Threads programmeren in Lazarus: een inleiding

Michaël Van Canneyt

April 11, 2013

Abstract

De ondersteuning van Thread programmeren in Free Pascal is grondig onder handen genomen, zodat het compatibel is met threading mogelijkheden in Delphi XE3. Een goede reden om het onderwerp threading nog eens te behandelen, en de mogelijkheden te bekijken, voor zij die niet erg bekend zijn met thread programmatie.

1 Introduction

Recent heeft de threading code in Free Pascal een flinke facelift gekregen. De mogelijkheden zijn nu identiek aan die van Delphi XE3, dus het moment is aangebroken om nog eens de threading mogelijkheden van Lazarus/Free Pascal te bekijken.

Threading wil zeggen dat de code van een programma zo geschreven wordt dat verschillende delen van het programma tegelijkertijd worden uitgevoerd, parallel. Elk deel van de code vormt een 'thread' (draad): Threading verondersteld hardware ondersteuning: verschillende processoren of een processor met verschillende cores. Indien de hardware het niet ondersteunt, zal het operating system gelijktijdige uitvoering simuleren door de verschillende delen van een programma beurtelings uit te voeren, net zoals het verschillende programmas beurtelings uitvoert en zo een illusie van gelijktijdige uitvoering creëert. De manier waarop het operating systeem de threads en programmas aan de beurt laat komen (de scheduling) verschilt van operating systeem tot operating systeem en is een onderwerp op zich.

Het verschil tussen threads en verschillende programmas is natuurlijk dat alle threads in een programma tegelijkertijd toegang hebben tot dezelfde gegevens en geheugenbereik. Programmas kunnen geheugen delen door shared memory te gebruiken, door middel van speciale aanroepen van het operating systeem.

Toegang tot gedeeld geheugen door de verschillende threads moet zorgvuldig ontworpen worden, of er kunnen zich vreemde dingen voordoen: Zolang er maar 1 thread schrijft, en een andere thread leest van een gemeenschappelijke geheugen locatie, is er weinig gevaar. Maar zodra 2 threads de inhoud van eenzelfde geheugenlocatie willen wijzigen,kunnen er problemen optreden. Om dit te voorkomen zijn speciale technieken nodig. In dit artikel tonen we hoe dit kan.

2 Classes voor thread programmatie in Free Pascal

De ondersteuning van threading in Free Pascal zorgt ervoor dat het vrij eenvoudig is multithreaded programmas te schrijven. FPC heeft verschillende classes en een voorziet een constructie in de Object Pascal taal zelf die samen het programmeren in Multiple threads mogelijk maken:

TThread Dit is het hart van het threading systeem: deze klasse stelt een thread in een programma voor. De klasse is gedefinieerd in de classes unit.

threadvar Dit is een sleutelwoord van de Pascal taal, en kan gebruikt worden om een speciaal type variabele te definiëren: Globale variabelen zijn normaal door alle threads in een programma te gebruiken. Als een globale variabele als `threadvar` gedeclareerd is in de plaats van als `var`, dan krijgt elke thread in the programma automatisch een copie van deze globale variabele.

TMultiReadExclusiveWriteSynchronizer Deze klasse kan gebruikt worden om een gegeven te beschermen zodanig dat het wel tegelijkertijd kan gelezen worden door verschillende threads in het programma, maar dat slechts 1 thread per keer het gegeven kan wijzigen. Deze klasse is gedeclareerd in de sysutils unit.

TThreadList Dit is een speciale vorm van `TList` die vergrendeld kan worden om exclusief door een thread gebruikt te worden. Deze klasse is gedefinieerd in de classes unit.

TCriticalSection Deze klasse kan gebruikt worden om ervoor te zorgen dat een stuk code slechts door 1 thread tegelijkertijd uitgevoerd kan worden. Deze klasse is gedefinieerd in de syncobjs unit.

3 De TThread klasse

Van alle klasses die hier gepresenteerd werden, is `TThread` veruit de belangrijkste. Deze klasse bevat op z'n minst de volgende publieke methodes:

Start Deze methode start de uitvoering van de thread indien ze 'suspended' (in opgeschorte toestand) gestart was.

Terminate Deze methode signaleert de thread dat uitvoering zo snel mogelijk gestopt moet worden. De thread dient zodanig geprogrammeerd te worden dat dit signaal op gezette tijden bekeken wordt.

WaitFor Deze methode wacht totdat de uitvoering van de thread stopt, en geeft dan de exit code van de thread terug.

Er zijn ook enkele properties:

FreeOnTerminate Deze property wordt hierna behandeld.

Handle De operating systeem handle van de thread.

ThreadID De operating systeem ID van de thread.

ExternalThread Een boolean die aangeecht of de thread buiten de RTL was gestart.

Priority An indicatie van de thread prioriteit.

