Multithreading

Objectives

On completion of the session you will be able to

- Define multitasking and multithreading
- List different thread states in the life cycle of a thread
- Write a simple multithreaded application
- Assign priorities to the threads
- Use Synchronization to access a resource by multiple threads
- Implement synchronization techniques like automatic, synchronized code regions and manual synchronization
- Use `Mutex` class to synchronize threads across inter process boundaries.
- Use `Timer` class to execute a method at specific intervals
- Use ThreadPool to increase efficiency of an application

Multitasking

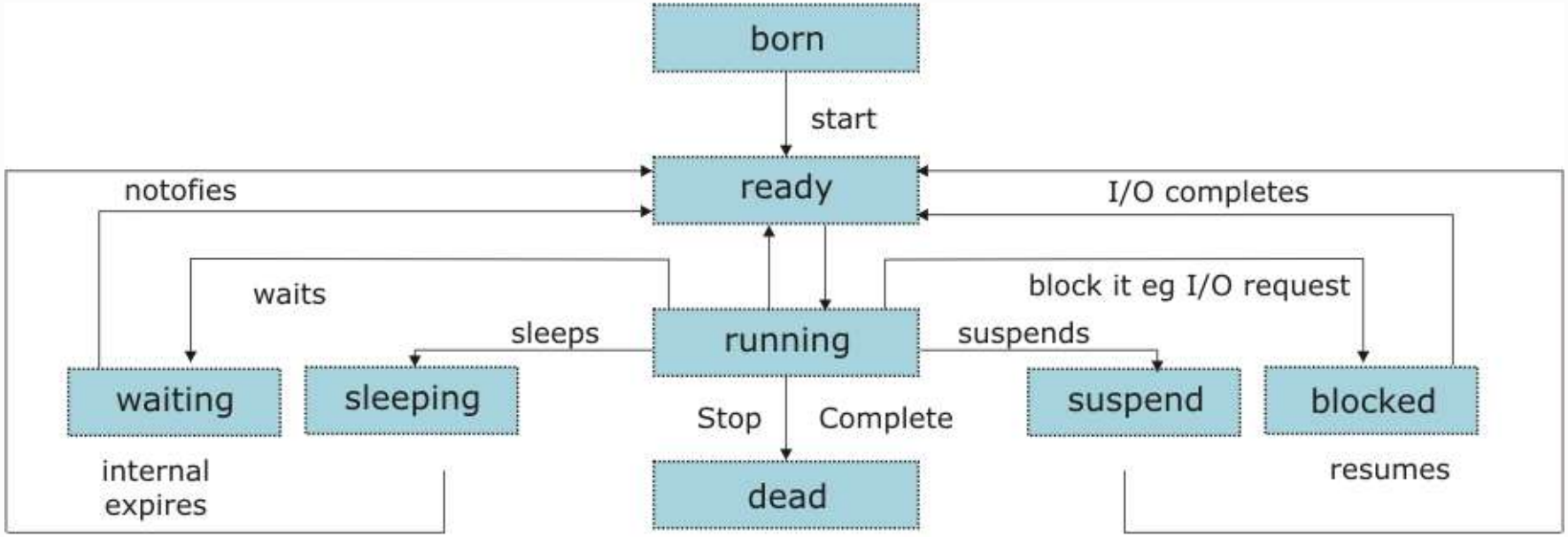
- Ability to have more than one program working at the same time.
- The objective is to utilize the idle time of the CPU.
- Multitasking can be done in 2 ways:
  - Non Pre-emptive
  - Pre-emptive

Multithreading

- Multithreading is the ability of an operating system to execute different parts of a program simultaneously
- When to use multithreading
  - Performing operations that take a large amount of time.
  - Prioritization of the tasks .
  - Application has to wait for some event to occur.

