

عنوان مضمون

Visual Programming-II

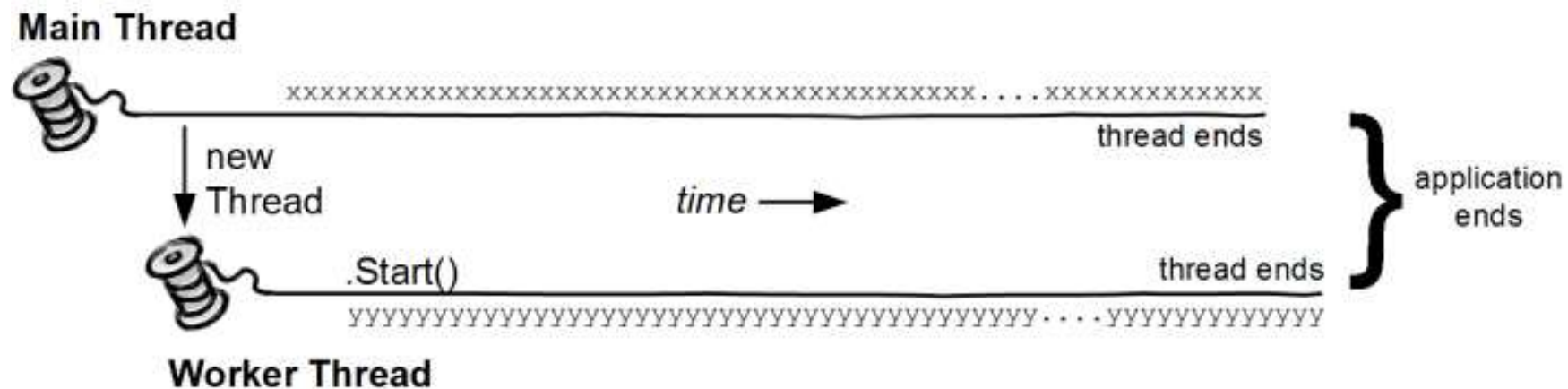
توسط : صفری

بهار 1397

Multithreading

Multithreading

- A **thread** is defined as the execution path of a program.
- Each thread defines a unique flow of control. If your application involves **complicated** and **time consuming** operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job

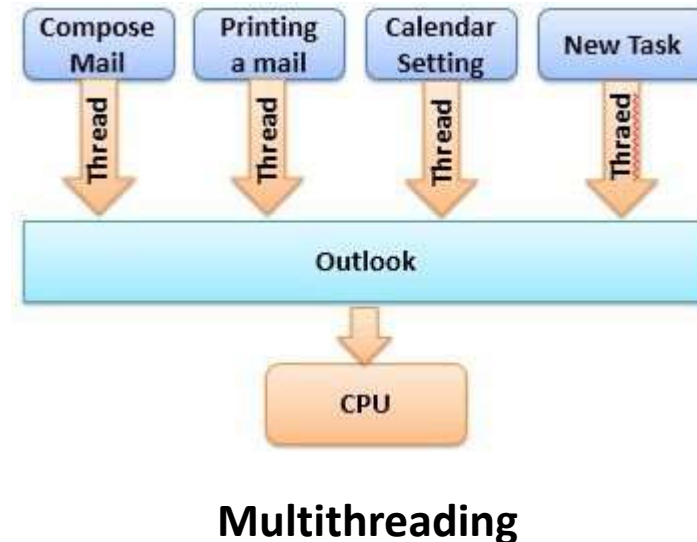
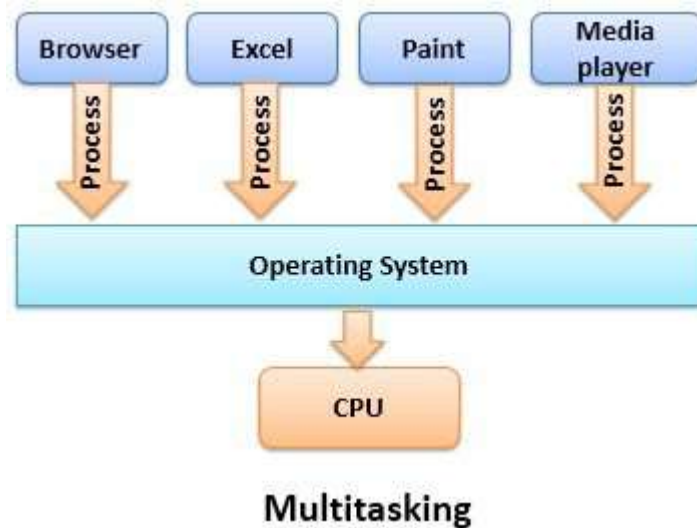


