car := Value mod 256;
    inherited Create(False);
end; // Tcar.create

procedure TCar.Execute;
var Volta: Integer;
begin
    for Volta:=1 to 4 do // Dê 4 voltas na pista
    begin
        PrintEvent(Equipe, Format('Carro %2.2d', [Car]));
        Sleep(1000); // Espera um certo tempo no Box
        WaitForSingleObject(hBox[Equipe], INFINITE);
        WaitForSingleObject(hPista, INFINITE);
        ClearEvent(Equipe, Car, Equipe);
        PrintEvent(5, Format('Carro %2.2d Equipe %2.2d', [Car, Equipe]));
        Sleep(10*(Random(200))); // corre durante certo tempo
        ReleaseSemaphore(hPista, 1, nil);
        ReleaseMutex(hBox[Equipe]);
        ClearEvent(5, Car, Equipe);
    end
end; // Tcar.execute

end.

```

O procedimento Tcar.PrintEvent escreve no listbox sem dificuldades. Observe que para tirar as mensagens do listbox o recurso encontrado foi reconstruir o string, e procurar por ele no listbox, encontrá-lo, e deletá-lo (**procedure** Tcar.ClearEvent).

O principal ponto a ser observado neste exemplo é a necessidade de se criar a classe Tmaster para criar os objetos da classe Tcar. Uma instância de Tmaster é criada, a qual por sua vez, cria como variável local 12 instâncias de Tcar e fica aguardando pelo término das threads. Quando todas as threads terminam, MasterThread também termina. Um método da thread principal não poderia ficar à espera do término das threads da classe Tcar, porque isto bloquearia a GUI thread e a impediria de processar as mensagens emitidas por Tcar, pedindo para escrever mensagens nas ListBoxes.

Melhorando o código: Uso de exceções:

Pode-se melhorar o código anterior utilizando-se exceções. Todas as classes do Delphi, que implementam funções da API geram exceções, o que possibilita tratar os erros sem verificar o retorno das funções, como fizemos com o procedimento `CheckForError`.

Exemplo:

```
try  
  
...  
except  
  on EZeroDivide do HandleZeroDivide;  
  on EOverflow do HandleOverflow;  
  on EMathError do HandleMathError;  
else  
  HandleAllOthers;  
end;
```

O código a seguir mostra o mesmo problema resolvido utilizando-se este conceito. Apenas o código alterado é mostrado.

Unit E41Semaphore

```
unit E41Semaphore;  
  
***  
  
type  
  
***  
private  
  // Deve ser um membro privado do form para ser apagado quando o programa termina  
  MasterThread: TMaster;  
  
  procedure OnMasterThreadTerminate(Sender: TObject);  
public  
  
  end;  
  
****  
procedure TTreinoF1.ButtonStartClick(Sender: TObject);  
begin  
  if Assigned(MasterThread) then // Se o handle já existe: libera  
    MasterThread.Free;  
  // As classes do Delphi que implementam funções da API  
  // retornam os erros como exceções  
try
```

```

// O botão de iniciar fica desabilitado enquanto as threads estiverem executando
ButtonStart.Enabled := False;
MasterThread:=TMaster.Create(True);
// Cria suspensão para atribuir uma propriedade da classe
// Define um evento sem parâmetro a ser chamado quando thread terminar.
MasterThread.OnTerminate := OnMasterThreadTerminate;
MasterThread.Resume; // inicia a execução
except
// se a construtora da thread gerou um erro, o botão deve ser habilitado manualmente
ButtonStart.Enabled := True;
raise; // retorna o erro ao usuário
end;
end;

```

```

procedure TTreinoF1.OnMasterThreadTerminate(Sender: TObject);
begin
// Este método é chamado no contexto da thread primária quando a
// MasterThread termina sua execução.
ButtonStart.Enabled := True;
end;

```

end.

Unit car

```

unit Car;

***

type
TCar = class(TThread)
protected
Index: Integer;
Equipe: Integer;
Car: Integer;
procedure Execute; override;
public
constructor Create (Value: Integer);
procedure PrintEvent(Janela: Integer; Str:string);
procedure ClearEvent(Janela:Integer; Car: Integer; Equipe: Integer);
end;

***

procedure TMaster.Execute;
var Cars: array[1..MaxCars] of Tcar;
Equipe: Integer;
i: Integer;
begin
// Cria Threads
for i:=1 to MaxCars do
begin

```

```

Equipe := 1 + Random(MaxCarrosPista); // Equipe: 1 a 4
try
  Cars[i]:=Tcar.Create(i + 256 * Equipe);
except
  on E: Exception do
    ShowMessage(Format("TMaster.Execute: Erro ao criar thread de carro. %s", [E.Message]));
  end; // try / except
end; // for
// Espera threads terminarem
// Não funciona se colocada na GUI thread porque quem atualiza o ListBox é a thread principal
for i:=1 to MaxCars do
begin
  Cars[i].WaitFor;
  Cars[i].Free;
end; // for
end; // Tmaster.execute

```

Timers

O timer mais básico para uso em Delphi é um componente da classe TTimer. Este componente serve para ativar um procedimento a um intervalo de tempo estipulado.

Para instanciar um componente da classe da TTimer basta arrastar o ícone com forma de relógio da aba System para o form do Delphi.



Figura 6: Inserindo um relógio em uma aplicação

Na janela de definição de eventos devemos definir o evento OnTimer. As principais propriedades de um objeto TTimer são **Interval** que fornece o período do relógio em milissegundos e **Enabled** que serve para habilitar e desabilitar o relógio.



Figura 7: Uso do timer da classe Ttimer

Exemplo de uso do objeto TTimer:

```
unit E41Timer;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, Menus;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    Button1: TButton;
    procedure Timer1Timeout(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure BtnClick(Sender: TObject);
  private
    Ticks: LongInt;
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Timer1Timeout(Sender: TObject);
begin
  Inc(Ticks);
  Caption := Format ('Timer: %d.%d segundos', [Ticks div 10, Ticks mod 10]);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Timer1.Interval := 100; // 100 ms
  Timer1.Enabled := True;
end;

procedure TForm1.BtnClick(Sender: TObject);
begin
  Timer1.Enabled := not(Timer1.Enabled);
  if (Timer1.Enabled)
  then Button1.Caption := 'Disable Clock'
  else Button1.Caption := 'Enable Clock';
end;

end.
```

Waitable Timers

Dos timers definidos na interface Win 32 nós iremos demonstrar apenas o uso de Waitable timers. Como estes timers estão disponíveis em todos os sistemas operacionais Windows à partir do 98, esta é uma solução bastante geral.

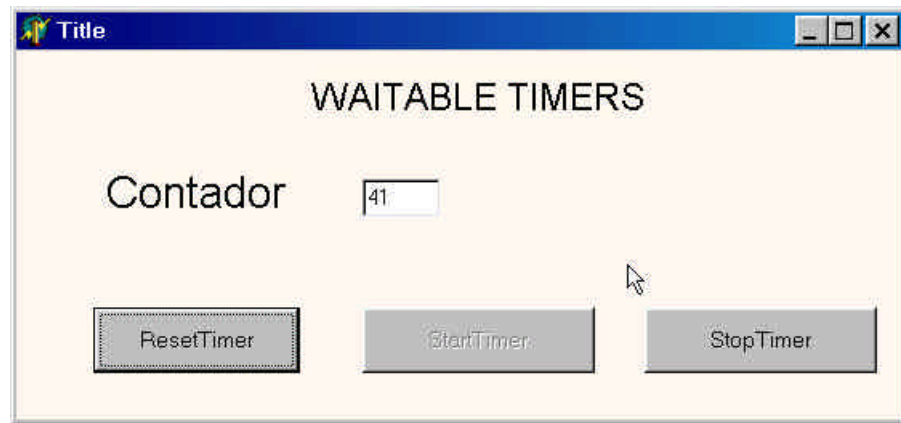


Figura 8: Waitable timers: janela da aplicação

O aplicativo cria um timer. O timer é programado para gerar um evento de temporização a cada segundo. Uma thread de serviço fica à espera deste evento e escreve o valor de um contador de eventos na tela.

