In a previous article named " Easy multi-thread programming Delphi ", the AsyncCalls library was used to process multiple images at the same time. However, the processing of every single image was still strictly serial, even if image processing kernels are quite easy to accelerate spreading the load over multiple threads.

In this article we will see how the OmniThreadLibrary can be used to split a simple image processing kernel across multiple threads.

```
  procedure TProcessedImage.EffectBlackWhite(Bitmap : TBitmap32);  var x, y : integer;
Color : TColor32;      Red, Green, Blue : Cardinal;      Gray : Cardinal;
begin
 Bitmap.BeginUpdate;
try

for
y :=
0
to
Pred(Bitmap.Height)
do

for
x :=
0
to
Pred(Bitmap.Width)
do

begin
          Color := Bitmap.Pixel[x, y];              Red := (Color
and
$00FF0000
)
shr
16
;             Green := (Color
and
$0000FF00
)
shr
8
;             Blue := Color
```

```
and
$000000FF
;            Gray := (Red *
299
+ Green *
587
+ Blue *
114
)
div
1000
;
//Io=(0.299Ri + 0.587Gi + 0.114Bi)
         Bitmap.Pixel[x, y] := Color32(Gray, Gray, Gray);
end
;
except

end
;    Bitmap.EndUpdate;
end
;
```

This procedure converts a RGB image into a gray-scale image by replicating the luma value into all the RGB channels. The processing of each pixel is independent from other ones, so it can be computed in parallel. Still, allocating a thread for each pixel would generate a huge unnecessary overhead, so it is more efficient to process each line, containing Bitmap.Width pixels, in a separate thread. Moving to a parallel solution is very easy thanks to the OmniThreadLibrary:

1. include OtlParallel into the uses list
2. add a Parallel.ForEach().Execute() line that specifies the number of iterations of the loop, in this case the vertical rows of the image to be processed, so the loop counter goes from 0 to Pred(Bitmap.Height) as the loop handled by the y variable in the serial code
3. move the bulk of the image processing code into a lambda inside the Execute invocation, and replace the variable y with the argument of the lambda called elem
4. move the variables that work on a single pixel inside the lambda, outside of the parent procedure, or they will be shared across all the threads executing the lambda. If you encounter wrong results that vary at every run, check that all variables used to perform computations are declared in the lambda.

```
  procedure TProcessedImage.EffectBlackWhite(Bitmap : TBitmap32);  begin
Bitmap.BeginUpdate;
try
   Parallel.ForEach(
```

```
0
, Pred(Bitmap.Height)).Execute(
procedure
(
const
elem: integer)
var
x : integer;          Color : TColor32;          Red, Green, Blue : Cardinal;          Gray : Cardinal;
begin

for
x :=
0
to
Pred(Bitmap.Width)
do

begin
        Color := Bitmap.Pixel[x, elem];          Red := (Color
and
$00FF0000
)
shr
16
;          Green := (Color
and
$0000FF00
)
shr
8
;          Blue := Color
and
$000000FF
;          Gray := (Red *
299
+ Green *
587
+ Blue *
114
)
div
1000
;
//Io=(0.299Ri + 0.587Gi + 0.114Bi)
        Bitmap.Pixel[x, elem] := Color32(Gray, Gray, Gray);
end
```

```
;
end
);
except

end
;    Bitmap.EndUpdate;
end
;
```

That's all! I told you this was going to be easy... Thanks to the OmniThreadLibrary, splitting a computational load over multiple CPU cores is now a task that can be accomplished in a matter of minutes.

Even better, we can use OmniThreadLibrary to build DUnit test cases that check for code correctness under multi-threaded usage. The following code fragment tests that adding status updates and queries can be run at the same time on the database backend:

```
  procedure TestTFWDDatabase.TestMultiThread3;  const UpdatedUserName : string = 'Updat
edUser'
;        UpdaterUserName :
string
=
'UpdaterUser'
;
var
UpdatedUser : TFWDUser;      UpdaterUser : TFWDUser;      CleanupStatusChangeList :
TList<TFWDStatusChange>;      StatusChangePtr : TFWDStatusChange;
begin

/// create user to be updated
 UpdatedUser := TFWDUser.Create;    UpdatedUser.ProtocolId := UpdatedUserName;
ObjectDatabase.AddUser(UpdatedUser);
/// create user that will signal updates
 UpdaterUser := TFWDUser.Create;    UpdaterUser.ProtocolId := UpdaterUserName;
ObjectDatabase.AddUser(UpdaterUser);
/// add updates and run queries on update table at the same time
 Parallel.ForEach(
1
,
100
).Execute(
procedure
```

```
(
const
elem: integer)
var
TestStatusUpdate : TFWDStatusUpdate;          StatusChangeList : TList<TFWDStatusChange>;

begin

if
(elem
and
1
) =
0

then
begin

/// status update
          TestStatusUpdate := TFWDStatusUpdate.Create(UpdaterUserName,
PROTOCOL_FACEBOOK);
TestStatusUpdate.AddFriendStatusUpdate(UpdatedUserName, false, elem);
TestStatusUpdate.Process;               TestStatusUpdate.Free;
end

else
begin

/// run query
          StatusChangeList := ObjectDatabase.GetStatusChangesFromUserId(UpdatedUser.Id);
          StatusChangeList.Free;
end
;
end
 );      CleanupStatusChangeList :=
ObjectDatabase.GetStatusChangesFromUserId(UpdatedUser.Id);
Check(CleanupStatusChangeList.Count =
50
,
'Wrong count of status changes'
);    Check(UpdaterUser.UpdateCount =
50
,
'Wrong count of status updates'
);
/// additional tests hidden...
```

```
/// cleanup

for
StatusChangePtr
in
CleanupStatusChangeList
do
    ObjectDatabase.EraseStatusChange(StatusChangePtr);    CleanupStatusChangeList.Free;
  ObjectDatabase.EraseUser(UpdatedUser);    ObjectDatabase.EraseUser(UpdaterUser);
end
;
```

The Parallel.ForEach runs 100 iterations of the lambda function, that uses the elem parameter to choose which action shall be performed in that invocation, choosing between adding a status update on even iterations and running a query on status tables on odd iterations. This code fragment is a part of the DUnit test suite of the Friend Watchdog server, and together with a large set of the other automated unit tests ensures that the server can support multiple requests from Friend Watchdog clients without getting stuck or corrupting data.