Concurrency Programming in .NET

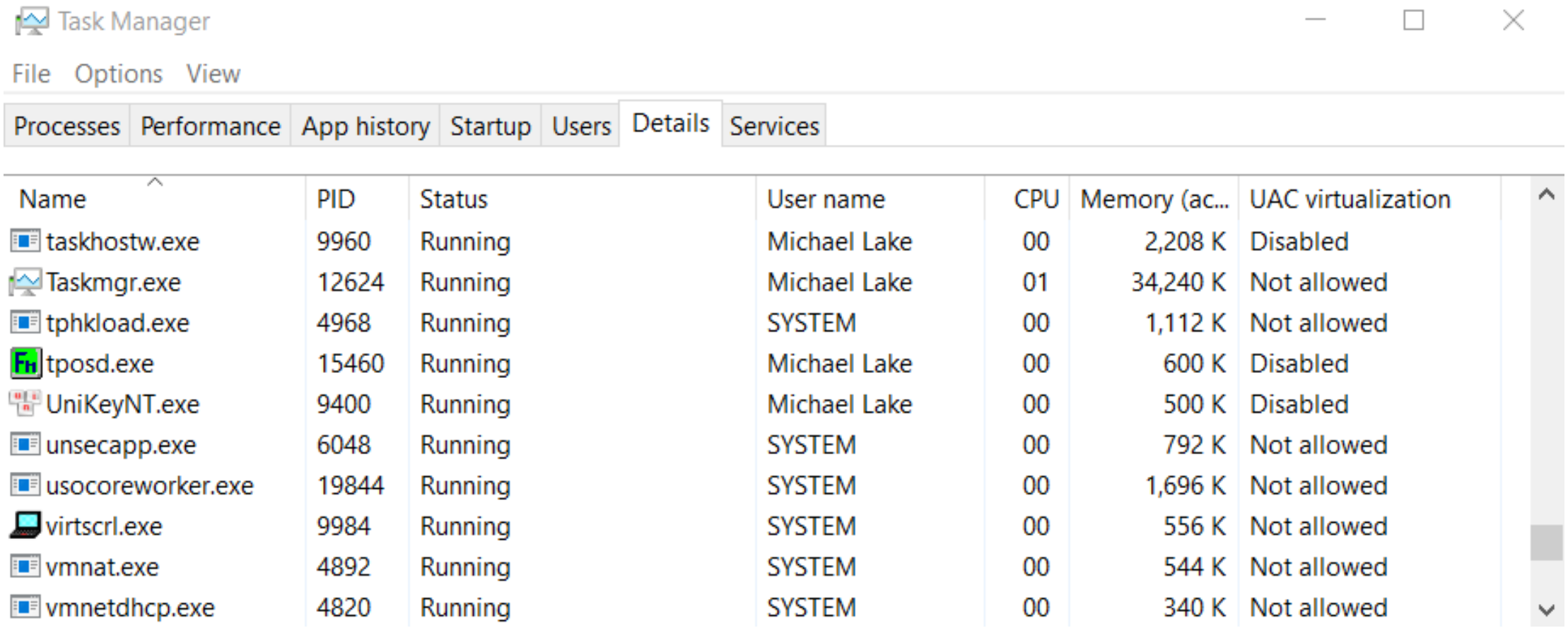
Objectives

- Overview Concurrency Programming
- Overview about MultiThreading
- Explain about Synchronization: lock and Monitor
- Explain and Demo about The Issue of Concurrency: Race Conditions
- Explain about ThreadPool and TimerCallback
- Demo MultiThreading application with C#
- Demo Synchronization in MultiThreading application
- Demo ThreadPool and TimerCallback

Processes and Multi Processing System

- A **process** has a self-contained execution environment
- A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space
- Multi Processing/ Multi Tasking System: System allows many processes executing concurrently
- A **thread** is a path of execution within an executable application
- By implementing additional threads, we can build more responsive (*but not necessarily faster executing*) applications