```
// Demonstração do uso de Waitable Timers em Delphi
//
// Autor: Constantino Seixas Filho      Data: 09/02/2002
//
// Comentários: Programa um timer multimídia para gerar um evento de
//               temporização periodicamente a cada 1s. A thread UpdateThread
//               fica a espera do evento através de WaitForSingleObject e
//               imprime o valor do contador de eventos.
//               Este contador pode ser resetado ou parado através da interface.
//
```

```
unit E41WaitableTimers;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, UpdateThread, SyncObjs;
```

```
type
```

```
TForm1 = class(TForm)  
  Label1: TLabel;  
  btnResetTimer: TButton;
```



```

    btnStopTimer: TButton;
    btnStartTimer: TButton;
    StaticText1: TStaticText;
    Edit1: TEdit;
procedure FormCreate(Sender: TObject);
procedure btnResetTimerClick(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure btnStopTimerClick(Sender: TObject);
procedure btnStartTimerClick(Sender: TObject);
private
    UpdateThread: TUpdateThread;
public
    { Public declarations }
end;

var
    Form1: TForm1;

Implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
var
    Preset: TLargeInteger; // LARGE_INTEGER Preset;
const
    // Define uma constante para acelerar cálculo do atraso e período
    nMultiplicadorParaMs: Integer = 10000;
begin
    Form1.Edit1.Text:= IntToStr(Count);

    // cria thread para esperar pelo timer e começa a executar imediatamente
    UpdateThread := TUpdateThread.Create(False);

    // Cria timer com reset automático
    hTimer := CreateWaitableTimer(nil, FALSE, 'MyTimer');
    // Programa o temporizador para que a primeira sinalização ocorra 2s
    // depois de SetWaitableTimer
    // Use - para tempo relativo
    Preset(*.QuadPart*) := -(1000 * nMultiplicadorParaMs);
    // Dispara timer
    SetWaitableTimer(hTimer, Preset, 1000, nil, nil, FALSE);

    CS:= TcriticalSection.Create;
end;

procedure TForm1.btnResetTimerClick(Sender: TObject);
begin
    CS.Enter;
    Count := 0;

```

```

    Form1.Edit1.Text:= IntToStr(Count);
    CS.Leave;
end;

procedure TForm1.btnStopTimerClick(Sender: TObject);
begin
    CS.Enter;
    EnableTimer := False;
    btnStopTimer.Enabled := False;
    btnStartTimer.Enabled := True;
    CS.Leave;
end;

procedure TForm1.btnStartTimerClick(Sender: TObject);
begin
    CS.Enter;
    EnableTimer := True;
    btnStopTimer.Enabled := True;
    btnStartTimer.Enabled := False;
    CS.Leave
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    CloseHandle(hTimer);
    CS.Free;
end;

end.

```

Unit UpdateThread

```

unit UpdateThread;
interface

uses
    Windows, Classes, SysUtils, SyncObjs;

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

var
    hTimer: THandle;
    Count: Integer = 0;
    EnableTimer: Bool = True;

```

```
CS: TCriticalSection;
```

implementation

```
uses E41WaitableTimers;
```

```
procedure TUpdateThread.Execute;
```

```
var
```

```
  i: Integer;
```

```
begin
```

```
  for i:=1 to 100 do
```

```
    begin
```

```
      WaitForSingleObject(hTimer, INFINITE);
```

```
      CS.Enter;
```

```
      if (EnableTimer)
```

```
        then begin
```

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

```
          Count := Count + 1;
```

```
        end;
```

```
      CS.Leave;
```

```
    end;
```

```
end;
```

```
end.
```

Uso de Mensagens Assíncronas

O uso de mensagens assíncronas constitui a primeira forma de comunicação entre aplicativos em ambiente Windows. Por esta forma de comunicação uma thread coloca uma mensagem na fila de mensagens de outra thread. O envio da mensagem é feito utilizando a mensagem `PostMessage`, ou `PostThreadMessage` que será a instrução que iremos utilizar. A outra tarefa recebe e retira a mensagem da fila através da instrução `PeekMessage` (assíncrona) ou `GetMessage` (síncrona). Apenas GUI threads possuem uma fila de mensagem associada. A thread de trabalho pode entretanto obter uma fila de mensagem através do uso da instrução:

```
PeekMessage(Mensagem, 0, 0, 0, PM_NOREMOVE);
```

Como foi visto no curso de programação multithreading em C++, o identificador de mensagem `WM_APP` deverá ser usado para identificar a mensagem.

No próximo exemplo um programa cria duas threads. A primeira recebe e envia a entrada de strings através de duas caixas de edição e as envia para a thread servidora, quando um botão é acionado. A thread servidora exibe o **string** recebido em um listbox.

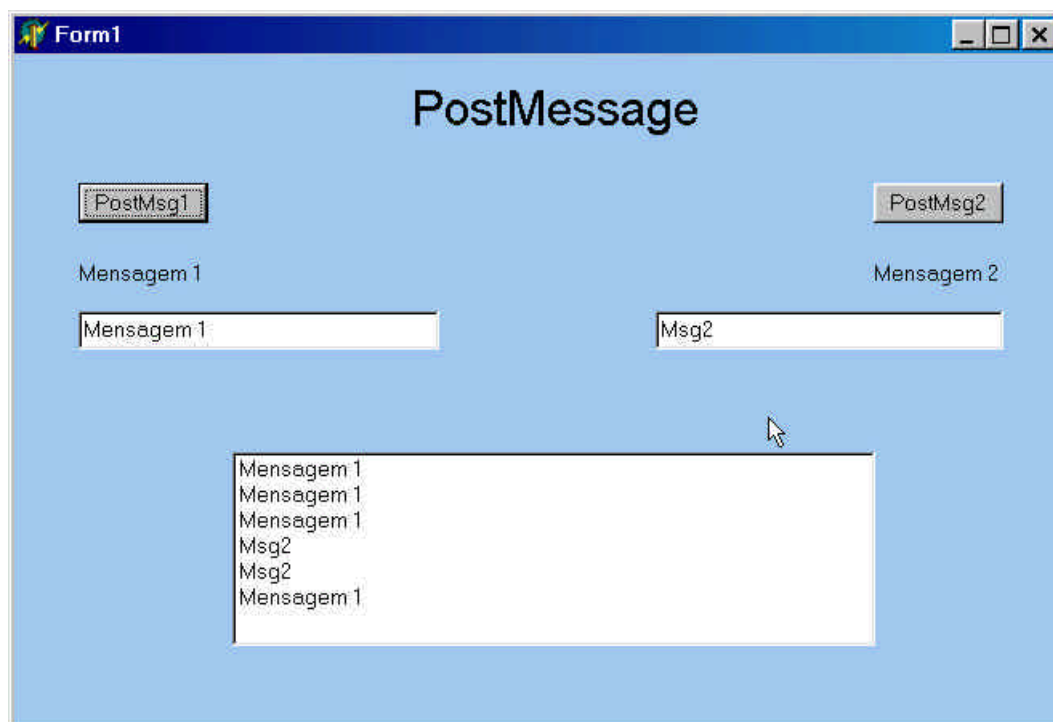


Figura 9: Janela do aplicativo PostMessage

PostMessage

```
BOOL PostThreadMessage(
```

```

    IdThread: DWORD; // Identificador da thread
    Msg: UINT; // Mensagem a ser enviada: WM_USER; WM_APP.
    WParam: WPARAM; // Primeiro parâmetro da mensagem
    LParam: LPARAM; // Segundo parâmetro da mensagem
);
```

Retorno da função:

Status	Interpretação
<> 0	Sucesso
0	Erro; Use GetLastError para identificar a causa do erro.

Exemplo: Cliente- GUI thread

```

//
// Demonstração do uso de PostThreadMessage em Delphi
//
// Autor: Constantino Seixas Filho    Data: 09/04/2002
//
// Comentários: A função executa da working thread cria uma fila de mensagens
```

