Document :: Threads
Author :: Jon Jenkinson
From "The Bits" Website
http://www.cbuilder.dthomas.co.uk
emails to : jon.jenkinson@mcmail.com

Threads

Threads are an integral part of Windows95, and as such should be used to improve the performance of any processor intensive or timer critical operations. Within this tutorial I will cover the basics of how to create and use threads, including their use with semaphores and the special function *Synchronize* which enables C++ Builder threads to interface with the VCL.

Colour Coding:

Information displayed in this colour is information provided by my copy of BCB.
Information displayed in this colour is information you need to type.
Information displayed in this colour is information which is of general interest.

****

Underlying Principals

The only underlying principals you require to know are Pre-Emptive Multi-Tasking, and semaphores. The first is part of the Windows95 Operating system, and as such your only knowledge is to know why it's there and how it affects your applications. Semaphores are far more important when it comes to using threads, so I'll cover that subject in more depth.

Pre-Emptive Multi-Tasking

This was the real advance in the Microsoft Windows mainstream product. Windows3 variants used what was effectively automatic application switching. It worked on the principal that any application would be well behaved enough to release resources to the system kernal so that a.n.other application could have a slice of the action. In theory, this meant that when a document was printing in the background you could get on with something else in the foreground. Unfortunatly, the words to note here are, *well behaved*. Anyone who remebers the days of Windows3 will be aware that the simplest of task could tie the machine up, and if one application fell over, the machine fell over.

Pre-Emptive Tasking has existed in the mainframe world since, well, time began. This was for two reasons, these systems were multi-user, that is one application hogging the mainframe would lock all other users out, and, mainframe programs were extemely processor, and thus, resource hungry. What Pre-Emptive Tasking actually does is remove the applications control of a machine and hand it to the operating system.

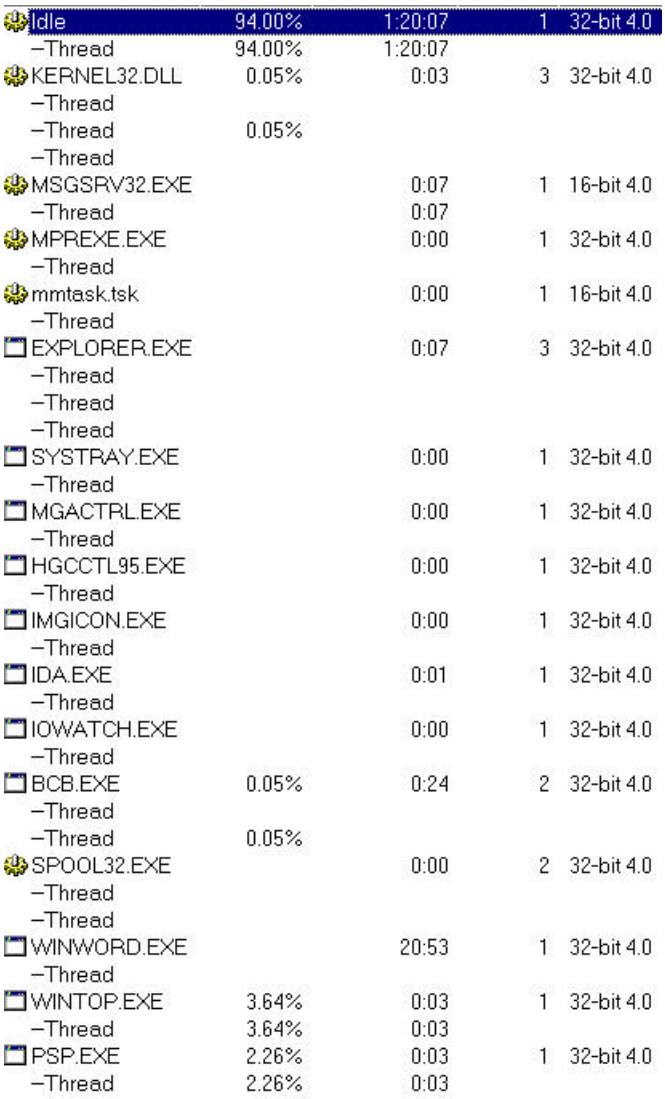
How does this affect yourselves ? Well, in the past, if you thought your application was important, and don't we all, you would keep control and not let Windows anywhere near the resources. Now, whilst you can still override certain aspects of Win95 system control, you are effectively at the control of the Win95 task management. That is to say, if you fire up a processor resource task, you will not get 100% of the processor. What you will get, is a time slice.

