Threading

Multithreading sounds like turbo charging. Does it speed up a program?

In the modern computers with multiple cores, multithreading does speed up a program, as the threads are picked up by different cores. Nonetheless, *speeding up a program is not the primary purpose of multithreading*. Additionally, multithreading will slow down the actual program execution time because the overhead of creating, registering and maintaining a separate thread and also jumping between the threads will add to the execution time.

Then why do we use multithreading?

We use multithreading when both of the following conditions are true:

1. We anticipate that a function is going to take a long time to execute. Let's call it `LengthyFunction()`.
2. We want the main program to remain responsive to requests even while `LengthyFunction()` is executing.

How do we use multithreading?

Multithreading involves just 2 steps:

1. Make a `Thread` instance out of `LengthyFunction`, say `Thr`.
2. `Start` the `Thr`.

Sample Code

Namespace needed for threading:

```
using System.Threading;

private void LengthyFunctionThroughAThread() {
    // Step 1
    Thread Thr = new Thread(LengthyFunction);
    // Step 2
    thr.Start();
```

```
}
```

Believe it or not, that is all that is needed to make a program multithreaded.

Concept

When a thread is created, as in step 1, the CPU creates a separate area for executing the function.

When the thread is started, as in step 2, the CPU gives half the process-allotted-time to the main program and half to the thread. The process-allotted-time itself is in milliseconds, so that half the time away from the main program is not palpable and therefore, the main program appears responsive.

Note: The above process is described to explain the concept. Nevertheless, it is not very far from the actual execution process.

The CPU takes only a few nanoseconds to create a thread, or manage a list of threads or jump between the threads. Let us magnify these times to 1 second each, in order to understand the concept better. Also, let's assume `LengthyFunction` takes 10 seconds to execute normally. In this case, this is how a non-threaded and a threaded operation will look like on a single-processor machine:

Non-threaded operation

Time	Execution
0:00	The main program is available.
0:01	`LengthyFunction()` is called.
0:01	`LengthyFunction()` starts running.
0:01 to 0:11	The main program is not available for any other function.
0:11	`LengthyFunction()` finishes running.
0:11 onwards	The main program is responsive/available after the `LengthyFunction()` has finished running.

Threaded operation

Time	Execution on main program	Execution on thread
0:00	The main program is available.	
0:01	`LengthyFunctionThroughAThread()` is called.	

0:02	A thread for `LengthyFunction()` is created.	Separate thread created.
0:02	The `Start()` method is called on the thread.	`LengthyFunction()` starts running.
0:02 onwards	The main program is responsive / available.	
0:02 to 0:23	The CPU gives half the time allotted to the process, to the main program. Managing the time between the threads delays the execution time of `LengthyFunction()`.	`LengthyFunction()` is running in a separate thread. The CPU gives half the time allotted to the process, to the `LengthyFunction()` thread.
0:23		`LengthyFunction()` finishes running.
0:23 onwards	The CPU gives the time allotted to the process, completely to the main program.	

So, although creating a separate thread delayed the completion of `LengthyFunction()`, the main program remains responsive throughout (available to be acted upon). Obviously, a user able to work with a responsive application and getting the result of his action in 22 seconds will be happier than a user forced to look at a frozen application for 10 seconds.

Once the scenario of multiple processors kicks in, the multithreaded application may take less than 13 seconds to get the complete processing time for the main program. But as mentioned earlier, multiprocessing is not the focus of this chapter.

Restrictions

A function can be called via a thread only if it conforms to one of the following two delegates:

Thread delegates	
Delegate	**Signature**
`ThreadStart`	`void FunctionName ( )`
`ParameterizedThreadStart`	`void FunctionName ( object Obj )`

As a matter of fact, the Step 1 in the sample code is a shortcut for:

```
ThreadStart DelegateForFunc = new ThreadStart(LengthyFunction);
Thread Thr = new Thread(DelegateForFunc);
```

Parameters

Well, a function usually does something based on some parameters. It usually gets a parameter from either

1. a field or property of its class,
2. or its argument

Obviously, the use of a class field does not call for something special in the creation of a thread. Its use is inherent in the function.

How is an argument sent to a threaded function?

The following two observations will make the process clear.

