Threads

PRATIAN
TECHNOLOGIES

PRATIAN TECHNOLOGIES

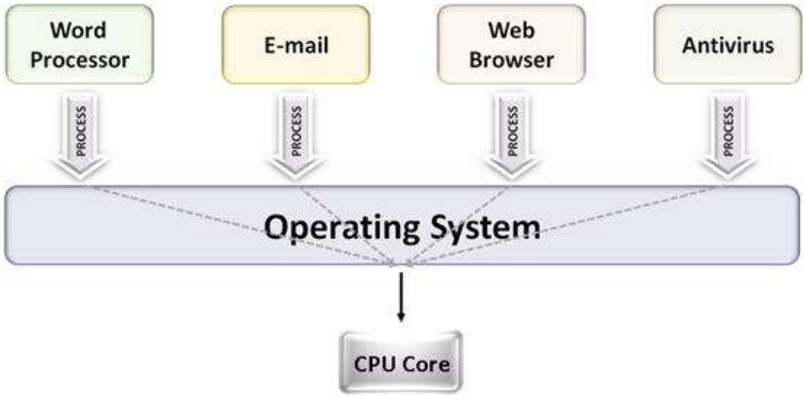
Objectives

- Introduction to Multithreading
  - What?
  - Why?
  - How?
- Foreground and Background Threads
  - What?
- Thread Synchronization Techniques
  - Various built-in classes in .NET framework
- Monitor Class
  - What?
  - Why?
  - How?

PRATIAN TECHNOLOGIES

Multitasking

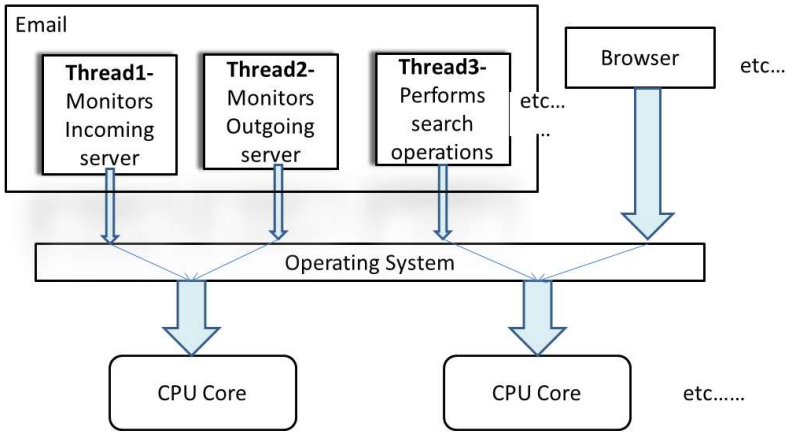
- Every application executes in its own address space or process.
  - E.g. MS Word (Creating a document) or WinAmp (listening songs)
- A processor can execute applications in multiple processes concurrently.
  - Ability to execute different applications or tasks by the processor is called multitasking
    - The scheduling algorithm of the operating system decides it
      - Based on time slicing – pre-emptive
- Operating System switches between the applications running in different processes
  - Each process has its own data which needs to be stored temporarily before switching to other process. This is called context.
  - Context switch is resource heavy



