Under Construction: Multi-Threading In Components

by Bob Swart

One of the benefits of the 32-bit Windows environment is that it supports multi-threading. Delphi 2 allows us to develop applications using multi-threading and this month we'll see what multi-threading is and how we can add support for it to our own components.

Threads

In a Win32 environment, we no longer have a *co-operative multi-tasking* system, instead, we have the notion of processes and threads, where each process has at least one thread (the primary thread) and terminates when the thread is terminated. Apart from the primary thread, however, processes can have other threads running in a *pre-emptive multi-tasking* environment, which means that each thread will get CPU time slices, no matter if and how other threads and processes behave.

This is the theory, at least: in practice some of the old 16-bit code in Windows 95 can and will quite happily let other threads wait seemingly forever for time slices when executed, but that's another story better told by the "experts" from that company in Redmond...

What are the benefits of multi-threading? Well, we can split off a thread for a long process (to be executed in the background), or even split the workload over multiple processors (on Windows NT), or just make sure the GUI is always available to the end-user while the application itself is using other threads to perform its tasks. I'm sure you can think of places in your apps or components where they could really benefit from this.

So, how does it work with Delphi? Fortunately (or should I say, as usual with Delphi), we don't have to use the Win32 API calls to work with multi-threading. To understand the implementation of multiple threads in Delphi, we just have to look at the `TThread` class.

The `TThread` class lets you create multiple threads of execution in your Delphi application. Like Experts and other abstract classes in Delphi, `TThread` can't be used as-is. You must derive a new thread class from `TThread` (for example `TDrBobThread`) and override two methods: `Execute` and `Synchronize`.

`Execute` is the method where we must put the code for the thread to execute. Returning from `Execute` terminates the thread, frees the thread's stack, and calls the optional `OnTerminate` event handler (see the on-line Help for details). Our `Execute` method must periodically check the `Terminated` property, because if this is set to `True`, we must return immediately (otherwise our thread won't terminate correctly when the `Terminate` method is called).

The `Synchronize` procedure has one argument of type `TThreadMethod` and lets you call this method of your thread object to avoid multi-thread conflicts with global VCL components.

VCL components can only be used from the main VCL thread: `Synchronize` calls the method you specify from within the main VCL thread so you can freely use properties and call methods of VCL components.

It is the `Synchronize` method that is allowed to use the global VCL visual components and call methods of these components. And if we add multiple threads to our application, we can safely access all VCL components and call their methods, as long as we only

➤ *Listing 1*

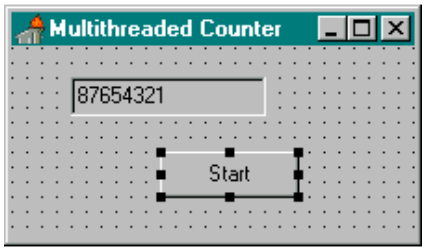
```
{ TThread }
EThread = class(Exception);
TThreadMethod = procedure of object;
TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher,
tpHighest,tpTimeCritical);
TThread = class
private
  FHandle: THandle;
  FThreadID: THandle;
  FTerminated: Boolean;
  FSuspended: Boolean;
  FMainThreadWaiting: Boolean;
  FFreeOnTerminate: Boolean;
  FFinished: Boolean;
  FReturnValue: Integer;
  FOnTerminate: TNotifyEvent;
  FMethod: TThreadMethod;
  FSynchronizeException: TObject;
  procedure CallOnTerminate;
  function GetPriority: TThreadPriority;
  procedure SetPriority(Value: TThreadPriority);
  procedure SetSuspended(Value: Boolean);
protected
  procedure DoTerminate; virtual;
  procedure Execute; virtual; abstract;
  procedure Synchronize(Method: TThreadMethod);
  property ReturnValue: Integer read FReturnValue write FReturnValue;
  property Terminated: Boolean read FTerminated;
public
  constructor Create(CreateSuspended: Boolean);
  destructor Destroy; override;
  procedure Resume;
  procedure Suspend;
  procedure Terminate;
  function WaitFor: Integer;
  property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
  property Handle: THandle read FHandle;
  property Priority: TThreadPriority read GetPriority write SetPriority;
  property Suspended: Boolean read FSuspended write SetSuspended;
  property ThreadID: THandle read FThreadID;
  property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
end;
```

do so within the method we passed to Synchronize.