A time slice is just what it says, a slice of the processor's time. If there are four applications running, you would see the following approximation,

Program	% of processor time
operating system	20%
Application1	20%
Application2	20%
Application3	20%
Application4	20%

The above is an indication of what would happen, in truth, there is more preference given to certain tasks. For now, we can assume that each of the five applications above have one fifth of the available time slices each. Expresses another way, they have one *thread* in five. This is the first leap of imagination you have to make if you've never heard of threads before. Those used to the old style of programming are used to seeing an application as a single entity. This is no longer true.

Now you have this concept of threads, and above we have five threads, one per application. Consider the following, a truer picture of what is happening when we run a 32-bit application under Win95,



Idle	94.00%	1:20:07	1	32-bit 4.0
–Thread	94.00%	1:20:07		
KERNEL32.DLL	0.05%	0:03	3	32-bit 4.0
–Thread				
–Thread	0.05%			
–Thread				
MSGSRV32.EXE		0:07	1	16-bit 4.0
–Thread		0:07		
MPREXE.EXE		0:00	1	32-bit 4.0
–Thread				
mmtask.tsk		0:00	1	16-bit 4.0
–Thread				
EXPLORER.EXE		0:07	3	32-bit 4.0
–Thread				
–Thread				
–Thread				
SYSTRAY.EXE		0:00	1	32-bit 4.0
–Thread				
MGACTRL.EXE		0:00	1	32-bit 4.0
–Thread				
HGCCTL95.EXE		0:00	1	32-bit 4.0
–Thread				
IMGICON.EXE		0:00	1	32-bit 4.0
–Thread				
IDA.EXE		0:01	1	32-bit 4.0
–Thread				
IOWATCH.EXE		0:00	1	32-bit 4.0
–Thread				
BCB.EXE	0.05%	0:24	2	32-bit 4.0
–Thread				
–Thread	0.05%			
SPOOL32.EXE		0:00	2	32-bit 4.0
–Thread				
–Thread				
WINWORD.EXE		20:53	1	32-bit 4.0
–Thread				
WINTOP.EXE	3.64%	0:03	1	32-bit 4.0
–Thread	3.64%	0:03		
PSP.EXE	2.26%	0:03	1	32-bit 4.0
–Thread	2.26%	0:03		

This is a screenshot from a program called, WinTop, part of the Microsoft PowerToys set up. What it shows are the programs running on my PC as I type this tutorial.

The first thing to notice is that every application has at least one thread, it can have no less than one. The second is that 16-bit applications usually have one thread only, *May not be true in all cases*. However, four of our 32-bit applications have more than one thread. Back to my original table, this time with threaded applications,

Program	% of processor time	#of threads	16/32 bit
operating system	33.3%	5	32 bit
Application1	6.7%	1	16 bit
Application2	6.7%	1	32 bit
Application3	6.7%	1	16 bit
Application4	46.6%	7	32 bit

Here we have 15 threads, and as such our processor is sliced into 15 sections. You can see that through a use of threads, Application4 has the most processor time, and as such will appear quicker to the user than Application2, a program which does exactly the same.

In reality, Microsoft recommends a limit on the number of threads an application uses, 16. If applications use threads without control, you can reach the stage where a single application appears to hang the system, when all it has done is launch so many threads that the system is time slicing an awful lot before a.n.other application gets a chance. It is rare, for instance, for an over-threaded program to control the system to the extent that the <ctrl-alt-del> sequence wont bring up the Win95 task manager.

Okay, you can now see the benefits of a logical use of threads, when printing for instance, fire a thread to manage the print whilst your user returns to a more important task. In fact, anything that doesn't require a user response should be sent off into a thread of its own, but be careful, it is easy to end up with more threads than you really need.

