Multithreaded Application Tutorial

Overview

This page will try to explain how to write and debug a multi-threaded application with Free Pascal and Lazarus. A multi-threaded application is one that creates two or more threads of execution that work at the same time. If you are new to multi-threading, please read the paragraph "Do you need multi-threading?" to determine whether it is really required; this may save you many headaches.

One of the threads is called the Main Thread. The Main Thread is the one that is created by the Operating System once our application starts. The Main Thread **must be** the only thread that updates the components that interfaces with the user: otherwise, the application may hang.

The main idea is that the application can do some processing in background in a second thread while the user can continue working using the main thread.

Another use of threads is just to have a better responding application. If you create an application, and when the user presses a button the application starts processing a big job... and while processing, the screen stops responding, and gives the user the impression that the application is frozen, a poor or misleading impression will be created. If the big job runs in a second thread, the application keeps responding (almost) as if it were idle. In this case it is a good idea, before starting the thread, to disable the buttons of the form to avoid the user starting more than one thread for the job.

Another use of multi-threading may be a server application that is able to respond to many clients at the same time.

Do you need multi-threading?

If you are new to multi-threading and you only want to make your application more responsive while your application performs moderately long-running tasks, then multi-threading may be more than is required. Multi-threaded applications are always more difficult to debug and they are often much more complex; in many cases you don't need multi-threading. A single thread is enough. If you can split up the time-consuming task into several smaller chunks, then instead you should use **Application.ProcessMessages**. This method allows the LCL to handle all waiting messages and returns. The central idea is to call Application.ProcessMessages at regular intervals during the execution of a long-running task to determine whether the user has clicked on something, or a progress indicator must be repainted, and so on.

For example: Reading a big file and process it. See examples/multithreading/singlethreadingexample1.lpi.

Multi-threading is only needed for

- blocking handles, like network communications
- using multiple processors simultaneously (SMP)
- algorithms and library calls that must be called through an API and as such cannot be split up into smaller parts.

If you want to use multi-threading to increase speed by using multiple processors simultaneously, check if your current program now uses all 100% resources of 1 core CPU (for example, your program can actively use input-output operations, e.g. writing to file; this takes a lot of time, but doesn't load CPU; in this case your program will not be faster with multiple threads). Also check if optimisation level is set to maximum (3). When switching optimisation level from 1 to 3, a program may become about 5 times faster.

Units needed for a multi-threaded application

You don´t need any special unit for this to work with Windows. However with Linux, macOS and FreeBSD, you need the cthreads unit and it *must* be the first used unit of the project (the program source, usually the .lpr file)! In cases where several units like cthreads, cmem and cwstrings are recommended to be placed first, due to how the units work a sensible order is cmem, cthreads and then cwstrings.

So, your Lazarus application code should look like:

```
program MyMultiThreadedProgram;
{$mode objfpc}{$H+}
uses
{$ifdef unix}
  cthreads,
  cmem, // the c memory manager is on some systems much faster for multi-
threading
{$endif}
  Interfaces, // this includes the LCL widgetset
  Forms
  { you can add units here },
```

If you forget this and you use TThread you will get this error on startup:

```
 This binary has no thread support compiled in.
 Recompile the application with a thread-driver in the program uses clause
before other units using thread.
```

Note: If you get a Linker error about "mcount" not found. Then you use some unit that contains some multithreaded code and you need to add the cthreads unit or use smart linking.

Note: If you get the error: "Project raised exception class 'RunError(232)'" in procedure SYSTEM_NOTHREADERROR then your code requires threading and you need to add the cthreads unit.