Thread

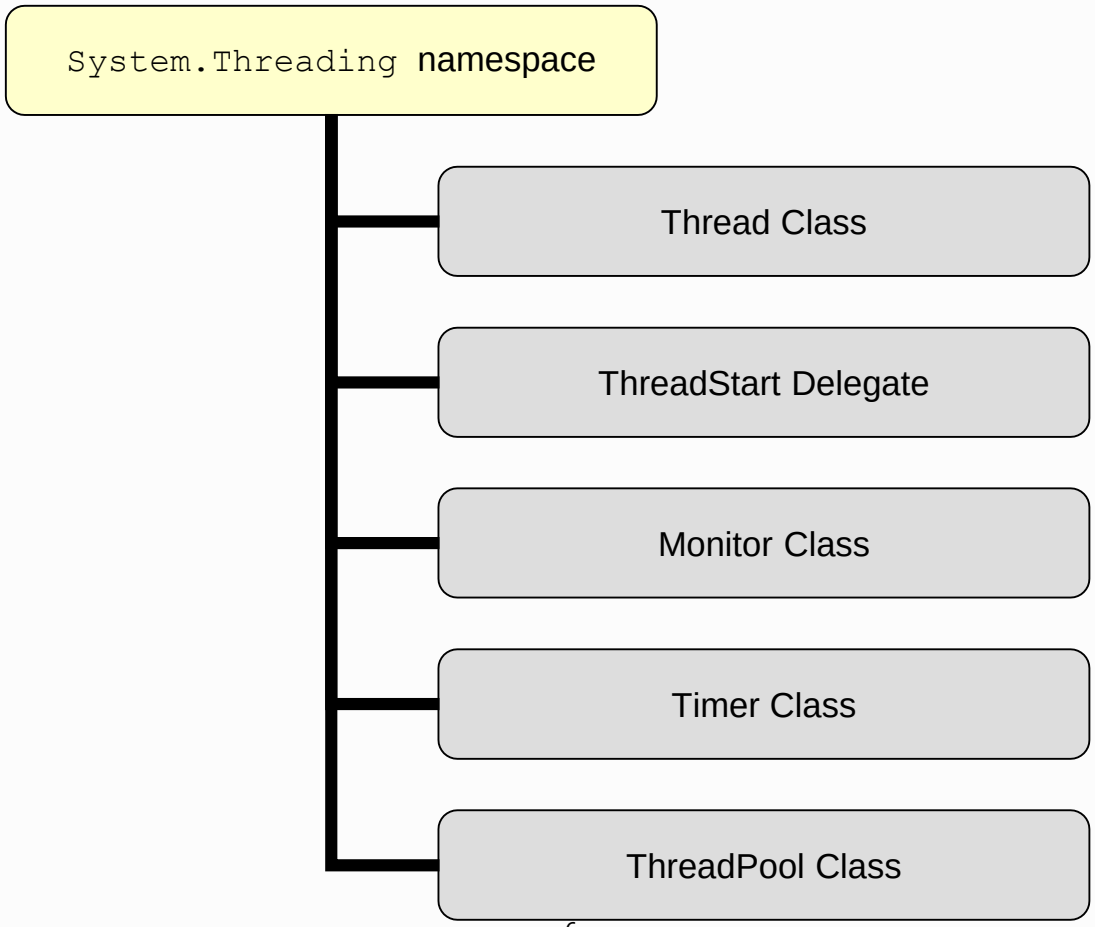
- Thread is a path of Execution in a running application.



Thread Life Cycle

`System.Threading` namespace

- Provides a number of classes and other types to support multithreading



Thread Class

```
class ThreadDemo {
private string fName,lName;
//parameterized  constructor defined here

public void Run(){
  Console.WriteLine("First Name is  " + fName);
  Thread.Sleep(500);
  Console.WriteLine("Last Name is  " + lName);
}
}
```

```
class TestThreading{
static void Main() {
  ThreadDemo t1 = new ThreadDemo ("Rahul", "Dravid");
  ThreadDemo t2 = new ThreadDemo ("Sourav", "Ganguly");
  Thread first = new Thread(new ThreadStart(t1.Run));
  Thread second = new Thread(new ThreadStart(t2.Run));
  first.IsBackground = true;
  second.IsBackground = true;
  first.Start();
  second.Start();
 }
}
```

Thread Scheduling

- Threads are scheduled for execution using priority
- Thread priorities are defined as `ThreadPriority` enumeration
  - `Highest`
  - `AboveNormal`
  - `Normal`
  - `BelowNormal`
  - `Lowest`

```
Thread UIThread=new Thread(new ThreadStart(StartMethod));
UIThread.Name = "Worker";
UIThread.Priority = ThreadPriority.AboveNormal;
UIThread.Start();
```

Why Thread Synchronization?