Suspended Een boolean die aangeeft of de uitvoering van de thread tijdelijk opgeschort (suspended) is.

Finished Een boolean die aangeeft of de uitvoering van de thread gestopt is.

OnTerminate Een event handler die opgeroepen wordt wanneer de thread stopt met uitvoeren.

FatalException Indien een exception niet door de gebruikerscode onderschept wordt tijdens de uitvoering van de thread, zal de RTL deze onderscheppen en wordt opgeslagen in deze property.

Er zijn ook enkele protected methodes, dus alleen beschikbaar in de implementatie van de `TThread` klasse en zijn afgeleiden:

Execute Een virtuele abstracte methode die moet worden geimplementeerd in een descendent.

Synchronize Deze methode staat to een procedure te laten uitvoeren in de hoofd thread van de applicatie. (de GUI thread). The thread zal wachten tot het uitvoeren van de procedure in de hoofd thread afgelopen is.

Queue Deze methode staat toe een procedure in deen wachtrij te plaatsen, opdat ze uitgevoerd kan worden in de main thread. In dit geval wacht de thread niet tot de uitvoering voltooid is, maar gaat gewoon verder met uitvoeren van verdere instructies.

Er zijn ook 2 protected properties:

ReturnValue Dit is een integer waarde die gezet kan worden om aan het einde van het uitvoeren van de thread een waarde terug te geven (de exit status van de thread). Deze waarde wordt door `WaitFor` teruggegeven.

Terminated Deze boolean vlag moet op regelmatige tijdstippen door de thread nagekeken worden; Zodra ze op `True` staat, moet de thread stoppen met de uitvoering van zijn taak. Deze property wordt door de `Terminate` methode gezet.

Hoe wordt deze klasse nu gebruikt om threads te programmeren ? In zijn eenvoudigste vorm kan code uitgevoerd worden in een aparte thread door een descendent te maken van de `TThread` klasse, en de `Execute` methode te implementeren (met een 'override' modifier) Alle code die in de aparte thread moet uitgevoerd worden, moet in deze methode opgeroepen worden.

Gegeven de volgende procedure:

```
Procedure DoeIetsLangdurigs;

begin
  // Doe hier iets
end;
```

Als deze procedure lang uitgevoerd wordt, kan de volgende code ze in een thread uitvoeren:

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

Procedure TMyThread.Execute;
begin
  FreeOnTerminate:=True;
  DoeIetsLangdurigs;
end;
```

```
Procedure DoeIetsLangdurigsInThread;
begin
  TMyThread.Create;
end;

begin
  DoeIetsLangdurigsInThread;
end.
```

De `DoeIetsLangdurigsInThread` routine zal dadelijk terugkeren, maar de thread die werd aangemaakt zal zo lang blijven uivoeren tot de taak voltooid is.

De `Execute` methode start met het zetten van de `FreeOnTerminate` property.

Deze property bepalt wat er gebeurt met de thread instance als de `Execute` methode afgelopen is. Standaard wordt het thread object niet vrijgegeven. Dit betekent dat de code die the thread aangemaakt heeft, een instance van de thread moet bijhouden en deze instance vrijgeven eens de uitvoering van de thread voorbij is. Bijgevolg moet het nagaan wanneer de thread zijn taak uitgevoerd heeft: Dit veronderstelt wat boekhouding.

Sommige taken zijn "fire&forget:" dit betekent dat ze gestart mogen worden, en dat ze dan aan hun lot overgelaten kunnen worden: ze kunnen de hun toegewezen taak uitvoeren en hoeven niets terug te rapporteren. Voor threads wil dit zeggen dat ze gestart kunnen worden, en wanneer ze klaar zijn mag de thread instance automatisch vrijgegeven worden. Het instellen van de `FreeOnTerminate` property op `True` zorgt er voor dat dit gebeurt.

In het voorgaande voorbeeld is het de thread code die beslist of de thread instance vrijgegeven wordt of niet. In het algemeen is het de oproepende code die beslist of de thread instance vrijgegeven moet worden of niet.

Dit zorgt voor een probleem: zodra de thread instance aangemaakt wordt, start de uitvoering in een nieuwe thread. Dit wil zeggen dat het mogelijk is dat de thread al klaar is voor de `FreeOnTerminate` property gezet kan worden: De uitvoering van de nieuwe thread is gestopt voor de property gezet is, en er treedt een meomry leak op.

De `FreeOnTerminate` property maar 1 voorbeeld. Er zijn andere properties die mogelijks gezet moeten worden voor de thread zijn uitvoering start. Bijvoorbeeld de `OnTerminate` of andere properties die nodig zijn voor het correct uitvoeren van de thread.

Het operating systeem bepaalt de details van het starten van de nieuwe thread: meerbepaald of de nieuwe thread eerst begint uit te voeren, of de oproepende code eerst voortgaat met het uitvoeren van zijn code alvorens de nieuwe thread start.

Om al deze problemen te voorkomen, heeft de constructor van de `TThread` klasse een boolean parameter `CreateSuspended`:

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

Deze kan gebruikt worden om de thread in *Suspended (opgeschorte) toestand aan te maken. De thread instance is aangemaakt in het geheugen, maar begint nog niet meteen aan zijn uitvoering. Dit geeft de oproepende code de kans om nog wat properties te zetten. Het eigenlijke uitvoeren van de thread kan dan beginnen met de `Start` methode:*