PRATIAN TECHNOLOGIES

Multithreading

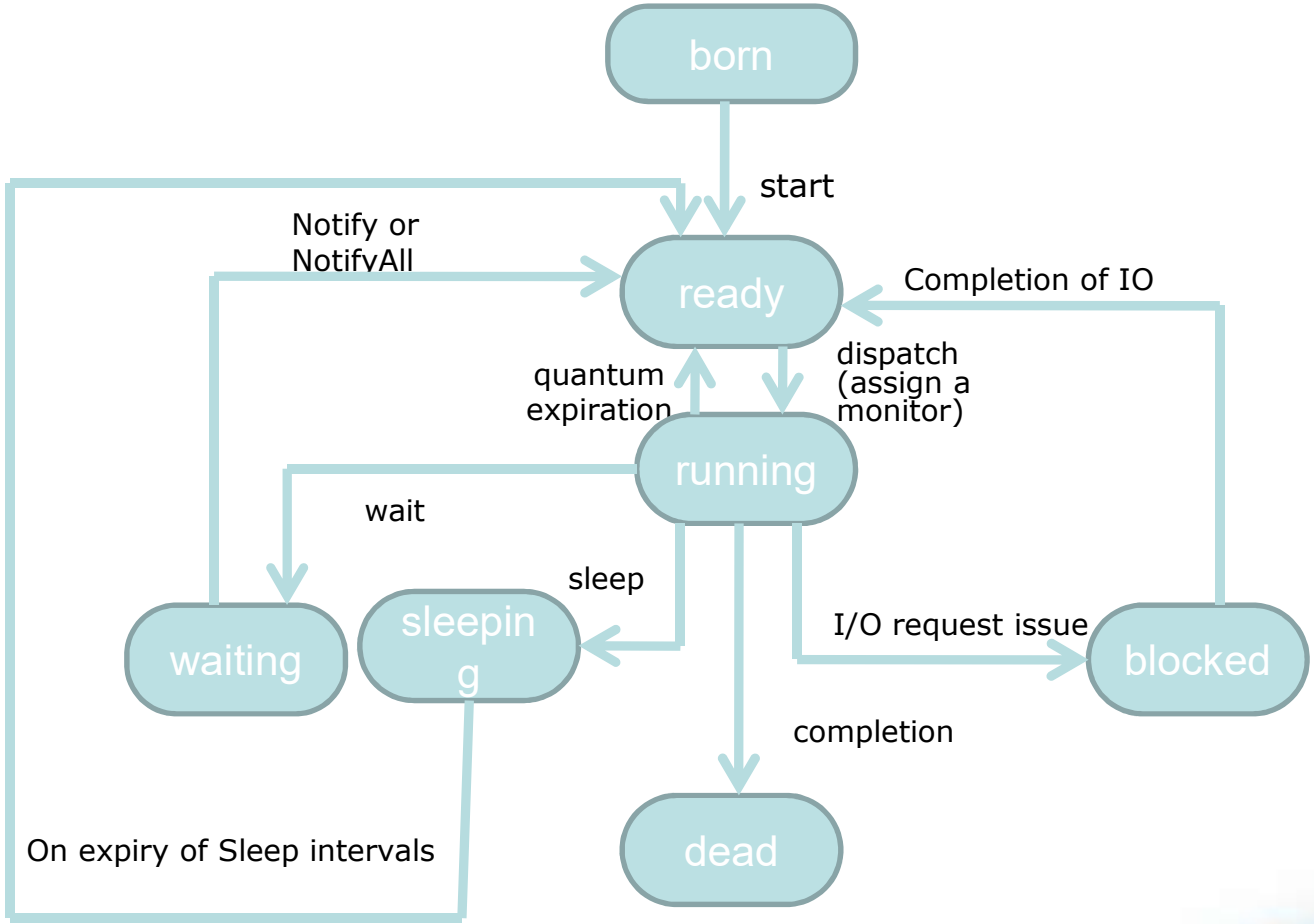
- CPU is idle between the context switch
- Consider a scenario of a single application - MS Word
  - Spell check, grammar check runs while the editing is being done.
- Ability of OS to run different parts of the same program or application concurrently is called multithreading.
  - Each part of the program is called “Thread”.
  - Thread can execute separate tasks independently with each task not dependent on other.
  - Two or more threads in an application can share some data.
    - Synchronization of threads is required in such a scenario as each thread can change the shared data.



PRATIAN TECHNOLOGIES

Thread Life Cycle

- Thread is a part of execution of an application. It has a life-cycle.



PRATIAN TECHNOLOGIES

Thread class

- *System.Threading* namespace provides a set of classes and other types that support in creating multithreaded applications.
- Thread Class
  - Represents a thread, helps to create it, control it, set its priority and get its status .

Method Name	Description
static void Sleep(int millisecondsTimeout)	Suspends the current thread for a particular time interval.
void Start()	Causes the state of thread to be in running state. It is an overloaded method.
void Join()	Blocks the calling thread until the thread terminates
void Abort()	Raises a *ThreadAbortException* in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
CurrentThread	property; gets the currently running thread
Name	gets or sets the name of the thread
ThreadPriority	Gets or sets a value indicating the scheduling priority of a thread.

PRATIAN TECHNOLOGIES

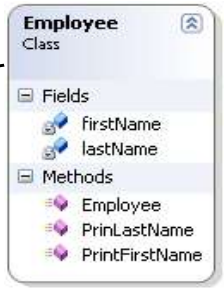
Thread class - example

A Simple Example of Multithreading

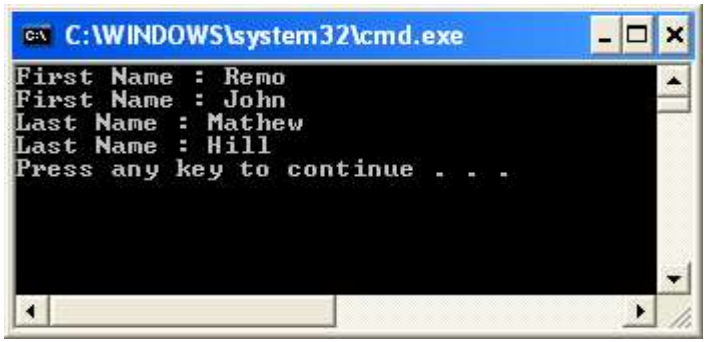
Call to the methods on threads

Normal method call from Main

```
static void Main(string[] args)
{
    Employee employee1 = new Employee("Remo", "Mathew");
    ThreadStart delegate1 = new ThreadStart(employee1.PrintFirstName);
    Thread firstThread = new Thread(delegate1);

    ThreadStart delegate2 = new ThreadStart(employee1.PrintLastName);
    Thread secondThread = new Thread(delegate2);

    firstThread.Start();
    secondThread.Start();


    Employee employee2 = new Employee("John", "Hill");
    employee2.PrintFirstName();
    employee2.PrintLastName();

}
```