Multithreading

- Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.
- So far we wrote the programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

Multitasking and Multithreading

1. The basic difference between multitasking and multithreading is that in **multitasking**, the system allows executing multiple programs and tasks at the same time, whereas, in **multithreading**, the system executes multiple threads of the same or different processes at the same time.



Multitasking and Multithreading

2. In multitasking **CPU** has to **switch** between **multiple programs** so that it appears that multiple programs are running simultaneously. On other hands, in multithreading **CPU** has to **switch** between **multiple threads** to make it appear that all threads are running simultaneously.
3. Multitasking allocates **separate memory and resources** for each process/program whereas, in multithreading threads belonging to the same process **shares the same memory and resources** as that of the process.

Thread Life Cycle

- The life cycle of a thread starts when an object of the **System.Threading** class is created and ends when the thread is terminated or completes execution.
- Following are the various states in the life cycle of a thread –
 - **The Unstarted State** – It is the situation when the instance of the thread is created but the Start method is not called.
 - **The Ready State** – It is the situation when the thread is ready to run and waiting CPU cycle.
 - **The Not Runnable State** – A thread is not executable, when
 - Sleep method has been called
 - Wait method has been called
 - Blocked by I/O operations
 - **The Dead State** – It is the situation when the thread completes execution or is aborted.

The Main Thread

- In C#, the **System.Threading** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.
- When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.

The following program demonstrates main thread execution

```
static void Main(string[] args) {  
    Thread th = Thread.CurrentThread;  
    th.Name = "MainThread";  
    Console.WriteLine("This is {0}", th.Name);  
    Console.ReadKey();  
}
```

When the above code is compiled and executed, it produces the following result

This is MainThread

Creating Threads

- Threads are created by extending the Thread class. The extended Thread class then calls the **Start()** method to begin the child thread execution.
- The following program demonstrates the concept

```
Thread childThread = new Thread(DoTask);  
    childThread.Start();
```

```
public static void CallToChildThread() {  
    Console.WriteLine("Child thread starts");  
}  
  
static void Main(string[] args) {  
    Console.WriteLine("In Main: Creating the Child thread");  
    Thread childThread = new Thread(CallToChildThread);  
    childThread.Start();  
    Console.ReadKey();  
}
```

