Chapter 8
Multithreading

Mrs. Swati Satpute
Fergusson College

Objective

- Threads
- System.Threading Namespace .
- Thread Class – its methods and properties.
- Thread Synchronization.
- Monitors
- C# Lock keyword.
- Reader/Writer Locks
- Conclusion

Threads

- Thread is the **fundamental unit of execution**.
- **More than one thread** can be executing code **inside the same process** (application).
- On a single-processor machine, the operating system is switching rapidly between the threads, giving the appearance of simultaneous execution.

Threads

- With threads you can :
  - **Maintain a responsive user interface** while **background tasks are executing**
  - **Distinguish tasks of varying priority**
  - **Perform operations that consume a large amount of time** without stopping the rest of the application

- System.Threading Namespace
  - Provides classes and interfaces that enable multithreaded programming.
  - **Consists of classes** for **synchronizing thread activities .**
  - Chief among the namespace members is Thread class

Process and Thread

- A **process** represents an **application** whereas a **thread** represents a **module of the application**.

- **Process is heavyweight** component whereas **thread is lightweight**.

- A **thread** can be termed as lightweight subprocess because it is **executed inside a process.**

Thread Class

- Implements various methods & properties that **allows to manipulate concurrently running threads.**

- Some of them are :

- **CurrentThread**: returns the instance of currently running thread.
- **IsAlive**: used to find the execution status of the thread.
- **IsBackground:**used to get or set value whether current thread is in background or not.
- **Name**:
- **Priority** : used to get or set the priority of the current thread.
- **ThreadState** :used to return a value representing the thread state.

System.Threading

Commonly used classes

- Thread
- Mutex
- Timer
- Monitor
- Semaphore
- ThreadLocal
- ThreadPool
- Volatile

Thread Life Cycle

- Each thread has a life cycle.

- The life cycle of a thread is started when instance of *System.Threading.Thread class* is created.

Thread Life Cycle

There are following states in the life cycle of a Thread in C#.

- **Unstarted**:

  When the instance of Thread class is created, it is in unstarted **state by default.**

- **Runnable** (Ready to run):

  When start() method on the thread is called, it is in **runnable** or **ready to run state.**

- **Running**

  Only one thread within a process can be executed at a time. **At the time of execution, thread is in running state.**

- **Not Runnable**

  The thread is in not runnable state, if **sleep() or wait()** method is called on the thread, or **input/output operation is blocked.**

- **Dead** (**Terminated**)

  After completing the task, thread **enters into dead or terminated state.**

Starting a thread

Thread thread = new Thread(new **ThreadStart** (ThreadFunc));
//Creates a thread object
// ThreadStart identifies the method that the thread executes when it
//starts

thread.Start();
//starts the thread running

Thread Priorities :
Controls the amount of CPU time that can be allotted to a thread.
ThreadPriority.Highest
ThreadPriority.AboveNormal
ThreadPriority.Normal
ThreadPriority.BelowNormal
ThreadPriority.Lowest

Suspending and Resuming Threads

- Thread.Suspend temporarily suspends a running thread.
- Thread.Resume will get it running again
- Sleep: A thread can suspend itself by calling Sleep.

- **Difference between Sleep and Suspend**

- A thread can call sleep only on itself.

- Any thread can call Suspend on another thread.

Terminating a thread

- **Thread.Abort()** terminates a running thread.
- In order to end the thread , Abort() throws a ThreadAbortException.

- Suppose a thread using SQL Connection ends prematurely, we can close the SQL connection by placing it in the finally block.

```
- SqlConnection conn …….
  try{

  conn.open();

  ….
        .....
   }
   finally{

  conn.close();//this gets executed first before the thread ends.
   }
```

- A thread can prevent itself from being terminated with Thread.ResetAbort.

```
- try{

…
                                                                    }
 catch(ThreadAbortException){

Thread.ResetAbort();
}
```

- Thread.Join()
  - When one thread terminates another, wait for the other thread to end.