Processes and Multi Processing System

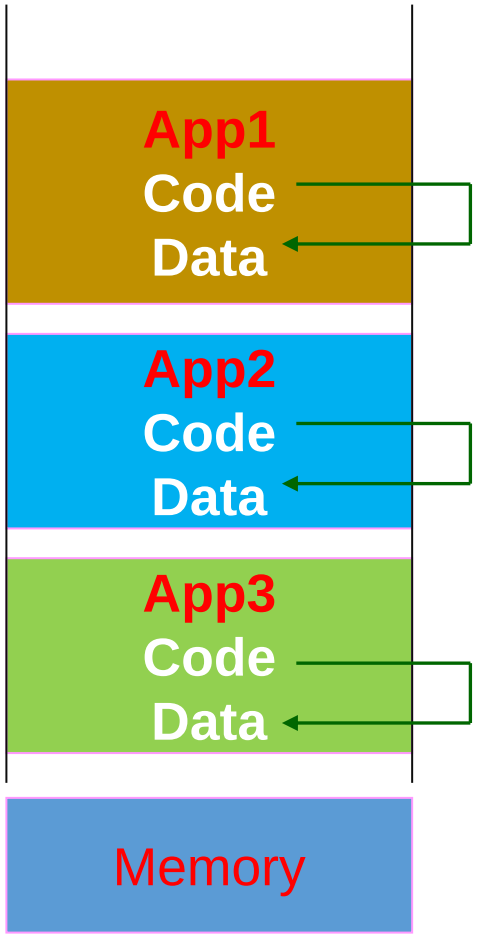


Name	PID	Status	User name	CPU	Memory (ac...	UAC virtualization
taskhostw.exe	9960	Running	Michael Lake	00	2,208 K	Disabled
Taskmgr.exe	12624	Running	Michael Lake	01	34,240 K	Not allowed
tphkload.exe	4968	Running	SYSTEM	00	1,112 K	Not allowed
tposd.exe	15460	Running	Michael Lake	00	600 K	Disabled
UniKeyNT.exe	9400	Running	Michael Lake	00	500 K	Disabled
unsecapp.exe	6048	Running	SYSTEM	00	792 K	Not allowed
usocoreworker.exe	19844	Running	SYSTEM	00	1,696 K	Not allowed
virtscrl.exe	9984	Running	SYSTEM	00	556 K	Not allowed
vmnat.exe	4892	Running	SYSTEM	00	544 K	Not allowed
vmnetdhcp.exe	4820	Running	SYSTEM	00	340 K	Not allowed

Concurrency in Operating System

- Concurrency is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel
- The running process threads always communicate with each other through shared memory or message passing. Concurrency results in sharing of resources result in problems like deadlocks and resources starvation
- Concurrency helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput

Concurrency in Operating System



App	Code Addr	Duration (mili sec)	CPU
App1	10320	15	1
App2	40154	17	2
App3	80166	22	1
…	…	…	…
…	…	…	…

Time-slicing Mechanism

A method of allocating CPU time to individual process in a priority schedule

Advantages of Concurrency

- **Running of multiple applications**: It enable to run multiple applications at the same time
- **Better resource utilization**: It enables that the resources that are unused by one application can be used for other applications
- **Better average response time**: Without concurrency, each application has to be run to completion before the next one can be run
- **Better performance**: It enables the better performance by the operating system. When one application uses only the processor and another application uses only the disk drive then the time to run both applications concurrently to completion will be shorter than the time to run each application consecutively

Issues of Concurrency

- **Non-atomic**: Operations that are non-atomic but interruptible by multiple processes can cause problems
- **Race conditions**: A race condition occurs of the outcome depends on which of several processes gets to a point first
- **Blocking**: Processes can block waiting for resources. A process could be blocked for long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable
- **Starvation**: It occurs when a process does not obtain service to progress
- **Deadlock**: It occurs when two processes are blocked and hence neither can proceed to execute

.NET Application Domains

- Under the .NET executables are not hosted directly within a Windows process that executables are hosted by a logical partition within a process called an *application domain*
- There are several benefits as follows :
  - AppDomains are a key aspect of the OS-neutral nature of the .NET Core platform, given that this logical division abstracts away the differences in how an underlying OS represents a loaded executable
  - AppDomains are far less expensive in terms of processing power and memory than a full-blown process. Thus, the CoreCLR is able to load and unload application domains much quicker than a formal process and can drastically improve scalability of server applications

