Threads

Life Cycle of a Thread

Thread

- This is a class in C#.net that can be found in the **`System.Threading`** namespace. It is used to create and control threads in a system or application, in which the properties and methods are already provided.
- This is an independent execution unit containing a piece of code. When a process begins, the main thread is started.
- All threads within a process share the same state and memory space and communicate with each other to perform different tasks.

Main Thread

- When using the **`Thread`** class, the first thread to be performed in a process is known as the **main thread**.
- The other threads that are made using the `Thread` class are known as the *child thread* of the main thread.

```
//Sample Main Thread
Thread basicThread = Thread.CurrentThread;
basicThread.Name = "Basic C# Thread";
Console.WriteLine("Current Thread: {0}", basicThread.Name);
```

Child Thread

- Creating a child thread for the main thread should write or create a delegate object, passing a callback method to it as a parameter.
- When the **thread** object is created, the **delegate** will be used to initialize the **thread** object.
- To define a callback method in the delegate, use **`ThreadStart`** to execute the code when the thread started.
- A **`ThreadStart`** delegate represents a method that runs in the `Thread` class. See sample code below.

```
//Sample Child Thread
private void btnRunThread_Click(object sender, EventArgs e)
{
   ThreadStart delThread = new ThreadStart(ChildThreadMethod);
   Thread childThread = new Thread(delThread);
   Thread.CurrentThread.Name = "Main Thread";
   Console.WriteLine(Thread.CurrentThread.Name);
   childThread.Start();
}
//method
public void ChildThreadMethod()
{
   Console.WriteLine("Calling child thread");
}
```

When using the `Thread` class, there are different ways to control it. The speed of execution can be paused, resumed, destroyed, or controlled. Thread has a state that determines when and where it undergoes during its life cycle.

Life Cycle of a Thread (Harwani, 2015)

1. A new thread begins its life cycle in the **`Unstarted`** state.
2. The thread remains in the `Unstarted` state until the Thread method **`Start`** is called, which places the thread in the `Started` (also known as `Ready` or `Runnable`) state.
3. The highest priority `Started` thread enters the **`Running`** state.
4. A `Running` thread enters the `Stopped` state when its job or task is over. Also, a `Running` thread can be forced to the `Stopped` state by calling the **`Abort`** method. The `Abort` method throws a `ThreadAbortException` in the thread, normally causing the thread to terminate.
5. A thread enters the **`Blocked`** state when the thread issues an input/output (I/O) request. In other words, the operating system blocks the thread to perform the I/O operations. The CPU time is not assigned to a `Blocked` thread.
6. After I/O operations are complete, the `Blocked` thread returns to the `Started` state so it can resume execution.
7. A `Running` thread may enter the **`WaitSleepJoin`** state either when it is asked to sleep for the specified number of

milliseconds or when the **Monitor** method **Wait** is called. From the **WaitSleepJoin** state, a thread returns to the Started state when another thread invokes the Monitor method **Pulse** or **PulseAll**. The **Pulse** method moves the next waiting thread back to the Started state. The **PulseAll** method moves all waiting threads back to the Started state.

8. A sleeping thread returns to the Started state when the specified sleep duration expires.
9. Any thread in the WaitSleepJoin state can return to the Started state if the sleeping or waiting thread's Interrupt method is called by another thread in the program.
10. If a thread cannot continue executing unless another thread terminates, it calls the other thread's **Join** method to join the two (2) threads. When two (2) threads are joined, the waiting thread leaves the WaitSleepJoin state when the other completes execution.

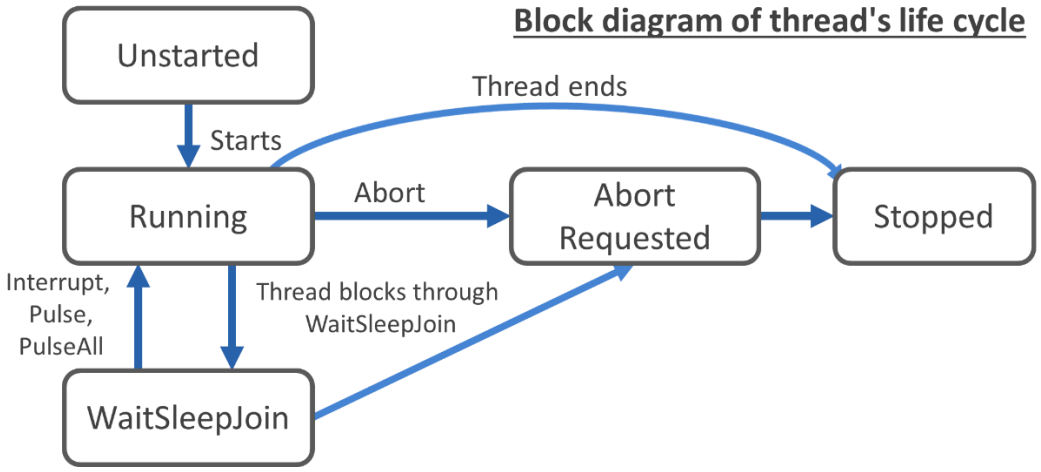

Figure 1. Block Diagram of Thread's Life Cycle
Source: Learning object-oriented programming in C# 5.0, 2015. p. 439

List of Thread States (Harwani, 2015)