1. As you can see from the table of the *Threaded operation*, when a thread is created, the underlying function is just marked for execution at a separate place. The function actually executes only when the `Start()` method is called on the Thread. Therefore, the argument to a function is passed via the `Start()` method.
2. Just as creating the thread for a function which runs without a parameter uses `ThreadStart` delegate, the thread for a function which runs with a parameter uses the `ParameterizedThreadStart` delegate. Looking at the signature of this delegate, we make two sub-observations:
   a) The underlying function can only have one parameter.
   b) This parameter can only be of type `object`. So, if the function expects an `int` or any other type, it must cast this parameter to that type in its body and then work with it.

These observations have been made only to clarify the concept. As far as the code is concerned, there is not much difference as we can see by comparing the following sample code with the basic one.

Sample Code

```
private void LengthyFuncWithParamThroughAThread() {
    // Step 1
    Thread Thr = new Thread(LengthyFuncWithParam);
    // Step 2 – Pass the argument
    Thr.Start("dummy");
}

private void LengthyFuncWithParam(object Obj) {
    // Cast the parameter into the type that is required
```

```
    string Param = (string) Obj;
    // Use the parameter
}
```

Multiple Parameters

The restriction that only one parameter can be passed to a function called via a thread poses a challenge when we want to pass multiple parameters. We circumvent this restriction with one of the following methods.

Method 1: Class fields or properties

This is a no-brainer. We simply add as many fields or properties to the class as the parameters that we expect to send to the function. We set these fields before calling the `Start()` method of the thread.

Sample code

```
// Class fields
private string sChk;
private int iStat;

private void LengthyFunc_ParamsFromFieldsThroughAThread() {
    // Step 1
    Thread Thr = new Thread(LengthyFunc_ParamsFromFields);
    // Argument Step – Set the class fields
    this.sChk = "LengthyFunc_ParamsFromFields";
    this.iStat = 1;
    // Step 2
    Thr.Start();
}

private void LengthyFunc_ParamsFromFields() {
    // Use the class fields as parameters
    string ParamCheck = this.sChk;
    int ParamStatus = this.iStat;
    // Use the parameters
}
```

Note: The restriction that the parameter of the function to be used via a thread is of type `object` can be used as a boon. Basically, this restriction means that we can pass any argument to the function since every type inherits from `object`. Since the value-types are boxed inherently on being used as an `object`, even they do not pose any challenge. We will use this boon in methods 2 and 3.

Method 2: Separate class for parameters

In this method, we first create a class and make the fields or properties of this class corresponding to the parameters that the function requires. Before calling the `Start()` method on the thread, instantiate this class, set the fields or properties of this instance and pass this instance to the `Start()` method. In the threaded function, we cast this parameter to this class, thus using its fields and properties as expected parameters.

Sample code

```
// A separate class containing the parameters
public class LengthyFunctionParams {
    public string Check;
    public int Status;
}

private void LengthyFunc_ParamsFromClassThroughAThread() {
    // Step 1
    Thread Thr = new Thread(LengthyFunc_ParamsFromClass);
    // Argument Step – Instantiate the separate class
    LengthyFunctionParams LFP = new LengthyFunctionParams();
    LFP.Check = "LengthyFunc_ParamsFromClassThroughAThread";
    LFP.Status = 10;
    // Step 2 – Pass the argument
    Thr.Start(LFP);
}

private void LengthyFunc_ParamsFromClass(object Obj) {
    // Cast the Obj parameter as the separate class
    LengthyFunctionParams LFP = (LengthyFunctionParams) Obj;
    // Use the fields/properties of Obj to obtain the parameters
    string ParamCheck = LFP.Check;
    int ParamStatus = LFP.Status;
    // Use the parameters
}
```

Method 3: Parameters in a collection or array

In this method, we use any collection (`Hashtable`, `ArrayList`, `DataTable`, `DataSet` or even the generic `HashSet`, `Dictionary`, etc.) or array.

Basically, we keep inserting parameters into this collection and pass this collection to the `Start()` method. In the threaded function, we cast the parameter to the chosen collection type. Then, we cast them into the appropriate types and then work with them.

Sample code

```
private void LengthyFunc_ParamsFromIEnumerableThroughAThread() {
    // Step 1
```

```
    Thread Thr = new Thread(LengthyFunc_ParamsFromIEnumerable);
    // Argument Step – Fill up any IEnumerable
    System.Collections.Hashtable HTblArgs =
        new System.Collections.Hashtable();
    HTblArgs.Add("Check", "LengthyFunc_ParamsFromIEnumerable");
    HTblArgs.Add("Status", 100);
    // Step 2 – Pass the argument
    Thr.Start(HTblArgs);
}

private void LengthyFunc_ParamsFromIEnumerable(object Obj) {
    // Cast the Obj parameter as the expected IEnumerable
    System.Collections.Hashtable HTblParams =
        (System.Collections.Hashtable) Obj;
    // Retrieve the parameters from the IEnumerable
    string ParamCheck = (string) HTblParams["Check"];
    int ParamStatus = (int) HTblParams["Status"];
    // Use the parameters
}
```

Return Variables

You may want a threaded function to return something once it finishes its task. Since the delegates available to us return `void`, how do we return something? The possible solutions are the same as for multiple parameters:

1. Use the class fields or properties as return variables, or
2. Use an instance of a separate class having fields or properties which can be used as return variables, or
3. Add return variables to a passed collection.