So, while we could say that the VCL as-is may not be thread-safe (ie you cannot just access VCL components in two threads' Execute methods simultaneously), we have just seen how to make sure we can use the VCL in a thread-safe way, by passing a custom method to the Synchronize procedure and use the global VCL components from there. So, the VCL itself may not be thread-safe, but we can use it in a thread-safe way!

'The abstract base class TThread is defined in unit Classes (see Listing 1). I call it an abstract base class, because the procedure Execute is abstract and needs to be overridden by a new derived class of TThread and given a meaningful implementation.

➤ *Figure 1*



TFirstThread

Creating our first thread class, TFirstThread, is very simple: we derive a new class TFirstThread from TThread then define and implement a new Execute method, as can be seen in Listing 2.

The purpose of TFirstThread is to count in the background while the main program is processing the primary thread. Apart from the Execute method (the one that counts), we also need a constructor to initialise the counter to 0 and set the maximum number to which we'll be counting. Normally, the constructor of a TThread class has an argument called CreateSuspended, to specify whether or not the thread should start right away or be created and suspended, after which you can set the priority level first and then resume it. We can skip that one and just pass False to the inherited constructor instead.

➤ *Listing 2*

```
unit thread1;
interface
uses Classes;
type
  TFirstThread = class(TThread)
  public
    MaxCounter,Counter: LongInt;
    constructor Create(MaxCount: LongInt);
    procedure Execute; override;
  end;
implementation
constructor TFirstThread.Create(MaxCount: LongInt);
begin
  inherited Create(False);
  MaxCounter := MaxCount;
  Counter := 0
end;
procedure TFirstThread.Execute;
begin
  while (Counter < MaxCounter) and not Terminated do
    Inc(Counter)
end;
end.
```