Lab Assignment

- Invoking threads
- Passing Parameters

Thread Synchronization

- **Thread Synchronization**:
  - Threads must be coordinated to prevent data corruption.

- **Monitors**
  - Monitors **allow us to obtain a lock** on a **particular object** and use that lock to restrict access to critical section of code.
  - While a thread owns a lock for an object, **no other thread can acquire that lock.**
  - **Monitor.Enter(object)** claims the lock but blocks if another thread already owns it.
  - Monitor.Exit(object) releases the lock.

Monitors

```
Void Method1()
    {
        ….
        Monitor.Enter(buffer);
        try
        {
                critical section;
        }
        finally
        {
                Monitor.Exit(buffer);
        }
    }
```

Calls to Exit are enclosed in finally blocks to ensure that they're executed even when an exception arises.

Lock Keyword

```
lock(buffer)
{
.....
}
```

is equivalent to

```
Monitor.Enter(buffer);
try
{
        critical section;
}
finally
{
        Monitor.Exit(buffer);
}
```

- Makes the code concise.
- Also ensures the presence of a finally block to make sure the lock is released.

Reader/Writer locks

- Prevent concurrent threads from accessing a resource simultaneously.
- Permit multiple threads to read concurrently.
- Prevent overlapping reads and writes as well as overlapping writes.

- Reader function uses :
  - AcquireReaderLock
  - ReleaseReaderLock

- Writer funciotn uses :
  - AcquireWriterLock
  - ReleaseReaderLock

- ReleaseLocks are enclosed in finally blocks to be absolutely certain that they are executed.

Drawback

Threads that need writer locks while they hold reader locks will result in deadlocks.

Solution is UpgradeToWriterLock and DowngradeFromWriterLock methods.

```
rwlock.AcquireReaderLock(Timeout.Infinite)
try{
        // read from the resource guarded by the lock
        ….
        //decide to do write to the resource

        LockCookie cookie = rwlock.UpgradeToWriteLock(Timeout.Infinite)

        try{
        // write to the resource guarded by the lock
        ….
        }
        finally{
                rwlock.DowngradeFromWriterLock(ref cookie);
        }
 }
finally{
        rwlock.ReleaseReaderLock();

}
```

MethodImpl Attribute

- **MethodImpl** Attribute
  - For synchronizing access to entire methods.
  - **To prevent a method from being executed by more than one thread at a time ,**

```
[MehtodImpl(MethodImplOptions.Synchronized)]
Byte[] TransformData(byte[] buffer)
{
……
}
```

Only one thread at a time can enter the method.

Lab Assignment

- Thread Synchronization

Thread Types

- 1. Foreground Thread
- 2. Background Thread

Foreground Thread

- Foreground threads are those threads which **keeps on running to complete its work even if the main thread quits.**
- **Lifespan** of worker thread is **not dependent** on the main thread.
- Worker thread can be alive without main thread**.**

Thread Types

Background Thread

- Background threads are those threads which quits if the main application method quits.
- lifespan of worker thread is dependent on the main thread.
- Worker thread quits if the main application thread quits
- Set “IsBackground" to true to make background thread.

Conclusion

- Using more than one thread, is the most powerful technique available **to increase responsiveness** to the user and process the data necessary to get the job done at almost the same time.

Thread Pool

Thread Pool - Concept

- A collection of pre-configured Threads sitting alive to serve incoming asynchronous task is called ***"ThreadPool"***.
- It is a pool of worker threads.
- ***"System.Threading"*** namespace
- If you have short tasks that require background processing, the managed thread pool is an easy way.
- The ***ThreadPool*** **improves the responsiveness of the application.**
  - **e.g** *Yahoo Mail Login Page*