Output – unpredicted depending on the scheduling algorithm of OS



PRATIAN TECHNOLOGIES

What are Foreground and Background Threads?

- Thread can be either foreground or background thread.
  - Foreground thread keep the execution environment running.
  - Background thread run in the background. It stops once the foreground thread stops.
- *IsBackround* property of a thread determines whether the thread is background.
  - By default the thread is foreground.
- An application runs on the main thread.
  - If some tasks are time consuming and their execution and output does not affect other threads in the application, such threads are made as background thread.

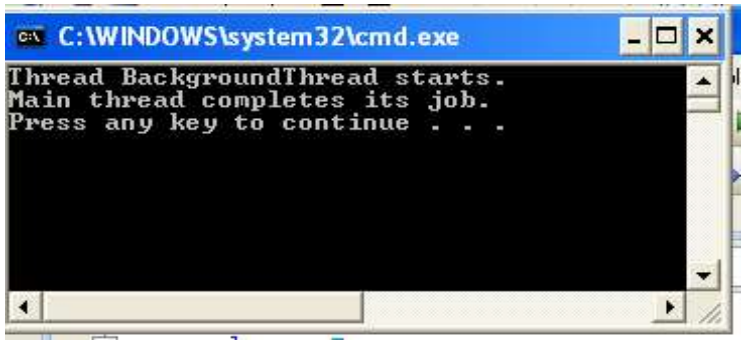
PRATIAN TECHNOLOGIES

Foreground and Background Threads - example

```
static void TaskOnThread()
{
    Console.WriteLine("Thread {0} starts.", Thread.CurrentThread.Name);
    Thread.Sleep(3000);
    Console.WriteLine("Thread {0} ends.", Thread.CurrentThread.Name);

}
```

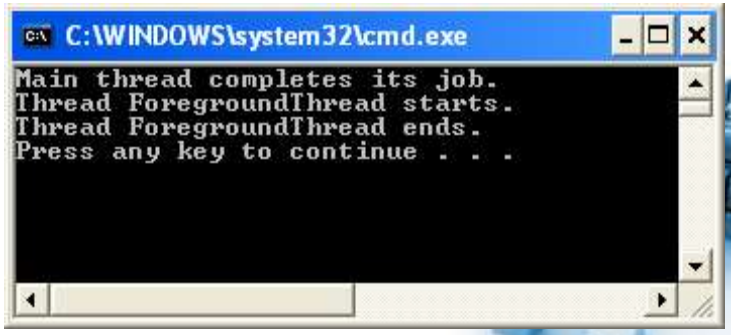
```
Thread t1 = new Thread(TaskOnThread);
t1.Name = "BackgroundThread";
t1.IsBackground = true; //make it as background thread
t1.Start();
Console.WriteLine("Main thread completes its job.");
```

I




```
Thread t1 = new Thread(TaskOnThread);
t1.Name = "ForegroundThread";
t1.IsBackground = false; //foreground thread
t1.Start();
Console.WriteLine("Main thread completes its job.");
```

II



PRATIAN TECHNOLOGIES

Why Synchronization?

- Two or more threads sharing the same data.
  - Data becomes inconsistent.
- Two important problems
  - Race Condition
    - Two threads try to reach a particular block of code first.
  - Deadlock
    - One thread tries to lock a resource which the other thread has already locked.
- Synchronization of threads should be done to avoid the problems.

PRATIAN TECHNOLOGIES

Thread Synchronization Techniques

- Synchronization of threads is required when two or more threads share same data.
- Thread Synchronization techniques
  - Synchronization context
    - *[Synchronization]* attribute is used
  - Synchronized Code Regions
    - Using Monitor class
  - Using *lock* keyword in C#
    - Statements which need synchronization are put in a block
  - Manual Synchronization
    - Using *Interlocked* class
- Inter – process Synchronization
  - Using *Mutex* class
- *MethodImplAttribute*
  - Synchronizes an entire method in one single command