Synchronisation

A very important concept when it comes to threads. Consider the paragraph above, your user has just sent something directly to a resource, and then, because they're users, they send something else to the same resource. Now, although Win95 handles printing for us, we will use the idea of a printer to prove the point.

You have two very small test documents,

"Hello I'm Document One, and I'm pleased to meet you!"
and
"Hello I'm The other Document and I hate your guts!"

In our application we have a button which allows the user to send all documents to the printer. We, being clever, have elected to launch a thread to print each document, Doc1.PrintThread, and Doc2.PrintThread. However, as we didn't use semaphores when we wrote our threads, we end up with the following output on the printer,
"Hello I'm The Hello Document Hello other One Document, and I'm pleased to and I hate your meet guts! You!" or worse?

What has happened here is that both threads have sent to the printer when they can, not allowing the printer to finish printing document one before the second starts. Again, as Win95 handles printers very well, and printers themselves do a reasonable job of telling us what's happening, you can probably control it in an inelegant way by responding to error messages.

Threads provide a better way by design. They have what is called a semaphore, at least in theory. Within C++ Builder, I have yet to find a property to match the semaphore, so below I describe their purpose, and later in the tutorial we build in a semaphore, out of a desire to achieve good design principals:).

A semaphore is nothing more than a flag. Used properly, it will provide you with built in error handling for your threads. Consider the following

No Semaphore

a)launch thread1	*(thread1 running)*
a)thread1.Action	(*do something)*
b)launch thread1	*(thread1 running twice)*
b)thread1.Action	**Collision**BAD**
!!!problems	

With Semaphore

a)launch thread1	*(thread1 running)*

a)check semaphore *(clear to continue)*
a)Grab semaphore *(we're in control)*
a)thread1.Action *(do something)*
b)launch thread1 *(thread1 running twice)*
b)check semaphore *(In Use, wait)*
a)thread1.finish *(we've finished)*
a)free semaphore *(we're no longer in control)*
b)Grab semaphore *(now we're in control)*
c)launch thread1 *(and try again)*
b)thread1.Action
c)check semaphore *(In Use, wait)*
b)thread1.finish
b)free semaphore
c)thread1.Action
c)thread1.finish
c)free semaphore.

As you can see, with the semaphore in place, we control completely what happens to our thread. You must remember, in a threaded application it is your responsability to take care of multiple occurances of your thread, just as it is your responsability to take care of multiple occurances of your application in normal programming terms.

Note : Normally one would handle threads with a semaphore in the above manner, allowing the second thread to pause until the semaphore is released, usually with a while, do/while, or unbreakable for. However, with C++ Builder and Win95, throwing a loop in this manner causes the thread containing the loop to grab processor resources, effectivly causing your application to deadlock, though the rest of Win95 functions normally. To avoid this, work with semaphores in the manner detailed here, that is, if you wish to limit the number of thread instances, use a semaphore and numeric semaphore, read on for a bit of lateral thinking.

And before we code:)

And finally, all threads are created dynamically at runtime. Whilst you can build them into class definitions, you must create the actual instance of your thread dynamically within the contstructor.

How to do Learn About Threads

Follow the steps below, which are self explanatory.

Step One : Create an application

Firstly, we'll create a simple application which will allow us to see our threads in action.

(a) Create a new application using File|New Application.
(b) Place the following components on the form, naming them as follows
    (1) Three TLabels, Label1, Label2, Label3
    (2) A TButton, Button1
(c) Run the application

Step Two : Create Thread Handling Semaphore

Before we actually place the variable which will act as our semaphore, a couple of things to be aware of. The semaphore is a flag, and as such we will use a boolean variable, that is, true =

thread in use, false = thread not in use. The location of the semaphore requires thinking about. If you place the semaphore within the thread itself, you will create a new semaphore *each* time the thread is launched. Thus, if you have multiple occurances of the thread, you will have multiple instances of the semaphore, unwise, and very difficult to synchronize.

For this reason we create our semaphore in the main application thread, or within our Form class. This way, our semaphore will be available to all subsequent forms, classes, and most importantly, to all threads, and all thread instances.