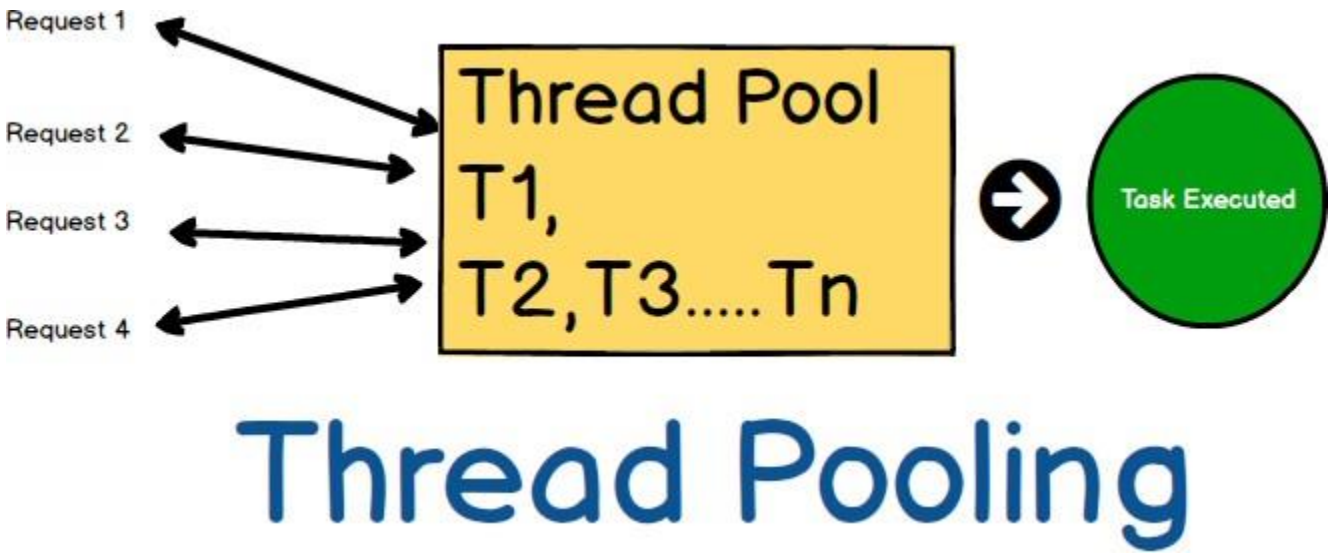
Thread pool characteristics

- Thread pool threads are background threads.
- Each thread uses the **default stack size**, runs at the **default priority**,
- It is in the **multithreaded apartment**.
- **Once** a thread in the thread pool **completes its task**, **it's returned to a queue of waiting threads**.
- After return, it can be reused.

