Introduction to thread programming in Lazarus

Michaël Van Canneyt

March 3, 2013

Abstract

Thread support in Free Pascal has received an update, so it is compatible to thread support in Delphi XE3. Time to re-visit the subject of threads, and look at the possibilities for those not familiar with thread programming.

1 Introduction

Recently, FPC received a substantial upgrade in its threading code. It has been made compatible with the Delphi XE3 threading possibilities. Time for a refreshment on how to do threading in Lazarus/Free Pascal.

Threading means writing the code of an application so that several parts of the code are executed simultaneously, in parallel. Each part of the code forms a thread of execution. Threading requires support by the hardware: several processors or a processor with various cores. If the hardware does not allow it, the operating system mimics simultaneous execution by giving the various parts of the program time to execute in an intertwined fashion, much like it allows multiple programs to execute 'simultaneously'. The scheduling of threads and programs by the operating system is a huge subject in itself, and is done differently by each OS.

The difference between threads and multiple programs is of course that all threads in a program all have access to the same data and memory space. Programs can share some memory by using shared memory and special operating system calls.

Access to the shared memory by the various threads must be carefully designed, or strange things may happen: As long as one thread only writes and another thread only reads from a shared memory location, things will usually progress without problems. However, if 2 threads will try to update the same memory location, strange things may happen. To prevent this, special programming is required. In this article, we'll show how this is done.

2 Thread support classes in Free Pascal

The threading support of Free Pascal makes it relatively simple to create multi-threaded programs. FPC provides several classes and a language construct to deal with multiple threads in a program:

TThread This is the heart of the threading system: This class encapsulates a thread in a program. It is declared in the classes unit.

threadvar This is a special kind of variable: global variables are normally accessible by all threads in a program. When they are declared as `threadvar` instead of `var`, each thread in the program will receive its own copy of the global variable.

TMultiReadExclusiveWriteSynchronizer This class serves to protect access to a piece of data such that it can be read simultaneously by various threads, but can be written only by one thread. It is declared in the sysutils unit.

TThreadList This is a special list which can be locked for exclusive access by a thread. It is declared in the classes unit.

TCriticalSection This class can be used to make sure a particular piece of code is executed only by one thread at a time. It is declared in the syncobjs unit.

3 The TThread class

From the classes presented above, the most important class is `TThread`. It has at least the following public methods:

Start this will start execution of the thread if it was created in a suspended state.

Terminate Tells the thread to stop executing at the earliest possible time.

WaitFor This waits for the thread to end, and returns the exit code of the thread.

And some properties:

FreeOnTerminate This property is described in detail below.

Handle An operating system handle for the thread.

ThreadID An operating system ID for the thread.

ExternalThread A boolean indicating whether this thread was started outside the RTL.

Priority An indicator of the thread's priority.

Suspended A Boolean indicating whether the thread is currently executing or not.

Finished A Boolean indicating whether the thread has finished executing.

OnTerminate An event handler that is calledwhen the thread has finished executing.

FatalException If an exception is raised during the execution of the thread, but is not caught by user code, then it is caught and held here for inspection.

There are also some protected methods, only available inside the `TThread` class and its descencents:

Execute A virtual abstract method that must be overridden.

Synchronize Allows to schedule a procedure for execution in the main thread. The thread waits till it has finished executing.

Queue Allows to schedule a procedure for execution in the main thread. The thread does not wait till it has finished executing, but continues at once.

And 2 protected properties:

ReturnValue This is an integer value that can be set to report back at the end of the thread. It is the value that is reported when `WaitFor` is used.

Terminated This is a flag that must be regularly checked during execution of the thread: when it is set to `True`, the thread should stop executing. It is set by the `Terminate` method.

Now, how can this class be used in thread programming ? In its simplest form, to execute code in a separate thread, a descendent of the `TThread` class must be made, and the `Execute` method must be overridden. All code that must run in a separathe thread should be called from inside this method.

For instance, given the following proceduree:

```
Procedure DoSomethingForLongTime;

begin
  // Do something here
end;
```

If it executes for a long time, the following code can be used to run the procedure in a separate thread:

```
Type
  TMyThread = Class(TThread)
    procedure Execute; Override;
  end;

Procedure TMyThread.Execute;
begin
  FreeOnTerminate:=True;
  DoSomethingForLongTime;
end;

Procedure DoSomethingForLongTimeInThread;
begin
  TMyThread.Create;
end;

begin
  DoSomethingForLongTimeInThread;
end.
```

The `DoSomethingForLongTimeInThread` routine will return at once, but the thread created in it will run as long as it takes to complete the task.

The `Execute` method starts by setting the `FreeOnTerminate` property. This property determines what happens with the thread instance when the `Execute` method has finished its work. By default, the thread object is not destroyed. This means that the code which has created the thread must keep an instance of the thread and free it once the thread has finished executing. By consequence, it must check when the thread has finished its work. This requires some bookkeeping code.