It is not important if your application can have multiple instances of the main form. Consider the following,

Application1	launches Mainform1A	which launches Thread1A
		which launches Thread2A
	launches Mainform1B	which launches Thread1B
		which launches Thread2B

Here, each instance of Mainform1 can launch and control instances of Thread1 and Thread2. This means that any action that Thread1 or Thread2 perform on Mainform1, will be limited to their own parent, either A or B. The only time a thread has to be aware of anything in the above scenario is if one of the threads actions on Application1, in which case your semaphore would have to be within Application1.

If you're playing with actions directly on the application level and not on the form level, you will understand the above. For most of us however, we will only ever have a single occurance of our Mainform, so placing the semaphore here makes sense.

Open up unit1.cpp's header and locate the TForm class definition, Add the code in red,

```
class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TButton *Button1;
private:       // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
        bool UsageFlag;
        int InstancesOfCounterThread;
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
```

All we have created here is a flag and a counter. The flag is a semaphore, and the counter will ultimatly become our workaround to allow us to control how many instances of our thread, CounterThread, occur.

Step Three : Adding Our First Thread

Here we will go overboard, and add a thread purely to update the display of Label2 and Label3. These displays are for the purpose of showing us how the flag and counter we have implemented above behave, and as such are for information only. The reason for putting these updates in a thread of their own is that we can have it running continuously, in other words being constantly updated without waiting for anything else in our application, therefore, we always know that the values displayed are as upto date as possible.

(a) Create A Thread,

(1) File|New, brings up the “New Items” dialog.
(2) Select, ThreadObject by double clicking on the icon
(3) Give the thread the Class Name “UpdateDisplayThread” and click ok.

You now have a thread, though it wont actually do anything. Take a moment to read the information created for you, paying specific note to the section describing the Synchronize function, more of which in a moment.

Currently the only two functions of our thread are the constructor, and an Execute() function. Do not place any code in the constructor, it seems to create havoc:) Treat the Execute() function as your constructor, in essence this function is a main execution point in itself.

```
void __fastcall UpdateDisplayThread::Execute()
{
        //---- Place thread code here ----
        do  //unbreakable from here,  only the app will set it to false on closing
        {
                Synchronize(UpdateCaptions);
        }while(Terminated == false);

        FreeOnTerminate = true; //not really neccasary as app will free
        Terminate();//not really neccasary as app will free
}
//---------------------------------------------------------------------------
```

The do while loop here is to ensure that the thread only terminates when given explicit permission by the application, or when called by you directly, as we will do in our next thread. For now, this is a safe way of ensuring that the calling application retains control of this thread.

Place the red code in the Execute() function, and I’ll explain:) The Synchronize function of a thread is C++ Builders way of ensuring that the thread updates as appropriate. A thread on its own can operate independantly of an application, once fired, and work directly with the Win95 system. However, when it wishes to interact with the rest of your application, it requires *synchronising*, hence the Synchronize function.

What this does is take the function call, *UpdateCaptions*, and wait until the referenced part of the application is ready for the actions within the called function. Now create the following function,

```
void __fastcall UpdateDisplayThread::UpdateCaptions()
{
        switch(Form1->UsageFlag)
        {
                case true:
                        Form1->Label2->Caption = “UsageFlag is true”;
                        break;
                case false:
                        Form1->Label2->Caption = “UsageFlag is false”;
                        break;
                default:
                        Application->MessageBox(“You’ve achieved the impossible”,
                                                “Called from within Thread2”,
                                                MB_OK|MB_ICONERROR);
                        break;
        }//end switch

        Form1->Label3->Caption = “NumberWaiting = ” +
          (String)Form1->InstancesOfCounterThread;
}
```

Next place the following into the header file,

```
class UpdateDisplayThread : public TThread
{
```

```
private:
protected:
        void __fastcall Execute();
public:
        __fastcall UpdateDisplayThread(bool CreateSuspended);
        void __fastcall UpdateDisplayThread::UpdateCaptions();
};
```

The function is declared as a fastcall, because the Synchronize function works with the VCL directly, and anything that works with the VCL should be called in this manner.

