# Easy Multi-threading with Visual FoxPro

*Kevin Ragsdale*
*The Net Results*
*Voice: 931-409-9899*
*Email: kevin@kevinragsdale.net*
*http://kevinragsdale.net*
*Twitter: @KevinRagsdale*

*This document provides an introduction to Christof Wollenhaupt's DMULT.DLL – a "helper" utility which provides multi-threaded capability to your Visual FoxPro applications. Christof's DLL is by far the easiest method I've found for adding multi-threaded capabilities to my own applications.*

# Introduction

It all started with a simple question in the September 2006 issue of *Advisor Guide to Microsoft Visual FoxPro*:

*Question: Many things would be easier to do if I could run them in a background thread. Is there a way to create multi-threaded applications in VFP?*

The question was answered by Contributing Editor Christof Wollenhaupt:

*Answer: There's a clear answer to that:* ***It depends****.*

Christof went on to explain that "it depends" really depends on what you consider to be a multi-threaded application.

We've had the ability to create multi-threaded DLLs in Visual FoxPro since Version 6, Service Pack 3. But this doesn't mean we're creating multi-threaded applications just because we are creating multi-threaded DLLs. When you instantiate a COM server located in a DLL, Visual FoxPro can only load it into the same thread as the main application.

In essence, if your DLL is performing "SomeLengthyProcess", Visual FoxPro (your application) must wait for "SomeLengthyProcess" to complete before it can proceed. This can confuse users as your app becomes unresponsive to user actions, and also opens the door for Windows to append a not-so-beautiful "(Not Responding)" message to the title bar of your application (Figure 1).

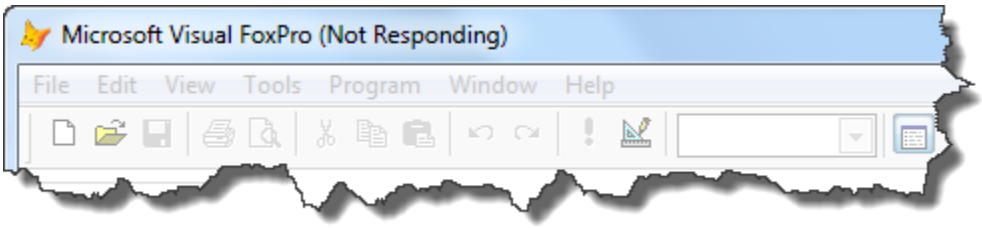


**Figure 1: Windows provides a "helpful" Not Responding message**

This article introduces a C++ DLL "helper utility" written by Christof, which he explained in further detail in the Advisor Answers section mentioned above.

## So, what is multi-threading anyway?

Based on all of the definitions I've seen on multi-threading via books, articles, and numerous Google searches, multi-threading is simply this: *the ability to perform multiple activities at the same time within an application*. This is even truer these days with our multi-core CPUs.

From WikiPedia: *On a single processor, **multithreading** generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multi-core system, the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.*

Source: http://en.wikipedia.org/wiki/Multithreaded

## What are the advantages of multi-threading?

For me, the number one advantage of multi-threading is simple: responsiveness of the application user interface.

It's been said many times before that for users, your application's user interface ***is*** the application. In my experience, nothing screams lousy UI (and lousy application) like a non-responsive UI.

Another advantage is the ability to perform necessary and/or optional functions without interrupting the users' workflow. I use multi-threaded requests in several apps to check for program updates in the background. If an update is found, the app will announce the availability to the user. Another app pulls a variety of information from several different websites in the background, parsing the information, writing to tables, etc., all while the user does other things in the app without ever realizing how much work is going on in the background. In short, the app remains responsive to the user, even while it's busy performing multiple tasks.

## A quick example of "SomeLengthyProcess"

Let's take a look at an unresponsive UI triggered by SomeLengthyProcess in a multi-threaded DLL.

I've got a project named SAMPLE.PJX, which contains a single PRG file named SAMPLEMAIN.PRG, and I've compiled the project as a multi-threaded DLL (SAMPLE.DLL in the source code for this session).