➤ *Listing 3*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if (Sender as TButton).Caption = 'Start' then begin
    FirstThread := TFirstThread.Create(StrToInt(Edit1.Text));
    Edit1.Text := '0';
    (Sender as TButton).Caption := 'Stop'
  end else begin
    { 'Stop' }
    FirstThread.Terminate;
    Edit1.Text := IntToStr(FirstThread.Counter);
    FirstThread.Free;
    FirstThread := nil;
    (Sender as TButton).Caption := 'Start';
  end
end;
```

Of course this is not a very meaningful example, but if you need one you can modify the TFirstThread to copy a (big) file from one location to another, counting the number of bytes while copying. That should provide a more complex framework for the remainder of the problems and solutions in this example.

Using an instance of TFirstThread is very simple, and can be done with a form, an editbox and one button, as in Figure 1.

If we click on the button, we use the contents of the editbox to create an instance of TFirstThread that will count up to the given number. If we click on the button again, the TFirstThread class is terminated (we can toggle the button caption to indicate which kind of processing is going on in the background). Listing 3 shows the code for the Button1.OnClick event to create and terminate the TFirstThread instance.

Testing the FirstThread component with this form shows us that the counter continues in the background (or, more specifically, in a background thread), while the primary thread (the GUI) continues to work. We can click on other buttons and they'll perform just fine. The only thing we miss is progress on the background counter. Can't the thread tell us how far it is along with counting, for example by updating a label's caption with this information? The short answer is yes, since the VCL is thread-safe. However, we need to work by the rules, as we've seen in the earlier discussion.

Synchronicity

In order to access a property or method (or just about anything) from a VCL component, we need to be sure we're inside the so-called Synchronize method. In practice, this means we just need to write a method that we can pass as an argument to the Synchronize method of the TThread parent class. This way, we'll be ensured that our method can access the VCL components.

In our example, we can add a method UpdateCounter to the TFirstThread class and make it

update the caption of a label on the form after the counter has reached a certain value (say, after every 1,000th increment in value). See Figure 2.

The special routine that updates a label on the form is called `UpdateCounter`. We can freely access the caption of `CountLabel` here. We can't call the `UpdateCounter` method directly, however, but must give it as a value argument to a call of the `Synchronize` method of the `TThread` base class. See Listing 4.

Note that I use a special value `Synchrone` (which can also be defined on the form itself) to experiment with different values of updating the counter. If we set `Synchrone` too low (for example to a value below 100), we'll see that the counter doesn't increase that quickly, since a lot of the time is spent synchronising the background thread with the primary GUI thread. If we set the value of `Synchrone` high (for example to a value over 1,000,000) we'll see that the counter will increase steadily without too much overhead. For a nice effect, I usually set the value of `Synchrone` to a value between 10,000 and 100,000, but you can experiment on your machine for your optimal settings. And of course we could have used a `TGauge` and set the value of the `Progress` property.

At least it should be clear that we can do just about anything inside the routine that is called by `Synchronize`, as long as we take care not to call this routine too often (ie not for every update), as in that case the act of actually synchronising the thread with the primary GUI thread will take up too much time, and your application will be far too slow to be useful to anyone.

Thread Priority

It's important to know that each thread can also have a priority. The priority can have one of the following values: `tpIdle`, `tpLowest`, `tpLower`, `tpNormal`, `tpHigher`, `tpHighest` and `tpTimeCritical`. The higher the priority, the more CPU time slices a thread will get from the system's thread scheduler. The `TThread` class already has a property called `Priority`, so if we need to adjust the priority level of our thread, all we need to do is assign a new value to this property.

Note that this is also the reason why the original constructor has a Boolean `CreateSuspended` parameter: we can create a suspended thread, set its priority to the required value and only then activate it (useful for real important or real unimportant threads that should start with the correct priority right away).

Thread Expert

So far we've created and expanded a new thread class called `TFirstThread`. However, Delphi 2 has a built-in expert that supports the creation of a thread object in a somewhat more straightforward manner (albeit only slightly). Just select `File | New` from the main menu in Delphi 2 and click on the `Thread Object` from the repository (Figure 3).

Next, we can specify the name of the new thread object. In this example we'll call it `TSpagettiThread` as we'll be using this second example for the remainder of the column. The code generated by Delphi looks very similar to the code that we've written by hand. Special comments are included to remind us that we should not call any VCL method or access VCL properties unless we're in a routine called by the `Synchronize` method.

We will use this framework to work on our final example: the dining philosophers.

Multiple Threads

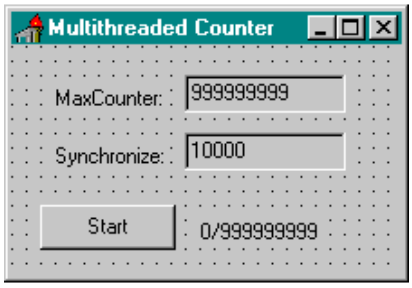
The problem of the dining philosophers is one where multiple concurrent processes (or threads) need to share common resources, such as printers or other devices. The biggest problem that can occur in these situations is the problem of deadlock, where one or more processes are waiting upon each other (eg to free a resource) before they can continue.

In the case of the dining philosophers we have a table with room for four monks to sit down. Each monk has a small bowl and one chopstick to the right of his bowl. There is a big bowl in the centre of the table containing rice, which can only be eaten by using two chopsticks. So, each monk must try to borrow the chopstick from the monk on his left. With a total of four monks and four chopsticks, this means that at most two monks at the same time can eat (the two sitting opposite each other). As long as the monks co-operate, nobody needs to starve. However, writing an algorithm that ensures that nobody starves is not easy.

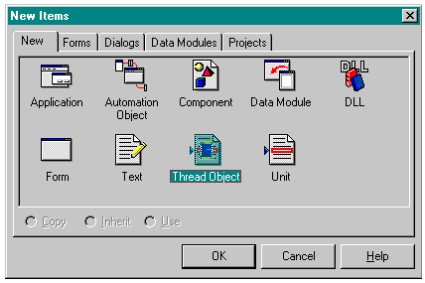
Brute Force Attempt

Let's first try a brute force approach, where every monk grabs

➤ *Figure 2*



➤ *Figure 3*



➤ *Listing 4*