Some tasks are fire&forget: this means that they can be started and then left to run to completion without reporting back. For threads, this means the thread can be started, it will do its work, and when finished, the thread instance may be freed automatically: setting the `FreeOnTerminate` property to `True` accomplishes just this.

In the example above it is the thread code that decides that the tread instance will be freed or not. In general, it is the calling code that will decide whether the thread instance should be freed or not.

This poses a bit of a problem: As soon as the thread instance is created, it starts executing in a separate thread. That means that there may be no time to set the `FreeOnTerminate` property: The new thread may have finished executing before the property is set, thus creating a memory leak.

The `FreeOnTerminate` property is just an example, there may be other properties that need to be set before the thread starts executing. For example `OnTerminate` or some properties that are needed for the thread to perform its task correctly.

It is the operating system that determines the details of exactly when the new thread starts executing and whether or not then caller is still allowed to execute instructions before the new thread starts.

To remedy this, the constructor of the `TThread` class has a boolean parameter `CreateSuspended`:

```
constructor Create(CreateSuspended: Boolean;
                   const StackSize: SizeUInt = DefaultStackSize);
```

This can be used to create a thread in *Suspended state. The thread is created, but does not yet immediatly start executing. This gives the parent code time to set some properties. Thread execution can then be started using the `Start` procedure:*

```
Procedure DoSomethingForLongTimeInThread;
begin
  With TMyThread.Create(True) do
    begin
    // Set some other properties...
    FreeOnTerminate:=True;
    Start;
    end;
end;
```

4 A practical example

To demonstrate the use of threads, an application is created which will display a histogram with the distribution of the various characters found in text files: The algorithm will search a directory (and subdirectories) and create the histogram based on the contents of the files it finds. The user may select a directory, specify a set of extensions of filenames that interest him, and finally the user can say whether subdirectories should be searched or not. For simplicity, the statistics are for ASCII codes only, no Unicode characters are allowed. The statistics will be displayed in a bar chart. It is clear that this is a task which can take some times, specially if many large files are involved.

The algorithm to examine a file is simple:

```
Procedure Updatestats(Var Stats : TSTats; AFileName : String);

Const
  MaxSize = 1024 * 1024 * 10;

Var
  S : Array of Byte;
  b : Byte;
  P : PByte;
  I,R : Integer;
  F : THandle;

begin
  SetLength(S,MaxSize);
  F:=FileOpen(AFileName,fmopenRead or fmShareDenyWrite);
  if F<0 then exit;
  try
    Inc(Stats[256]);
    Repeat
      R:=FileRead(F,S[0],MaxSize);
      P:=PByte(@S[0]);
      For I:=1 to R do
        begin
        Inc(Stats[P^]);
        Inc(P);
        end;
    Until R<MaxSize;
  finally
    FileClose(F);
  end;
end;
```

`TStats` is a simple array type that keeps a count per ASCII code:

```
Type
  TStats = Array[0..256] of Int64;
  Pstats = ^TStats;
```

As ASCII codes run from 0 to 255, element 256 in the array will be used to store the number of treated files.

The following routine will traverse a directory, performing the counts on all files it finds.

```
Function GetStats(Var Stats : TSTats;
                  ADir,AExt : String;
                  Recurse : Boolean) : Integer;
```

`ADir` is the name of the directory, `AExt` is a list of extensions, separated by dots. `Stats` is the array that must be checked.

The algorithm starts by checking all files in the given directory:

```
Function GetStats(Var Stats : TSTats;
                  ADir,AExt : String;
                  Recurse : Boolean) : Integer;

Var
  Info : TSearchRec;
  E : String;

begin
  Result:=0;
  If FindFirst(ADir+'*.*',0,Info)=0 then
    try
      repeat
```

```
        E:=LowerCase(ExtractFileExt(Info.Name))+'.';
        If Pos(E,AExt)<>0 then
          begin
          inc(Result);
          UpdateStats(Stats,ADir+Info.Name);
          end;
      Until FindNext(Info)<>0;
    finally
      FindClose(Info);
    end;
```

If the `Recurse` *parameter is true, then the subdirectories are searched as well:*

```
  if Recurse then
    If FindFirst(ADir+AllFilesMask,faDirectory,Info)=0 then
      try
        repeat
          if ((Info.Attr and faDirectory)<>0)
              and (Info.Name<>'..') and (info.name<>'.') then
            Result:=Result+GetStats(Stats,
                    ADir+Info.Name+PathDelim,AExt,Recurse);
        Until FindNext(Info)<>0;
      finally
        FindClose(Info);
      end;
end;
```