Here's the code from SAMPLEMAIN.PRG:

```
DEFINE CLASS EasyMTServer As Session OLEPUBLIC
  PROCEDURE SomeLengthyProcess (toCallback)

    ** We'll declare the Windows API Sleep
    ** function, so we can ensure the
    ** process runs for 20 seconds.
    DECLARE Sleep IN WIN32API Long

    ** Create a variable for
    ** our FOR/ENDFOR loop.
    LOCAL lnCount As Integer

    ** Print a string to the main VFP window.
    FOR lnCount = 1 TO 20
      ** Create a SYS(2015) value, and
      ** print it back to the VFP window.
      toCallback.DoCmd("? + ALLTRIM(SYS(2015))")

      ** Let's sleep for 1 second
      Sleep(1000)
    ENDFOR

    ** Write out a Done!
    toCallback.DoCmd("? + 'Done!'")

    CLEAR DLLS "Sleep"
  ENDPROC
ENDDEFINE
```

Looks like a lengthy process, right? It should take 20 seconds to run. After instantiating the COM server (DLL), I'm going to call the SomeLengthyProcess method and pass in the _VFP object as the callback. For 20 seconds, the DLL will create a SYS(2015) string (once every second) and tell _VFP to write the string out to the main VFP window.

Calling the DLL from the VFP Command Window looks like this:

```
loDLL = CREATEOBJECT("sample.EasyMTServer")
loDLL.SomeLengthyProcess(_VFP)
```

Perfect! I've got a multi-threaded DLL that stays busy for 20 seconds, writing back to my VFP session so I can see what it's doing. But, there's a problem:

My VFP session is unresponsive (Figure 2). I can click, double-click and even triple-click all I want, but *nothing is happening while SomeLengthyProcess is running*!

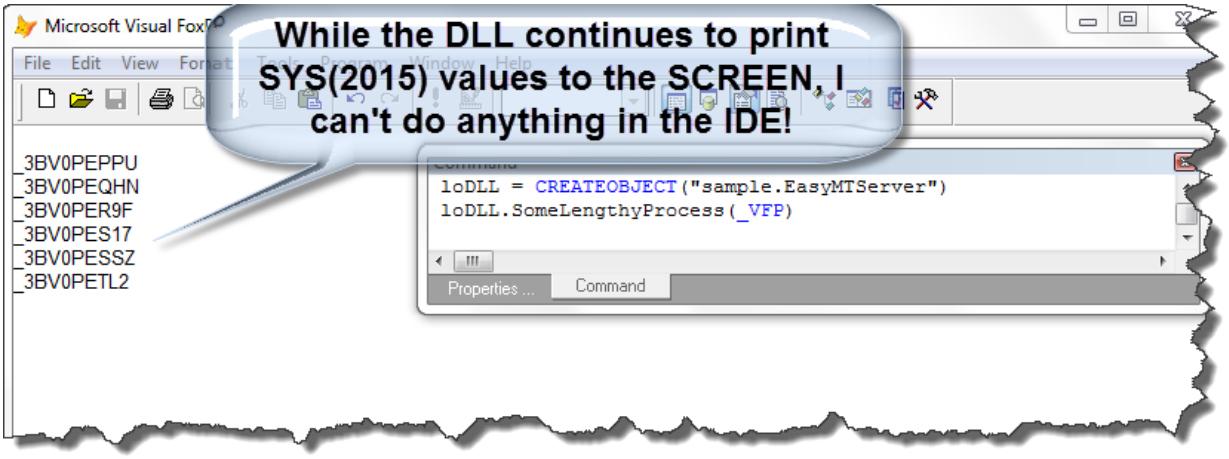


**Figure 2: A multi-threaded DLL doesn't run in a separate thread**

Clicking around the VFP window can cause Windows to append the previously shown "Not Responding" message to the title bar. At times, the VFP window (or your application) can fade (Figure 3) and in Windows Vista/Windows 7 you (or your users) may see a message like the one seen in Figure 4.