```
procedure TFirstThread.Execute;
begin
  while (Counter < MaxCounter) and not Terminated do begin
    Inc(Counter);
    if (Counter mod Synchrone) = 0 then Synchronize(UpdateCounter)
  end
end;
procedure TFirstThread.UpdateCounter;
begin
  CountLabel.Caption := Format('%d/%d',[Counter, MaxCounter])
end;
```

the chopstick on his right (his own) and then tries to grab the chopstick on his left, in order to eat his bowl of rice. If the chopstick on his right isn't available, he waits until it becomes available before grabbing for the other one (this helps to prevent a deadlock situation where every monk has one chopstick and everyone waits for another to become available). We can implement this algorithm as shown in Listing 5, using four instances of the `TSpagettiThread` as four monks and a global array of `0..3` of `chopstick` to indicate whether a chopstick is free (-1) or in use (a value between 0 and 3).

We use a set of four `TGauge` controls to show the percentage of food reserve the monks still have (inside themselves, not in their bowls). After eating, this reserve has a value of 100 and it slowly degrades to zero (meditation takes a lot of energy!). If a monk's reserve reaches zero before he is able to eat again, the monk dies...

The constructor gives each monk his own unique ID and sets up the `FoodValue` `TGauge` control (note that each monk only uses a pointer to a `Gauge` that has been placed on a form somewhere else). The `ShowFood` method is the one called by `Synchronize` that updates the `Progress` properties of the gauges so we can see how much food reserve the monks still have.

There are two very important methods when sharing resources: the `Wait` and `Signal` routines. `Wait` is used by a monk to wait for a chopstick to become available, while `Signal` is used to signal that a given chopstick is available again, after which the other monk can pick it up. Note that each chopstick can only be used by two monks, so if one monk signals the availability of a chopstick, there can be only one other monk ready to pick it up; the first monk should go back and meditate for a while again before needing to eat anyway...

Using the `Wait` and `Signal` methods, our first attempt at the `Execute` routine can be implemented as follows: we grab the chopstick on our right (or wait until it becomes available), then reach for the one on our left (or wait), start to eat, put down the chopsticks again (first the one on our left, then the one on our right), think for awhile and then try to eat again. This carries on until we're terminated *[Yikes, what kind of Monasteries do you have in The Netherlands! Editor]* or die from lack of food (ie when our `FoodValue` reaches zero before we can replenish it with rice again). See Listing 6.

When creating four monks this way and running the example program (which is on this month's disk of course), it sure looks like nothing is wrong with this algorithm: the monks are eating a lot and their food reserve seldom gets below 70%. Since these guys eat fairly quickly they have lots of time to meditate and wait for a chopstick to become available so they can eat again. See Figure 4.

Note that a colour of gray (in the gauge) means that no chopstick is in use by the monk, blue means that the monk is currently holding one chopstick and is waiting for another, while yellow means that this monk is now holding two chopsticks and is happily eating rice before putting the sticks down again.

➤ *Listing 5*