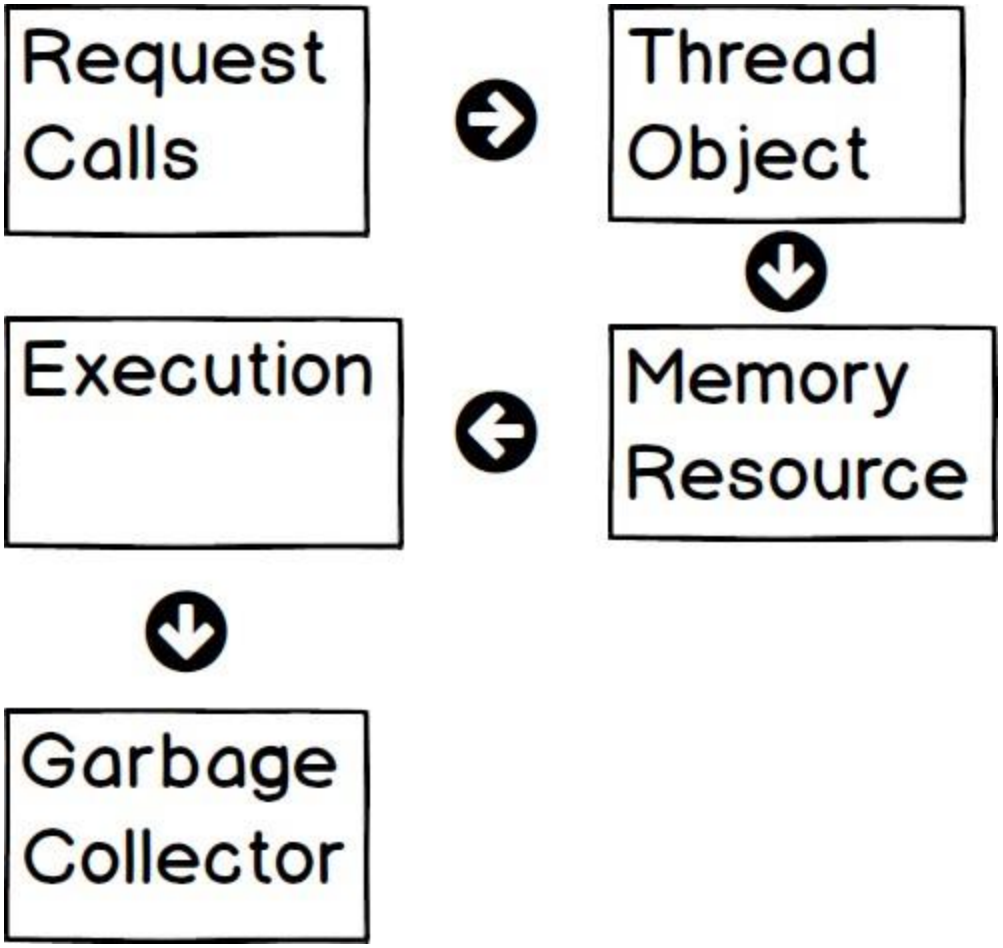
Thread Pool - Functions

- ***QueueUserWorkItem -*** is used to submit the task to the ThreadPool.
- ***SetMaxThreads()"*** and ***"SetMinThreads()"*** methods are used to control the ThreadPool's load