As our thread interacts with Form1, we must tell it what Form1 is, so include the header "Unit1.h" at the top of Unit2.cpp,

```
#include "Unit2.h"
#include "Unit1.h"
```

Now our thread has something to do, let's place the code to create it and fire it.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
        UpdateDisplayThread *NewUpdateDisplayThread;

        NewUpdateDisplayThread = new UpdateDisplayThread(true);

        NewUpdateDisplayThread->Resume();
}
```

What we have here is actually a lot more simple than it looks. We firstly create a pointer to hold our thread's instance. Next we dynamically lauinch our thread, setting its suspend property to true. What this actually does is start the thread running, and then suspend it until we are ready to start it, which we do next by calling the threads Resume() function. This is just good practice, it allows threads to be fired in a particular order, but be suspended imediatly if we require to set other variable, or threads, before we continue.

Finally, we set the UsageFlag variable from within the Form's constructor, and include Unit2.h in Unit1.cpp

```
#include "Unit1.h"
#include "Unit2.h"


__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
        UsageFlag = false;
}
```

Okay, run the program, and if you have a program which shows threads such as WinTop, you will see that you now have two threads running under project one.

Note: Here we have not implemented a semaphore in this thread, we are relying on the fact that this thread is linked to Form1, and not the application. As we create the thread within the form's OnCreate function, we can reasonably assume that the thread can only be created in the ratio, one thread to one form1. Thus, we should not be able to run the thread a second time, but beware of doing this, I have done it this way for simplicity, you should handle your own threads in the manner described below, unless you are *Certain* that it cannot run twice within the same scope.

Step Four : Adding a thread which does something

Now we will add a thread which actually does something, nothing much, just runs a counter,

but the major thing we will do is control the way that the thread gains access to the counting part of itself, and thus the display update.

Firstly, we will create the thread, and run it to cause problems, so you know what can go wrong if we're not careful.

(a) Create a new thread,
    (1) Select File|New, selecting New..Thread Object from the dialog,
    (2) Call it CounterLoopThread.
    (3) Select ok.

You should now have a unit3.cpp, which will be our third thread, and all we'll do now is create the functionality we want within it. We will not make use of the semaphore yet, we will enable ourselves to create problems:)

Firstly, lets include Unit1's header in Unit3.cpp

```
#include "Unit3.h"
#include "Unit1.h"
```

Now place the following code in unit3.cpp

```
void __fastcall CounterLoopThread::Execute()
{
        //---- Place thread code here ----
        for(int x=0; x < 10000; x++)
        {
                Counter = x;
                Synchronize(UpdateCaption);
                if(Terminated == true)
                {//terminate requested
                        x = 200000;//ensure loop is broken
                }
        }
        FreeOnTerminate = true;
        Terminate();
}
```

Right, we have now created a threaad which actually does something. All this does is count within a for loop, and send a *Synchronized* call to update the caption of Form1, which we'll declare in a moment. The first thing we must do is pay attention to the if statement marked by the line, //terminate requested. This request is thread independant, that is to say, each instance of any thread, as with any instance of any class, has its own terminate functions. Later we will learn how to set the terminate setting of a thread, but for now understand that unless you call it explicitly, the only other two things which can ask your thread to terminate are the operating system, or the calling thread, in our case form1. *Either way, your thread must ensure that it handles the termination properly, it is your responsibility.*

There is one other thing to note about a thread termination. If the operating system elects to terminate your thread, that's it, it will be shutdown. However, when your thread is asked to terminate by the calling thread, and you don't provide a method for your thread to terminate, the main application will not close, it will simply wait for the thread to finish, which may not happen at all. Consider,

In our first thread, *UpdateDisplayThread*, we have a do-while loop which is unbreakable until a terminate request is sent from the calling thread. If we had a simple unbreakable loop however, and ignored the termination request, we would be unable to close our program, the calling thread would simply be waiting for our thread to terminate, which would be never. Thus, we have this simple boolean flag, Terminated, which allows the main application or ourselves to terminate the thread as needed.

Note how the control of termination is purely at the mercy of your own thread however. Here, we simply step outside our for loop by forcing the value, but we could easily write data out to files etc., if that's what is required.

Okay, our loop increments the integer Counter whilst the integer x is below the value 10000. We break the loop if the Terminated flag is set to true, by setting the value of x to 200000. And we also call the function below, to update the display of Form1.