```
Procedure DoeIetsLangdurigsInThread;
begin
  With TMyThread.Create(True) do
    begin
    // Set some other properties...
```

```
    FreeOnTerminate:=True;
    Start;
    end;
end;
```

4 Een practisch voorbeeld

Om het gebruik van threads te demonstreren, maken we een programma dat een histogram toont met de verdeling van de karakters in tekstbestanden. Het algortime doorzoekt alle bestanden in een map (en alle submappen) en maakt een histogram gebaseerd op de bestanden die het vindt.

De gebruiker kan een map selecteren, en de bestandsextensies die doorzocht moeten worden. De gebruiker kan ook aangeven of submappen doorzocht moeten worden. Voor de eenvoud van het algoritme zullen de statistieken alleen ASCII codes tellen, geen Unicode karakters. De statistieken worden in een blokdiagram getoond.

Het is duidelijk dat het opbouwen van deze statistiek een taak is die even kan duren, zeker als er veel en grote bestanden zijn.

Het algoritme om een bestand te doorzoeken is eenvoudig:

```
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;
```

Het `TStats` type is een eenvoudige array die een teller per ASCII code bevat:

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

De array bevat 257 elementen. Aangezien ASCII codes waardes van 0 tot 255 zijn, zal element 256 in de array gebruikt worden om het aantal doorzochte bestanden op te slaan.

De volgende routine doorzoekt een map, en doorzoekt elk bestand in de map die een goede bestandsextensie heeft:

```
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;
```

Als de `Recurse` parameter `True` is, worden de submappen ook doorzocht:

```
  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;
```

Dit algoritme is niet zo moeilijk. Het hoofdscherm van het programma heeft enkele edit controls waarmee de gebruiker de map, extensies kan instellen. Een checkbox staat toe aan te geven dat de zoekoperatie recursief is. Een klik op de `GO` knop start de zoekoperatie:

```
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;
```

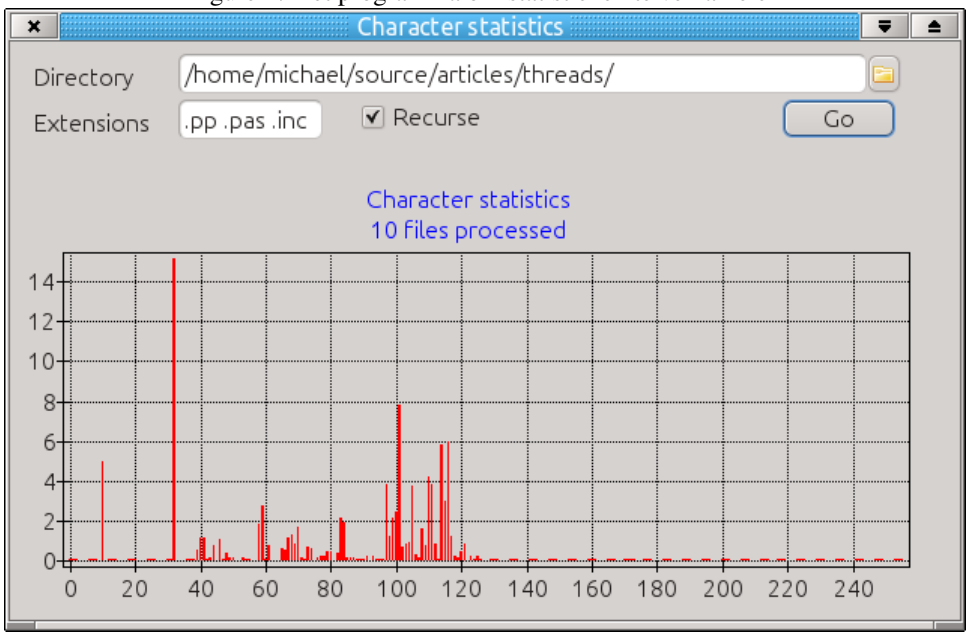
De eerste lijnen zetten de map en extensies om in een bruikbare vorm voor het zoekalgoritme Daarna wordt de stats array leeggemaakt, en alle elementen worden doorgegeven aan de `GetStats` routine. Wanneer deze routine terugkeert worden de statistieken getoond op het scherm:

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

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;
```

De waardes worden als een percentage van het totaal aantal karakters getoond. Wanneer het programma uitgevoerd wordt, ziet het er min of meer uit zoals in figure **??** *on page* **??***:*

Een druk op de `GO` bevriest het programma totdat alle bestanden doorzocht zijn. Het programma reageert niet totdat de statistieken verzameld zijn. Het is duidelijk dat dit niet erg gebruiksvriendelijk is. De oplossing is het `GetStats` algoritme in een aparte thread

Figure 1: Het programma om statistieken te verzamelen



te starten. Daardoor is de hoofd thread van het programma vrij om op gebruikersacties te reageren, het scherm opnieuw te tekenen indien nodig. Als de thread klaar is, kunnen de statistieken getoond worden.