PRATIAN TECHNOLOGIES

Synchronizing Code Regions - *Monitor* class

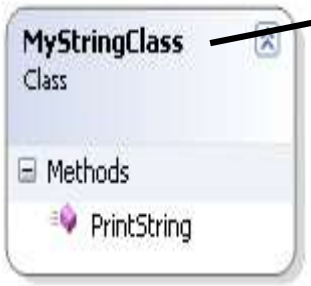
- This class helps to control lock on an object for a particular region of the code for a single thread.
  - Region is called critical section.
  - No other thread can access the critical section till the lock is released.
  - Only reference types can be locked and not the value types.
    - Value type have to be boxed if used
- Monitor class maintains information
  - Reference to the thread that holds the lock
  - Reference to the threads waiting in the queue to obtain the lock
  - Reference to the queue itself
- Methods of Monitor class
  - *Enter() , TryEnter()*
  - *Wait()*
  - *Pulse(), PulseAll()*
  - *Exit()*

PRATIAN TECHNOLOGIES

Example of using *Monitor* class

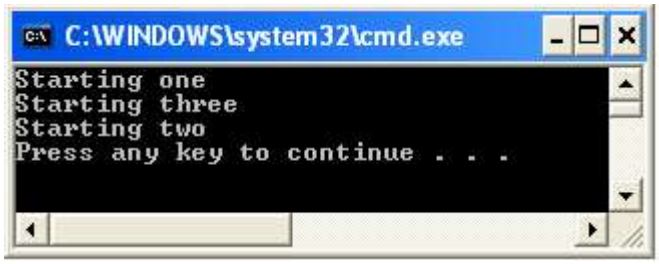
```
class CustomThread
{
    MyStringClass _shared;
    string myString;

    public CustomThread(MyStringClass shared, string str)
    {
        this._shared = shared;
        this.myString = str;
    }
    public void Run()
    {
        Monitor.Enter(_shared);
        try
        {
            _shared.PrintString(myString);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        Monitor.Exit(_shared);
    }
}
```

Synchronizing using Monitor class

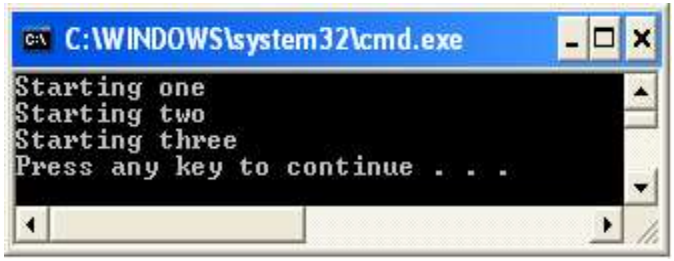



```
MyStringClass sharedObject = new MyStringClass();

CustomThread first = new CustomThread(sharedObject,"one");
CustomThread second = new CustomThread(sharedObject, "two");
CustomThread third = new CustomThread(sharedObject, "three");

Thread thread1 = new Thread(new ThreadStart(first.Run));
Thread thread2 = new Thread(new ThreadStart(second.Run));
Thread thread3 = new Thread(new ThreadStart(third.Run));
```

O/P if Monitor class is not used (inconsistent O/P)



O/P when Monitor class is used (synchronization of code region)



PRATIAN TECHNOLOGIES

lock keyword

- *Lock k*eyword helps to control lock on a particular object for a particular section of code.
  - Uses *Enter()* and *Exit()* methods
  - Simplified version of *Monitor* class

```
class SampleClass
{
    private object object1 = new object();
    private int i;
    public void Method1()
    {
      lock (object1)
      {
          // code which needs locking
          // . . .
          // lock is released automatically
           //after the last executable statement
      }
    }
}
```

PRATIAN TECHNOLOGIES

Inter-process synchronization - *Mutex* Class

- A synchronization technique that is used for inter-process synchronization.
  - Gives exclusive access to a shared resource to only one thread. The other thread that also needs access is suspended till the first thread releases the mutex.
- Mutex are of two types
  - Local Mutex
    - Mutex that exists only within the process
    - Unnamed mutex
  - System Mutex
    - Mutex that is visible throughout operating system i.e. all processes
    - Named mutex

PRATIAN TECHNOLOGIES

Mutex Class - Example

```
static Mutex mutex = new Mutex();
void WriteToFile()
{
    mutex.WaitOne();

    String ThreadName = Thread.CurrentThread.Name;
    Console.WriteLine("{0} using resource", ThreadName);

    // code to to write to a file

    Console.WriteLine("{0} releasing resource", ThreadName);

    mutex.ReleaseMutex();
}
```

I

II

```
static void Main(string[] args)
{
    bool appInstance;

    using (Mutex mutex = new Mutex(true, "App", out appInstance))
    {
        if (!appInstance)
        {
            Console.WriteLine("Application is already open");

            //return;
        }
        else
        {
            Console.WriteLine("Code in the application executing...");

        }
       Console.ReadKey();
    }
}
```