```
//---------------------------------------------------------------------------
void __fastcall CounterLoopThread::UpdateCaption()
{
        char *TestStr;

        TestStr = new char[255];

        sprintf(TestStr, "Updated in a thread -- %ld %%", (int)(Counter/100));
        Form1->Caption = TestStr;
        sprintf(TestStr, "Last Update by thread  -- %ld", ThreadID);
        Form1->Label1->Caption = TestStr;
        Form1->Update();
}
```

All this does is update the Caption of Form1, and Label1->Caption. ThreadID is a property of each thread instance, and we call it for display purposes only.

Next we update the header files, place the following in Unit3.h

```
class CounterLoopThread : public TThread
{
private:
protected:
        void __fastcall Execute();
public:
        __fastcall CounterLoopThread(bool CreateSuspended);
        int Counter;
        void __fastcall CounterLoopThread::UpdateCaption();
};
//---------------------------------------------------------------------------
```

Finally, we must dynamically call the thread from within Form1. We do this from within the OnClick event handler of Button1, simulating a user, and users always do things there not supposed to.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        CounterLoopThread *CountingLoopInstance;//Test MultiThreaded Item

        CountingLoopInstance = new CounterLoopThread(true);

        CountingLoopInstance->Resume();
}
```

As before, we create a thread and suspend it, and then allow it to resume.

Finally, add Unit3.h to Unit1.cpp thus

```
#include "Unit1.h"
#include "Unit2.h"
#include "Unit3.h"
```

Okay, run the application, and then press the button, waiting for the thread to finish counting **before** pressing the button again.

If you have WinTop or the equivalent, check and you'll see three threads running, when the counter finishes, the third thread will close.

Now let's cause a problem. This will not cause the program to crash, and be aware, if we were not updating the display, you would be blissfully unaware of the conflict, until something didn't work. Start the counter running using Button1. Check WinTop, 3 threads. Click the button again, WinTop, 4 threads, Click the button again, WinTop 5 threads, and so on.

If you look at the display, you will see flashing thread numbers and form caption. We are now running 5 threads, our main thread, our flag display thread, and 3 instances of our counter. Each counter is working perfectly, but as you can see the results are not really what we wanted, and because of the way C++ Builder *Synchronizes*, the counters are going very slowly. This is because we are updating the form's display, and as such, each instance of our counter thread has to wait for the previous instance to update before it gains access to the display itself. And this is slow*ish*.

You can either wait for the counters to finish, another minute or so, (*I've typed the previous three paragraphs of the document, and the first counter is still only on 96%)*, or you can close the application using ctrl-F2 from within Builder itself.

Once you're application has finished, return to the IDE, and we will now use a semaphore to provide access to our thread's action, and gain control over what's happening.

Firstly, add the following code to the Execute funtion in Unit3.

```
void __fastcall CounterLoopThread::Execute()
{
//---- Place thread code here ----
  Synchronize(CheckUsage);
  if(Terminated == false)
  {
    for(int x=0; x < 10000; x++)
    {
        Counter = x;
        Synchronize(UpdateCaption);
        if(Terminated == true)
        {//terminate requested
           x = 200000;//ensure loop is broken
        }
    }
  }

  if(Terminated == false)
  {
    Application->MessageBox("We are here too",
      "From the one that's finished counting",MB_OK);
    Synchronize(FreeFlag);
  }
  else
  {
    Application->MessageBox("We are here","From One that can't Count",
      MB_OK|MB_TASKMODAL);
  }

  FreeOnTerminate = true;
  Terminate();
}
```

Now we have taken a large leap of code here to allow us to handle the semaphore, yes real programming is still required with C++ Builder. The first thing we now ask our thread to do is check the usage flag, done with the synchonised call to the CheckUsage function, defined below.

```
void __fastcall CounterLoopThread::CheckUsage()
{
  switch(Form1->UsageFlag)
  {
        case true:     //there is a thread instance already running
               Application->MessageBox("This thread is already running","From Thread
               Instance", MB_OK);
               Terminate();
               break;
        case false:    //no other thread has the semaphore:)
               Form1->UsageFlag = true;             //Now we have the semaphore
               break;
        default:              //We really need to know,  this would be bad
              Application->MessageBox("Terminal Error. Closing Error",
                 "From thread",MB_OK|MB_ICONERROR);
              Form1->Close(); //Force the application to close
              break;
  }
}
```

and declare the function in the thread class,

```
class CounterLoopThread : public TThread
{
private:
protected:
        void __fastcall Execute();
public:
        __fastcall CounterLoopThread(bool CreateSuspended);
        int Counter;
        void __fastcall CounterLoopThread::UpdateCaption();
        void __fastcall CounterLoopThread::CheckUsage();
};
```