Om dit te kunnen doen moet het hoofdprogramma verwittigd worden als the thread klaar is: Het `OnTerminate` *event van de* `TThread` *klasse wordt opgeroepen als de* `Execute` *methode afgelopen is. De parameters voor de oproep van* `GetStats` *moeten aan de thread doorgegeven worden, voordat de* `Execute` *methode uitgevoerd wordt.*

Een manier om dat te doen is alle argumenten door te geven aan de thread constructor:

```
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;
```

De constructor bewaart deze waarden zodat ze in de `Execute` *methode gebruikt kunnen worden:*

```
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;
```

De `Execute` methode gebruikt alle parameters om `GetStats` op te roepen. Merk op dat alleen het adres van de stats array doorgegeven wordt aan de thread, niet de echte array:

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

De `OnClick` methode van de form heeft nu als laatste instructie het volgende:

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

De `ThreadDone` methode die doorgegeven wordt aan de thread wordt opgeroepen wanneer de thread afgelopen is. Deze methode roept slechts de routine op die de statistieken op het scherm toont:

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

begin
  ShowStats(0);
end;
```

Wanneer de `Go` knop nu wordt ingedrukt, begint het programma statistieken te verzamelen, maar het scherm blijft reageren en wordt mooi hertekend indien nodig als de grootte wijzigt of wanneer het verplaatst wordt. Als de thread klaar is, wordt het scherm opnieuw getekend en worden de statistieken getoond.

5 Synchronisatie

Het programma heeft nog een tekortkoming: hoewel het zijn taak vervult, heeft de gebruiker geen idee van wat er aan de hand is. Het zou beter zijn als het programma de statistieken toont terwijl ze verzameld worden, bijvoorbeeld eenmaal per afgewerkte map.

Dit zorgt voor een extra probleem: de LCL (of de VCL in Delphi) is niet multi-thread. Dat wil zeggen dat alleen de hoofd thread van het programma het scherm opnieuw mag tekenen; andere threads mogen de GUI elementen niet wijzigen. Een of andere vorm van communicatie tussen de hoofd thread en de thread die het werk doet is nodig. De `Synchronize` methode van `TThread` is hiervoor gemaakt: De `Synchronize` methode staat een thread toe een de hoofd thread een taak te laten uitvoeren: terwijl de hoofd thread bezig is, wacht de thread tot de taak afgerond is.

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

`TThreadMethod` is een eenvoudige procedure.

Om dit te kunnen gebruiken, moet de routine die een map doorzoekt, een extra parameter krijgen, een callback routine:

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

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

Aan het eind van het afwerken van een map wordt de callback opgeroepen, en de naam van de map wordt doorgegeven:

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

De thread klasse geeft een methode door aan de `GetStats` routine:

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

In deze methode, gebeuren er 2 dingen: Eerst wordt de huidige map opgeslagen waarna de `Synchronize` methode opgeroepen wordt. Omdat `Synchronize` geen parameters aanvaardt, is een 2de methode nodig: `DoOnDir`:

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

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

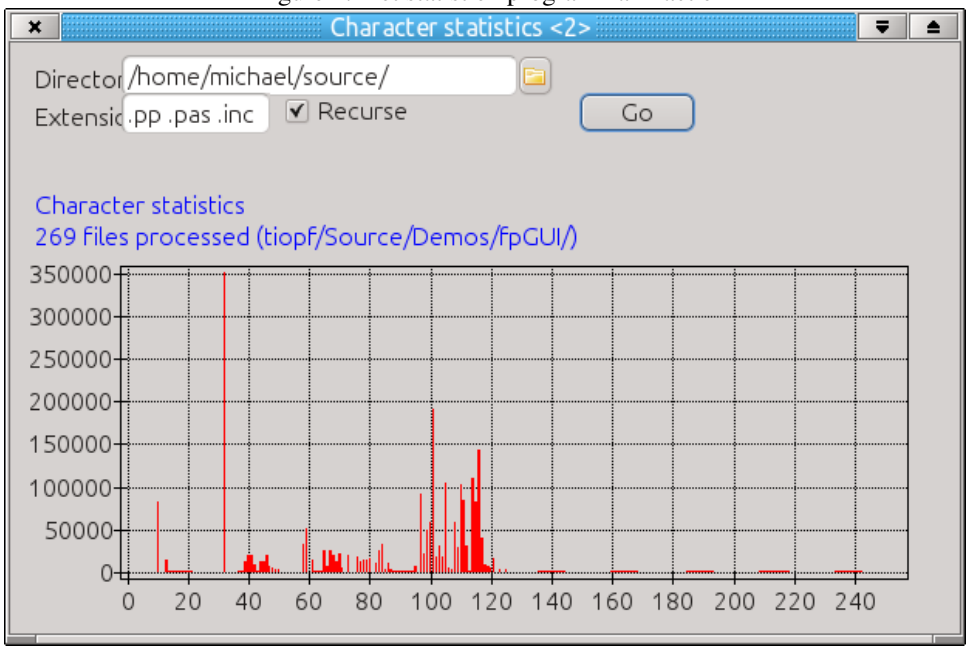
De `FOnDir` variable is een handler die door het hoofdscherm is doorgegeven aan de thread: Maar deze event handler kan niet doorgegeven worden aan `Synchronize`, en moet dus opgeroepen worden met de correcte parameters voor `Synchronize`: `DoOnDir`. De `DoOnDir` methode wordt opgeroepen vanuit de hoofd thread, en roept gewoon de event handler op:

```
procedure TStatsThread.DoOnDir;

begin
  FOnDir(FCurrentDir);
end;
```

D event handler in het scherm ziet er als volgt uit:

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

Figure 2: Het statistiek programma in actie



Om de statistieken iets duidelijker te maken, wordt de naam van de huidige map berekend, relatief t.o.v. de door de gebruiker aangegeven start map: De `Showstats` *routine gebruikt deze naam om de titel van het histogram aan te passen.*

Het resultaat van al deze wijzigingen is zichtbaar in figure 2 on page 11.

6 Thread methodes

Het programma zoals het nu is, is bruikbaar. Het toont de statistieken terwijl ze berekend worden. Wat nog ontbreekt is een optie om het zoekproces te onderbreken.

De `TThread` *class biedt de* `Terminate` *methode aan om de thread te verwittigen dat de uitvoering gestopt moet worden. Maar de* `GetStats` *methode heeft geen weet van de* `Terminate` *property.*

Een extra event handler zou aan `GetStats` *doorgegeven kunnen worden, zodat deze op gezette tijden kan kijken of de zoekoperatie gestopt moet worden of niet.*

De resulterende code zou er niet er mooi uit zien: callbacks allerhande. Zelfs zonder dit heeft de thread constructor al vrij veel parameters: 6.

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

Het is tijd om de code wat te herwerken.

Het is beter een klasse te maken dat alle argumenten voor de thread constructor bevat als properties, en dit object door te geven aan de thread constructor.

Hetzelfde geldt voor de GetStats methode: hier kan ook een object doorgegeven worden.

Een ander pijnpunt is het feit dat het hoofdscherm het gebruik van threads expliciet maakt: dit is slechte design.

Het zou beter zijn als het hoofdscherm een object kan aanmaken, de instructie geven om de statistieken te berekenen en op gezette tijden terug te rapporteren. Het is dan de beslissing van het object om threads the gebruiken of niet, en ervoor te zorgen dat het terug rapporteren in de main thread gebeurt.

Dit leidt op bijna natuurlijke wijze tot de volgende klasse:

```
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;
```

Deze klasse heeft een hoop properties: de meeste zijn simpelweg de parameters die aan de thread werden doorgegeven. De `GetStats` procedure die de mappen doorzoekt is een methode geworden van de klasse. Dat heeft het voordeel dat de meeste parameters die doorgegeven werden aan deze routine ook overbodig geworden zijn: ze zijn beschikbaar als properties van de klasse. Slechts de naam van de map wordt nog doorgegeven .

De `Terminate` methode zorgt ervoor dat het hoofdscherm de zoekroutine kan onderbreken.

De `Execute` methode van deze klasse is zeer eenvoudig:

```
procedure TStatsJob.Execute;

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

`FThread` is een private veld van de `TStatsJob` klasse. De thread code ziet er nu als volgt uit:

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

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

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

```
end;
```

Dit is zeer eenvoudig. De `DoExecute` is ook de eenvoud zelve:

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

De `GetStats` methode is een kopie van de oude procedure, maar de code is aangepast zodat de properties `Extensions` en `Recurse` en de callback `OnDir` van de `TStatsJob` klasse gebruikt worden, in de plaats van ze door te krijgen als parameters.

Deze structuur staat toe een kleine wijziging aan te brengen in de `Execute` methode van `TStatsJob`:

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

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

De oproeper van de `Execute` methode heeft nu de optie om threads the gebruiken of niet.