Pure FPC example

Below code gives a very simple example. Tested with FPC 3.0.4 on Win7.

```
Program ThreadTest;
{test multi threading capability }
{
      OUTPUT
thread 1 started
thread 1 thri 0 Len(S)= 1
thread 1 thri 1 Len(S)= 2
thread 1 thri 2 Len(S)= 3
thread 1 thri 3 Len(S)= 4
thread 1 thri 4 Len(S)= 5
thread 1 thri 5 Len(S)= 6
thread 1 thri 6 Len(S)= 7
thread 1 thri 7 Len(S)= 8
thread 1 thri 8 Len(S)= 9
thread 1 thri 9 Len(S)= 10
thread 1 thri 10 Len(S)= 11
thread 1 thri 11 Len(S)= 12
thread 1 thri 12 Len(S)= 13
thread 1 thri 13 Len(S)= 14
thread 1 thri 14 Len(S)= 15
thread 2 started
thread 3 started
thread 1 thri 15 Len(S)= 16
thread 2 thri 0 Len(S)= 1
thread 3 thri 0 Len(S)= 1
thread 1 thri 16 Len(S)= 17
...
...
thread 5 thri 997 Len(S)= 998
thread 5 thri 998 Len(S)= 999
thread 5 thri 999 Len(S)= 1000
thread 5 finished
thread 10 thri 828 Len(S)= 829
```

```
thread 9 thri 675 Len(S)= 676
thread 4 thri 656 Len(S)= 657
thread 10 thri 829 Len(S)= 830
thread 9 thri 676 Len(S)= 677
thread 9 thri 677 Len(S)= 678
thread 10 thri 830 Len(S)= 831
thread 10 thri 831 Len(S)= 832
thread 10 thri 832 Len(S)= 833
thread 10 thri 833 Len(S)= 834
thread 10 thri 834 Len(S)= 835
thread 10 thri 835 Len(S)= 836
thread 10 thri 836 Len(S)= 837
thread 10 thri 837 Len(S)= 838
thread 10 thri 838 Len(S)= 839
thread 10 thri 839 Len(S)= 840
thread 9 thri 678 Len(S)= 679
...
...
thread 4 thri 994 Len(S)= 995
thread 4 thri 995 Len(S)= 996
thread 4 thri 996 Len(S)= 997
thread 4 thri 997 Len(S)= 998
thread 4 thri 998 Len(S)= 999
thread 4 thri 999 Len(S)= 1000
thread 4 finished
10

}

uses
  {$ifdef unix}cthreads, {$endif} sysutils;

const
  threadcount = 10;
  stringlen = 1000;

var
   finished : longint;

threadvar
   thri : ptrint;

function f(p : pointer) : ptrint;
var
```

```
  s : ansistring;
begin
  Writeln('thread ',longint(p),' started');
  thri:=0;
  while (thri<stringlen) do begin
    s:=s+'1'; { create a delay }
    writeln('thread ',longint(p),' thri ',thri,' Len(S)= ',length(s));
        inc(thri);
  end;
  Writeln('thread ',longint(p),' finished');
  InterLockedIncrement(finished);
  f:=0;
end;


var
   i : longint;

Begin
   finished:=0;
   for i:=1 to threadcount do
     BeginThread(@f,pointer(i));
   while finished<threadcount do ;
   Writeln(finished);
End.
```

The TThread Class

The following example can be found in the examples/multithreading/ directory.

To create a multi-threaded application, the easiest way is to use the TThread Class. This class permits the creation of an additional thread (alongside the main thread) in a simple way. Normally you are required to override only 2 methods: the Create constructor, and the Execute method.

In the constructor, you will prepare the thread to run. You will set the initial values of the variables or properties you need. The original constructor of TThread requires a parameter called Suspended. As you might expect, setting Suspended = True will prevent the thread starting automatically after the creation. If Suspended = False, the thread will start running just after the creation. If the thread is created suspended, then it will run only after the Start method is called.

Note: Method Resume is deprecated since FPC 2.4.4. It is replaced by Start.

As of FPC version 2.0.1 and later, TThread.Create also has an implicit parameter for Stack Size. You can now change the default stack size of each thread you create if you need it. Deep procedure call

recursions in a thread are a good example. If you don't specify the stack size parameter, a default OS stack size is used.

In the overridden Execute method you will write the code that will run on the thread.

The TThread class has one important property: Terminated : boolean;

If the thread has a loop (and this is typical), the loop should be exited when Terminated is true (it is false by default). Within each pass, the value of Terminated must be checked, and if it is true then the loop should be exited as quickly as is appropriate, after any necessary cleanup. ~~Bear in mind that the Terminate method does not do anything by default: the .Execute method must explicitly implement support for it to quit its job.~~ The Terminate method only sets the Terminated property to True.

As we explained earlier, the thread should not interact with the visible components. Updates to visible components must be made within the context of the main thread.