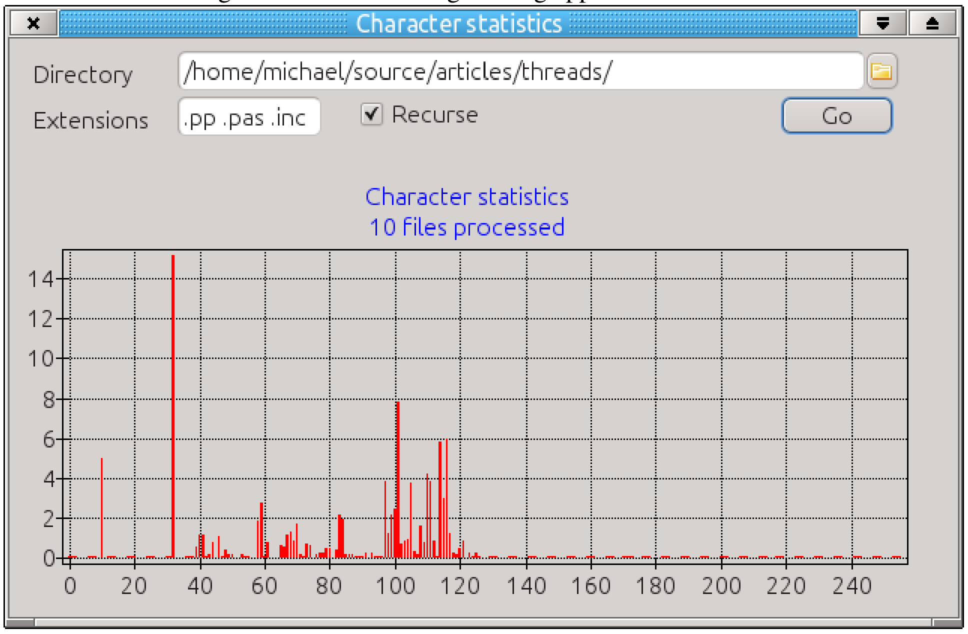
There is nothing surprising in this algorithm. The main form of the program has some edit controls that allow the user to specify the directory, extensions, and a checkbox to indicate that the search should be recursive. A click on the button `BGO` *will trigger the search:*

```
procedure TMainForm.BGoClick(Sender: TObject);

Var
  E,D : String;
  I : integer;

begin
  D:=IncludeTrailingPathDelimiter(DEDir.Directory);
  E:=EExt.Text;
  For I:=1 to Length(E) do
    If E[i]=' ' then E[I]:='.';
  E:='.'+E+'.';
  E:=StringReplace(E,'..','.',[rfReplaceAll]);
  FillWord(FStats,SizeOf(FStats) div 2,0);
  i:=GetStats(FStats,D,E,CBRecurse.Checked);
  ShowStats(i);
end;
```

The first lines are some cleanup of the directory name and the extensions. After that the stats array is cleared, and everything is passed to the `GetStats` *call. When that returns, the statistics are shown:*

```
procedure TMainForm.ShowStats(ACount : Integer);
```

Figure 1: The statistics gathering application in action



/home/michael/source/articles/threads/

.pp .pas .inc

```
Var
  B : TBarSeries;
  I : Integer;
  C : Int64;

begin
  B:=CChars.Series[0] as TBarSeries;
  C:=FStats[256];
  CChars.Title.Text[1]:=Format('%d files processed',[C]);
  C:=0;
  For I:=0 to 255 do
    C:=C+FStats[I];
  For I:=0 to 255 do
    B.SetYValue(I,FStats[i]/C*100);
end;
```

As can be seen, the values are displayed as a percentage of the total number of characters. When the program executes, it looks more or less like figure **??** *on page* **??***:*

When the GO *button is pressed, the program will freeze, till all files have been processed. No interaction is possible, till the statistics have been gathered. Clearly this is not desirable. The solution is to run the* `GetStats` *algorithm in a separate thread. This means the main thread is free to respond to user actions, update the display and more. When the thread has finished, the statistics are displayed.*

To do this, the main program needs to be notified when the thread is finished. The `OnTerminate` *event of the* `TThread` *class is triggered when the* `Execute` *method has stopped. The arguments needed fort the* `GetStats` *call must be passed on to the thread, and they must be passed on before the thread starts executing.*

One way of doing that is passing all needed arguments to the thread consructor:

```
TStatsThread = Class(TThread)
private
  FDirectory : String;
  FExtensions : String;
  FRecurse : Boolean;
  FStats : PStats;
Public
  Constructor Create(AStats : PSTats;
                     ADirectory,AExtensions : String;
                     Recurse : Boolean;
                     AOnDestroy : TNotifyEvent);
  Procedure Execute; override;
end;
```

The constructor just stored these arguments so they can be used later on by the `Execute` *method:*

```
constructor TStatsThread.Create(AStats: PSTats; ADirectory,
  AExtensions: String; Recurse: Boolean; AOnDestroy: TNotifyEvent);
begin
  FDirectory:=ADirectory;
  FExtensions:=AExtensions;
  FRecurse:=Recurse;
  FStats:=ASTats;
  OnTerminate:=AOndestroy;
  FreeOnTerminate:=True;
  Inherited Create(False);
end;
```

