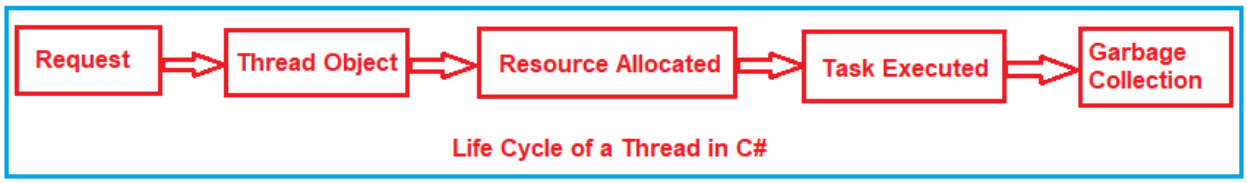
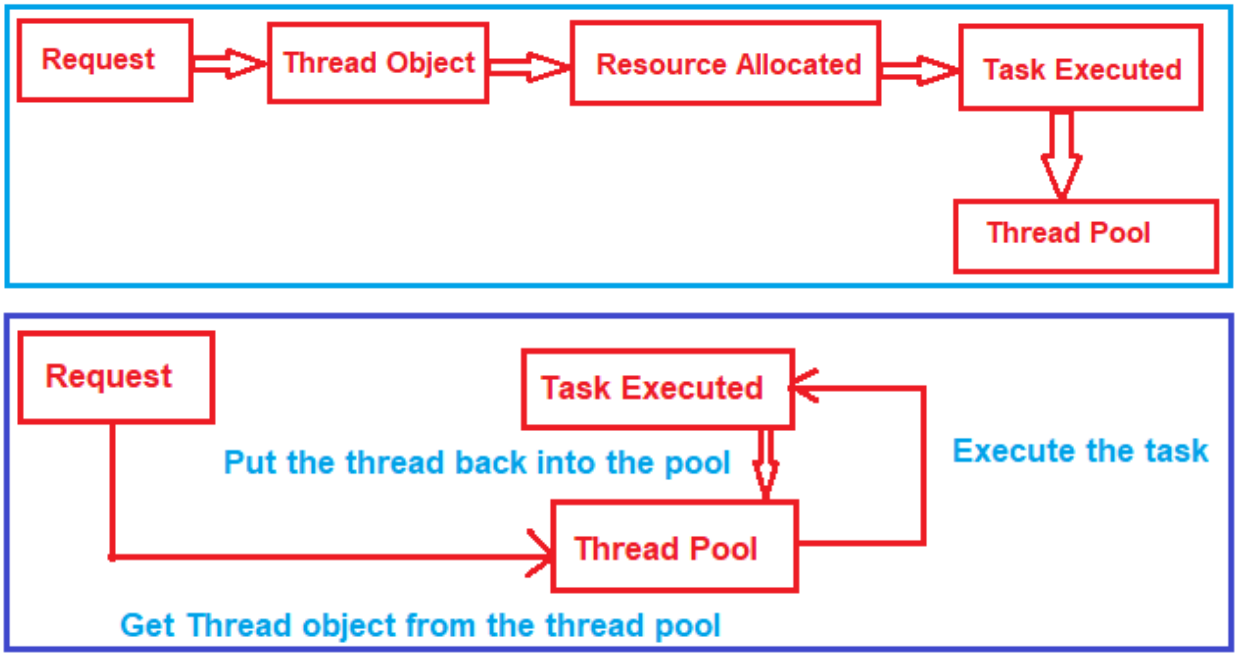
Thread Pooling in C#

Creation and destruction of new threads comes with a cost and it affects an application's performance. Threads can also be blocked or go into sleep or other unresolved states.



If your app doesn't distribute workload properly, worker threads may spend the majority of their time sleeping. This is where thread pool comes in handy. A thread pool is a pool of worker threads that have already been created and are available for apps to use them as needed. Once thread pool threads finish executing their tasks, they go back to the pool.



.NET provides a managed thread pool via the **ThreadPool** class that is managed by the system. As a developer, we don't need to deal with the thread management overhead. For any short background tasks, the managed thread pool is a better choice than creating and managing your own threads. Thread pool threads are good for background process only. There is only one thread pool per process. Use of thread pool is **not recommended when**,

- You need to prioritize a thread.
- Thread is a foreground thread.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.

- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

ThreadPool class in C#

Once you import the System.Threading namespace, then you need to use the **ThreadPool** class and using this class you need to call the **QueueUserWorkItem** static method. If you go to the definition of the **QueueUserWorkItem** method, then you will see that this method takes one parameter of type **WaitCallback** object.

```
public static bool QueueUserWorkItem (System.Threading.WaitCallback callBack);
```

```
public static bool QueueUserWorkItem (System.Threading.WaitCallback callBack, object? state);
```

While creating the object of the **WaitCallback** delegate, you need to pass the method name that you want to execute.

```
public delegate void WaitCallback(object state);
```

Here, the **QueueUserWorkItem** method Queues the function for execution and that function executes when a thread becomes available from the thread pool. If no thread is available then it will wait until one thread gets freed.

Properties of ThreadPool class

1. `CompletedWorkItemCount`
   Gets the number of work items that have been processed so far.
2. `PendingWorkItemCount`
   Gets the number of work items that are currently queued to be processed.
3. `ThreadCount`
   Gets the number of thread pool threads that currently exist.

Maximum and Minimum Thread Pool Threads

Thread pool size is the number of threads available in a thread pool. The thread pool provides new worker threads or I/O completion threads on demand until it reaches the minimum for each category. By default, the minimum number of threads is set to the number of processors on a system. When the minimum is reached, the thread pool can create additional threads in that category or wait until some tasks complete. The thread pool creates and destroys threads to optimize throughput, which is defined as the number of tasks that complete per unit of time. Too few threads might not make optimal use of available resources, whereas too many threads could increase resource contention.