To do this, a TThread method called Synchronize exists. Synchronize requires a method within the thread (that takes no parameters) as an argument. When you call that method through Synchronize(@MyMethod), the thread execution will be paused, the code of MyMethod will be called from the main thread, and then the thread execution will be resumed.

The exact working of Synchronize depends on the platform, but basically it does this:

- it posts a message onto the main message queue and goes to sleep
- eventually the main thread processes the message and calls MyMethod. This way MyMethod is called without context, that means not during a mouse down event or during paint event, but after.
- after the main thread executed MyMethod, it wakes the sleeping Thread and processes the next message
- the Thread then continues.

There is another important property of TThread: FreeOnTerminate. If this property is true, the thread object is automatically freed when the thread execution (.Execute method) stops. Otherwise the application will need to free it manually.

Example:

```
Type
  TMyThread = class(TThread)
  private
    fStatusText : string;
    procedure ShowStatus;
  protected
    procedure Execute; override;
  public
    Constructor Create(CreateSuspended : boolean);
  end;
constructor TMyThread.Create(CreateSuspended : boolean);
begin
```

```
    inherited Create(CreateSuspended);
    FreeOnTerminate := True;
  end;

  procedure TMyThread.ShowStatus;
  // this method is executed by the mainthread and can therefore access all
GUI elements.
  begin
    Form1.Caption := fStatusText;
  end;

  procedure TMyThread.Execute;
  var
    newStatus : string;
  begin
    fStatusText := 'TMyThread Starting...';
    Synchronize(@Showstatus);
    fStatusText := 'TMyThread Running...';
    while (not Terminated) and ([any condition required]) do
      begin
        ...
        [here goes the code of the main thread loop]
        ...
        if NewStatus <> fStatusText then
          begin
            fStatusText := newStatus;
            Synchronize(@Showstatus);
          end;
      end;
  end;
```

In the application:

```
  var
    MyThread : TMyThread;
  begin
    MyThread := TMyThread.Create(True); // This way it doesn't start
automatically
    ...
    [Here the code initialises anything required before the threads starts
executing]
    ...
    MyThread.Start;
  end;
```

If you want to make your application more flexible you can create an event for the thread; this way your synchronized method won't be tightly coupled with a specific form or class: you can attach listeners to the thread's event. Here is an example:

```
  Type
    TShowStatusEvent = procedure(Status: String) of Object;

    TMyThread = class(TThread)
    private
      fStatusText : string;
      FOnShowStatus: TShowStatusEvent;
      procedure ShowStatus;
    protected
      procedure Execute; override;
    public
      Constructor Create(CreateSuspended : boolean);
      property OnShowStatus: TShowStatusEvent read FOnShowStatus write
FOnShowStatus;
    end;

  constructor TMyThread.Create(CreateSuspended : boolean);
  begin
    inherited Create(CreateSuspended);
    FreeOnTerminate := True;
  end;

  procedure TMyThread.ShowStatus;
  // this method is executed by the mainthread and can therefore access all
GUI elements.
  begin
    if Assigned(FOnShowStatus) then
    begin
      FOnShowStatus(fStatusText);
    end;
  end;

  procedure TMyThread.Execute;
  var
    newStatus : string;
  begin
    fStatusText := 'TMyThread Starting...';
    Synchronize(@Showstatus);
    fStatusText := 'TMyThread Running...';
    while (not Terminated) and ([any condition required]) do
      begin
```

```
      ...
      [here goes the code of the main thread loop]
      ...
      if NewStatus <> fStatusText then
        begin
          fStatusText := newStatus;
          Synchronize(@Showstatus);
        end;
    end;
end;
```

In the application:

```
Type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { private declarations }
    MyThread: TMyThread;
    procedure ShowStatus(Status: string);
  public
    { public declarations }
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  inherited;
  MyThread := TMyThread.Create(true);
  MyThread.OnShowStatus := @ShowStatus;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  MyThread.Terminate;

  // FreeOnTerminate is true so we should not write:
  // MyThread.Free;
  inherited;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
   MyThread.Start;
  end;

  procedure TForm1.ShowStatus(Status: string);
  begin
    Label1.Caption := Status;
  end;
```

Special things to take care of

Stack checking under Windows

There is a potential headache in Windows with Threads if you use the -Ct (stack check) switch. For reasons not so clear the stack check will "trigger" on any TThread.Create if you use the default stack size. The only work-around for the moment is to simply not use -Ct switch. Note that it does NOT cause an exception in the main thread, but in the newly created one. This "looks" like if the thread was never started.