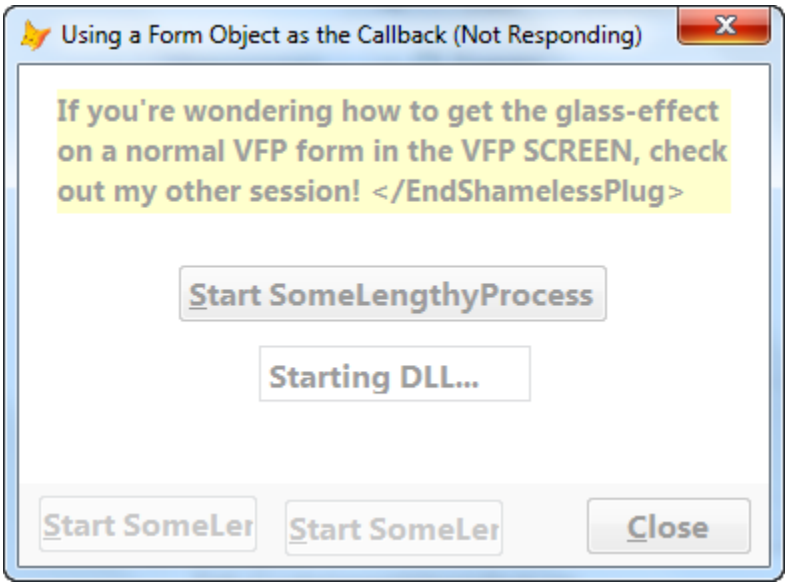


**Figure 3: An application that is unresponsive and faded by Windows**

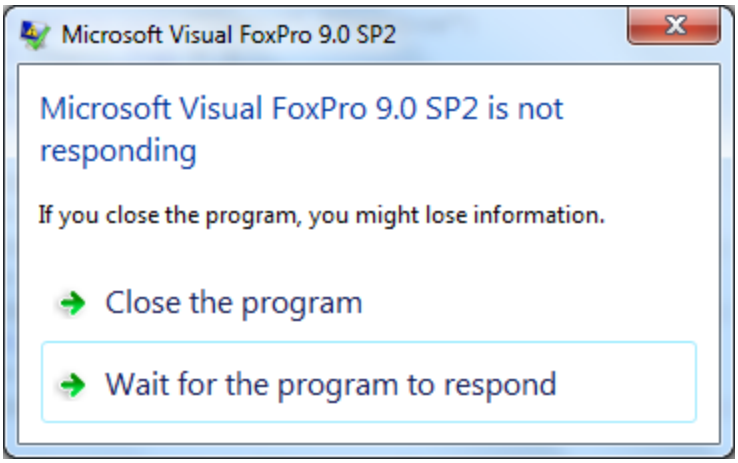


**Figure 4: Not a lot of fun when it's your app that is not responding**

As I wrote in the Introduction (based on information from Christof's Advisor Answer), when you instantiate a COM server located in a DLL, VFP can only load it into the same thread as the main application. So, while SomeLengthyProcess is running, the VFP IDE is unavailable.

## Enter "helper" utilities

All is not lost, however, as Christof further explains: *Even if you create an MTDLL in VFP, you need an additional helper utility that creates a new thread, initializes the apartment, and loads the COM server*.

There are three such "helper" utilities available that I know of (they were all mentioned by Christof in his Advisor Answer):

- **Remus Rusano's VFPMTAPP.DLL**
  Remus published his DLL and had it posted at http://multithreadedvfp.com for a while. At some point, that address began to redirect to a 404 page on Geocities. I found an archived version of the page (which includes some documentation and a download for his helper utility) at http://www.oocities.org/rremus/multithreaded_vfp.htm

  I have not used Remus' helper to date (primarily because Christof's is so easy to use, hence the title of this session).

- **Calvin Hsia's code from his blog**
  Calvin published a routine on his blog at http://blogs.msdn.com/b/calvin_hsia/archive/2006/05/16/599108.aspx which creates assembler code and writes it into memory at runtime. Amazing? Yes. Unreadable? To me, yes. Feasible? Not really, thanks to DEP (Data Execution Prevention). DEP prevents code execution in data-only memory blocks.

  With DEP turned on, Calvin's code dies a horrific "Microsoft Visual FoxPro has stopped working" error (similar to the window in Figure 4). If your app is using this code and DEP triggers the failure, your app will die the same horrific death.

  DEP notwithstanding, Claude Fox ran with Calvin's code and even created a Codeplex project for it at http://mtmyvfp.codeplex.com/.

  For me though, this should simply not be used in production code. **Ever**.

  This leaves us with one more helper:

- **Christof's DMULT.DLL**
  Christof's DMULT.DLL is a C++ DLL that allows you to instantiate a COM DLL object in another thread. It is also the focus of this session and whitepaper.

  With Christof's permission, I've included DMULT.DLL in the sample code for this session. You can download the complete version (including the DLL source code, slides from a presentation he did several years ago, and more) at his website:

  http://www.foxpert.com/download/DMULT.ZIP

## Getting started with DMULT.DLL

First things first: all code you want to run in a different thread needs to go in a project of its own. Using the sample we looked at earlier, this is already done.

We built the SAMPLE.PJX into a multi-threaded DLL, ran it directly from the VFP IDE, and found it was just not feasible thanks to the unresponsiveness of the IDE while waiting for SomeLengthyProcess to complete.