```

//      através da função PeekMessage e trata todas as mensagens
//      recebidas diretamente.
//      Neste exemplo strings são enviados diretamente.
//

unit E61PostMsg;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    lstListBox1: TListBox;
    Edit1: TEdit;
    Edit2: TEdit;
    btnSendMsg1: TButton;
    btnSendMsg2: TButton;
    Label2: TLabel;
    Label3: TLabel;
    procedure btnSendMsg1Click(Sender: TObject);
    procedure btnSendMsg2Click(Sender: TObject);
    procedure Edit1Click(Sender: TObject);
    procedure Edit1Exit(Sender: TObject);
    procedure Edit2Click(Sender: TObject);
    procedure Edit2Exit(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    ServerThread: TServerThread;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Strings: array[1..2] of string;
  MasterUp: TEvent; // Sinaliza quando Server estiver pronto
  MyCs: TCriticalSection;

implementation

{$R *.dfm}

procedure TForm1.Edit1Click(Sender: TObject);
begin
  Strings[1] := ";

```

```
MyCs.Enter;  
Edit1.Text:= Strings[1];  
MyCS.Leave;  
end;
```

```
procedure TForm1.Edit1Exit(Sender: TObject);  
begin  
    Strings[1] := Edit1.Text;  
end;
```

```
procedure TForm1.Edit2Click(Sender: TObject);  
begin  
    Strings[2] := "  
    MyCS.Enter;  
    Edit2.Text:= Strings[2];  
    MyCS.Leave;  
end;
```

```
procedure TForm1.Edit2Exit(Sender: TObject);  
begin  
    Strings[2] := Edit2.Text;  
end;
```

```
procedure TForm1.btnSendMsg1Click(Sender: TObject);  
begin  
    // Envia Mensagem  
    PostThreadMessage(ServerThread.ThreadID,WM_APP,Wparam(Strings[1],0);  
    // ShowMessage('MsgSent='+ Msg); // Ative para ver tamanho da mensagem  
end;
```

```
procedure TForm1.btnSendMsg2Click(Sender: TObject);  
begin  
    // Envia Mensagem  
    PostThreadMessage(ServerThread.ThreadID,WM_APP,Wparam(Strings[2],0);  
    // ShowMessage('MsgSent='+ Msg); // Ative para ver tamanho da mensagem  
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    MasterUp := TEvent.Create(nil, True, FALSE, 'MasterUp'); // Reset Manual  
    MyCS := TCriticalSection.Create;  
    // Cria thread servidora e começa a executar  
    ServerThread := TServerThread.Create(False);  
    MasterUp.WaitFor(INFINITE); // Espera Servidor estar pronto  
end;
```

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    MasterUp.Free;  
    MyCS.Free;
```

```

    // Pede para thread terminar e espera término da thread
    PostThreadMessage(ServerThread.ThreadID, WM_QUIT, 0, 0);
    ServerThread.WaitFor;
end;

end.

```

Servidor: Working Thread

```

unit PostMsgServer;

interface

uses
    Classes, Windows, SyncObjs, SysUtils, QDialogs, Messages;

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

implementation
uses
    E61PostMsg;
procedure TServerThread.Execute;
const
    MaxMsgSize: Integer = 40;
var
    MsgBuffer: string;
    Mensagem: TMSG;
begin
    // Cria fila de mensagens, mesmo não sendo GUI Thread
    PeekMessage(Mensagem, 0, 0, WM_APP, PM_NOREMOVE);
    MasterUp.SetEvent; // Servidor está pronto

    while not Terminated do
    begin
        GetMessage(Mensagem, 0, 0, WM_APP + 5);
        if (Mensagem.message = WM_APP)
        then begin
            MsgBuffer := Pchar(Mensagem.wParam);
            // Exibe Mensagem
            MyCS.Enter;
            Form1.lstListBox1.Items.Add(MsgBuffer);
            MyCS.Leave;
        end
    end

```

```

        else if (Mensagem.message = WM_QUIT)
            then ExitThread(0);
    end
end;

end.

```

Esta solução peca por um inconveniente. É quase que uma transcrição do programa construído em C++ para Delphi. A linguagem Delphi possui uma diretiva de mais alto nível que define um *handler* para mensagens Windows definidas pelo usuário. A chamada do cliente é exatamente a mesma. Uma mensagem do tipo WM_APP + n ou WM_USER + n é enviada através da diretiva PostThreadMessage. Do lado do servidor um handler para a mensagem deve ser definido em qualquer componente Delphi que receba a mensagem, no nosso caso o objeto thread.

```

const
    TH_MSG = WM_APP + 1;
type
    TMyComponent = class(...)
        ...
    private
        procedure HandleMessage(var Message: Tmessage); message TH_MSG;
    public
    end;

procedure TMyComponent.TH_MSG(var msg: TMessage);
begin

end;

```

Uma segunda solução para este problema será agora mostrada. Nesta solução as mensagens são formatadas na forma de estruturas (records) para ilustrar como definir mensagens heterogêneas em Delphi. Observe no exemplo do servidor que um loop de despacho de mensagens teve que ser construído na procedure TServerThread.Execute. Não seria necessária a construção deste loop de mensagem caso a thread de destino para a mensagem fosse uma GUI thread.

Exemplo: Cliente - GUI thread

```

// Demonstração do uso de PostThreadMessage em Delphi
//
// Versão 2: Uso de Records
//
// Autor: Constantino Seixas Filho    Data: 21/05/2002
//
// Comentários: A função execute da working thread cria uma fila de mensagens
//               através da função PeekMessage e despacha as mensagens WM_APP
//               recebidas para uma função dedicada.

```



```
// Neste exemplo é criado um record contendo os membros da mensagem
// a ser enviada.
//
```

```
unit E61PostMsg2;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, PostMsgServer2, SyncObjs;
```

```
type
```

```
TForm1 = class(TForm)  
  Label1: TLabel;  
  lstListBox1: TListBox;  
  Edit1: TEdit;  
  Edit2: TEdit;  
  btnSendMsg1: TButton;  
  btnSendMsg2: TButton;  
  Label2: TLabel;  
  Label3: TLabel;  
  procedure btnSendMsg1Click(Sender: TObject);  
  procedure btnSendMsg2Click(Sender: TObject);  
  procedure Edit1Click(Sender: TObject);  
  procedure Edit1Exit(Sender: TObject);  
  procedure Edit2Click(Sender: TObject);  
  procedure Edit2Exit(Sender: TObject);  
  procedure FormCreate(Sender: TObject);  
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```
private
```

```
  ServerThread: TServerThread;  
  Strings: array[1..2] of string;
```

```
public
```

```
  { Public declarations }
```

```
end;
```

```
PStdMsg = ^StdMsg;
```

```
StdMsg = record
```

```
  MyStr: string;
```

```
  Length: Integer;
```

```
end;
```

```
var
```

```
  Form1: TForm1;
```

```
  MasterUp: TEvent; // Sinaliza quando Server estiver pronto
```

```
  MyCs: TCriticalSection;
```

```
implementation
```

```
{ $R *.dfm }
```

```
procedure TForm1.Edit1Click(Sender: TObject);
```

```
begin
```

```
  Strings[1] := '';
```

```
  MyCs.Enter;
```

```
  Edit1.Text:= Strings[1];
```

```
  MyCS.Leave;
```

```
end;
```

```
procedure TForm1.Edit1Exit(Sender: TObject);
```

```
begin
```

```
  Strings[1] := Edit1.Text;
```

```
end;
```

```
procedure TForm1.Edit2Click(Sender: TObject);
```

```
begin
```

```
  Strings[2] := '';
```

```
  MyCS.Enter;
```

```
  Edit2.Text:= Strings[2];
```

```
  MyCS.Leave;
```

```
end;
```

```
procedure TForm1.Edit2Exit(Sender: TObject);
```

```
begin
```

```
  Strings[2] := Edit2.Text;
```

```
end;
```

```
procedure TForm1.btnSendMsg1Click(Sender: TObject);
```

```
var
```

```
  P: PStdMsg;
```

```
begin
```

```
  new(P);
```

```
  P^.MyStr:= Strings[1];
```

```
  P^.Length:= Length(Strings[1]);
```

```
  // Envia Mensagem
```

```
  PostThreadMessage(ServerThread.ThreadID, WM_APP, Wparam(P), 0);
```

```
  // ShowMessage('MsgSent= '+ P^.MyStr); // Ative para ver mensagem
```

```
end;
```

```
procedure TForm1.btnSendMsg2Click(Sender: TObject);
```

```
var
```

```
  P: PStdMsg;
```

```
begin
```

```
  new(P);
```

```
  P^.MyStr:= Strings[2];
```

```
  P^.Length:= Length(Strings[2]);
```

```
  // Envia Mensagem
```

```
  PostThreadMessage(ServerThread.ThreadID, WM_APP, Wparam(P), 0);
```

```
  // ShowMessage('MsgSent= '+ P^.MyStr); // Ative para ver mensagem
```

```

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    MasterUp := TEvent.Create(nil, True, FALSE, 'MasterUp'); // Reset Manual
    MyCS := TCriticalSection.Create;
    // Cria thread servidora e começa a executar
    ServerThread := TServerThread.Create(False);

    MasterUp.WaitFor(INFINITE); // Espera Servidor estar pronto
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    MasterUp.Free;
    MyCS.Free;
    PostThreadMessage(ServerThread.ThreadID, WM_QUIT, 0, 0);
    // Pede para thread terminar
    ServerThread.WaitFor;
end;

end.