A good code to check for this and other exceptions which can occur in thread creation is:

```
MyThread := TThread.Create(False);

if Assigned(MyThread.FatalException) then
  raise MyThread.FatalException;
```

This code will assure that any exception which occurred during thread creation will be raised in your main thread.

GetDC function in Windows

The GetDC (and GetDCEx) function retrieves a handle to a device context (DC) for the client area of a specified window or for the entire screen. It can be used, for example, to take a screenshot. Note however that it is not thread safe: the handle to the DC can only be used by a single thread at any one time as documented here.

Multithreading in packages

Packages which uses multi-threading should add the **-dUseCThreads** flag to the custom usage options. Open the package editor of the package, then Options > Usage > Custom and add *-dUseCThreads*. This will define this flag to all projects and packages using this package, including the IDE. The IDE and all new applications created by the IDE have already the following code in their .lpr file:

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  cmem, // the c memory manager is on some systems much faster for multi-
```

```
threading
  {$ENDIF}{$ENDIF}
```

Heaptrc

You can not use the -gh switch with the *cmem* unit. The -gh switch uses the heaptrc unit, which extends the heap manager. Therefore the **heaptrc** unit must be used **after** the **cmem** unit.

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  cmem, // the c memory manager is on some systems much faster for multi-
threading
  {$ENDIF}{$ENDIF}
  heaptrc,
```

Initialization and Finalization

To initialize the thread object itself, you can either start it suspended and set its properties and/or create a new constructor and call the inherited constructor.

Note: Using the AfterConstruction when CreateSuspended=false is dangerous, as the thread has already started.

On the other hand, the destructor may be used to finalize the object's ressources.

```
type
  TMyThread = class(TThread)
  private
    fRTLEvent: PRTLEvent;
  public
    procedure Create(SomeData: TSomeObject); override;
    destructor Destroy; override;
  end;

procedure TMyThread.Create(SomeData: TSomeObject; CreateSuspended: boolean);
begin
  // example: set up events, critical sections and other ressources like
files or database connections
  RTLEventCreate(fRTLEvent);
  inherited Create(CreateSuspended);
end;

destructor TMyThread.Destroy;
begin
  RTLeventDestroy(fRTLEvent);
```

```
  inherited Destroy;
end;
```

Non LCL program

TThread.Synchronize requires that the main thread regularly calls CheckSynchronize. The LCL does that in its loop. If you don't use the LCL event loop you must call it yourself.

SMP Support

The good news is that if your application works properly multi-threaded this way, it is already SMP enabled!

Debugging Multi-threaded Applications with Lazarus

The debugging on Lazarus requires GDB and is rapidly becoming more and more fully featured and stable. However, there still exists a few Linux distributions with some problems.

Debugging output

In a single threaded application, you can simply write to console/terminal/whatever and the order of the lines is the same as they were written. In multi-threaded application things are more complicated. If two threads are writing, say a line is written by thread A before a line by thread B, then the lines are not necessarily written in that order. It can even happen, that a thread writes its output, while the other thread is writing a line.While under linux (maybe) you'll get proper DebugLn() output, under win32 you can get exceptions (probably DiskFull) because of DebugLn() usage outside of main thread.So, to avoid headaches use DebugLnThreadLog() mentioned below.

The LCLProc unit contains several functions, to let each thread write to its own log file:

```
  procedure DbgOutThreadLog(const Msg: string); overload;
  procedure DebuglnThreadLog(const Msg: string); overload;
  procedure DebuglnThreadLog(Args: array of const); overload;
  procedure DebuglnThreadLog; overload;
```

For example: Instead of *writeln('Some text ',123);* use

```
 DebuglnThreadLog(['Some text ',123]);
```

This will append a line 'Some text 123' to **Log<PID>.txt**, where <PID> is the process ID of the current thread.