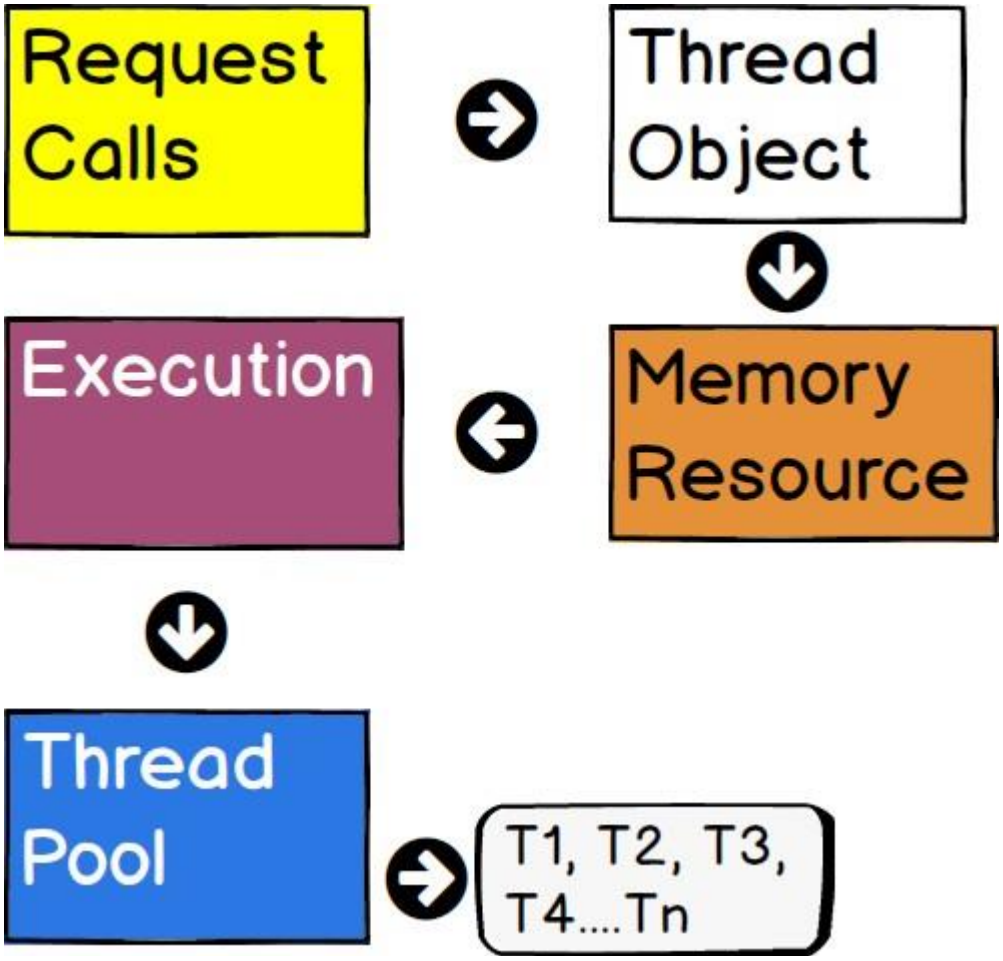
Thread Pool - Concept



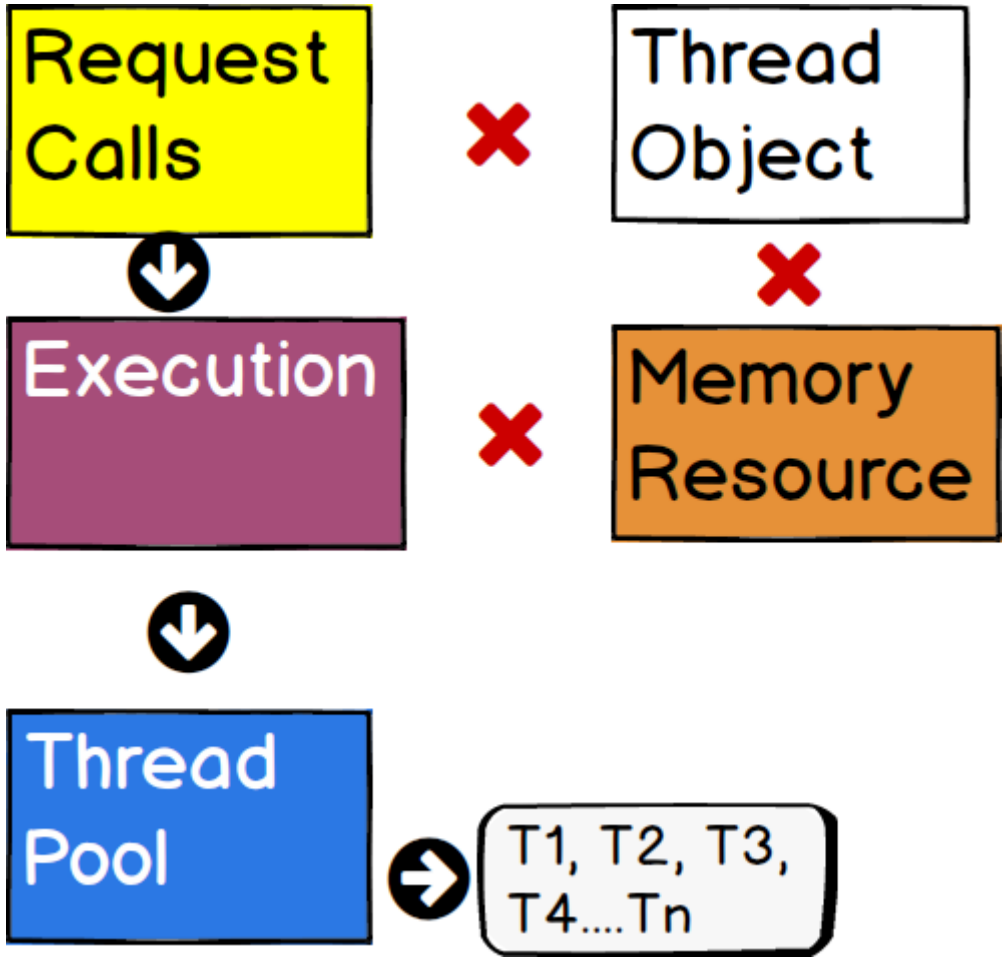