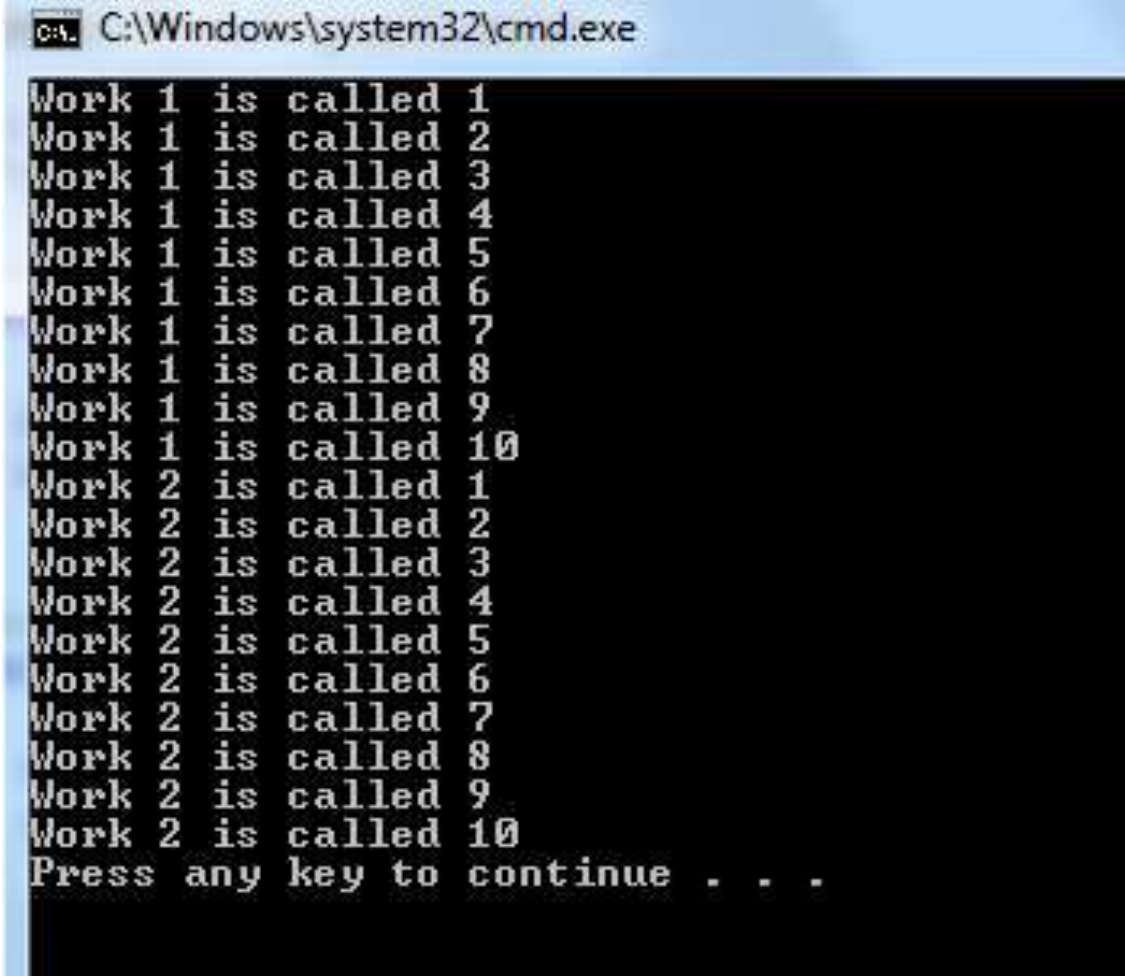
```
public static void CallToChildThread() {  
    Console.WriteLine("Child thread starts");  
}  
static void Main(string[] args) {  
    ThreadStart childref = new ThreadStart(CallToChildThread);  
    Console.WriteLine("In Main: Creating the Child thread");  
    Thread childThread = new Thread(childref);  
    childThread.Start();  
    Console.ReadKey();  
}
```

Before Multithreading

```
static void Main(string[] args)
{
    Work1();
    Work2();
}

static void Work1()
{
    for(int i = 1; i <= 10; i++)
    {
        Console.WriteLine("Work 1 is called " + i.ToString());
    }
}

static void Work2()
{
    for (int i = 1; i <= 10; i++)
    {
        Console.WriteLine("Work 2 is called " + i.ToString());
    }
}
```