.NET Application Domains

- Under the .NET platform, there is not a direct one-to-one correspondence between application domains and threads that a given AppDomain can have numerous threads executing within it at any given time
- To programmatically gain access to the AppDomain that is hosting the current thread, using the static *Thread.GetDomain()* method
- A single thread may also be moved into a particular execution context at any given time, and it may be relocated within a new execution context at the whim of the CoreCLR
- The CoreCLR is the entity that is in charge of moving threads into (and out of) execution contexts

Enumerating Assemblies In AppDomain Demo

```
using System;
using System.Reflection;
namespace DemoEnumeratingLoadedAssemblies {
    class Program{
        static void Main(string[] args){
            //Get access to the AppDomain for the current thread.
            AppDomain defaultAD = AppDomain.CurrentDomain;
            //Get all loaded assemblies in the efault AppDomain.
            Assembly[] loadedAssemblies = defaultAD.GetAssemblies();
            Console.WriteLine("The assemblies loaded in {0}",defaultAD.FriendlyName);
            foreach (Assembly a in loadedAssemblies) {
                Console.WriteLine($"--Name, Version: {a.GetName().Name}:{a.GetName().Version}");
            }
            Console.ReadLine();
        }
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Programming\DemoEnumeratingLoadedAssemblies\bin\Del
The assemblies loaded in DemoEnumeratingLoadedAssemblies
--Name, Version: System.Private.CoreLib:5.0.0.0
--Name, Version: DemoEnumeratingLoadedAssemblies:1.0.0.0
--Name, Version: System.Runtime:5.0.0.0
--Name, Version: System.Console:5.0.0.0
```

Interacting with Processes Using .NET

- The System.Diagnostics namespace defines a number of types that allow you to programmatically interact with processes and various diagnostic-related types such as the system event log and performance counters

Process-Centric Types of the System.Diagnostics Namespace	Description
Process	Provides access to local and remote processes and enables you to start and stop local system processes.
ProcessModule	Represents a.dll or .exe file that is loaded into a particular process.
ProcessModuleCollection	Provides a strongly typed collection of ProcessModule objects.
ProcessStartInfo	Specifies a set of values that are used when you start a process.
ProcessThread	Represents an operating system process thread.
ProcessThreadCollection	Provides a strongly typed collection of ProcessThread objects.

Interacting with Processes Using .NET

- The System.Diagnostics.Process class allows us to analyze the processes running on a given machine (local or remote) and also provides members to programmatically start and terminate processes, view (or modify) a process's priority level, and obtain a list of active threads and/or loaded modules within a given process

Properties of the Process Type	Description
ExitTime	Gets the time that the associated process exited
Handle	Gets the native handle of the associated process
Id	Gets the unique identifier for the associated process
MachineName	Gets the name of the computer the associated process is running on
Modules	Gets the modules that have been loaded by the associated process
StartTime	Gets the time that the associated process was started

Interacting with Processes Using .NET

Methods of the Process Type	Description
CloseMainWindow()	Closes a process that has a user interface by sending a close message to its main window
GetCurrentProcess()	Gets a new Process component and associates it with the currently active process
GetProcesses()	Creates a new Process component for each process resource on the local computer
Kill()	Immediately stops the associated process
Start()	Starts (or reuses) the process resource that is specified by the StartInfo property of this Process component and associates it with the component.

- More **Processes class**
  https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process?view=net-5.0

Enumerating Running Processes Demo

```
using System;
using System.Diagnostics;
using System.Linq;
class Program{
    static void Main(string[] args) {
        int no = 1;
        string info;
        // Get all the processes on the local machine, ordered by PId.
        var runningProcs = from proc in Process.GetProcesses(".")
        orderby proc.Id
        select proc;
        // Print out Pid and Name of each process.
        foreach (var p in runningProcs){
            info = $"#{no++}. PID: {p.Id}\tName: {p.ProcessName}";
            Console.WriteLine(info);
        }
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Programming\
#1. PID: 0      Name: Idle
#2. PID: 4      Name: System
#3. PID: 8      Name: svchost
#4. PID: 96     Name: Registry
#5. PID: 388    Name: smss
#6. PID: 420    Name: conhost
#7. PID: 436    Name: svchost
.....
.....
#269. PID: 19788        Name: AppVShNotify
#270. PID: 19812        Name: sihost
#271. PID: 19844        Name: usocoreworker
#272. PID: 19936        Name: POWERPNT
#273. PID: 20100        Name: msedge
#274. PID: 20400        Name: fontdrvhost
```