Normal Thread Life Cycle



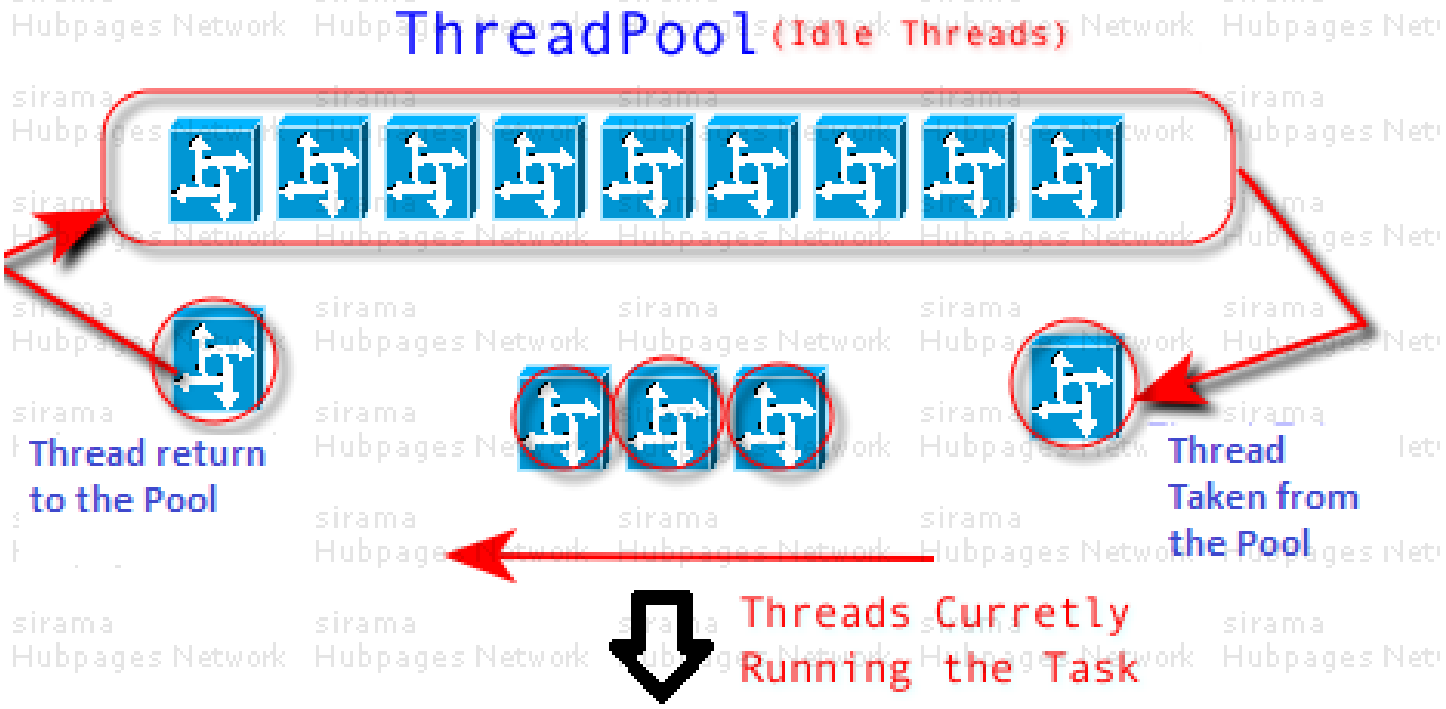
Thread Pool Life Cycle – **first initialization**