The execute method uses all supplied arguments to call `GetStats`*. Note that only the address of the stats array is passed to the thread, not the actual array.*

```
procedure TStatsThread.Execute;
begin
  GetStats(FStats^,FDirectory,FExtensions,FRecurse);
end;
```

The `OnClick` *method of the form will now have as it's last line the following:*

```
TStatsThread.Create(@FStats,D,E,CBRecurse.Checked,@ThreadDone);
```

The `ThreadDone` *method that is passed on to the thread is called when the thread terminates. It just calls the routine to show the statistics on screen:*

```
procedure TMainForm.ThreadDone(Sender: TObject);

begin
  ShowStats(0);
end;
```

When the 'Go' button is pressed now, the program will start to collect statistics, and at the same time the display remains responsive; the window can be resized, moved etc. When the thread is finished working, the display will be updated and the statistics shown.

5 Synchronization

However, the program still has a flaw: while the program is working, the user has no indication of what is happening. It would be much better if the program displayed the statistics while they are being built, for instance, once for each processed directory.

This presents us with a difficulty: the LCL (or VCL in Delphi) is not thread safe. That means that only the main thread can handle update of the display: the second thread is not allowed to mainpulate the GUI elements. So some form of communication between the main thread and the thread doing the work is required. The `Synchronize` *method of* `TThread` *allows to do just that. The* `Synchronize` *method allows the thread to let the main program thread execute a task, during which the thread waits till the main program thread has finished the task:*

```
Procedure Synchronize(AMethod: TThreadMethod);
```

`TThreadMethod` *is a simple procedure.*

To use this, the directory traversing algorithm must get an additional callback:

```
Type
  TDirectoryCallBack =
    Procedure(Const ADirectory: String) of object;

Function GetStats(Var Stats : TSTats;
            ADirectory,AExtensions : String;
            Recurse : Boolean;
            OnDirectoryDone : TDirectoryCallBack) : Integer;
```

At the end of a directory, the callback is called, and it gets passed the currently finished directory:

```
  // scan of files in directory
  If Assigned(OnDirectoryDone) then
    OnDirectoryDone(ADirectory);
  if Recurse then
    // Rest of code
```

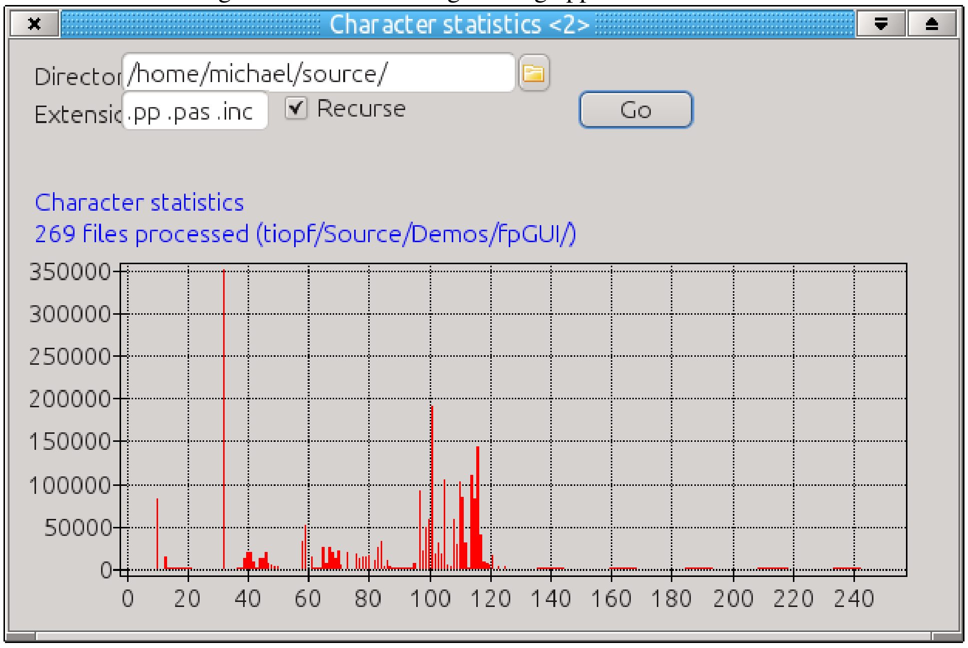
The thread passes a method to the `GetStats` *call:*

```
procedure TStatsThread.Execute;
begin
  FTotal:=GetStats(FStats^,FDirectory,FExtensions,FRecurse,@DirDone);
end;
```