ThreadPool.**GetAvailalbeThreads** returns the number of threads that are available currently in a pool. It is the number of maximum threads minus currently active threads. **ThreadPool**.**GetMaxThreads** and **ThreadPool**.**GetMinThreads** returns the maximum and minimum threads available in a thread pool. **ThreadPool**.**SetMaxThreads** and **ThreadPool**.**SetMinThreads** are used to set maximum and minimum

number of threads on demand as needed in a thread pool. By default, the minimum number of threads is set to the number of processors on a system.

Example:

```
using System;
using System.Threading;
class ThreadPoolSample
{
    // Background task
    static void BackgroundTask(Object stateInfo)
    {
        Console.WriteLine("Hello! I'm a worker from ThreadPool");
        Thread.Sleep(1000);
    }
    // Background task with additional object
    static void BackgroundTaskWithObject(Object stateInfo)
    {
        Person data = (Person)stateInfo;
        Console.WriteLine($"Hi {data.Name} from ThreadPool.");
        Thread.Sleep(1000);
    }
    static void Main(string[] args)
    {
        ThreadPool.QueueUserWorkItem(BackgroundTask);
        // Create an object and pass it to ThreadPool worker thread
        Person p = new Person("Dhaval", 30, "Male");
        ThreadPool.QueueUserWorkItem(BackgroundTaskWithObject, p);
        // Get maximum number of threads
        ThreadPool.GetMaxThreads(out int workers, out int ports);
        Console.WriteLine($"Maximum worker threads: {workers} ");
        Console.WriteLine($"Maximum completion port threads: {ports}");
        // Get available threads
        ThreadPool.GetAvailableThreads(out int workers, out int ports);
        Console.WriteLine($"Availalbe worker threads: {workers} ");
        Console.WriteLine($"Available completion port threads: {ports}");
        // Set minimum threads
        ThreadPool.GetMinThreads(out int minWorker, out int minIOC);
        ThreadPool.SetMinThreads(4, minIOC);
        // Get total number of processes availalbe on the machine
        Console.WriteLine($"No. of processors available on the system: {
Environment.ProcessorCount}");
        // Get minimum number of threads
        ThreadPool.GetMinThreads(out int workers, out int ports);
        Console.WriteLine($"Minimum worker threads: {workers} ");
        Console.WriteLine($"Minimum completion port threads: {ports}");
        Console.ReadKey();
    }
    // Create a Person class
    public class Person
    {
```

```
        public string Name { get; set; }
        public int Age { get; set; }
        public string Sex { get; set; }
        public Person(string name, int age, string sex)
        {
             this.Name = name;
             this.Age = age;
             this.Sex = sex;
        }
    }
}
// Output:
//    Hello! I'm a worker from ThreadPool
//    Maximum worker threads: 32767
//    Maximum completion port threads: 1000
//    Available worker threads: 32765
//    Available completion port threads: 1000
//    No. of processors available on the system: 4
//    Minimum worker threads: 4
//    Minimum completion port threads: 4
//    Hi Dhaval from ThreadPool.
```

Performance testing using and without using Thread Pool in C#

Let us see an example to understand the performance benchmark. Here, we will compare how much time does the thread object takes and how much time does the thread pool thread takes to do the same task i.e. to execute the same methods.

In order to do this, what we are going to do is, we will create a method called **Test** as shown below. This method takes an input parameter of type object and as part of that Test method we are doing nothing means an empty method.

```
public static void Test(object obj)
{
}
```

Then we will create two methods such as **MethodWithThread** and **MethodWithThreadPool** and inside these two methods, we will create one for loop which will execute 10 times. Within for loop, we are going to call the Test as shown below. As you can see, the **MethodWithThread** method uses the Thread object to call the Test method while the **MethodWithThreadPool** method uses the ThreadPool object to call the Test method.

```
public static void MethodWithThread()
{
for (int i = 0; i < 10; i++)
      {
            Thread thread = new Thread(Test);
      }
}
public static void MethodWithThreadPool()
{
```

```
        for (int i = 0; i < 10; i++)
        {
                ThreadPool.QueueUserWorkItem(new WaitCallback(Test));
        }
}
```

Now we need to call the above two methods (**MethodWithThread** and **MethodWithThreadPool**) from the main method. As we are going to test the performance benchmark, so we are going to call these two methods between the stopwatch start and end as shown below. The Stopwatch class is available in **System.Diagnostics** namespace. The for loop within the Main method is for warm-up. This is because when we run the code for the first time, compilation happens and compilation takes some time and we don't want to measure that.

```
static void Main(string[] args)
{
        for (int i = 0; i < 10; i++)
        {
                MethodWithThread();
                MethodWithThreadPool();
        }
        Stopwatch stopwatch = new Stopwatch();
        Console.WriteLine("Execution using Thread");
        stopwatch.Start();
        MethodWithThread();
        stopwatch.Stop();
        Console.WriteLine($"Time consumed by MethodWithThread is :
                                {stopwatch.ElapsedTicks.ToString()}");
        stopwatch.Reset();
        Console.WriteLine("Execution using Thread Pool");
        stopwatch.Start();
        MethodWithThreadPool();
        stopwatch.Stop();
        Console.WriteLine($"Time consumed by MethodWithThreadPool is :
                                {stopwatch.ElapsedTicks.ToString()}");
        Console.Read();
}
```

Exercise: Execute a console app to check performance improvements using ThreadPool over Thread class on your system as described in previous section.