```
unit spagetti;
interface
uses Classes, Gauges;
const
  MaxPlate = 3; { four monks: 0, 1, 2 and 3 }
type
  TSpagettiThread = class(TThread)
  private
    Plate: Integer;
    FoodValue,Sticks: Integer; { 0..100 }
    Food: TGauge;
  public
    constructor Create(ID: Integer; Gauge: TGauge);
  protected
    procedure Execute; override;
  private
    procedure ShowFood;
    procedure Wait(Chop: Integer);
    procedure Signal(Chop: Integer);
  end;
implementation
uses Graphics;
var
  Chopstick: Array[0..MaxPlate] of Integer; { -1 = FREE }
{ TSpagettiThread }
constructor TSpagettiThread.Create(ID: Integer; Gauge: TGauge);
begin
  inherited Create(False);
  Plate := ID;
  Food := Gauge;
  FoodValue := 100;
end {Create};
procedure TSpagettiThread.ShowFood;
begin
  Food.Progress := FoodValue;
  case Sticks of
    1: Food.ForeColor := clNavy;
    2: Food.ForeColor := clYellow;
    else Food.ForeColor := clGray;
  end;
  if FoodValue <= 0 then Food.Visible := False
end {ShowFood};
procedure TSpagettiThread.Wait(Chop: Integer);
var i: Integer;
begin
  if Chopstick[Chop] >= 0 then begin
    repeat
      for i:=1 to 10000 do if (i div 10000) = 1 then Dec(FoodValue);
      Synchronize(ShowFood) { wait }
    until (Chopstick[Chop] < 0) or (FoodValue <= 0)
  end;
  if FoodValue > 0 then begin
    Chopstick[Chop] := Plate { taken by me };
    Inc(Sticks)
  end
end {Wait};
procedure TSpagettiThread.Signal(Chop: Integer);
begin
  if Chopstick[Chop] = Plate then begin
    { did I take it? }
    Chopstick[Chop] := -1 { release chopstick again };
    Dec(Sticks)
  end
end {Signal};
var i: Integer;
begin
  for i:=0 to MaxPlate do Chopstick[i] := -1;
end.
```

However, after a certain amount of time (this can take up to a minute, depending on random factors and the speed of your machine), suddenly it seems there's something wrong: the monks have each gotten hold of one chopstick and each is waiting for another one to become available. While waiting, no-one is willing to give up his own chopstick (this was not foreseen: we should only need to wait for a chopstick if another monk is eating, which shouldn't take long anyway). See Figure 5.

The food reserve of each monk drops fast and it doesn't take long before one or more just die from lack of food. Of course, the other(s) just pick up the available chopstick(s) and start eating again. This may, however, result in a situation where only one monk is left, who should not be afraid of running out of chopsticks ever again, of course... See Figure 6.

Surely, four wise men eating rice with chopsticks should be able to co-operate a little bit more and see to it that they all survive! This is one of the classic examples of concurrent programming, by the way, and if you're interested, I challenge you to try to solve this little puzzle before reading on to the solution.

Sharing The Table

The solution for the dining philosophers is to realise that they not only need to share the chopsticks, but also the room.

If they want to ensure that at least one monk is always able to eat (in other words, no deadlock can occur), then at most three monks can be allowed to sit at the table at the same time. That way, we can ensure that at least one monk is allowed to eat, after which he needs to put down his chopsticks and get off the table. The moment this monk leaves the table, the fourth one can join the table and pick up a chopstick. At that time, another of the two monks will be eating, so the monk only has to wait for the second chopstick to become available, which can be proven to become available as the other monks eat and leave the table as well.

Running the simulation with this additional constraint (the `{$IFDEF ROOM}` compiler directive) results in a setting in which the monks never run out of food reserves. To test it, you can run the program for a few days and see that the monks are all still alive. Of course, this is but one solution and you're welcome to try your own solution instead. For a more theoretical background on the principles of concurrent programming, I can recommend the book with the same title by M Ben-Ari (Prentice Hall, ISBN 0-13-701078-8). A bit old (1982: we used it at University) but with Pascal syntax and still very good on the theoretical stuff.

Efficiency

Are threads efficient? Yes and no. Of course, it may seem at first that a program is now faster since a background thread is executing a lot of the workload from the program itself. However, if the main GUI thread still has to wait for the answer to provide to the end-user, the program hasn't gotten any faster than before. And no matter how many threads we start, we still need to perform the same amount of work; using threads mean we're going to use more CPU cycles than without threads (there's no denying that).

Concurrent programming techniques offer us a whole new world of experimenting with background processing that can also help to take the workload off the main thread and into the background.

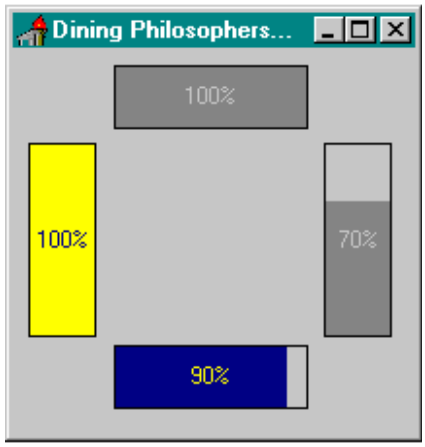
➤ *Listing 6*