So, let's create a new PRG file which will use Christof's DMULT.DLL to instantiate the SAMPLE.DLL in a separate thread (the following code is contained in a FUNCTION named MultiThreaded in the BASICSAMPLE.PRG file included with this paper).

```
Declare Long CreateThreadWithObject in DMult.DLL ;
  String lpszClass, ;
  String lpszMethod, ;
  Object oRef,  ;
  Long @lpdwThreadId

Declare CloseHandle in Win32API LONG

LOCAL lnHandle, lnThreadID
lnThreadID = 0

lnHandle = CreateThreadWithObject( ;
                Strconv("sample.EasyMTServer"+Chr(0),5), ;
                Strconv("Main"+Chr(0),5), ;
                _VFP, ;
                @lnThreadID ;
           )
=CloseHandle(m.lnHandle)
```

And that's it! Run the program and you will see the same results as before, but with one major difference: ***the VFP IDE is available immediately after starting this program***!

## How did that work?

DMULT.DLL has two methods available: CreateThreadWithObject(), and CreateThreadWithString(). For purposes of this session, we will be using CreateThreadWithObject() exclusively.

When we called CreateThreadWithObject(), we sent in three parameters:

- The COM server DLL we want to instantiate in a different thread (note the DLL must be registered in order for this to work)
- The method within the COM server DLL we want to run
- A callback object

Note: the server name and the method to run need to be passed in Unicode. Luckily, this is easy to do with the STRCONV() function in VFP:

For the server name, we append a NULL character (CHR(0)) then STRCONV() with a conversion setting parameter of 5 (DBCS to Unicode).

We do the same for the method within the DLL that we want to run.

Lastly, we send in a callback object. As Christof noted in his article, we can only send COM objects to API functions. Luckily, _VFP is a COM object, which makes the preceding demo quite easy to implement.

You should note, however, that you can really send *any* VFP object (forms, textboxes, custom classes, etc.) to CreateThreadWithObject() as a callback. You just need to convert the object to a COM object. Christof shows a neat trick you can use to convert the VFP reference to a COM reference.

Let's say you want the callback to be a textbox on a form. The textbox is named txtMyCallback, and the COM server DLL will write back to the Value property of the textbox (instead of our earlier example of writing to the _VFP IDE window). How can we convert the VFP reference to a COM reference? It could look something like this:

```
loTextBox = ThisForm.txtMyCallback
loCOMRef = _VFP.Eval("loTextBox")
```

After that, we send in loCOMRef as the callback object (the third parameter in CreateThreadWithObject()).

To make this work from within the DLL, we simply change the following line:

```
toCallback.DoCmd("? + ALLTRIM(SYS(2015))")
```

to this:

```
toCallback.Value = ALLTRIM(SYS(2015))
```

Since toCallback is now a textbox on a form, the value is updated once every second for twenty seconds with a SYS(2015) value.

I normally would not use this type of callback functionality, though. I tend to create my own custom callback objects, which act as event handlers for the multi-threaded DLL.

We'll take a deeper look at callback objects in the section titled "Let's try a multi-threaded approach (with event handlers)".

## This is cool stuff, but where's the real-world examples?

I'm glad you asked! Let's take a look at something that *could* take a while to process: HTTP requests.

I've created a sample form, GETBLOGS.SCX, which calls out to my web server to retrieve a list of FoxPro-related blogs. I ~~stole~~ leveraged some data from Ted Roche's PlanetFox site, which is available at http://www.tedroche.com/planetfox/ , and put it in a web page on my server.

The form (Figure 5), contains a button with the caption "Get Blogs from Planet Fox". When you click this button, the form uses Craig Boyd's VFPConnection.fll to call the page on my server. When the response is returned, it parses the response to grab the URL for approximately 60 blogs, inserts the URLs into a cursor, and updates the grid on the form.

*Note: This session does not cover VFPConnection.fll in detail. If you want to learn more about it, I recommend Rick Schummer's "How Craig Boyd Makes Me A Hero!" session.*

Here's the code from the cmdGetBlogs button Click() event:

```
This.Enabled = .F.
ThisForm.cmdCancel.Enabled = .T.

SELECT xBlogs
SET SAFETY OFF
ZAP
SET SAFETY ON

ThisForm.grdBlogs.RecordSource = ""
ThisForm.grdBlogs.RecordSource = "xBlogs"
ThisForm.grdBlogs.Refresh()

LOCAL loDLL As Object
loDLL = CREATEOBJECT("BlogStuff.BlogRetriever")
loDLL.GetBlogs(THISFORM)
```

I've sent THISFORM as a parameter to the GetBlogs function in the BLOGSTUFF.DLL. The GetBlogs function will call THISFORM's ProcessBlogs method, which will parse the result and add records to the xBlogs cursor.