It is a good idea to remove the log files before each run:

```
 rm -f Log* && ./project1
```

Linux

If you try to debug a multi-threaded application on Linux, you will have one big problem: the Desktop Manager on X server can hang. This happens for instance when the application has captured the mouse/keyboard and was paused by gdb and the X server waits for your application. When that happens you can simply log in from another computer and kill the gdb or exit out of that session by pressing CTRL+ALT+F3 and kill gdb. Alternatively you can restart the window manager: enter sudo /etc/init.d/gdm restart. This will restart the desktop manager and get you back into your desktop.

Since it depends where gdb stops your program in some cases some tricks may help: for Ubuntu x64 set the Project options for debugging required extra information file...

```
 Project Options -> Compiler Options -> Linking -> Debugging: Check Use external gdb debug symbols file (-Xg).
```

The other option is to open another X desktop, run the IDE/gdb on one and the application on the other, so that only the test desktop freezes. Create a new instance of X with:

```
 X :1 &
```

It will open, and when you switch to another desktop (the one you are working with pressing CTRL+ALT+F7), you will be able to go back to the new graphical desktop with CTRL+ALT+F8 (if this combination does not work, try with CTRL+ALT+F2... this one worked on Slackware).

Then you could, if you want, create a desktop session on the X started with:

```
 gnome-session --display=:1 &
```

Then, in Lazarus, on the run parameters dialog for the project, check "Use display" and enter :1.

Now the application will run on the second X server and you will be able to debug it on the first one.

This was tested with Free Pascal 2.0 and Lazarus 0.9.10 on Windows and Linux.

Instead of creating a new X session, one can use Xnest. Xnest is a X session on a window. Using it X server didn't lock while debugging threads, and it's much easier to debug without keeping changing terminals.

The command line to run Xnest is

```
 Xnest :1 -ac
```

to create a X session on :1, and disabling access control.

Lazarus Widgetset Interfaces

The win32, the gtk and the carbon interfaces support multi-threading. This means, TThread, critical sections and Synchronize work. But they are not thread safe. This means only one thread at a time can access the LCL. And since the main thread should never wait for another thread, it means only the main

thread is allowed to access the LCL, which means anything that has to do with TControl, Application and LCL widget handles. There are some thread safe functions in the LCL. For example most of the functions in the FileUtil unit are thread safe.

Using SendMessage/PostMessage to communicate between threads

Only one thread in an application should call LCL APIs, usually the main thread. Other threads can make use of the LCL through a number of indirect methods, one good option being the usage of SendMessage or PostMessage. LCLIntf.SendMessage and LCLIntf.PostMessage will post a message directed to a window in the message pool of the application.

See also the documentation for these routines:

- SendMessage
- PostMessage

The difference between SendMessage and PostMessage is the way that they return control to the calling thread. Like Synchronize, SendMessage blocks and control is not returned until the window that the message was sent to has completed processing it; however under certain circumstances SendMessage might attempt to optimise processing by remaining in the context of the thread that called it. With PostMessage control is returned immediately up to some system-defined maximum number of enqueued messages and as long as space remains on the heap for attached data.

In both cases the procedure handling the message (see below) should avoid calling application.ProcessMessages, since this might cause a second message to be dispatched which will be handled reentrantly. If this is unavoidable then it would probably be preferable to use some other mechanism to transfer serialised events between threads.

Here is an example of how a secondary thread could send text to be displayed in an LCL control to the main thread:

```
const
  WM_GOT_ERROR           = LM_USER + 2004;
  WM_VERBOSE             = LM_USER + 2005;

procedure VerboseLog(Msg: string);
var
  PError: PChar;
begin
  if MessageHandler = 0 then Exit;
  PError := StrAlloc(Length(Msg)+1);
  StrCopy(PError, PChar(Msg));
  PostMessage(formConsole.Handle, WM_VERBOSE, Integer(PError), 0);
end;
```