De code in het hoofdscherm kan nu herwerkt worden tot het volgende:

```
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;
```

Merk op dat de procedure stop als er al een taak bezig is. De code in het scherm bevat verder geen enkele referentie naar threads: dit is een implementatie detail dat volkomen verborgen is in de `TStatsJob` klasse.

De `Synchronize` methode is een protected methode van `TThread`. Dat wil zeggen dat ze niet opgeroepen kan worden vanuit de `TStatsJob` klasse. Gelukkig is er ook een publieke versie van deze methode.

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

Het is een class method en moet dus als volgt opgeroepen worden:

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

Als `FThread` `Nil` is, zal de methode nog steeds werken. Dit wil zeggen dat elke klasse de GUI kan bijwerken zonder te na te gaan of de klasse in de hoofd of in een andere thread uitgevoerd wordt: het enige wat moet gebeuren is de `TThread` class method `Synchronize` oproepen.

De `TStatsJob` klasse heeft een `Terminate` methode, die, als ze uitgevoerd wordt, ervoor zorgt dat de klasse het zoekproces onderbreekt. Aangezien de `TStatsJob` instance bewaard wordt in het `FJob` veld, kan een knop 'Cancel' op het scherm gezet worden. Als er op geklikt wordt, roept de onclick handler de `Terminate` methode van de `TStatsJob` klasse opgeroepen:

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

Het enige wat de `Terminate` methode doet, is een vlag zetten: `Terminated`. De `GetStats` methode is gewijzigd zodat de vlag op gezette tijdstippen nagekeken wordt, na elk verwerkt bestand:

```
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

De `Synchronize` methode gebruikt om het scherm bij te werken heeft een nadeel: Deze routine wacht tot de hoofd threadn het scherm heeft bijgewerkt. Pas na terugkeer worden de

statistieken verder verzameld. Het zou efficienter zijn de statistieken verder te verzamelen terwijl het hoofdscherm de laatst gekende statistieken aan het tekenen is.

Dit kan gedaan worden met behulp van de `Queue` methode. De `Queue` methode doet hetzelfde als de `Synchronize` methode: Het zet een taak in de wachtrij om te worden uitgevoerd in de hoofd thread. Het verschil met `Synchronize`, is dat de `Queue` methode niet wacht tot de main thread de taak volbracht heeft. Net zoals `Synchronize`, bestaat `Queue` in 2 vormen:

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

Er moet opgelet worden bij het gebruik van `Queue`. Er is geen garantie dat de de taak uitgevoerd wordt voor de thread klaar is. Als `FreeOnTerminate True` is, kan het zijn dat de thread niet langer in het geheugen is.

Daarom moet, voordat de thread klaar is, moet het alle taken die het in de wachtrij had gezet, er terug uithalen. Dit kan met de `RemoveQueuedEvents` class methode van `TThread`, die bestaat in 3 vormen:

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

De laatste vorm verwijdert alle methodes die door de thread in the wachtrij gezet werden.

Gebruik makend van `Queue` ipv `Synchronize` staat de `TStatsJob` klasse toe verder statistieken te verzamelen, en wanneer alles afgelopen is, moet `RemoveQueuedEvents` opgeroepen worden.

In het voorbeeld van het tonen van de statistieken in de main form is het nutteloos een bijwerking van het scherm in de wachtrij te zetten, als de vorige nog niet afgewerkt is: De hoofd thread zal dezelfde methode 2 maal uitvoeren, met dezelfde statistieken.

Om dit te verhinderen, wordt een vlag gedefinieerd in de `TStatsJob` klasse, `ShowScheduled`:

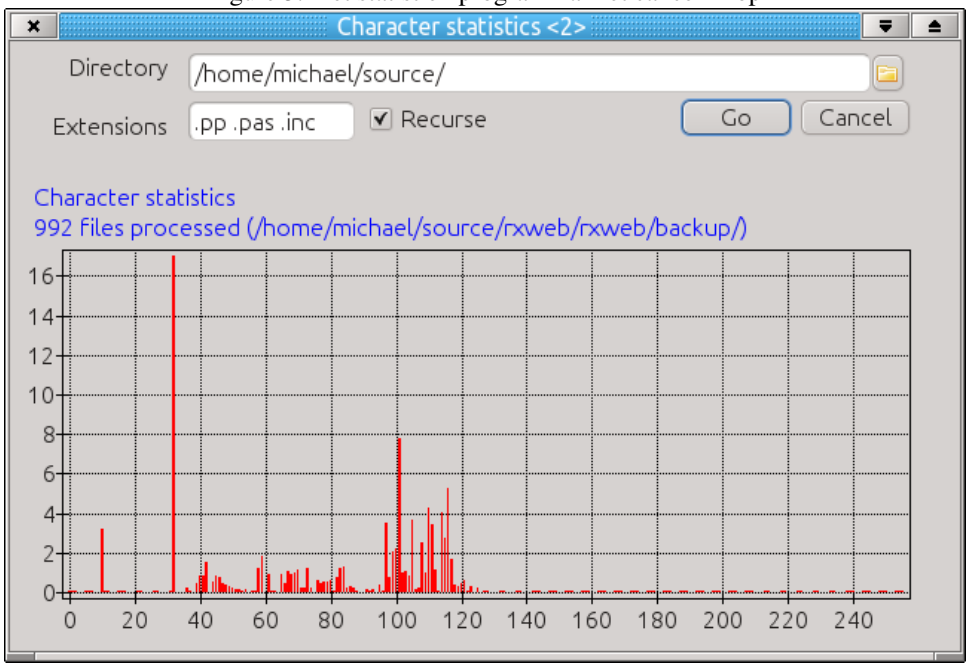
```
procedure TStatsJob.DoneDir(ADir: String);
begin
  FCurrentDir:=ADir;
  If Not ShowScheduled then
    begin
    ShowScheduled:=True;
    TThread.Queue(FThread,@ShowDir);
    end;
end;
```

Deze vlag wordt leeggemaakt zodra de statistieken op het scherm bijgewerkt zijn:

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

De `ThreadDone` methode, opgeroepen als de thread afgelopen is, wordt aangepast zodat alle methodes uit de wachtrij gehaald worden:

Figure 3: Het statistiek programma met cancel knop




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

Met deze wijzigingen zal het verzamelen van statistieken onderbroken worden, en het zal sneller werken, want er moet niet meer gewacht worden op het bijwerken van het scherm.

Het result van deze wijzigingen - met de bijkomende cancel knop ziet er uit zoals in figure 3 on page 16.

8 Data beschermen

Het programma zoals het tot nu toe werd uitgewerkt, gebruikt 1 thread om statistieken te verzamelen. Zodra deze thread gestart is, zal de `Go` knop geen nieuwe thread meer starten. Maar wat als we de gebruiker willen toestaan zoveel taken te starten als hij wil ?

Daarvoor moet een lijst `TStatJob` instances bijgehouden worden. Telkens de gebruiker een nieuwe taak start, wordt een nieuwe `TStatJob` instance aan de lijst toegevoegd. Als een taak beëindigd wordt, wordt de correcte instance uit de lijst gehaald en vrijgegeven.

Er zit echter een addertje onder het gras. De verschillende taken (threads) zullen allemaal dezelfde `TStats` array bijwerken. Tot nu toe was er slechts 1 taak die in de array schreef, en de hoofd thread toonde de gegevens uit de array, maar wijzigde deze niet.

Als meerdere taken tegelijkertijd werken, dan wordt de `TStats` array tegelijkertijd bijgew-

erkt door verschillende threads. Dit wil ook zeggen dat er fouten kunnen optreden. Om dit te begrijpen, kijken we even wat er in `UpdateStats` *gebeurt:*

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

Wat er achter de schermen gebeurt is dat de huidige waarde van `Stats[P]`$\hat{}$ *uit et geheugen gehaald wordt, en in een register van de CPU opgeslagen wordt. Het wordt dan opgehoogd, en het resultaat wordt daarna terug in het geheugen opgeslagen. Dit wil zeggen, dat in het geval van 2 threads het volgende kan gebeuren: De volgende stappen worden in volgorde uitevoerd:*

1. *Veronderstel dat de beginwaarde van* `Stats[32]` *gelijk is aan 10.*
2. *Thread 1 zet de waarde 10 in de CPU.*
3. *Thread 1 verhoogt de waarde naar 11.*
4. *Thread 2 haalt de waarde 10 in de CPU.*
5. *Thread 2 verhoogt de waarde naar 11.*
6. *Thread 1 zet de nieuwe waarde (11) in* `Stats[32]`
7. *Thread 2 zet de nieuwe waarde (11) in* `Stats[32]`
8. *De uiteindelijke waarde in* `Stats[32]` *is 11.*

Het resultaat is fout, de uiteindelijke waarde moet 12 zijn.

In het geval van de statistieken leidt dit mechanisme tot verkeerde statistieken. In andere gevallen kan het tot het afbreken van het programma leiden.

Het bijwerken van de data moet beschermd worden tegen dit soort fouten: Een manier van werken is alle schrijfoperaties door threads coordineren zodanig dat slechts 1 thread kan schrijven. Dit kan gebeuren door een `TCriticalSection` *klasse te gebruiken. Een* `TCriticalSection` *kan gebruikt worden om een barriere rond een stuk code te zetten met met de* `Enter` *en* `Leave` *methodes. All code uitgevoerd tussen deze 2 oproepen kan door slechts 1 thread tegelijkertijd uitgevoerd worden:*