This code actually runs pretty fast, so I added some special code to the Page_Load event on the web page to slow it down a bit. I put the ASP.NET thread on the web server to sleep for 5 seconds, just to make sure it takes a while to return the response to the HTTP request.

While the request is taking place, you can click all over the form (try the Cancel button), or even within the VFP IDE, but until the request is complete, the form (and IDE) is unresponsive.

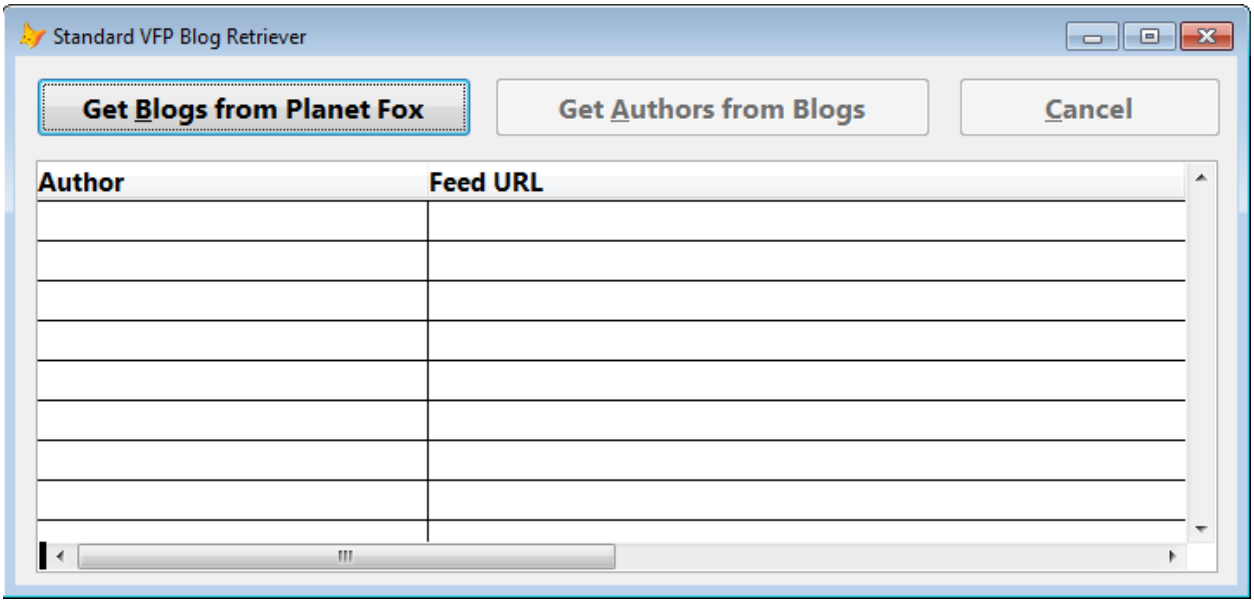


**Figure 5: The GETBLOGS.SCX form**

Once the process has finished, we will have a list of feed URLs in the grid (Figure 6).


**Figure 6: The GETBLOGS.SCX form after retrieving Feed URLs**

Let's click the "Get Authors from Blogs" button. This will walk us through the list of URLs, and for each one we'll again use Craig Boyd's FLL to hit each URL and extract the Title (or Author) information from each.

Good luck with this one. It could take a while to complete. On my system at home, it usually takes between 65 and 90 seconds. That's 65 to 90 seconds of ***non-responsive*** UI, which can really cause users to believe the app just isn't very good.

Here's the code from the cmdGetAuthors button Click() event:

```
LOCAL lnStart, lnEnd
lnStart = SECONDS()

This.Enabled = .F.
ThisForm.cmdCancel.Enabled = .T.

SELECT xBlogs
GO TOP

SET LIBRARY TO vfpconnection.fll

SCAN
  LOCAL lcHTTP As String
  lcHTTP = HTTPToStr(ALLTRIM(xBlogs.cURL))

  lcAuthor = STREXTRACT(lcHTTP,[<author><name>],[</name>],1)
  IF !EMPTY(ALLTRIM(lcAuthor))
    REPLACE cAuthor WITH STRTRAN(ALLTRIM(m.lcAuthor),"&#039;","'")
  ELSE
    lcAuthor = STREXTRACT(lcHTTP,[<title>],[</title>],1)
    IF !EMPTY(lcAuthor)
      REPLACE cAuthor WITH STRTRAN(ALLTRIM(m.lcAuthor),"&#039;","'")
    ELSE
      DELETE
    ENDIF
  ENDIF
  ThisForm.grdBlogs.Refresh()
ENDSCAN

ThisForm.grdBlogs.Refresh()
ThisForm.cmdCancel.Enabled = .F.

lnEnd = SECONDS()
This.Caption = ALLTRIM(TRANSFORM((lnEnd - lnStart),"99.99") + "  seconds")
```

## Let's try a multi-threaded approach (with event handlers)

We've seen a process that can take a very long time to complete, and we've seen the UI completely unusable until the process has completed. Let's "take out the slow parts" by re-working this for DMULT.DLL.