And an example of how to handle this message from a window:

```
const
  WM_GOT_ERROR           = LM_USER + 2004;
  WM_VERBOSE             = LM_USER + 2005;

type
  { TformConsole }

  TformConsole = class(TForm)
    DebugList: TListView;
    // ...
  private
    procedure HandleDebug(var Msg: TLMessage); message WM_VERBOSE;
  end;

var
  formConsole: TformConsole;

implementation

....

{ TformConsole }

procedure TformConsole.HandleDebug(var Msg: TLMessage);
var
  Item: TListItem;
  MsgStr: PChar;
  MsgPasStr: string;
begin
  MsgStr := PChar(Msg.wparam);
  MsgPasStr := StrPas(MsgStr);
  Item := DebugList.Items.Add;
  Item.Caption := TimeToStr(SysUtils.Now);
  Item.SubItems.Add(MsgPasStr);
  Item.MakeVisible(False);

// Followed by something like

  TrayControl.SetError(MsgPasStr);
  StrDispose(MsgStr)
end;

end.
```

When you get on the Linux x64 machine an error: "Project XY raised exception class 'External: SIGSEGV'" you have to change "Integer(PError)" to "PtrInt(PError)" in procedure "VerboseLog". Modified line have to look like:

```
 PostMessage(formConsole.Handle, WM_VERBOSE, PtrInt(PError), 0);
```

Critical sections

A *critical section* is an object used to make sure, that some part of the code is executed only by one thread at a time. A critical section needs to be created/initialized before it can be used and be freed when it is not needed anymore.

There are two ways you use critical sections, from the RTL or from LCL. The text below shows the RTL model and shows the LCL alternatives as a comment, the only difference is the method names.

Critical sections are normally used this way:

```
// uses LCLIntf, LCLType;
```

the necessary RTL units will almost certainly already be mentioned.

Declare the section (globally for all threads which should access the section):

```
MyCriticalSection: TRTLCriticalSection;
// MyCriticalSection : TCriticalSection;
```

Create the section:

```
InitCriticalSection(MyCriticalSection);
//InitializeCriticalSection(MyCriticalSection);
```

Run some threads. Doing something exclusively:

```
EnterCriticalSection(MyCriticalSection);
try
  // access some variables, write files, send some network packets, etc
finally
  LeaveCriticalSection(MyCriticalSection);
end;
// RTL and LCL use same syntax, thats nice.
```

After all threads terminated, free it:

```
DoneCriticalSection(MyCriticalSection);
// DeleteCriticalSection(MyCriticalSection);
```

As an alternative, you can use a TCriticalSection object. The creation does the initialization, the Enter method does the EnterCriticalSection, the Leave method does the LeaveCriticalSection and the

destruction of the object does the deletion.

Note that a critical section does not protect against the same thread entering the same block of code, only against different threads. For that reason it cannot be used to protect against e.g. reentry of a message handler (see the section above).

For example: 5 threads incrementing a counter. See lazarus/examples/multithreading/criticalsectionexample1.lpi

Warning: There are two sets of the above four functions. The RTL and the LCL ones. The LCL ones are defined in the unit LCLIntf and LCLType. Both work pretty much the same. You can use both at the same time in your application, but you should not use a RTL function within an LCL Critical Section and vice versa.

Sharing Variables

If some threads share a variable, that is read only, then there is nothing to worry about. Just read it. But if one or several threads changes the variable, then you must make sure, that only one thread accesses the variables at a time.

For example: 5 threads incrementing a counter. See lazarus/examples/multithreading/criticalsectionexample1.lpi

Waiting for another thread

If a thread A needs a result of another thread B, it must wait, till B has finished.

Important: The main thread should never wait for another thread. Instead use Synchronize (see above).

See for an example: lazarus/examples/multithreading/waitforexample1.lpi

```
{ TThreadA }

procedure TThreadA.Execute;
begin
  Form1.ThreadB:=TThreadB.Create(false);
  // create event
  WaitForB:=RTLEventCreate;
  while not Application.Terminated do begin
    // wait infinitely (until B wakes A)
    RtlEventWaitFor(WaitForB);
    writeln('A: ThreadB.Counter='+IntToStr(Form1.ThreadB.Counter));
  end;
end;
```

```
{ TThreadB }

procedure TThreadB.Execute;
var
  i: Integer;
begin
  Counter:=0;
  while not Application.Terminated do begin
    // B: Working ...
    Sleep(1500);
    inc(Counter);
    // wake A
    RtlEventSetEvent(Form1.ThreadA.WaitForB);
  end;
end;
```

Note: RtlEventSetEvent can be called before RtlEventWaitFor. Then RtlEventWaitFor will return immediately. Use RTLeventResetEvent to clear a flag.