```

Exemplo Servidor

```

unit PostMsgServer2;

interface

uses
    Classes, Windows, SyncObjs, SysUtils, QDialogs, Messages, Forms;

type
    TServerThread = class(TThread)
    private
        { Private declarations }
    protected
        procedure Execute; override;
        procedure WMAp(var msg: TMessage); message WM_APP;
    end;

implementation
uses
    E61PostMsg2;

procedure TServerThread.Execute;
var
    Msg: TMsg;
    DMsg: TMessage;

```

```

begin
  // Cria fila de mensagens, mesmo não sendo GUI Thread
  PeekMessage(Msg, 0, 0, 0, PM_NOREMOVE);
  MasterUp.SetEvent;          // Servidor está pronto

  while not Terminated do begin
    if GetMessage(Msg, 0, 0, WM_APP + 5)
    then begin
      DMsg.Msg:=Msg.message;
      DMsg.wParam:=Msg.wParam;
      DMsg.lParam:=Msg.lParam;
      DMsg.Result:=0;
      Dispatch(DMsg);
    end
    else ExitThread(0);      // WM_QUIT foi recebido: termina thread
  end;

end;

procedure TServerThread.WMApp(var msg: TMessage);
const
  MaxMsgSize: Integer = 40;
var
  MsgBuffer: string;
  P: PStdMsg;
begin
  P := PStdMsg(msg.wParam);
  MsgBuffer := P^.MyStr;
  Dispose(P); // Libera memória alocada

  // Exibe Mensagem
  MyCS.Enter;
  Form1.lstListBox1.Items.Add(MsgBuffer);
  MyCS.Leave;
end;

end.

```

Uso de Mailslots

Mailslots são usados para a comunicação unidirecional entre aplicativos, em um ambiente distribuído. Neste exemplo vamos solucionar o mesmo problema de comunicar mensagens entre uma aplicação que lê strings definidas pelo usuário e outra que os recebe e os exibe.

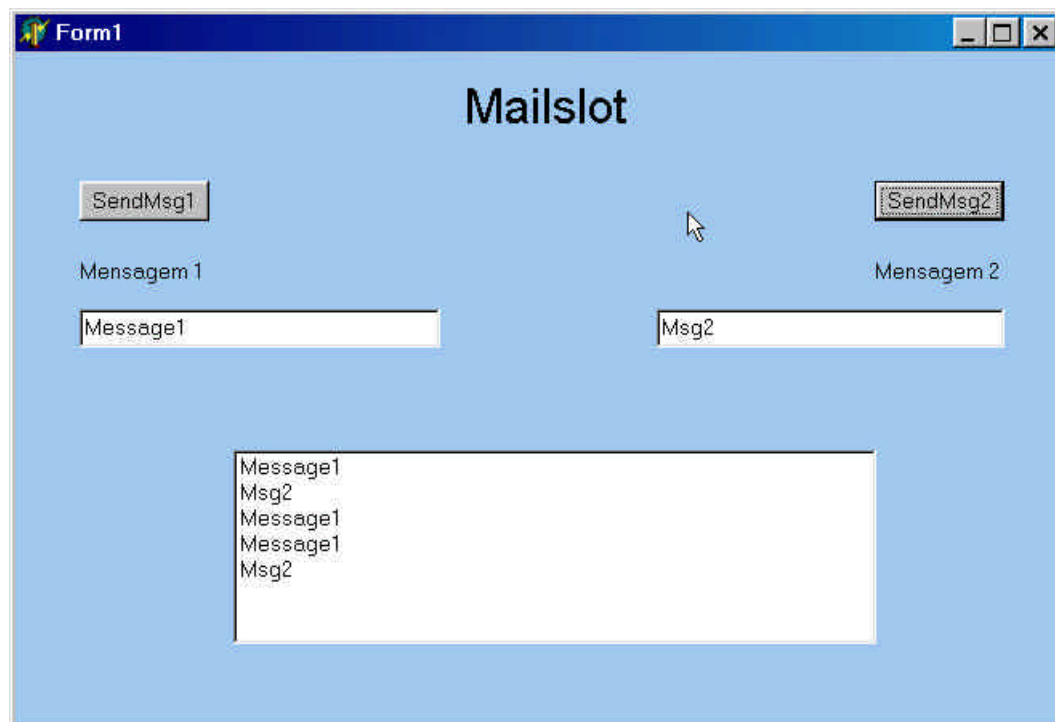


Figura 10: Troca de mensagens entre aplicativos usando Mailslot

Embora Mailslots possam ser usados para comunicar processos diferentes, neste caso está sendo usado na comunicação entre threads. A thread servidora irá criar um Mailslot e ativar um evento avisando que está apta a aceitar mensagens. A thread cliente esperará por este evento e à partir daí passará a enviar mensagens diretamente para o Mailslot através da instrução WriteFile. Como as mensagens podem ter um comprimento variável e estão sendo enviadas como strings, um evento está sendo ativado para avisar ao servidor que a mensagem está disponível no buffer de recepção. A thread servidora pode então perguntar o tamanho da mensagem recebida e realizar uma leitura síncrona.

As principais funções utilizadas são:

Instrução GetMailSlotInfo

Function GetMailSlotInfo(

HMailslot: THandle,	// Handle para Mailslot
lpMaxMessageSize: Pointer,	// Endereço do buffer contendo o maior // tamanho possível da mensagem.
var lpNextSize: DWORD,	// Tamanho da próxima mensagem.
lpMessageCount: Pointer,	// Endereço do número de mensagens
lpReadTimeout: Pointer	// Endereço de <i>timeout</i> de leitura
); BOOL;	

Retorno da função:

Status	Interpretação
<> 0	Sucesso
0	Erro; Use GetLastError para identificar a causa do erro.

Instrução ReadFile

```
function ReadFile(
```

```
hFile: THandle, // Handle para arquivo a ser lido  
  
var Buffer: DWORD, // Buffer de recepção da mensagem  
var nNumberOfBytesToRead: // Número de bytes a serem lidos  
    DWORD,  
var lpNumberOfBytesRead: // Número de bytes lidos  
    DWORD,  
lpOverlapped: POverlapped // Apontador para estrutura Overlapped  
): BOOL;
```

Retorno da função:

Status	Interpretação
<> 0	Sucesso
0	Erro; Use GetLastError para identificar a causa do erro.

Instrução WriteFile

```
function WriteFile(
```

```
hFile: THandle, // Handle para arquivo a ser escrito  
  
const Buffer, // Buffer com dados a serem transmitidos  
nNumberOfBytesToWrite: // Número de bytes a serem escritos  
    DWORD,  
var lpNumberOfBytesWritten: // Número de bytes escritos  
    DWORD,  
lpOverlapped: POverlapped // Apontador para estrutura Overlapped  
): BOOL;
```

Retorno da função:

Status	Interpretação
<> 0	Sucesso
0	Erro; Use GetLastError para identificar a causa do erro.

Observe que o buffer é passado como um *const parameter*, isto é o seu conteúdo não pode ser modificado pela rotina WriteFile, nem ele poderia ser passado como um *var parameter* para outra rotina dentro do corpo de WriteFile.