First, we're going to need a COM server DLL. I've created a project named BLOGSTUFF.PJX, which contains a PRG file named BLOGSTUFFMAIN.PRG. This DLL is actually used on the GETBLOGS.SCX form, when you click the "Get Blogs from PlanetFox" button. Here's the code listing for this PRG:

```
DEFINE CLASS BlogRetriever As Session OLEPUBLIC
  PROCEDURE Init
    ** This assumes vfpconnection.fll is
    ** located in your path by the DLL!
    SET LIBRARY TO vfpconnection.fll
  ENDPROC

  PROCEDURE Destroy
    ** Release the fll
    SET LIBRARY TO
  ENDPROC

  PROCEDURE GetBlogs(toCallback)

    ** This function calls the web server and
    ** returns a string to be parsed by the
    ** callback object.

    LOCAL lcHTTP As String, lcList As String
    lcHTTP = HTTPToStr("www.kevinragsdale.net/FoxBlogs.aspx")
    lcList = STREXTRACT(lcHTTP,[<h2>Subscriptions</h2>],[</ul>],1)

    ** toCallback is the form. It contains a method
    ** named ProcessBlogs which will parse this result.
    IF VARTYPE(m.toCallback) == "O"
      TRY
        toCallback.ProcessBlogs(m.lcList)
      CATCH
      ENDTRY
    ENDIF
  ENDPROC

  PROCEDURE GetAuthor(toCallback)
    LOCAL lcHTTP As String, lcURL As String, llContinue As Boolean
    lcHTTP = ""
    lcURL = ""
    llContinue = .T.

    IF VARTYPE(m.toCallback) == "O"
      TRY
        m.lcURL = m.toCallback.cURL
      CATCH
        m.llContinue = .F.
      ENDTRY
```

```
    ELSE
      m.llContinue = .F.
    ENDIF

    IF m.llContinue
      m.lcHTTP = HTTPToStr(m.lcURL)
    ENDIF

    ** toCallback is a custom object with a
    ** method named ProcessResult which will
    ** parse this result.
    IF VARTYPE(m.toCallback) == "O"
      TRY
        m.toCallback.ProcessResult(m.lcHTTP)
      CATCH
      ENDTRY
    ENDIF
  ENDPROC
ENDDEFINE
```

This code assumes that VFPConnection.fll exists and can be located by the DLL. This DLL consists simply of one object with two methods. Both methods do basically one thing: they each make an HTTP request then send back the result to the callback object that is sent in as a parameter.

Now, let's go back to the form. This time we'll use GETBLOGS_MT.SCX. It looks the same, in fact it is the same as the previous form, except we've moved a couple of bits of code around.

In the cmdGetBlogs button Click() event, we've removed everything past the "zap the cursor and set the grid's RecordSource" part and replaced it with this:

```
Declare Long CreateThreadWithObject in DMult.DLL ;
  String lpszClass, ;
  String lpszMethod, ;
  Object oRef,  ;
  Long @lpdwThreadId

Declare CloseHandle in Win32API LONG

Local lnHandle, lnThreadID, loCallback
lnThreadID = 0
loCallback = THISFORM

lnHandle = CreateThreadWithObject( ;
                Strconv("BlogStuff.BlogRetriever"+Chr(0),5), ;
                Strconv("GetBlogs"+Chr(0),5), ;
                _VFP.Eval("loCallback"), ;
                @lnThreadID ;
          )

=CloseHandle(m.lnHandle)
```

That looks easy, doesn't it? Note the loCallback object: we've set it to THISFORM, and we're converting the VFP reference of the form to a COM reference using Christof's _VFP.Eval("loCallBack") trick.

One thing you'll notice when you click the "Get Blogs from Planet Fox" button is control is returned almost immediately back to you. You can click on the form (try the Cancel button), or even in the VFP IDE (Command Window, menu, etc.) - they're all responsive!