- **Unstarted** – A thread is created within the Common Language Runtime (CLR) but has not started.
- **Ready** – A thread is ready to run and is waiting for the CPU time.
- **Running** – A thread is in running mode after invoking its Start method.
- **WaitSleepJoin** – A running thread is suspended temporarily by invoking either the Sleep method or the monitor's Wait method.
- **Started** – A suspended thread resumes to Started state when the conditions for which is it was suspended are no longer valid.
- **Blocked** – A thread is blocked when it is waiting for a resource or I/O operations.
- **Stopped** – A thread has finished its task.

```
private Thread _childThread;

private void btnCheck_Click(object sender, EventArgs e)
{
   ThreadStart delThreadObj = new ThreadStart(ThreadCycle);
   Console.WriteLine("Will Create Child Thread In Main Thread…");
   _childThread = new Thread(delThreadObj);
   _childThread.Start();
   Thread.Sleep(3000);//3 seconds before aborting the thread.
   Console.WriteLine("Aborting Child Thread");
   _childThread.Abort();
}
public void ThreadCycle()
{
   try
```

```
    {
        Console.WriteLine("Thread starts here!");
        Console.WriteLine("Paused for 2 seconds: child thread");
        Console.Write("Countdown: ");
        for(int x = 2; x > 0; x--){
            Console.Write(x + " ");
            Thread.Sleep(1000);
        }
        Console.WriteLine("\nChild thread resume");
        Console.WriteLine("WaitSleepJoin State: (Child Thread):" +
          (_childThread.ThreadState == ThreadState.WaitSleepJoin));
        Thread.Sleep(1000); //Thread sleep or pause for 1 second
        Console.WriteLine("Child Thread Done.");
    }
    catch(ThreadAbortException ex)
    {
        Console.WriteLine("Exception Message: " + ex.Message);
    }
    finally{
        Console.WriteLine("Finally Block the Thread!");
    }
}
```

The code above explains how to process different states of a child thread or the life cycle itself. If the thread is aborted, an exception will be thrown named **`ThreadAbortException`**. This exception can be caught but is automatically rethrown at the end of the **`catch`** block.

The Thread class also provides some properties and methods to access information about the thread.

Properties

- **`CurrentThread`** – It returns the current thread that is running.
- **`IsAlive`** – It returns a Boolean value indicating the execution status of the recent thread.
- **`IsBackground`** – It is used to get or set a value that indicates whether the thread is a background thread or not.
- **`Name`** – It is used to get or set the name of the thread.
- **`Priority`** – It is used to get or set a value that represents the priority of a thread.
- **`ThreadState`** – It is used to get the value that contains the states of the recent thread.

Methods

- **`public void Abort()`** – It terminates the thread when calling this method and raises `ThreadAbortException` in the thread.
- **`public void Interrupt()`** – It interrupts the thread that is in the state of `WaitSleepJoin`.
- **`public void Join()`** – It is used to stop the calling thread until a thread terminates.
- **`public static void ResetAbort()`** – It is used to withdraw an abort request for the ongoing thread.
- **`public void Start()`** – It is used to start a thread.
- **`public static void Sleep()`** – It is used to pause a thread for the stated number in milliseconds.

Multithreading

In operating systems, multithreading is a common feature that allows your application to have more than one (1) execution path at the same time. In multithreading, the CPU is assigned to each thread for a time slice before moving on to the next thread. In other words, the CPU serves each thread or a given time interval in a round-robin fashion (Harwani, 2015, p. 438).

```
private int x = 2;
private int y = 2;
```

```
private Thread _childThread1, _childThread2, _childThread3;

private void btnStart_Click(object sender, EventArgs e)
{
   Console.WriteLine("--------");
   Console.WriteLine("Main Thread Running…");
   ThreadStart delObjThread = new ThreadStart(Method1);

   _childThread1 = new Thread(delObjThread);
   _childThread1.Name = "Child Thread 1";
    _childThread2 = new Thread(delObjThread);
    _childThread2.Name = "Child Thread 2";

    _childThread3 = new Thread(new ThreadStart(Method2));
    _childThread3.Name = "Child Thread 3";

    _childThread1.Start();
    _childThread2.Start();
    _childThread3.Start();

    _childThread1.Join();
    _childThread2.Join();
    _childThread3.Join();
    Console.WriteLine("Value of x: " + x);
    Console.WriteLine("Value of y: " + y);
}
///Methods to be called in ThreadStart delegate
public void Method1(){
   Console.WriteLine(Thread.CurrentThread.Name + " running...");
    x++;
}
public void Method2(){
   Console.WriteLine(Thread.CurrentThread.Name + " running...");
   y++;
}
```

The sample program above shows how multithreading is implemented. It contains two (2) int variables and three (3) child threads. This program uses the `Join()` method to make the main thread wait until the three (3) child threads finish their task. The two (2) child threads (`_childThread1` and `_childThread2`) are gotten from the same resource simultaneously for manipulation, which is known as **race condition**.

REFERENCES:
Deitel, P. & Deitel, H. (2015). *Visual C# 2012 how to program* (5th ed.). USA: Pearson Education, Inc.
Gaddis, T. (2016). *Starting out with Visual C#* (4th ed.). USA: Pearson Education, Inc.
Harwani, B. (2015). *Learning object-oriented programming in C# 5.0.* USA: Cengage Learning PTR.
Miles, R. (2016). *Begin to code with C#.* Redmond Washington: Microsoft Press.
Doyle, B. (2015). *C# programming: from problem analysis to program design* (5th ed.). Boston, MA: Cengage Learning.