Mailslot: Programa principal

```
// Demonstração do uso de Mailslots em Delphi
//
// Autor: Constantino Seixas Filho      Data: 07/03/2002
//
// Comentários: O programa servidor irá criar um Mailslot e ficar à espera de
//              mensagens. O programa cliente formata mensagens e as envia para
//              o Mailslot designado.
//
unit E61Mailslot;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
    lstListBox1: TListBox;
    Edit1: TEdit;
    Edit2: TEdit;
    btnSendMsg1: TButton;
    btnSendMsg2: TButton;
    Label2: TLabel;
    Label3: TLabel;
    procedure btnSendMsg1Click(Sender: TObject);
    procedure btnSendMsg2Click(Sender: TObject);
    procedure Edit1Click(Sender: TObject);
    procedure Edit1Exit(Sender: TObject);
    procedure Edit2Click(Sender: TObject);
    procedure Edit2Exit(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    ServerThread: TServerThread;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Strings: array[1..2] of string = ('Mystring1','MyString2');
```

```
DataSentEvent: TEvent; // Sinaliza que mensagem foi colocada no Mailslot
MasterUp: TEvent;      // Sinaliza quando Mailslot for criado
MyCs: TCriticalSection;
hMailslot: THandle; // Handle do servidor para Mailslot
```

implementation

```
{ $R *.dfm }
```

```
procedure TForm1.Edit1Click(Sender: TObject);
```

```
begin
```

```
  Strings[1] := "";
  MyCs.Enter;
  Edit1.Text:= Strings[1];
  MyCS.Leave;
```

```
end;
```

```
procedure TForm1.Edit1Exit(Sender: TObject);
```

```
begin
```

```
  Strings[1] := Edit1.Text;
```

```
end;
```

```
procedure TForm1.Edit2Click(Sender: TObject);
```

```
begin
```

```
  Strings[2] := "";
  MyCS.Enter;
  Edit2.Text:= Strings[2];
  MyCS.Leave;
```

```
end;
```

```
procedure TForm1.Edit2Exit(Sender: TObject);
```

```
begin
```

```
  Strings[2] := Edit2.Text;
```

```
end;
```

```
procedure TForm1.btnSendMsg1Click(Sender: TObject);
```

```
var
```

```
  Msg: string;
```

```
  BytesWritten: DWORD; // Número de bytes escritos no Mailslot
```

```
begin
```

```
  Msg := Strings[1];
  // Escreve mensagem no Mailslot
  WriteFile(hMailslot, Pointer(Msg)^, Length(Msg), BytesWritten, nil);
  // ShowMessage('MsgSent='+ Msg + ' Size =' + IntToStr(BytesWritten));
  // Ative para ver tamanho da mensagem
  // Avisa que mensagem de tamanho variável foi enviada
  DataSentEvent.SetEvent;
```

```
end;
```

```
procedure TForm1.btnSendMsg2Click(Sender: TObject);
```

```
var
```



```

Msg: string;
BytesWritten: DWORD; // Número de bytes escritos no Mailsot
begin
    Msg := Strings[2];
    // Escreve mensagem no Mailslot
    WriteFile(hMailslot, Pointer(Msg)^, Length(Msg), BytesWritten, nil);
    // ShowMessage('MsgSent='+ Msg + ' Size =' + IntToStr(BytesWritten));
    // Ative para ver tamanho da mensagem
    // Avisa que mensagem de tamanho variável foi enviada
    DataSentEvent.SetEvent;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    MasterUp := TEvent.Create(nil, True, FALSE, 'MasterUp'); // Reset Manual
    DataSentEvent:=TEvent.Create(nil, False, FALSE,
    'EventoDadoEnviadoMailSlot');
    MyCS := TCriticalSection.Create;

    // Cria thread servidora e começa a executar
    ServerThread := TServerThread.Create(False);

    MasterUp.WaitFor(INFINITE); // Espera Mailsot ser criado
    // Abre Mailslot
    hMailslot := CreateFile("\\.\mailslot\Windows\Temp\MyMailslot',
        GENERIC_WRITE,
        FILE_SHARE_READ,
        nil,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        0); // Handle para arquivo com atributos a serem copiados
    if hMailslot = INVALID_HANDLE_VALUE then
        raise exception.create('Não consegui abrir Mailslot');
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    MasterUp.Free;
    DataSentEvent.Free;
    MyCS.Free;
    CloseHandle(hMailslot);
end;

end.

```

Mailslot: Programa Servidor

```
unit MailslotServer;
```

```
interface
```

```

uses
  Classes, Windows, SyncObjs, SysUtils, QDialogs;
type
  TServerThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation
uses
  E61Mailslot;

procedure TServerThread.Execute;
const
  MaxMsgSize: Integer = 40;
var
  BytesLidos: DWORD;    // Bytes lidos do Mailslot
  MsgBuffer: string;
  NextMsgSize: DWORD; // Tamanho da próxima msg no buffer
  MsgCount: DWORD;    // Número de Mensagens no Mailslot
  hMailslot: THandle;

begin
  hMailslot := CreateMailslot("\\.\mailslot\Windows\Temp\MyMailslot',
    MaxMsgSize, 0, nil);
  if hMailslot = INVALID_HANDLE_VALUE then
    raise exception.create('Nao consegui criar Mailslot');
  MasterUp.SetEvent; // Mailslot foi criado

  while True do
  begin
    DataSentEvent.WaitFor(INFINITE); // Para evitar espera ocupada
    // Verifica tamanho da próxima mensagem
    GetMailslotInfo(hMailSlot, nil, NextMsgSize, @MsgCount, nil);
    ShowMessage('MsgSize=' + IntToStr(NextMsgSize));
    // Ative para ver tamanho da mensagem

    // Ajusta tamanho do string convenientemente
    SetLength(MsgBuffer, NextMsgSize);
    ReadFile(hMailSlot, PChar(MsgBuffer)^, NextMsgSize, BytesLidos, nil);
    // Exibe Mensagem
    MyCS.Enter;
    Form1.lstListBox1.Items.Add(MsgBuffer);
    MyCS.Leave;
  end
end;
end.

```

Pipes

O mesmo programa utilizado para demonstrar o uso de Mailslots foi adaptado para demonstrar o uso de pipes. Uma diferença básica é que o servidor de pipe tem que necessariamente rodar uma das versões do WNT ou sucessores.