- Controlled access to resources needs to be given.
- Provided by lock on the object to prevent second thread modifying it.

```
enum Operation{credit,debit}
class AccountUser {
. . .
BankAccount ac;
public void run()
{lock(ac){
 if(operation == Operation.debit)
    ac.Debit(amt);
  else
    ac.Credit(amt);
     }
 }
}
```

Thread Synchronization

- Common problems of multithreading
  - Deadlocks
    - Avoided by managing timeouts using threading classes like Monitor class
  - Race condition
    - Avoided by using `InterLocked` class

- Different strategies to synchronize access to instance and static methods and instance fields
  - Synchronized contexts
  - Synchronized code regions
  - Manual synchronization

Synchronization Context

- `[Synchronization]` enables simple, automatic synchronization for instances of the class.
  - Static fields and methods are not protected from concurrent access by multiple threads

```
using System.Runtime.Remoting.Contexts;
. . .
[Synchronization]
class BankAccount
{
static int sCount = 0; //multiple threads can access
int iCount = 0;//only 1 thread can access at a time
 public void Debit (float amt){
       //… only one thread can access at a time
    }
}
```

Synchronized Code regions

- Restricted access to a block of code, commonly called a critical section is required.
- `Monitor` class
  - controls access to these objects by granting a lock on an object to a single thread.
  - Exposes methods like `Enter(), Exit(), Wait() Pulse() and PulseAll()`

```
public void Credit()
{
   . . .
Monitor.Enter(x);
try {   . . .
    }
finally {
 Monitor.Exit(x);
}
```

Manual Synchronization

- Access to a variable shared by multiple threads is synchronized manually using `InterLocked` class.
- InterLocked class exposes methods like `Add()`,`Increment()`,`Decrement()`,`CompareExchange()` , etc. to perform atomic operations on such variables.

```
class AccountUser { static int count;
. . .
public AccountUser(){
Interlocked.Increment(ref count);
}
~AccountUser(){
Interlocked.Decrement(ref count);
}
}
```

Inter-process Synchronization

- Use `Mutex` class
  - to synchronize between threads ross processes
- Use `ReaderWriterLock`
  - in scenarios with a single “writer” and multiple “readers”

```
private static Mutex mutex = new Mutex();
private static void UseResource()
{
 mutex.WaitOne();
 // Place code to access resources her
 Thread.Sleep(500);
 Console.WriteLine("{0} is leaving the protected area\r\n",
                   Thread.CurrentThread.Name);

 mutex.ReleaseMutex();
 }
```

`Timer` Class

- Used to periodically execute a method

```
public class Test
{
public static void OnTimer()
   {
    // background task .....
   }

Public static void Main()
   {
    TimerCallback dcallback = new TimerCallback(Test.
                                                OnTimer);

    long dTime = 15 ; // wait before the first tick (in ms)
    long pTime = 130 ; // timer during subsequent invocations (in ms)

    Timer timer = new Timer(dcallback, null,dTime, pTime) ;
    // do some thing with the timer object
    ...
    timer.Dispose() ;
   }
}
```

Thread Pool

- Provides a pool of threads that can be used to post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.
- Used to improve efficiency
- Thread Pooling should not be used when
  - Need a task to have a particular priority
  - Have a task that might run for a long time
  - Need to place threads into a single-threaded apartment
  - Need a stable identity to be associated with the thread

Think before Multithreading

- Keeping track of and switching between threads consumes memory resources and CPU time.
- Programming with multiple threads can be complex.
- Shared resources utilization problem.

Quick Recap …

- Multithreading is the ability of an operating system to execute different parts of a program simultaneously.
- `Thread` class is used to create a simple thread.
- Threads are scheduled for execution using priority
- controlled access to resources can be given using Synchronization.
- Synchronization can be implemnted using lock statement, `Monitor` class, `Interlocked` class, `Mutex` class and by applying `MethodImplAttribute`.
- `Timer` class is used to execute method at regular intervals.
- `ThreadPool` class provides a pool of worker threads that are managed by the system.