```
CS.Enter;
try
  // Doe dingen
finally
  CS.Leave
end
```

Als de eerste thread de `CS.Enter` *uitvoert, zal het de code erachter dadelijk uitvoeren. Zodra een tweede thread de* `CS.Enter` *uitvoert wanneer de eerste thread het* `CS.Leave` *commando nog niet heeft uitgevoerd, wordt deze tweede thread geblokkeerd. Wanneer de eerste threade de* `CS.Leave` *uitvoert, wordt de tweede thread gedeblokkeerd en zet de uitvoering van de code na het* `CS.Enter` *commando voort.*

Alle threads moeten dezelfde `TCriticalSection` *instance gebruiken om de toegang tot een gedeelde resource te beschermen. Dat wil zeggen dat de TCriticalSection aangemaakt moet worden in het hoofdprogramma, en moet doorgegeven worden aan alle taken. De* `TStatsJob` *krijgt een nieuwe property:*

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

Deze wordt gezet wanneer de instance aangemaakt wordt in het hoodscherm:

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

De `TCriticalSection` instance wordt aangemaakt in de `OnCreate` event handler van het scherm.

De statistieken worden bijgewerkt in de `UpdateStats` routine:

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

De naieve manier om de critical section te gebruiken zou dus zijn:

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

Het effect is dus dat er maar 1 thread tegelijkertijd een bestand kan behandelen: geen enkele andere thread kan statistieken verzamelen omwille van de `TCriticalSection`. Het effect is dat we terug in de situatie zitten waar er maar 1 thread statistieken verzamelt.

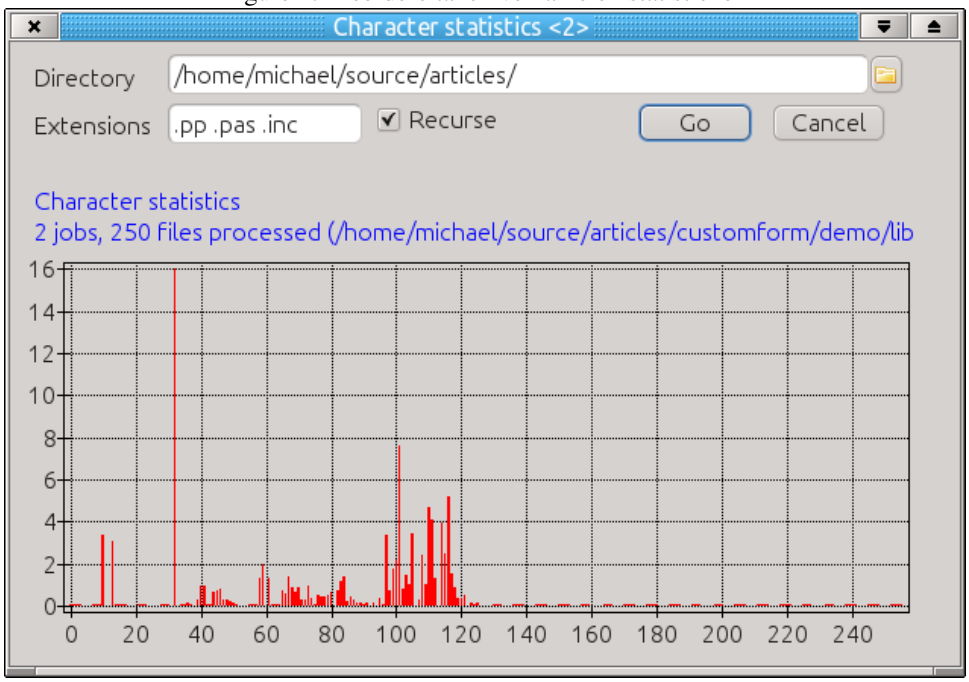
De oplossing is elke thread statistieken te laten verzamelen in een lokale `TStats` array, en als dit gedaan is, het resultaat aan de globale `TStats` array toe te voegen. Alleen dit tweede deel - dat zeer snel kan gebeuren - moet beschermd worden met een `TCriticalSection`:

```
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;
```

Met al deze wijzigingen klaar voor gebruik, kan de `ShowStats` methode nu gewijzigd worden zodat het ook het aantal taken in uitvoering kan tonen:

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

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

Figure 4: Meerdere taken verzamelen statistieken




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

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;
```

Merk op dat het hoofdscherm de TCriticalSection niet gebruikt bij het lezen van de data. Het resultaat is zichtbaar in figure 4 on page 19. We laten het aan de lezer over om uit te vissen waarom het correcter zou zijn het tonen van de statistieken in een TCriticalSection te behandelen.

9 Conclusie

Threads kunnen zeer nuttig zijn bij het uitvoeren van langdurige taken in de achtergrond. De ondersteuning voor threads in Free Pascal maakt dit tot een eenvoudige taak. Meerdere threads die dezelfde gedeelde gegevens wijzigen in de achtergrond, zijn iets moeilijker, maar kunnen net ook met standaard beschikbare klassen uitgevoerd worden.