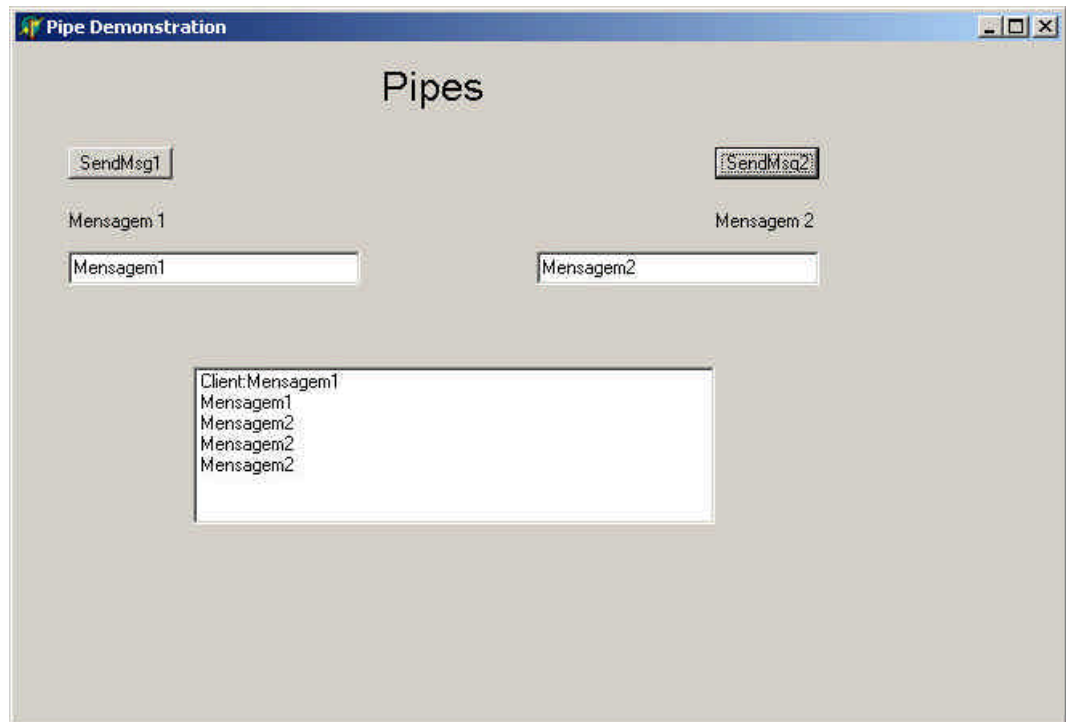


Figura 11: Demonstrativo do uso de Pipes em Delphi

Programa Cliente

```
//  
// Demonstração do uso de Pipes em Delphi  
//  
// Autor: Constantino Seixas Filho      Data: 07/03/2002  
//  
// Comentários: O programa servidor irá criar um Pipe e ficar à espera de  
//              mensagens. O programa cliente formata mensagens e as envia para  
//              o Pipe designado. O servidor de pipes só pode ser executado no  
//              Windows NT ou Windows 2000.  
//
```

```
unit E61Pipe;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs, ExtCtrls, Menus, PipeServer, SyncObjs, StdCtrls;
```

```

type
  TForm1 = class(TForm)
    Label1: TLabel;
    lstListBox1: TListBox;
    Edit1: TEdit;
    Edit2: TEdit;
    btnSendMsg1: TButton;
    btnSendMsg2: TButton;
    Label2: TLabel;
    Label3: TLabel;
    procedure btnSendMsg1Click(Sender: TObject);
    procedure btnSendMsg2Click(Sender: TObject);
    procedure Edit1Click(Sender: TObject);
    procedure Edit1Exit(Sender: TObject);
    procedure Edit2Click(Sender: TObject);
    procedure Edit2Exit(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    ServerThread: TServerThread;
    procedure GetOS(var OSId: integer; var OSName: string);
  public
    procedure CreateWnd; override;
  end;

var
  Form1: TForm1;
  Strings: array[1..2] of string = ('Mystring1','MyString2');
  MasterUp: TEvent; // Sinaliza quando Pipe for criado
  MyCs: TCriticalSection;
  hPipe: THandle; // Handle do servidor para Pipe
  Pipename: string = '\\.\pipe\Pipe63';

```

implementation

```

{$R *.dfm}

procedure TForm1.GetOS(var OSId: integer; var OSName: string);
var
  VersionInfo: OsVersionInfo;
const
  OS: array[0..6] of string = ('Inválido','Win95','Win98','WinMe','WNT',
    'Win2000','WinXP');
begin
  VersionInfo.dwOSVersionInfoSize := sizeof(VersionInfo);
  GetVersionEx(VersionInfo);

  with VersionInfo do
    begin

```

```

If (dwPlatformId = VER_PLATFORM_WIN32_WINDOWS) // Windows 95
and (dwMajorVersion = 4) and (dwMinorVersion = 0)
then OSId:= 1

else If (dwPlatformId = VER_PLATFORM_WIN32_WINDOWS) // Win 98
and (dwMajorVersion > 4) or ((dwMajorVersion = 4) and
(dwMinorVersion > 0))
then OSId:= 2

else If (dwPlatformId = VER_PLATFORM_WIN32_WINDOWS) // Win Me
and (dwMajorVersion = 4) and (dwMinorVersion = 90)
then OSId:= 3

else If (dwPlatformId = VER_PLATFORM_WIN32_NT) // WNT (NT 4)
and (dwMajorVersion = 4)
then OSId:= 4

else If (dwPlatformId = VER_PLATFORM_WIN32_NT) // Win2000 (NT 5 )
and (dwMajorVersion = 5) and (dwMinorVersion = 0)
then OSId:= 5

else If (dwPlatformId = VER_PLATFORM_WIN32_NT) // WinXP (NT 5.1)
and (dwMajorVersion = 5) and (dwMinorVersion = 1)
then OSId:= 4;
end; // with
OSName:= OS[OSId];
end; // GetOS

// Antes de começar verifique se o sistema operacional suporta pipes
// O servidor de pipes deve rodar em > Windows NT
procedure TForm1.CreateWnd;
var
    OSId: Integer;
    OSName: string;
begin
    inherited CreateWnd;

    GetOS(OSId, OSName);
    ShowMessage('Sistema Operacional '+ OSName );
    If (OsId < 4)
        then begin
            ShowMessage('Sistema Operacional '+ OSName +
            ': Pipe não disponível');
            ExitProcess(0);
        end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    MasterUp := TEvent.Create(nil, True, FALSE, 'MasterUp'); // Reset Manual

```

```

MyCS := TCriticalSection.Create;

// Cria thread servidora e começa a executar
ServerThread := TServerThread.Create(False);

MasterUp.WaitFor(INFINITE); // Espera Pipe ser criado

// Todas as instâncias estão ocupadas, então espere pelo tempo default
if (WaitNamedPipe(Pchar(Pipename), NMPWAIT_USE_DEFAULT_WAIT) =
False)
then ShowMessage('Esperando por uma instância do pipe...');

hPipe := CreateFile(
    Pchar(Pipename), // nome do pipe
    GENERIC_WRITE, // acesso para escrita
    0, // sem compartilhamento
    nil, // IpSecurityAttributes
    OPEN_EXISTING, // dwCreationDistribution
    FILE_ATTRIBUTE_NORMAL, // dwFlagsAndAttributes
    0); // hTemplate
if hPipe = INVALID_HANDLE_VALUE
then raise exception.create('Não consegui abrir Pipe');
end;

procedure TForm1.Edit1Click(Sender: TObject);
begin
    MyCs.Enter;
    Strings[1] := '';
    Edit1.Text:= Strings[1];
    MyCS.Leave;
end;

procedure TForm1.Edit1Exit(Sender: TObject);
begin
    MyCs.Enter;
    Strings[1] := Edit1.Text;
    MyCS.Leave;
end;

procedure TForm1.Edit2Click(Sender: TObject);
begin
    MyCS.Enter;
    Strings[2] := '';
    Edit2.Text:= Strings[2];
    MyCS.Leave;
end;

procedure TForm1.Edit2Exit(Sender: TObject);
begin
    MyCs.Enter;

```

```

    Strings[2] := Edit2.Text;
    MyCS.Leave;
end;

procedure TForm1.btnSendMessage1Click(Sender: TObject);
var
    Msg: string;
    BytesWritten: DWORD; // Número de bytes escritos no Pipe
begin
    MyCs.Enter;
    Msg := Strings[1];
    MyCS.Leave;
    // Escreve mensagem no Pipe: escrita síncrona
    WriteFile(hPipe, Msg, Length(Msg), BytesWritten, nil);
    // Imprime o que enviou (Melhor que ShowMessage)
    MyCS.Enter;;
    Form1.lstListBox1.Items.Add('Client:'+ Msg);
    MyCS.Leave;
end;

procedure TForm1.btnSendMessage2Click(Sender: TObject);
var
    Msg: string;
    BytesWritten: DWORD; // Número de bytes escritos no Pipe
    P: ^byte;
begin
    MyCs.Enter;
    Msg := Strings[2];
    MyCS.Leave;
    P := @Msg;
    // Escreve mensagem no Pipe
    WriteFile(hPipe, P^, Length(Msg), BytesWritten, nil);
    // ShowMessage('MsgSent='+ Msg + ' Size = ' + IntToStr(BytesWritten));
    // Ative para ver tamanho da mensagem
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var Msg: Byte;
    BytesWritten: DWORD; // Número de bytes escritos no Pipe
begin
    // Escreve mensagem no Pipe para abortar thread secundária
    WriteFile(hPipe, Msg, 0, BytesWritten, nil);
    ServerThread.WaitFor; // espera pela morte da thread
    MasterUp.Free;
    MyCS.Free;
    if hPipe <> 0 then CloseHandle(hPipe);
end;
end.