Fork

When forking in a multi-threaded application, be aware that any threads created and running BEFORE the fork (or fpFork) call, will NOT be running in the child process. As stated on the fork() man page, any threads that were running before the fork call, their state will be undefined.

So be aware of any threads initializing before the call (including on the initialization section). They will NOT work.

Parallel procedures/loops

A special case of multi threading is running a single procedure in parallel. See Parallel procedures.

Distributed computing

The next higher steps after multi threading is running the threads on multiple machines.

- You can use one of the TCP suites like synapse, Inet or indy for communications. This gives you maximum flexibility and is mostly used for loosely connected Client / Server applications.
- You can use message passing libraries like MPICH, which are used for HPC (High Performance Computing) on clusters.

External threads

To make Free Pascal's threading system work properly, each newly created FPC thread needs to be initialized (more exactly, the exception, I/O system and threadvar system per thread needs to be initialized so threadvars and heap are working). That is fully automatically done for you if you use BeginThread (or indirectly by using the TThread class). However, if you use threads that were created without BeginThread (i.e. external threads), additional work (currently) might be required. External threads also include those that were created in external C libraries (.DLL/.so).

Things to consider when using external threads (might not be needed in all or future compiler versions):

- Do not use external threads at all - use FPC threads. If can you can get control over how the thread is created, create the thread by yourself by using BeginThread.

If the calling convention doesn't fit (e.g. if your original thread function needs cdecl calling convention but BeginThread needs pascal convention, create a record, store the original required thread function in it, and call that function in your pascal thread function:

```
type
 TCdeclThreadFunc = function (user_data:Pointer):Pointer;cdecl;

 PCdeclThreadFuncData = ^TCdeclThreadFuncData;
 TCdeclThreadFuncData = record
   Func: TCdeclThreadFunc;  //cdecl function
   Data: Pointer;           //original data
 end;

// The Pascal thread calls the cdecl function
function C2P_Translator(FuncData: pointer) : ptrint;
var
  ThreadData: TCdeclThreadFuncData;
begin
  ThreadData := PCdeclThreadFuncData(FuncData)^;
  Result := ptrint(ThreadData.Func(ThreadData.Data));
end;

procedure CreatePascalThread;
var
  ThreadData: PCdeclThreadFuncData;
begin
  New(ThreadData);
  // this is the desired cdecl thread function
  ThreadData^.Func := func;
  ThreadData^.Data := user_data;
  // this creates the Pascal thread
```

```
  BeginThread(@C2P_Translator, ThreadData );
end;
```

- Initialize the FPC's threading system by creating a dummy thread. If you don't create any Pascal thread in your app, the thread system won't be initialized (and thus threadvars won't work and thus heap will not work correctly).

```
type
   tc = class(tthread)
     procedure execute;override;
   end;

   procedure tc.execute;
   begin
   end;

{ main program }
begin
  { initialise threading system }
   with tc.create(false) do
   begin
     waitfor;
     free;
   end;
   { ... your code follows }
end.
```

(After the threading system is initialized, the runtime may set the system variable "IsMultiThread" to true which is used by FPC routines to perform locks here and there. You should not set this variable manually.)

- If for some reason this doesn't work for you, try this code in your external thread function:

```
function ExternalThread(param: Pointer): LongInt; stdcall;
var
  tm: TThreadManager;
begin
  GetThreadManager(tm);
  tm.AllocateThreadVars;
  InitThread(1000000); // adjust inital stack size here

  { do something threaded here ... }
```

```
  Result:=0;
end;
```

Identifying external threads

Sometimes you even don't know if you have to deal with external threads (eg if some C library makes a callback). This can help to analyse this:

1. Ask the OS for the ID of the current thread at your application's start

```
GetCurrentThreadID() // Windows;
GetThreadID() // Darwin (macOS); FreeBSD;
TThreadID(pthread_self) // Linux;
```

2. Ask again for the ID of the current thread inside the thread function and compare this with the result of step 1.

Give up some time slice

ThreadSwitch - Note: may be ignored.

Note: Do not use the Windows trick *Sleep(0)* as this won't work on all platforms.

See also