```
procedure TSpagettiThread.Execute;
var i: Integer;
begin
  repeat
    for i:=1 to 50000 do
      if (i mod 10000) = 0 then
        Dec(FoodValue);
    Synchronize(ShowFood);
    Wait(Plate);
    Synchronize(ShowFood);
    if FoodValue > 0 then begin
      Wait((Plate+1) mod MaxPlate);
      if FoodValue > 0 then
        for i:=1 to 10000 do
          if (i div 10000) = 1 then
            FoodValue := 100; {eat}
      Synchronize(ShowFood);
      Signal(
        (Plate+1) mod MaxPlate);
      Synchronize(ShowFood)
    end;
    Signal(Plate);
    Synchronize(ShowFood);
  until Terminated or
    (FoodValue <= 0);
  Food.Visible := False
end {Execute};
```

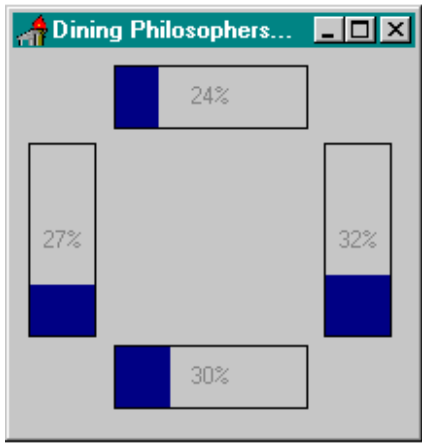
Conclusions

Multi-threading is a powerful feature of Win32. I hope to have shown that the implementation of threads in Delphi 2 is very elegant and easy to use. We can implement our own derived classes from `TThread` and work with them in a thread-safe way, when we play by the rules of

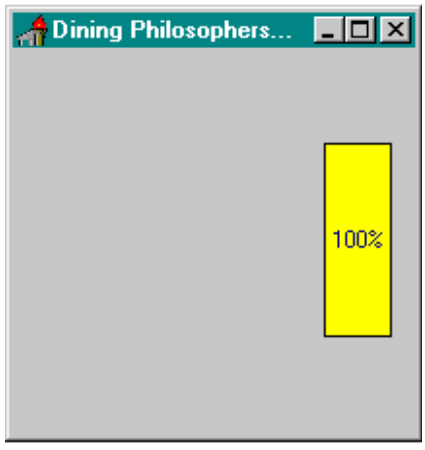
➤ *Figure 4*



➤ *Figure 5*



➤ *Figure 6*



using `Synchronize`. We've also explored some basic issues of concurrent programming, which will become more and more important as multi-threading and resource sharing becomes more common.

Next time, we'll go back to components and experts, as we'll be building a Wizard Component.

Bob Swart (Dr.Bob, home.pi.net/ ~drbob/) is a professional knowledge engineer and technical consultant using Delphi and C++ for Bolesian (www.bolesian.com) and co-author of *The Revolutionary Guide to Delphi 2*. He is now co-working on *Delphi Internet Solutions*, a new book about Delphi and the internet.