```

Servidor

```

unit PipeServer;

interface

uses
  Classes, Windows, SyncObjs, SysUtils, Dialogs;

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

implementation
uses
  E61Pipe;

procedure TServerThread.Execute;
const
  MaxMsgSize: Integer = 255;
var
  BytesLidos: DWORD;    // Bytes lidos do Mailslot
  MsgBuffer: string;
  hPipe: THandle;
  ErrorCode: DWORD;
  Status: Boolean;
  Abort: Boolean;
  bStatus: Boolean;
  P: ^Byte;
begin
  Abort := False;
  hPipe := CreateNamedPipe( // Cria pipe nomeado
    Pchar(PipeName),
    PIPE_ACCESS_DUPLEX, // Comunicação Full Duplex
    PIPE_TYPE_MESSAGE + PIPE_READMODE_MESSAGE +
    PIPE_WAIT,
    1, // Número de instâncias
    0, // nOutBufferSize
    0, // nInBufferSize
    INFINITE, // Timeout para esperar por cliente
    nil); // Atributos de segurança
  if hPipe = INVALID_HANDLE_VALUE then
    raise exception.create('Nao consegui criar Pipe');
  MasterUp.SetEvent; // Pipe foi criado

  Status := ConnectNamedPipe(hPipe, nil); // Espera por uma conexão
  if (Status)
    then ShowMessage('Cliente se conectou com sucesso')

```



```

else begin
    ErrorCode := GetLastError();
    if (ErrorCode = ERROR_PIPE_CONNECTED)
        then ShowMessage('Cliente já havia se conectado')
        else if (ErrorCode = ERROR_NO_DATA)
            then begin
                ShowMessage('Cliente fechou seu handle');
                Abort := True;
            end // if
        else ShowMessage('Erro ' + IntToStr(ErrorCode) );
            // 536 = ERROR_PIPE_LISTENING
end; // else

P := @MsgBuffer;

while not Abort do
begin
    // Espera Mensagem: Leitura síncrona
    bStatus := ReadFile(hPipe, P^, MaxMsgSize, BytesLidos, nil);
    If not bStatus then
    begin
        MyCS.Enter;
        Form1.lstListBox1.Items.Add('Erro de leitura: ' + IntToStr(GetLastError));
        MyCS.Leave;
    end;
    //SetLength(MsgBuffer, BytesLidos);
    if (BytesLidos = 0)
    then Abort:= True
    else begin
        // Exibe Mensagem
        MyCS.Enter;
        Form1.lstListBox1.Items.Add(MsgBuffer);
        MyCS.Leave;
    end
end;
ExitThread(0);
end;

end.

```

TMultiReadExclusiveWriteSynchronizer

Esta classe proporciona uma solução perfeita para o problema dos leitores e escritores, garantindo que N threads possam ler de forma concorrente um objeto compartilhado, mas que apenas uma thread possa acessar o objeto para escrita.

var

RdWrLock: TMultiReadExclusiveWriteSynchronizer;

Cada thread desejando ler o objeto deve executar o protocolo:

RdWrLock.BeginRead

Acesso à variável

RdWrLock.EndRead

Cada escritor deverá executar:

RdWrLock.BeginWrite

Acesso à variável

RdWrLock.EndWrite

Este tipo de controle é muito mais eficiente que o controle utilizando sessões críticas para resolver o problema de múltiplos leitores e escritores. Apenas um escritor poderá acessar a variável de cada vez. Quando um escritor estiver acessando, nenhum leitor o fará. Vários leitores podem conviver pacificamente lendo a variável.

Os principais métodos desta classe são:

Método	Função
Create	Istancia o objeto do tipo TmultiReadExclusiveWriteSynchronizer. Deve ser declarado na mesma thread onde está a memória que ele protege.
BeginRead	Fica bloqueado até não haja nenhuma outra thread realizando a escrita na memória compartilhada.
EndRead	Encerra leitura protegida à memória compartilhada. Cada chamada a BeginRead deve ser pareada a uma chamada a EndRead.
BeginWrite	Fica bloqueada até que nenhuma outra thread esteja lendo ou escrevendo na memória protegida. Retorna True se a memória protegida não foi modificada por nenhuma outra thread depois que BeginWrite foi emitido e False caso contrário.
EndWrite	Encerra escrita protegida à memória compartilhada. Cada chamada a BeginWrite deve ser pareada a uma chamada a EndWrite.
Free	Destrói o objeto e libera a memória associada.

Programa de exemplo:

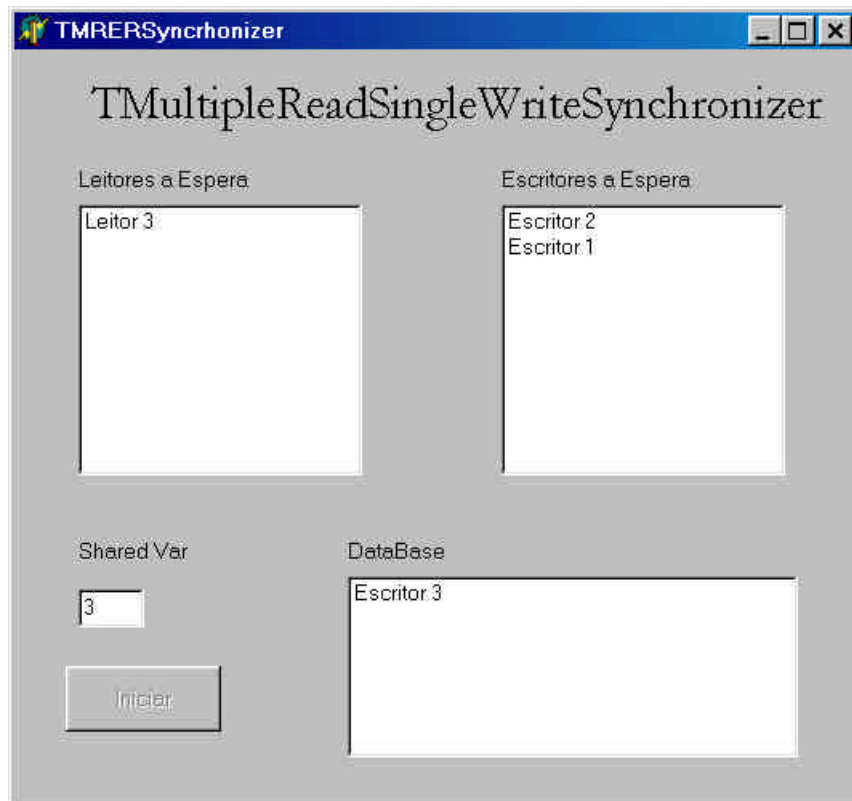


Figura 12: Programa para teste com múltiplos leitores e escritores

No programa de exemplo são criados 5 leitores e 3 escritores que tentam aleatoriamente realizar operações numa base de dados compartilhada. Os leitores ou escritores querendo realizar uma operação chamam a função `TmultipleReadSingleWriteSynchronizer` e são mostrados numa janela que representa a fila de threads à espera para leitura ou Escrita. A terceira janela mostra as threads que estão acessando a variável compartilhada num dado instante.

Programa Principal

```
//
// Demonstração do uso do recurso TMultiReadExclusiveWriteSynchronizer
//
// Autor: Constantino Seixas Filho      Data: 26/05/2002
//
// Comentários: Este programa ilustra o problema dos leitores e escritores.
//               Threads leitoras e escritoras tentam acessar uma variável.
//               O programa exibe os leitores e os escritores em suas filas.
//               Quando um leitor ou escritor consegue entrar na área
//               compartilhada, a sua identificação é exibida.
//
```

unit E41MRERSync;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, Reader, Writer;

const

MaxReaders = 5;
MaxWriters = 3;
LISTBOX_READER = 1;
LISTBOX_WRITER = 2;
LISTBOX_DATABASE = 3;

type

TForm1 = **class**(TForm)
 btnIniciar: TButton;
 lstLeitores: TListBox;
 lstEscritores: TListBox;
 lstDatabase: TListBox;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Label6: TLabel;
 EditVar: TEdit;
 procedure FormCreate(Sender: TObject);
 procedure FormClose(Sender: TObject; **var** Action: TCloseAction);
 procedure btnIniciarClick(Sender: TObject);
private
 ReaderThreads: **array** [1..MaxReaders] of TReaderThread;
 WriterThreads: **array** [1..MaxWriters] of TWriterThread;
public
 procedure PrintEvent(Janela: Integer; Str:**string**);
 procedure ClearEvent(Janela:Integer; Texto: **string**);
 procedure UpdateField;
end;