In this method, 2 things happen: First, the current directory is stored, and secondly, the `Synchronize` *method is called: Synchronize can only handle a procedure without parameters, which is* `DoOnDir`*:*

```
procedure TStatsThread.DirDone(Const ADir : String);

begin
  FCurrentDir:=ADir;
  If Assigned(FOnDir) then
    Synchronize(@DoOnDir);
end;
```

Figure 2: The statistics gathering application in action



The `FOnDir` variable is an event handler that is set by the main form; Since it cannot be passed to `Synchronize`, it must be called from a procedure with the correct signature for `Synchronize`: `DoOnDir`. The `DoOnDir` method will be called from within the main thread and simply calls the callback:

```
procedure TStatsThread.DoOnDir;

begin
  FOnDir(FCurrentDir);
end;
```

The event handler in the form looks as follows:

```
procedure TMainForm.DirDone(const ADir: String);
begin
  FCurrentDir:=ExtractRelativePath(DEDir.Directory,ADir);
  ShowStats(-1);
end;
```

To make the display of the current directory somewhat nicer, it extracts the relative path of the current directory. Showstats will use this to set the caption of the bar chart.

The result of all these changes look like figure figure 2 on page 10.

6 Thread methods

The resulting program is now both responsive, and updates the statistics almost live. What is missing, is the option to stop the search process.

The `TThread` class offers the `Terminate` call to tell it to stop searching. However, the `GetStats` method does not know about the `Terminate` property. It would be possible *to add another event handler to* `GetStats`*, which would enable it to check at regular intervals whether it should stop or not.*

The resulting code would not look very nice: callbacks everywhere. Even without this, the thread constructor becomes a bit impressive, it gets passed 6 arguments:

```
Constructor Create(AStats : PSTats;
                   ADirectory,AExtensions : String;
                   Recurse : Boolean;
                   AOnDestroy : TNotifyEvent;
                   AOnDir : TDirectoryCallBack);
```

Time to refactor the code somewhat.

All this can be remedied by creating an object that has all the arguments to the thread constructor as properties and passing that object to the constructor.

The same is true for the GetStats call: it too can be passed an object.

At the same time, the fact that the main form "knows" that threads are used, is bad design.

It would be better if the main form could simply create an object, tell it to calculate the statistics, and report back at regular intervals, and let the object worry about whether or not threads should be used.

This leads - almost naturally - to the following object:

```
TStatsJob = Class(TObject)
Protected
  Procedure GetStats(Const ADirectory : String);
public
  Procedure Execute;
  Procedure Terminate;
  Property Stats : PStats Read FStats Write FStats;
  Property Dirs : Integer Read FDirs;
  Property CurrentDir : String Read FCurrentDir;
  Property OnDir : TNotifyEvent Read FOnDir Write FOndir;
  Property Extensions : String Read FExtensions Write FExtensions;
  Property StartDir : String Read FStartDir Write FStartDir;
  Property Recurse : Boolean Read FRecurse Write FRecurse;
  Property OnDone : TNotifyEvent Read FOnDone Write FOnDone;
end;
```

The object has a lot of properties: most of them are simply the arguments that were passed on to the thread. The `GetStats` *procedure that traverses the directories has also been made a method of the object: this means that all arguments which were passed on to the method, can be removed: they are accessible as properties.*

The `Terminate` *method will allow the main form to terminate the search routine.*

The `Execute` *method of this object is very simple:*

```
procedure TStatsJob.Execute;

begin
  FThread:=TStatsThread.Create(Self);
end;
```

`FThread` *is a private field of the* `TStatsJob` *object. The thread code now looks as follows:*

```
constructor TStatsThread.Create(AJob : TStatsJob);

begin
  FJob:=AJob;
  OnTerminate:=@FJob.ThreadDone;
  Inherited Create(False);
end;

procedure TStatsThread.Execute;
begin
  FJob.DoExecute
end;
```

Which is very simple. The `DoExecute` is also simplicity itself:

```
procedure TStatsJob.DoExecute;
begin
  FCurrentDir:='';
  GetStats(StartDir);
end;
```

The `GetStats` method is a copy of the old procedure, but the code is adapted so that it uses the properties `Extensions` and `Recurse` and the callback `OnDir` from the `TStatsJob` instance instead of getting them passed as parameters.

This structure allows a small change in the `Execute` method of `TStatsJob`:

```
procedure TStatsJob.Execute(UseThreads : Boolean = true);

begin
  if UseThreads then
    FThread:=TStatsThread.Create(Self)
  else
    begin
    FThread:=Nil;
    DoExecute;
    ThreadDone(Self);
    end;
end;
```

Giving the caller the option of using threads or not.