In other words, *nothing special is required*.

Not so fast

Observe that the above possible solutions are the same as returning something from a function which returns `void`. Nevertheless, multithreading poses an issue different from a `void`-returning-function.

In a non-threaded operation, the main program waits till the called function finishes and then works with the return variables.

In the threaded operation, on the other hand, the main program can start working on the return variables as soon as the function has been called via a thread. This is not desirable. Most likely, the threaded function is still working on the return variables. If it were quick to return these variables, we would not have to use threading at the first place.

To work around this situation, we use one of the following two methods.

Method 1: WaitHandle

Basically: *wait for a signal from the threaded function*.

Here are the steps we follow:

a) Create a class-level instance of one of the many child classes of `WaitHandle`. These classes act as signals and provide us with at least one method to give the *ON* signal and one method to give the *OFF* signal. In the sample code, we create an instance of the child class `AutoResetEvent`. Let's call it `ARE`.

b) Set the signal `ARE` to *OFF* before calling the threaded function. In case of an `AutoResetEvent`, it is done by calling the `Reset` method.

c) In the main program, where we want to work with the return variables, wait for the signal to turn *ON*. This is done by using the `WaitOne` method.

d) In the threaded function, set the signal to *ON* when the function is ready to return the variables.

Sample Code

```
// Class Fields
//     Parameters
private string sChk;
private int iStat;
//     Return variables
private string sRet;
//     Signal
private AutoResetEvent ARE;

private void LengthyFunc_WaitHandleThr() {
    // Step 1
    Thread Thr = new Thread(LengthyFunc_WaitHandle);
    // Argument Step
    this.sChk = "LengthyFunc_WaitHandleThr";
    this.iStat = 100;
    // Step a) Create the signal
    this.ARE = new AutoResetEvent(false);
    // Step b) Signal OFF
    this.ARE.Reset();
    // Step 2
    Thr.Start();
}

// This function can be called at any time
private void UseReturnVariables_WaitHandle() {
```

```
    // Step c) Wait for the signal to turn ON
    this.ARE.WaitOne();
    // Use the return variable(s)
    UseReturnVariables(this.sRet);
}

private void LengthyFunc_WaitHandle(object Obj) {
    // Use the parameter(s) and set the return variable(s)
    this.sRet = this.sChk + " returns " + 2 * this.iStat;
    // Do other things which make the function lengthy
    //Thread.Sleep(10000);
    // Step d) Signal ON
    this.ARE.Set();
}
```

Method 2: Callback

Basically: *make the threaded function execute the function which needs to use the return variables*. This involves a few steps:

a) Create a function that will use the return variables and make the return variables as the parameters of the function. Let's call it `UseReturnVariables`. This step is not shown in the Sample Code.

b) Create a delegate with the same signature as the `UseReturnVariables`.

c) As one of the arguments to the threaded function, pass the function via the delegate. We have done this using the class field.

d) In the threaded function, retrieve the delegate, just as any parameter.

e) At the end of the function, invoke the delegate, passing the return values as its arguments.

Sample Code

```
// Step b) Delegate with the expected return values as parameters
//             inside or outside of the class
public delegate void ReturnVarsHandler(string sReturn);

// Class Fields
//     Parameters
private string sChk;
private int iStat;
//     Return variables
private string sRet;
//     Delegate to the function which uses the return variables
ReturnVarsHandler ArgFunc;

private void LengthyFunc_CallbackThr() {
    // Step 1
    Thread Thr = new Thread(LengthyFunc_Callback);
    // Argument Step
```

```
    this.sChk = "LengthyFunc_CallbackThr";
    this.iStat = 100;
    // Step c) Setup the delegate to the function
    //           which uses the return variables
    //           just like setting up other arguments
    this.ArgFunc = new ReturnVarsHandler(UseReturnVariables);
    // Step 2
    Thr.Start();
}

private void LengthyFunc_Callback(object Obj) {
    // Use the parameter(s) and set the return variable(s)
    string sReturn = this.sChk + " returns " + 2 * this.iStat;
    // Do other things which make the function lengthy
    //Thread.Sleep(10000);
    // Steps d and e) Retrieve the callback function
    //   and invoke it with the return value(s)
    this.Invoke(this.ArgFunc, new object[] { sReturn });
}
```