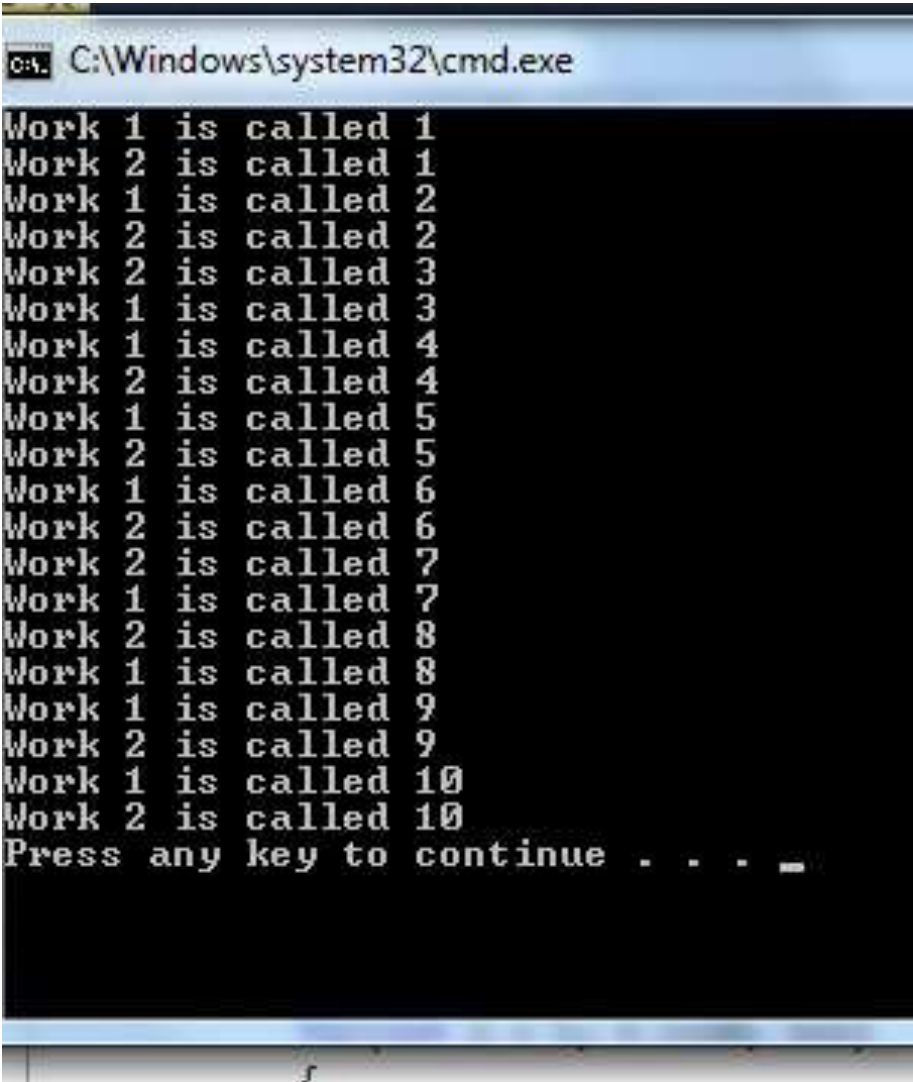
```
C:\Windows\system32\cmd.exe
Work 1 is called 1
Work 1 is called 2
Work 1 is called 3
Work 1 is called 4
Work 1 is called 5
Work 1 is called 6
Work 1 is called 7
Work 1 is called 8
Work 1 is called 9
Work 1 is called 10
Work 2 is called 1
Work 2 is called 2
Work 2 is called 3
Work 2 is called 4
Work 2 is called 5
Work 2 is called 6
Work 2 is called 7
Work 2 is called 8
Work 2 is called 9
Work 2 is called 10
Press any key to continue . . .
```

After Multithreading

```
static void Main(string[] args)
{
    Thread oThreadone = new Thread(Work1);
    Thread oThreadtwo = new Thread(Work2);
    oThreadone.Start();
    oThreadtwo.Start();
}

static void Work1()
{for(int i = 1; i <=10; i++)
    {Console.WriteLine("Work 1 is called " + i.ToString());
    }
}

static void Work2()
{ for (int i = 1; i <= 10; i++)
    { Console.WriteLine("Work 2 is called " + i.ToString());
    }
}
```



```
C:\Windows\system32\cmd.exe
Work 1 is called 1
Work 2 is called 1
Work 1 is called 2
Work 2 is called 2
Work 2 is called 3
Work 1 is called 3
Work 1 is called 4
Work 2 is called 4
Work 1 is called 5
Work 2 is called 5
Work 1 is called 6
Work 2 is called 6
Work 2 is called 7
Work 1 is called 7
Work 2 is called 8
Work 1 is called 8
Work 1 is called 9
Work 2 is called 9
Work 1 is called 10
Work 2 is called 10
Press any key to continue . . . _
```

Thread Methods

Method	Action
<u>Start</u>	Causes a thread to start to run.
<u>Sleep</u>	Pauses a thread for a specified time.
<u>Suspend</u>	Pauses a thread when it reaches a safe point.
<u>Abort</u>	Calling this method usually terminates the thread..
<u>Resume</u>	Restarts a suspended thread
<u>Join</u>	Causes the current thread to wait for another thread to finish. If used with a time-out value, this method returns True if the thread finishes in the allocated time.