The code in the main form can now be refactored to the following:

```
procedure TMainForm.BGoClick(Sender: TObject);

Var
  E,D : String;
  I : integer;
  J : TStatsJob;

begin
  if FJob<>Nil then
    exit;
  E:=EExt.Text;
  For I:=1 to Length(E) do
    If E[i]=' ' then E[I]:='.';
  E:='.'+E+'.';
  E:=StringReplace(E,'..','.',[rfReplaceAll]);
  J:=TStatsJob.Create;
  J.StartDir:=IncludeTrailingPathDelimiter(DEDir.Directory);
  J.Stats:=@FStats;
  J.Extensions:=E;
  J.Recurse:=CBrecurse.Checked;
  J.OnDone:=@JobDone;
  J.OnDir:=@DirDone;
  FillWord(FStats,SizeOf(FStats) div 2,0);
  J.Execute;
  Fjob:=J;
end;
```

Note that the procedure exits at once if a job is already running. The form is completely unaware of the fact that `TStatsJob` is using threads to do the actual work.

The `Synchronize` method is a protected method of `TThread`. That means it cannot be called from within the `TStatsJob` class. Luckily, there is also a version of this call which is public:

```
class procedure Synchronize(AThread: TThread; AMethod: TThreadMethod);
```

As can be seen, it is a class method, meaning it must be called as follows:

```
procedure TStatsJob.DoneDir(ADir: String);
begin
  FCurrentDir:=ADir;
  TThread.Synchronize(FThread,@ShowDir);
end;
```

If `FThread` is nil, the method will still work. This means that any object can do update of a GUI without bothering to check if it is running in a thread or not: all it needs to do is call the `TThread` class method `Synchronize`.

The `TStatsJob` class has a `Terminate` method, which, when executed, will cause the class to interrupt the process. Since the `TStatsJob` instance is saved in a `FJob` field means that a button 'Cancel' can be put on the form, which, when clicked, will call the `Terminate` method of the `TStatsJob` class:

```
procedure TMainForm.BCancelClick(Sender: TObject);
begin
  If Assigned(FJob) then
    FJob.Terminate;
end;
```

All the `Terminate` method does, is set a flag: `Terminated`. The `GetStats` method is changed, so it checks this flags at regular intervals, after each file:

```
repeat
  E:=LowerCase(ExtractFileExt(Info.Name))+'.';
  If Pos(E,FExtensions)<>0 then
    UpdateStats(FStats^,ADirectory+Info.Name);
Until (FindNext(Info)<>0) or Terminated;
```

7 Queuing methods

The `Synchronize` method used to update the display has a disadvantage: it waits for the main thread to have updated the display before continuing to gather statistics. It would be more efficient to continue gathering statistics while the main form is displaying the last known statistics. This can be done with the aid of the `Queue` method. The `Queue` method will do the same thing as the `Synchronize` method: it schedules a task to be executed in the main thread. In difference with `Synchronize`, the `Queue` method will not wait for the main thread to have completed the task. Like `Synchronize`, it comes in 2 forms:

```
procedure Queue(aMethod: TThreadMethod);
class procedure Queue(aThread: TThread; aMethod: TThreadMethod);
```

Care must be taken when using `Queue`. There is no guarantee that the scheduled method will execute before the thread finishes executing. if `FreeOnTerminate` is `True`, the thread instance may no longer be in memory.

So, before the thread is done, it should remove any jobs that it had scheduled. This can be done with the `RemoveQueuedEvents` class method of `TThread`, which exists in 3 forms:

```
class procedure RemoveQueuedEvents(aThread: TThread;
                                   aMethod: TThreadMethod);
class procedure RemoveQueuedEvents(aMethod: TThreadMethod);
class procedure RemoveQueuedEvents(aThread: TThread);
```

The last form removes all methods queued by the thread.

So, using `Queue` instead of `Synchronize` will allow the `TStatsJob` class to continue gathering statistics, and when done, it should call `RemoveQueuedEvents`.

In the example of updating the statistics shown in the main form, it makes no sense to schedule an update of the display if the previous update was not yet handled. The main thread will simply execute the same method twice, one after the next, with the same statistics.

To prevent this from happening, we introduce a flag in `TStatsJob` called `ShowScheduled`:

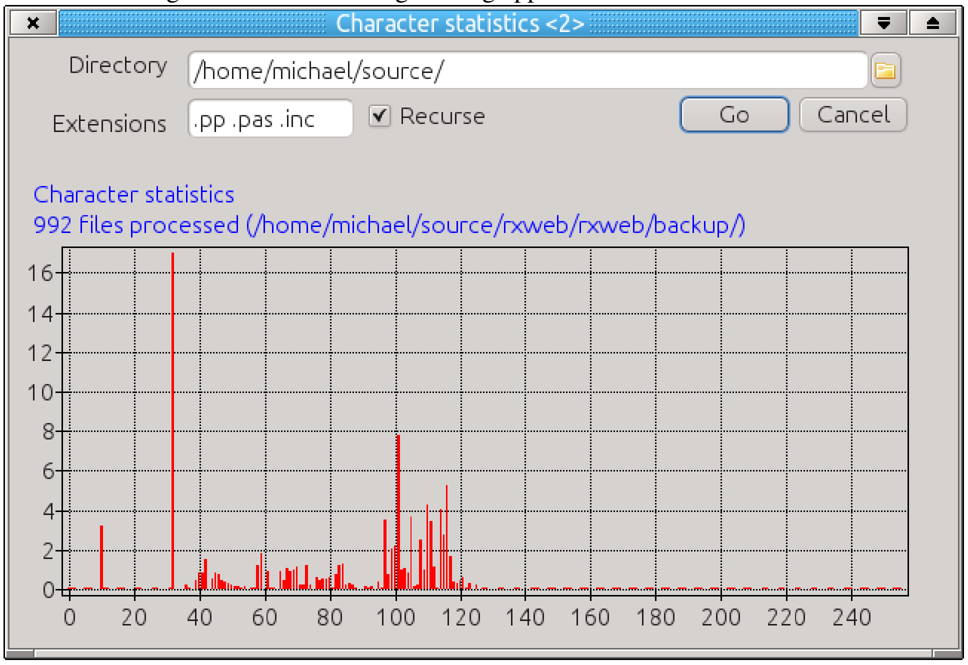
```
procedure TStatsJob.DoneDir(ADir: String);
begin
  FCurrentDir:=ADir;
  If Not ShowScheduled then
    begin
    ShowScheduled:=True;
    TThread.Queue(FThread,@ShowDir);
    end;
end;
```

The flag is cleared as soon as the statistics have been shown:

```
procedure TStatsJob.ShowDir;
begin
  If Assigned(FOnDir) then
    FOnDir(Self);
  ShowScheduled:=False;
end;
```

The `ThreadDone` method, called when the thread finishes, is adapted so it removes any queued events:

Figure 3: The statistics gathering application with cancel button



```
procedure TStatsJob.ThreadDone(Sender : TObject);
begin
  FCurrentDir:='';
  if ShowScheduled then
    TThread.RemoveQueuedEvents(FThread,@ShowDir);
  FThread:=Nil;
  If Assigned(OnDone) then
    OnDone(Self);
end;
```

With these changes, the process of gathering statistics can be cancelled, and will work faster, since it no longer needs to wait for the update of the display.

The result of all these changes - with the additional cancel button, look like figure figure 3 on page 15.

8 Protecting data

The program presented till now allows only 1 thread to gather statistics. As soon as it starts, the `Go` *button will no longer start a new thread. What if we wanted to allow the user to start as many jobs as he wants ?*

This would mean keeping a list of `TStatJob` *instances. Each time the user starts a new job, a new* `TStatJob` *instance is added to the list. When a job stops, the correct instance is removed from the list and freed.*

However, there is a small caveat. The various jobs (threads) will all work on the same `TStats` *array. Till now, the job thread was the only one updating the stats array, while the main form only showed the result.*

If multiple jobs run at the same time, they will be updating the `TStats` *array at the same* *time. That also means that errors can happen. To understand this, consider what happens in `UpdateStats`*

```
Inc(Stats[P^]);
```

What happens behind the scenes is that the current value of `Stats[P]̂` is fetched from memory, and stored in a register of the CPU. Then it is increased, and the result is again stored in memory. This opens the possibility of the following scenario in case of 2 threads. The following steps are executed in order:

1. *Suppose the initial value of `Stats[32]` is 10.*
2. *Thread 1 fetches the value 10 into the CPU.*
3. *Thread 1 increases the value to 11*
4. *Thread 2 fetches the value 10 into the CPU.*
5. *Thread 2 increases the value to 11*
6. *Thread 1 stores the new value (11) to `Stats[32]`*
7. *Thread 2 stores the new value (11) to `Stats[32]`*
8. *The final value of `Stats[32]` is 11.*

The result is wrong, the final value should be 12.

In the case of statistics, this results simply in wrong statistics. In other situations, this may lead to a crash of the program.

The updating of the data must be protected. One way of doing this is coordinating all write access so only one thread can change it at the same time. This can be done with a `TCriticalSection` object. A `TCriticalSection` can be used to put a barrier around a piece of code using it's `Enter` and `Leave` methods. All code executed between these 2 calls can only be executed by 1 thread:

```
CS.Enter;
try
  // Do things
finally
  CS.Leave
end
```

As soon as the first thread reaches the `CS.Enter`, it will continue to execute. If a second thread comes to `CS.Enter` while the first thread has not reached the `CS.Leave` command, the second thread will be blocked. As soon as the first thread executes `CS.Leave`, the second thread will be unblocked and will continue executing.

All threads must use the same `TCriticalSection` instance to guard access to a shared resource like this. That means that the critical section must be created in the main program, and passed to all jobs. For this, the `TStatsJob` gets a new property:

```
Property Sync : TCriticalSection Read FSync Write FSync;
```

Which is set when the job is created in the main form:

```
  J.Sync:=FSync;
```

The `TCriticalSection` instance is created in the `OnCreate` event handler of the form.