Now then, if we look at the CheckUsage function, it shows how we will handle multiple instances of our thread. We firstly have a look at the UsageFlag on Form1. If it is free, (*false*), it means no other instance of our thread, *or for that matter, anything else*, has taken control of our target resource. We then grab the semaphore by setting it to true, and return to our main Execute() function.

If however, the semaphore is taken, we acknowledge the fact, and then set the Terminated flag of our instance with a call to the Terminate function. *(all this does is set the Flag for **our instance of the thread**, and our instance only.)*

Now when return to the Execute() function, we find that that the Terminated flag is true, and not false, so we will not enter the loop which does the counting. At the end of the Execute function we have a little bit of sillyness, so you can free the semaphore. Note, you need to ensure again, that the instance which set the flag releases it, or no other thread will run. We do this with a simple call to the FreeFlag function declared below.

```
void __fastcall CounterLoopThread::FreeFlag()
{
        Form1->UsageFlag = false;//free the flag
        Form1->Caption = "No Threads Running";
}
```

and again add this function to the header file,

```
class CounterLoopThread : public TThread
{
private:
protected:
        void __fastcall Execute();
public:
        __fastcall CounterLoopThread(bool CreateSuspended);
        int Counter;
        void __fastcall CounterLoopThread::UpdateCaption();
```

```
        void __fastcall CounterLoopThread::CheckUsage();
        void __fastcall CounterLoopThread::FreeFlag();
};
```

Now run the application, and set a thread running by hitting the button. Once the counter starts, hit the button again, and you get a message box. Close the message box and you get another, from the end of the second instance's Execute() function, saying, "we are here". If you wait for the counter to finish, the first instance will throw a message saying "we are here too", and both thread instances are still running, as can be checked from WinTop.

Accept both boxes and confirm to yourself that you understand how the thread instances are interacting.

Step Five : Automating our thread Handling.

Now you'll agree, that whilst we can now control the thread instances, it isn't really ideal. What we are now saying to our user is that they've tried to fire a thread, and tuff, it's busy, where it would be better if we accept that they wish to fire the action as many times as they like, and we just have to control how it's done.

To prove to yourself that I haven't found an easier way, we will use the first method which comes into my head, a while flag in use loop. ***Be warned, this will lock your application, and require closing from the IDE. It may even crash windows. Only do the following if you do not trust my word. If you do, jump to step six below!!!!!!!!!***

Change our CheckFlag function so it matches the following,

```
void __fastcall CounterLoopThread::CheckUsage()
{
  switch(Form1->UsageFlag)
  {
    case true: //there is a thread instance already running
      while(Form1->UsageFlag == true)
      {
       Label3->Caption = "waiting..waiting";
       Update();
      }
      Form1->UsageFlag = true;
      break;
    case false:        //no other thread has the semaphore:)
      Form1->UsageFlag = true;              //Now we have the semaphore
      break;
    default:           //We really need to know,  this would be bad
      Application->MessageBox("Terminal Error. Closing Error","
        From thread",MB_OK|MB_ICONERROR);
      Form1->Close();  //Force the application to close
      break;
  }
}
```

Now the theory here appears correct, we are effectivley waiting for the flag to free before we grab it. Maybe I've missed something in the way C++ Builder handles threads, but I can't find a way around what happens next.

Run the application, and set a thread running using the button. Works as before. Now hit the button again, and watch in horror as your application freezes. If you have WinTop or similar, look at the processor usage, you will see that you're application hasn't in fact frozen, it's just got so tight into the while loop that it has taken over your application. If you wait long enough, about a week:), it may get back to normal and continue, but it isn't really what was intended. ***Do not use a while, for or do loop within any call that interacts with another VCL component, the thread will die if you do:(***

Now switch to the IDE, and press ctrl-F2 to terminate the application.