Managing Threads

- The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time

```
public static void CallToChildThread() {  
    Console.WriteLine("Child thread starts");  
    // the thread is paused for 5000 milliseconds  
    int sleepfor = 5000;  
    Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);  
    Thread.Sleep(sleepfor);  
    Console.WriteLine("Child thread resumes");  
}  
  
static void Main(string[] args) {  
    ThreadStart childref = new ThreadStart(CallToChildThread);  
    Console.WriteLine("In Main: Creating the Child thread");  
    Thread childThread = new Thread(childref);  
    childThread.Start();  
    Console.ReadKey();  
}
```


Destroying Threads

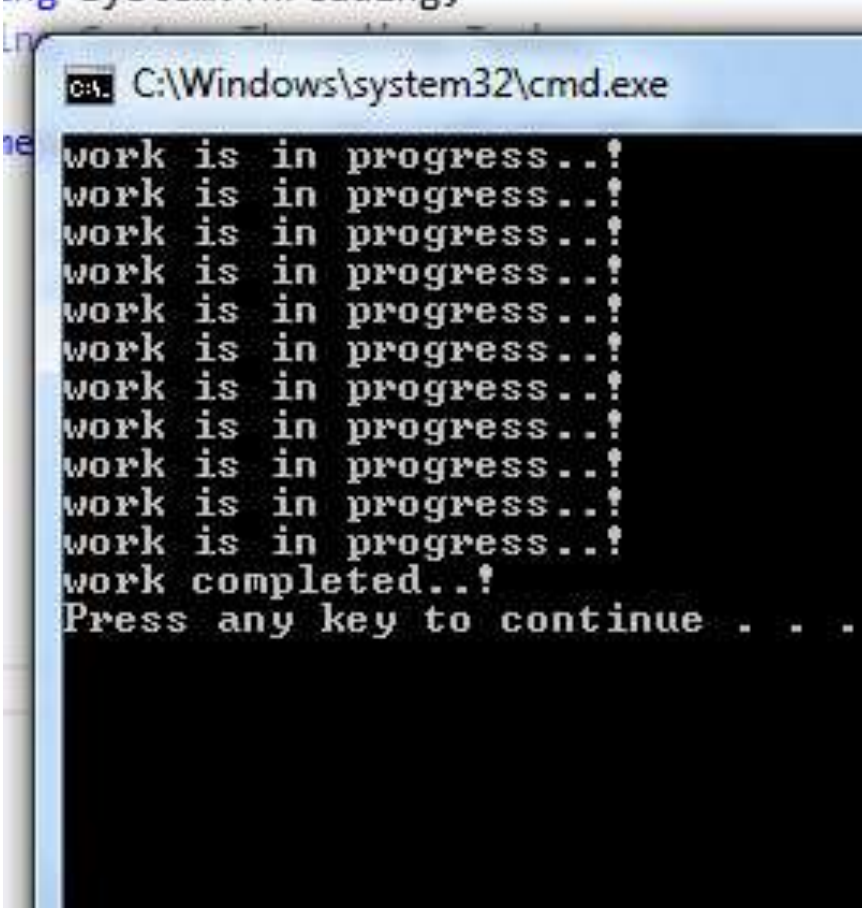
- The **Abort()** method is used for destroying threads.
- The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

```
public static void CallToChildThread() {
    Console.WriteLine("Child thread starts"); // do some work, like counting to 10
    for (int counter = 0; counter <= 10; counter++) {
        Thread.Sleep(500);
        Console.WriteLine(counter);
    }
    Console.WriteLine("Child Thread Completed");
}

static void Main(string[] args) {
    ThreadStart childref = new ThreadStart(CallToChildThread);
    Console.WriteLine("In Main: Creating the Child thread");
    Thread childThread = new Thread(childref);
    childThread.Start();
    Thread.Sleep(2000); //stop the main thread for some time
    Console.WriteLine("In Main: Aborting the Child thread"); //now abort the child
    childThread.Abort();
    Console.ReadKey();
}
```

```
static void Main(string[] args)
{
    Thread oThread = new Thread(MethodJoin);
    oThread.Start();
    oThread.Join();
    Console.WriteLine("work completed..!");
}

static void MethodJoin()
{
    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine("work is in progress..!");
    }
}
```



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\system32\cmd.exe". The command prompt displays the output of the C# program shown in the previous block. It shows 11 lines of "work is in progress..!" followed by "work completed..!" and a prompt "Press any key to continue . . .".

```
C:\Windows\system32\cmd.exe
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work is in progress..!
work completed..!
Press any key to continue . . .
```