Thread Pool Life Cycle



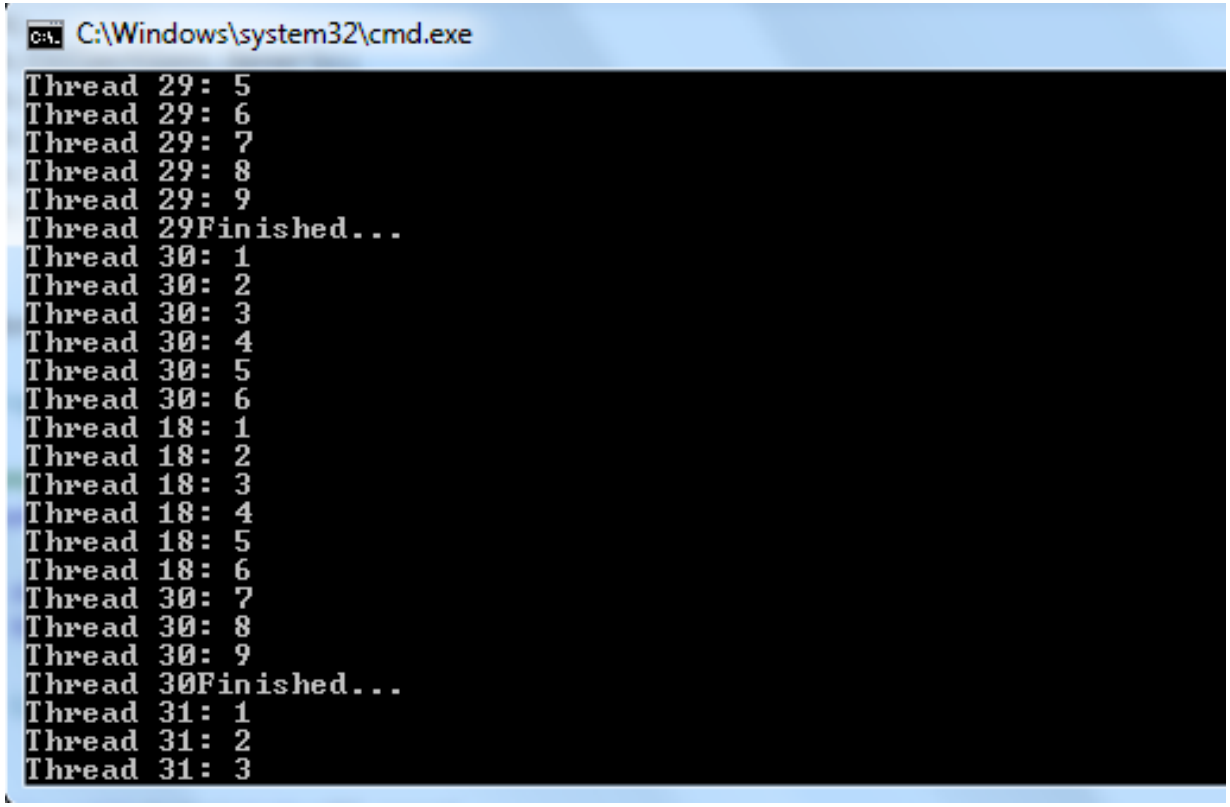
Thread Pool - Concept



ThreadPool *- Example*

Steps

- Using System.Threading;
- Define task/function to invoke through thread
- Create threadpool and queue threads


C:\Windows\system32\cmd.exe
Thread 29: 5
Thread 29: 6
Thread 29: 7
Thread 29: 8
Thread 29: 9
Thread 29Finished...
Thread 30: 1
Thread 30: 2
Thread 30: 3
Thread 30: 4
Thread 30: 5
Thread 30: 6
Thread 18: 1
Thread 18: 2
Thread 18: 3
Thread 18: 4
Thread 18: 5
Thread 18: 6
Thread 30: 7
Thread 30: 8
Thread 30: 9
Thread 30Finished...
Thread 31: 1
Thread 31: 2
Thread 31: 3

Demo