You'll notice that we may not have locked windows, or any other application you may have had running, just our own application, *but then again*!!!! This suggests that the way that the threads interact within the application is at fault, not Win95 itself. Until someone tells me what I've missed, below is a workaround which achieves our objective.

Step Six : Automation that Works:)

We will now use the integer variable *InstancesOfCounterThread* to control how many threads we need to run to satisfy our user.

Change the following functions as follows,

Change the OnClick handler of Button1 to the following,

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  //launch the thread.
  InstancesOfCounterThread++; //add one to our stack of needed

  CounterLoopThread *CountingLoopInstance;//Test MultiThreaded Item
  CountingLoopInstance = new CounterLoopThread(true);
  CountingLoopInstance->Resume();
}
```

the Execute function of Unit3 to the following,

```
void __fastcall CounterLoopThread::Execute()
{
  //---- Place thread code here ----
  Synchronize(CheckUsage);
  if(Terminated == false)
  {
    for(int x=0; x < 10000; x++)
    {
        Counter = x;
        Synchronize(UpdateCaption);
        if(Terminated == true)
        {//terminate requested
           x = 200000;//ensure loop is broken
        }
    }
  }

  if(Terminated == false)
  {
    Synchronize(FreeFlag);
  }
  else
  {
        //Don't do anything :)
  }

  FreeOnTerminate = true;
  Terminate();
}
```

the CheckUsage function of Unit3 to the following

```
void __fastcall CounterLoopThread::CheckUsage()
{
  switch(Form1->UsageFlag)
```

```
  {
        case true:     //there is a thread instance already running
                Terminate();
                break;
        case false:    //no other thread has the semaphore:)
                Form1->UsageFlag = true;             //Now we have the semaphore
                break;
        default:               //We really need to know,  this would be bad
                Application->MessageBox("Terminal Error. Closing Error",
                                                "From thread",MB_OK|MB_ICONERROR);
                Form1->Close();       //Force the application to close
                break;
  }
}
```

and finally the FreeFlag function of unit3 to the following.

```
void __fastcall CounterLoopThread::FreeFlag()
{
  Form1->UsageFlag = false;   //free the flag
  Form1->InstancesOfCounterThread--;
  if(Form1->InstancesOfCounterThread == 0)
  {
    Form1->Caption = "No Threads Running";
  }
  else
  {
    Form1->Caption = "Reloading Thread,  please watch";
    Form1->Button1Click(Form1);
    //decrease the count
    Form1->InstancesOfCounterThread--;
  }
}
```

Okay, now run the application, and press the button. This time look at the third label, the one which should now read "*Number Waiting =1*". The counter is running, so click the button again. Notice now that Label3 now reads "*Number Waitiing =2"*, click the button again, and we now have 3 waiting.

Wait for the first count to finish, and watch as the counters decrement. The logic for this is quite straightforward, the only thing to notice is how we launch another thread. Note that we fire it before our thread finishes, but this is not a problem, as we have freed the semaphore. The actual call to Form1->Button1Click is important. As the calling object, we pass Form1, basically put, we ensure that the new thread is called from within the main application thread, with Button1Click returning to the sender, in this case Form1. If we called the Click handler with the sender as our thread, we may run into problems later, as our thread will have terminated. Better be safe than sorry.

In Summary

There's been a lot to take in during this tutorial, but ensure you take away the following key points when dealing with threads,

- Each thread is created dynamically at runtime
- There is nothing to stop a thread running twice unless you implement it
- Threads can easily deadlock your application, especially when using while, for and do/while loops
- Never call a function within another unit without using the built-in Synchronize function.
- Think of your user, if they want to run your thread twice, let them, just control the way it's done
- Semaphores are the control to your thread, and are used when you have a resource which you wish to control access to.

Finally, I hope I have cleared up why we have threads, and some of the problems you must face. Be careful to control thread access, as it is *very* easy to put your application into deadlock.

JpmJenkinson

****

A Final Note

No liability is accepted by the author(s) for anything which may occur whilst following this tutorial

****