Note: The delegate has not been called directly as `this.ArgFunc(sReturn);`. It has been called using the `Invoke` function. Basically, introducing the `Invoke` runs the underlying function on the main thread. Without `Invoke`, we will not be returning the variables but only using them. If we did not want to return the variables, we might as well have called the function `UseReturnVariables` directly.

Why go back?

A natural question that now arises is that why should we wait for the thread to finish when the whole point of multithreading is the freedom from this wait. There are various scenarios in which this is desirable:

1. The function that started the thread is different from the function that expects the return variables from the thread. The second function may be called when a user is finished working on other areas of the application and is now ready to wait for the return variables. We will see an example in the section *Concurrency*.
2. The main function itself may be a long running program and by the time it finishes, it wants the return variables from the thread. To an experienced developer, this would make sense when the threaded function mainly waits for (a) response(s) from some outside application(s).

Concurrency

In the section *Return Variables*, we saw the issue of a parent and a child thread concurrently trying to work with the return variable. We now generalize the issue to multiple threads concurrently trying to work on the same variable(s). This is called *Concurrency*.

We look at the two commonly used solutions to this issue: `lock` and `ReaderWriterLockAsync`.

Lock

Let's say a variable is to be used by various threads. In all the functions, just before this variable is going to be used, we lock it. When the function has finished using the variable, it unlocks it.

Initially, a thread comes to one such function; it requests the CLR to lock this variable for it. The CLR does that, and the thread starts working with the variable. While the first thread is still working with this variable, another thread tries to use the same variable.

The second thread may or may not have the same underlying function. Since we have followed the rule of applying the lock to the variable before using it, in all the functions, this thread requests the CLR to lock the variable for it. But since the variable is already locked, the CLR is unable to do so and makes that thread stand still at that point of its function. When the first thread finishes its work with the shared variable and requests the CLR to unlock it, the CLR does so. Then, the CLR locks the variable for the second thread and it continues it execution. Similar to the first thread, once the second thread finishes working with the variable, it requests the CLR to unlock it, to which the CLR obliges.

Method

We lock a variable by using the keyword `lock`, followed by putting that variable in parentheses. This is followed by a section of code in braces. The closing brace automatically unlocks the variable.

In the following sample code, three functions try to work on two datatables. All these functions are called via threads, so we may have the issue of concurrency. The lock mechanism solves this for us, as explained above.

Sample Code

```
// Class fields
DataTable DTblWork, DTblWorkDone;
// Class constructor
public frmSync() {
    InitializeComponent();
    this.DTblWork = new DataTable();
    this.DTblWork.Columns.Add("ID", Type.GetType("System.String"));
    this.DTblWork.Columns.Add(
        "WorkStarted", Type.GetType("System.DateTime"));
    this.DTblWorkDone = DTblWork.Clone();
    this.DTblWorkDone.Columns.Add(
        "WorkFinished", Type.GetType("System.DateTime")
    );
}

private void MimicGettingWorkRequests() {
    while (true) {
        // Mimic that a work request came in at a random time
        Thread.Sleep((new Random()).Next(2000) + 1000);
        string sDt = DateTime.Now.ToString("MMyyddmmHHss");
        lock (this.DTblWork) {
            this.DTblWork.Rows.Add(sDt, DateTime.Now);
        } // End of lock (this.DTblWork)
    }
}

private void WorkOnRequests() {
    while (true) {
        lock (this.DTblWork) {
            if (this.DTblWork.Rows.Count > 0) {
                DataRow DRowWork = DTblWork.Rows[0];
                // Say the work takes 3 seconds to complete
                Thread.Sleep(3000);
                lock (this.DTblWorkDone) {
                    this.DTblWorkDone.Rows.Add(
                        DRowWork["ID"], DRowWork["WorkStarted"],
                        DateTime.Now
                    );
                } // End of lock (this.DTblWorkDone)
                this.DRowWork.Delete();
            }
        } // End of lock (this.DTblWork)
    }
}
```

Note: Make the IsBackground property of the threads true

Because of the current workload, the work on teaching modules has been suspended indefinitely. This particular teaching module is incomplete as well.