var

Form1: TForm1;
RdWrLock: TMultiReadExclusiveWriteSynchronizer;
ListBoxes: **array**[1..3] of TListBox;
// Memória protegida. Deve estar na mesma thread que cria o objeto
SharedMem: Integer = 0;

implementation

{ \$R *.dfm }

procedure TForm1.FormCreate(Sender: TObject);

begin

 ListBoxes[1]:= lstLeitores;
 ListBoxes[2]:= lstEscritores;
 ListBoxes[3]:= lstDatabase;

```

EditVar.AutoSize:= False;
// Cria Objeto de sincronização
RdWrLock := TMultiReadExclusiveWriteSynchronizer.Create;

EditVar.Text:= IntToStr(SharedMem);
EditVar.ReadOnly:= True; // Não permite ao usuário modificar o texto
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
    Index: Integer;
begin
    for Index:= 1 to MaxReaders do
        ReaderThreads[Index].WaitFor;
    for Index:= 1 to MaxWriters do
        WriterThreads[Index].WaitFor;
    RdWrLock.Free;
end;

procedure TForm1.PrintEvent(Janela: Integer; Str:string);
begin
    if (Janela in [1..3])
    then with ListBoxes[Janela] do
        begin
            Canvas.Lock;
            Items.Add(Str);
            Canvas.Unlock;
        end
    else ShowMessage('PrintEvent: Index '+ IntToStr(Janela) + ' desconhecido');
end; // PrintEvent

procedure TForm1.ClearEvent(Janela:Integer; Texto: string);
var
    i: Integer;
    Index: Integer;
begin
    Index := -1; // Flag não encontrado
    if (Janela in [1..3])
    then with ListBoxes[Janela] do
        begin
            Canvas.Lock;
            for i:=0 to (Items.count - 1) do
                if Items[i] = Texto then
                    begin
                        Index:= i;
                        break;
                    end; // if
                if (index > -1) then Items.Delete(Index);
            Canvas.Unlock;
        end
    end

```

```
    else ShowMessage('PrintEvent: Index '+ IntToStr(Janela) + ' desconhecido');  
end; // ClearEvent
```

```
procedure TForm1.btnIniciarClick(Sender: TObject);  
var  
    Index: Integer;  
begin  
    // Cria Leitores  
    for Index:= 1 to MaxReaders do  
        ReaderThreads[Index]:= TReaderThread.Create(Index);  
    // Cria Escritores  
    for Index:= 1 to MaxWriters do  
        WriterThreads[Index]:= TWriterThread.Create(Index);  
    // Ativa leitores e escritores  
    for Index:= 1 to MaxReaders do  
        ReaderThreads[Index].Resume;  
    for Index:= 1 to MaxWriters do  
        WriterThreads[Index].Resume;  
    btnIniciar.Enabled:= False;  
end;
```

```
procedure TForm1.UpdateField();  
begin  
    EditVar.Text:= IntToStr(SharedMem);  
end;
```

```
end.
```

Thread Escritor

```
unit Writer;  
  
interface  
  
uses  
    Classes, SysUtils, Dialogs, Windows;  
  
type  
    TWriterThread = class(TThread)  
    private  
        { Private declarations }  
    protected  
        Index: Integer;  
        procedure Execute; override;  
    public  
        constructor Create(Value: Integer);  
    end;  
  
implementation
```

```

uses E41MRERSync;

// Implementa passagem de parâmetro para inicializar a thread
constructor TWriterThread.Create(Value: Integer);
begin
    Index := Value;
    inherited Create(True); // Cria em estado suspenso
end; // TWriter.create

procedure TWriterThread.Execute;
var
    Msg: string;
begin
    try
    while not Terminated do
    begin
        Msg := 'Escritor ' + IntToStr(Index);
        Sleep(10*(Random(200))); // Espera tempo aleatório para entrar em cena
        Form1.PrintEvent(LISTBOX_WRITER, Msg);
        Sleep(300); // Mostra que entrou na fila

        RdWrLock.BeginWrite;
        Form1.ClearEvent(LISTBOX_WRITER, Msg);
        SharedMem := Index; // Modifica Memória compartilhada
        Form1.UpdateField; // Pedre para atualizar tela
        Form1.PrintEvent(LISTBOX_DATABASE, Msg);
        Sleep(1000); // Escreve durante 1 segundo
        RdWrLock.EndWrite;

        Form1.ClearEvent(LISTBOX_DATABASE, Msg);
        ExitThread(0);
    end
    except
        ShowMessage('Erro em Thread Escritor');
        // raise
    end
end;

end.

```

Thread Leitor

```

unit Reader;

interface
uses
    Classes, SysUtils, Dialogs, Windows;

type
    TReaderThread = class(TThread)

```

```

private
  { Private declarations }
protected
  Index: Integer;
  procedure Execute; override;
public
  constructor Create(Value: Integer);
end;

implementation

uses E41MRERSync;

// Implementa passagem de parâmetro para inicializar a thread
constructor TReaderThread.Create(Value: Integer);
begin
  Index := Value;
  inherited Create(True); // Cria em estado suspenso
end; // Treader.create

procedure TReaderThread.Execute;
var
  Msg: string;
  Value: Integer;
begin
  try
  while not Terminated do
  begin
    Msg := 'Leitor ' + IntToStr(Index);
    Sleep(10*(Random(200))); // Espera tempo aleatório para entrar em cena
    Form1.PrintEvent(LISTBOX_READER, Msg);
    Sleep(300); // Mostra que entrou na fila

    RdWrLock.BeginRead;
    Form1.ClearEvent(LISTBOX_READER, Msg);
    Value := SharedMem; // Lê Memória compartilhada
    Form1.PrintEvent(LISTBOX_DATABASE, Msg);
    Sleep(1000); // Lê durante 1 segundo
    RdWrLock.EndRead;

    Form1.ClearEvent(LISTBOX_DATABASE, Msg);
    ExitThread(0);
  end
  except
    ShowMessage('Erro em Thread Leitor');
    // raise
  end
end;
end.

```


Exercícios

- 1) Modifique o exemplo 3 para usar `CriticalSections` ao invés da classe `TCriticalSections`. Confira o resultado com o programa `E31CSDemo2`.
- 2) Escreva uma versão para o problema do jantar dos filósofos em Delphi.
- 3) Escreva uma versão do programa que demonstra o uso de `Timers` Multimídia em Delphi (programa 45 do livro texto).
- 4) Construa um programa para demonstrar o uso de buffers circulares, usando o modelo dos produtores e consumidores em Delphi.
- 5) Resolva o problema do banheiro Unisex usando Delphi.
- 6) Construa uma classe `TPipe` para encapsular todas as funções úteis de pipes.
- 7) Construa um componente que substitua a classe `TPipe`. Desenvolva um programa de teste para este componente.

BIBLIOGRAFIA

- [Cantù 99] Marco Cantù, Mastering Delphi 5, Sybex, 1999.
- [Inprise 99] Delphi Developer's Guide: Chapter 37: Handling Messages, 1999,
www.borland.com/techpubs/delphi/delphi5/dg/messages.html.
- [De Cleen 2000] Wim De Cleen, Delphi threading by example, An example with the Windows API, July 8th 2000, Article ID: 22411.
- [Kastner 99] Christian Kastner, Mailslots, April 99,
www.kaestnerpro.de/mailslot.htm
- [Leigh 2000] Andrew Leigh, ALMailSlot,
<http://www.digicraft.com.au/delphi/>

Sites a serem visitados

- Torry's Delphi Pages homepages.borland.com/torry/tools_documentation.htm
- Página oficial do livro Mastering Delphi www.marcocantu.com/
- Better Threads For The VCL www.midnightbeach.com/jon/pubs/MsgWaits/MsgWaits.html
- Componentes em Delphi delphi.about.com/library/weekly/aa122899a.htm
- Programação OPC em Delphi www.opcconnect.com/delphi.shtml
- CriticalSections e Mutexes www.pergolesi.demon.co.uk/prog/threads/Ch6.html
- Handling Messages info.borland.com/techpubs/delphi/delphi5/dg/messages.html
- Comunicação serial com Delphi www.torry.net/modems.htm