Show

Visual APL Programming Guide

How to: Create and Terminate Threads (Visual APL Programming Guide)

This example demonstrates how an auxiliary or worker thread can be created and used to perform processing in parallel with the primary thread. Making one thread wait for another and gracefully terminating a thread are also demonstrated. For background information on multi-threading, see Managed Threading and Using Threading (Visual APL Programming Guide).

The example creates a class named `Worker` that contains the method that the worker thread will execute called `DoWork`. The worker thread will begin execution by calling this method, and terminate automatically when this method returns. The `DoWork` method looks like this:

Visual APL

```
    public function DoWork() {
        a = Form()
        t = TextBox()
        t.Multiline = true
        t.Size = Size(200, 200)
        a.Controls.Add(t)
        a.Show()
        for (i=0;i<100;i++) {
            if (_shouldStop) {
                break;
            }
            t.AppendText(i+". worker thread: working...\n")
        }
        MessageBox.Show("Worker thread: terminating gracefully")
    }
```

The `Worker` class contains an additional method that is used to indicate to `DoWork` that it should return. This method is called `RequestStop`, and looks like this:

Visual APL

```
public void RequestStop()
{
    _shouldStop = true;
}
```

The `RequestStop` method merely assigns the `_shouldStop` data member to **true**. Because this data member is checked by the `DoWork` method, this has the indirect effect of causing `DoWork` to return, thereby terminating the worker thread. However, it is important to note that `DoWork` and `RequestStop` will be executed by different threads. `DoWork` is executed by the worker thread, and `RequestStop` is executed by the primary thread, so the `_shouldStop` data member is declared **volatile**, like this:

Visual APL
```
private volatile bool _shouldStop;
```

The **volatile** keyword alerts the compiler that multiple threads will access the `_shouldStop` data member, and therefore it should not make any optimization assumptions about the state of this member. For more information, see volatile (Visual APL Reference).

The use of **volatile** with the `_shouldStop` data member allows us to safely access this member from multiple threads without the use of formal thread synchronization techniques, but only because `_shouldStop` is a **bool**. This means that only single, atomic operations are necessary to modify `_shouldStop`. If, however, this data member were a class, struct, or array, accessing it from multiple threads would likely result in intermittent data corruption. Consider a thread that changes the values in an array. Windows regularly interrupts threads in order to allow other threads to execute, so this thread could be halted after assigning some array elements but before assigning others. This means the array now has a state that the programmer never intended, and another thread reading this array may fail as a result.

Before actually creating the worker thread, the `test` function creates a `Worker` object and an instance of Thread. The thread object is configured to use the `Worker.DoWork` method as an entry point by passing a reference to this method to the **Thread** constructor, like this:

Visual APL
```
wo = Worker();
wt = new Thread((ThreadStart)wo.DoWork);
```

At this point, although the worker thread object exists and is configured, the actual worker thread has yet been created. This does not happen until `test` calls the Start method:

Visual APL
```
wt.Start();
```

At this point the system initiates the execution of the worker thread, but it does so asynchronously to the primary thread. This means that the `test` function continues to execute code immediately while the worker thread simultaneously undergoes initialization. To insure that the `test` function does not try to terminate the worker thread before it has a chance to execute, the `test` function loops until the worker thread object's IsAlive property gets set to **true**:

Visual APL
```
while (!wt.IsAlive);
```

Next, the primary thread is halted briefly with a call to Sleep. This insures that the worker thread's `DoWork` function will execute the loop inside the `DoWork` method for a few iterations before the `test` function executes any more commands:

Visual APL
```
Thread.Sleep(300);
```

After the 300 millisecond elapses, `test` signals to the worker thread object that it should terminate using the `Worker.RequestStop` method introduced previously:

Visual APL
```
wo.RequestStop();
```

It is also possible to terminate a thread from another thread with a call to Abort, but this forcefully terminates the

affected thread without concern for whether it has completed its task and provides no opportunity for the cleanup of resources. The technique shown in this example is preferred.

Finally, the `test` function calls the Join method on the worker thread object. This method causes the current thread to block, or wait, until the thread that the object represents terminates. Therefore **Join** will not return until the worker thread returns, thereby terminating itself:

Visual APL
```
wt.Join();
```

At this point only the primary thread executing `test` exists. It displays one final message, and then returns, terminating the primary thread as well.

The complete example appears below.

Example

Visual APL
```
using System
using System.Threading
refbyname System.Windows.Forms
using System.Windows.Forms
refbyname System.Drawing
using System.Drawing


public class Worker {

    // This method will be called when the thread is started.
    public function DoWork() {

        a = Form()
        t = TextBox()
        t.Multiline = true
        t.Size = Size(200, 200)
        a.Controls.Add(t)
        a.Show()
        for (i=0;i<100;i++) {
            if (_shouldStop) {
                break;
            }
            t.AppendText(i+". worker thread: working...\n")
        }
        MessageBox.Show("Worker thread: terminating gracefully")
    }
    public void RequestStop() {
        _shouldStop = true;
    }

    // Volatile is used as hint to the compiler that this data
    // member will be accessed by multiple threads.
    private volatile _shouldStop = false;
```

```
}
static function test() {

    // Create the thread object. This does not start the thread.
    wo = Worker()
    wt = Thread((ThreadStart)wo.DoWork)

    // Start the worker thread.
    wt.Start()
    print "main thread: starting worker thread..."

    // Put the main thread to sleep for 300 milliseconds to
    // allow the worker thread to do some work:
    Thread.Sleep(300)

    // Request that the worker thread stop itself:
    wo.RequestStop()

    // Use the Join method to block the current thread
    // until the object's thread terminates.
    wt.Join()
    print "main thread: worker thread has terminated"
}
```

Sample Output

```
main thread: starting worker thread...
1. worker thread: working...
2. worker thread: working...
3. worker thread: working...
4. worker thread: working...
5. worker thread: working...
6. worker thread: working...
7. worker thread: working...
8. worker thread: working...
9. worker thread: working...
10. worker thread: working...
11. worker thread: working...
Shown in MessageBox -  worker thread: terminating gracefully...
main thread: worker thread has terminated
```