In the CreateThreadWithObject() above, we're telling DMULT.DLL to create a new thread for the BlogStuff.dll (with the BlogRetriever object), run the GetBlogs method, and use THISFORM as the callback.

The GetBlogs method hits my website (again using Craig Boyd's FLL), extracts some info from it, then does this:

```
toCallback.ProcessBlogs(m.lcList)
```

Since toCallback is THISFORM, we need a ProcessBlogs method on the form to handle the response from GetBlogs. Hence, ProcessBlogs becomes the event handler. Here's the code from the forms ProcessBlogs method:

```
LPARAMETERS tcHTMLString As String
  LOCAL lcTable As String
  lcTable = tcHTMLString

  lnRecords = OCCURS([<li>],lcTable)

  ** For each <li>, check for <a>'s
  IF lnRecords > 0
    FOR i = 1 TO lnRecords
      lcLI = STREXTRACT(lcTable,[<li>],[</li>],i)
      lcURL = STREXTRACT(lcLI,[<a href="],["],1)

      IF !EMPTY(ALLTRIM(m.lcURL))
        INSERT INTO xBlogs (cAuthor,cURL) VALUES ("",ALLTRIM(m.lcURL))
      ENDIF
      ThisForm.grdBlogs.Refresh()
    ENDFOR
  ENDIF

  ThisForm.cmdCancel.Enabled = .F.
  ThisForm.cmdGetAuthors.Enabled = .T.

  This.cmdGetBlogs.Caption = "Found " + ;
                             ALLT(TRANSFORM(RECC("xBlogs"),"99")) + ;
                             " Blogs"
```

This is the same code as in the previous form example.

Now we're ready for the fun part. Let's click the "Get Authors from Blogs" button. Remember the previous example, when it took over a minute to complete?

Check.

This.

Out.

Pretty fast, eh? Did you notice the UI (including the VFP IDE) was totally responsive? The whole process takes about 5 to 6 seconds on my system at home, which is a tremendous improvement over the 60+ seconds from the previous form.

The code in the cmdGetAuthors button Click() event has changed in this form versus what was in the previous form. Here's the new code:

```
This.Enabled = .F.
ThisForm.cmdCancel.Enabled = .T.

SELECT xBlogs
GO TOP

LOCAL lnHandle, lnThreadID, loCallback

SCAN
  SCATTER MEMVAR
  lnThreadID = 0

  loCallback = NEWOBJECT("AuthorsHandler","bloghandler.prg","",m.cURL)
  lnHandle = CreateThreadWithObject( ;
                    Strconv("BlogStuff.BlogRetriever"+Chr(0),5), ;
                    Strconv("GetAuthor"+Chr(0),5), ;
                    _VFP.Eval("loCallback"), ;
                    @lnThreadID ;
                )

  =CloseHandle(m.lnHandle)
ENDSCAN

ThisForm.cmdCancel.Enabled = .F.
```

Notice the new callback this time? loCallback is an object named AuthorsHandler which is located in the BLOGHANDLER.PRG file:

```
DEFINE CLASS AuthorsHandler AS CUSTOM
  cURL = ""

  PROCEDURE Init (tcURL As String)
    This.cURL = ALLTRIM(tcURL)
  ENDPROC

  PROCEDURE ProcessResult (tcString AS String)
    LOCAL lcString As String
    lcString = tcString

    lcAuthor = STREXTRACT(lcString,[<author><name>],[</name>],1)
    IF !EMPTY(ALLTRIM(lcAuthor))
```

```
      lcAuthor = STRTRAN(ALLTRIM(m.lcAuthor),"&#039;","'")
    ELSE
      lcAuthor = STREXTRACT(lcString,[<title>],[</title>],1)
      IF !EMPTY(lcAuthor)
        lcAuthor = STRTRAN(ALLTRIM(m.lcAuthor),"&#039;","'")
      ENDIF
    ENDIF

    IF USED("xBlogs")
      IF !EMPTY(lcAuthor)
        UPDATE xBlogs SET cAuthor = ALLTRIM(m.lcAuthor) ;
          WHERE ALLTRIM(cURL) = ALLTRIM(This.cURL)
      ELSE
        UPDATE xBlogs SET cAuthor = "Unknown" ;
          WHERE ALLTRIM(cURL) = ALLTRIM(This.cURL)
      ENDIF
    ENDIF

    ** We're done with processing the result, so release this
    This.Release()
  ENDPROC

  PROCEDURE Release
    RELEASE THIS
  ENDPROC
ENDDEFINE
```

We've taken the "processing" code out of the Click() event (from the previous form) and moved it to this objects' ProcessResult method.