Statistics are updated in the `UpdateStats` routine:

```
   UpdateStats(ADirectory+Info.Name);
```

So the naive method would be to do

```
  CS.Enter;
  try
     UpdateStats(ADirectory+Info.Name);
  finally
    CS.Leave
  end
```

However, the `UpdateStats` method may take quite some time to complete. During all this time, no other thread can gather statistics, because of the critical section. This, in effect, recreates the situation where there is only 1 thread gathering statistics.

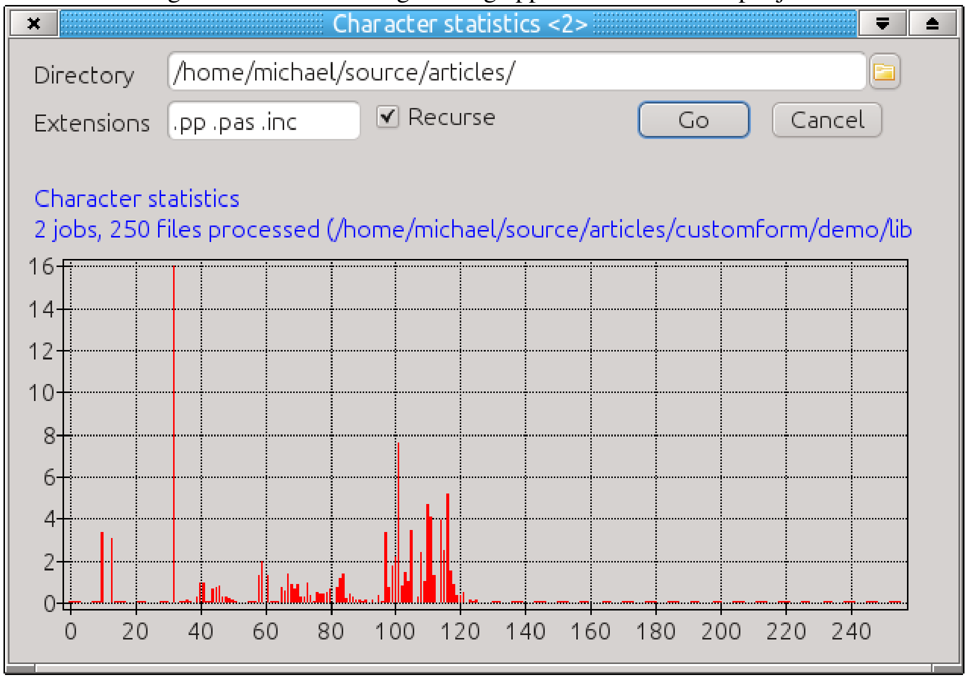
The solution is to let each thread gather statistics from a file in a local `TStats` array, and when it is done, add the result to the global `TStats` array. Only the second part - which executes very fast - needs to be protected by a critical section:

```
procedure TStatsJob.UpdateStats(const AFileName: String);

Var
  S : TStats;
  I : integer;

begin
  FillWord(S,SizeOf(S) div SizeOf(Word),0);
  ReadStats.UpdateStats(S,AFileName);
  FSync.Enter;
  try
    For I:=0 to 256 do
      FStats^[i]:=FStats^[i]+S[I];
  finally
    FSync.Leave;
  end;
end;
```

With all these changes in place, the `ShowStats` method can be changed so it also shows the number of currently running jobs:

```
procedure TMainForm.ShowStats(AJob : TStatsJob);

Const
  SJobFiles    = '%d jobs, %d files processed';
  SJobFilesDir = '%d jobs, %d files processed (%s)';

Var
  B : TBarSeries;
  I,C : Integer;
  S : String;
```

Figure 4: The statistics gathering application with multiple jobs



```
begin
  B:=CChars.Series[0] as TBarSeries;
  C:=FStats[256];
  If Assigned(AJob) and (AJob.CurrentDir<>'') then
    S:=Format(SJobsFilesDir,[FJobs.Count,C,AJob.CurrentDir])
  else
    S:=Format(SJobsFiles,[FJobs.Count,C]);
  CChars.Title.Text[1]:=S
  C:=0;
  For I:=0 to 255 do
    C:=C+FStats[I];
  For I:=0 to 255 do
    B.SetYValue(I,FStats[i]/C*100);
  Application.ProcessMessages;
end;
```

Note that the main form does not use the critical section when reading the data. The result can be seen in figure 4 on page 18. It is left as an exercise to the reader to find out why it would be more correct to update the display in a critical section as well.

9 Conclusion

Threads can be very useful when performing lengthy tasks in the background. The thread support in Free Pascal makes it very easy to do. Having multiple threads perform tasks in the background while accessing and modyfing the same shared data is a bit more tricky, but can be done as well with standard classes provided by Free Pascal.