So, for each record in the xBlogs cursor, we create a new object (AuthorHandler), and call CreateThreadWithObject() in DMULT.DLL. In this case, approximately 60 *almost simultaneous* requests/threads (as fast as VFP can run through the SCAN loop, literally).

If you watch Windows TaskManager when you first click the Get Authors button, you'll see the thread count skyrocket, and (almost as quickly) come back down to the count it was at before you clicked the button.

Let's go back to the GetAuthors method in BLOGSTUFF.DLL:

```
PROCEDURE GetAuthor(toCallback)
  LOCAL lcHTTP As String, lcURL As String, llContinue As Boolean
  lcHTTP = ""
  lcURL = ""
  llContinue = .T.

  IF VARTYPE(m.toCallback) == "O"
    TRY
      m.lcURL = m.toCallback.cURL
    CATCH
      m.llContinue = .F.
    ENDTRY
  ELSE
```

```
    m.llContinue = .F.
  ENDIF

  IF m.llContinue
    m.lcHTTP = HTTPToStr(m.lcURL)
  ENDIF

  ** toCallback is a custom object with a
  ** method named ProcessResult which will
  ** parse this result.
  IF VARTYPE(m.toCallback) == "O"
    TRY
      m.toCallback.ProcessResult(m.lcHTTP)
    CATCH
    ENDTRY
  ENDIF
ENDPROC
```

It gets the AuthorsHandler callback object as a parameter, then refers back to that object to retrieve the URL it needs to hit with the VFPConnection.fll. After getting the HTTP response, it calls the ProcessResult method of the AuthorHandler, which extracts the necessary data and updates the xBlogs grid.

And, while all of this is going on for one single URL, it is actually happening on all 60+ URLs at *virtually the same time*. And **that** is the power of multi-threading.

Other examples during this session include 100+ almost simultaneous requests to one web server, and unzipping a very large file using Craig Boyd's VFPCompression.fll. The source code from these examples are not included with the session materials.

## Holy cow, this is awesome! But, this looks like some tight-coupling is involved…

You are correct. The examples I've shown in the demos are tightly-coupled. I'm sure many of you will find ways to properly manage threads and provide more generic functionality. But, in an effort to keep it simple/easy, I have taken a tightly-coupled approach – not only for this session, but in my own apps.

## Are there any disadvantages to multi-threading?

Yes, there are some disadvantages. But in my experience, the advantages have far outweighed the disadvantages. There are some things you should be mindful of when working with multi-threading (everything in this section that is in *italics* is from Christof's Advisor answer – the rest are comments from me):

*As a VFP developer, you're used to variables, global objects, work areas, data sessions, etc. Here, multi-threading is kind of disappointing. Every thread is a new instance of the VFP runtime with its own set of variables, work areas, etc. You can't create a cursor in one thread and access it directly in another thread without converting it to some other format (XML, string, file), passing it to the other thread, and converting it back.*

I've done a bit of this in my own apps where the DLL will end up with a cursor that I need to pass back to the app. A simple CURSORTOXML() in the DLL, followed by a XMLTOCURSOR() in the event handler in my app takes care of this very quickly.

*The same applies to object references. The only object the other thread sees is the object you passed to it when you created the thread.*

I had some trouble with this early on, but found that if my event handler objects contain properties which are object references (i.e. my "application object"), I get access to all of that from within the new thread, too.

*Multi-threaded applications provide some challenges. For one, the MTDLL runtime is a different one than the EXE runtime (there are subtle differences).*

I've never ran into an issue with this – at least as far as I know. Most of the differences in the runtimes that I know of have to do with display/UI issues.

*It is also not as easy to debug an MTDLL.*

Can I get an "AMEN!" to this one? This is the biggest hurdle I faced when I started playing around with multi-threading. It really can be hard to debug an MTDLL. My favorite debugging method is a truckload of STRTOFILE() statements in the DLL code. Not exactly a best practices approach, but it works for me.

*A fast multi-threaded application requires significantly more thought up front regarding the architecture and the communication flow.*

For me, this was a tough nut to crack. Time? More time? Upfront? Thinking? Personally, I'd rather be coding than thinking any day. It does take time to think some of this stuff through, but believe me - your users, boss, and co-workers will be pleased with the result!

## Summary

In this session, we took a look at Christof's DMULT.DLL, and seen the positive effects multi-threading can have on our apps.

With DMULT.DLL, multi-threading in Visual FoxPro is not only *possible* – it's **easy**!