System.Threading Namespace

- The System.Threading namespace provides a number of types that enable the direct construction of multithreaded applications
- It provides types that allow us to interact with a particular CoreCLR thread, this namespace defines types that allow access to the CoreCLR-maintained **thread pool**, a simple (non-GUI-based) **Timer class**, and numerous types used to provide synchronized access to shared resources

System.Threading.Thread Class

- The most primitive of all types in the System.Threading namespace is **Thread**
- It represents an object-oriented wrapper around a given path of execution within a particular AppDomain
- It also defines several methods (both static and instance level) that allow us to create new threads within the current AppDomain, as well as to suspend, stop, and destroy a particular thread

System.Threading.Thread Class

```
using System.Threading;
using static System.Console;
namespace DemoStatistics {
    class Program {
        static void Main(string[] args){
            // Obtain and name the current thread.
            Thread primaryThread = Thread.CurrentThread;
            primaryThread.Name = "ThePrimaryThread";
            WriteLine($"ID of current thread: { primaryThread.ManagedThreadId}");
            WriteLine($"Thread Name: {primaryThread.Name}");
            WriteLine("Has thread started?: {primaryThread.IsAlive}");
            WriteLine("Priority Level: {primaryThread.Priority}");
            WriteLine("Thread State: {primaryThread.ThreadState}");
            ReadLine();
        }
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Program
ID of current thread: 1
Thread Name: ThePrimaryThread
Has thread started?: True
Priority Level: Normal
Thread State: Running
```

Manually Creating Secondary Threads

- **Steps to Create a Thread**

1) Create a method to be the entry point for the new thread

2) Create a new ParameterizedThreadStart (or ThreadStart) delegate, passing the address of the method defined in step 1 to the constructor

3) Create a Thread object, passing the ParameterizedThreadStart/ThreadStart delegate as a constructor argument

4) Establish any initial thread characteristics (name, priority, etc.)

5) Call the Thread.Start() method. This starts the thread at the method referenced by the delegate created in step 2 as soon as possible

Working with the ThreadStart Delegate

```
//Step 01
public class Printer{
    public void PrintNumbers(){
        // Display Thread info.
        Console.WriteLine($"{Thread.CurrentThread.Name} is executing PrintNumbers()");
        // Print out numbers.
       for (int i = 1; i <= 5; i++){
            Console.WriteLine($"Second thread: {i}");
            Thread.Sleep(2000);
        }
        Console.WriteLine();
    }
}
```

Working with the ThreadStart Delegate

```
class Program {
    static void Main(string[] args){
        Thread primaryThread = Thread.CurrentThread;
        primaryThread.Name = "Primary";
        Console.WriteLine($"{Thread.CurrentThread.Name} is executing Main()");
        Printer p = new Printer(); //Step 02
        Thread backgroundThread = new Thread(new ThreadStart(p.PrintNumbers)); //Step 03
        backgroundThread.Name = "Secondary";  //Step 04
        backgroundThread.Start();  //Step 05
        // Do some additional work.
        for (int i = 1; i <= 5; i++) {
            Console.WriteLine($"Main thread  : {i}");
            Thread.Sleep(1000);
        }
        Console.WriteLine("The main thread has finished.");
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Programming\DemoTh
Primary is executing Main()
Main thread  : 1
Secondary is executing PrintNumbers()
Second thread: 1
Main thread  : 2
Second thread: 2
Main thread  : 3
Main thread  : 4
Second thread: 3
Main thread  : 5
The main thread has finished.
Second thread: 4
Second thread: 5
```

Working with the ParameterizedThreadStart Delegate

```
class MyParams {
    public int value01 { get; set; }
    public int value02 { get; set; }
}
class Program {
    static AutoResetEvent waitHandle = new AutoResetEvent(false);
    static void AddNumber(object data){
        if (data is MyParams p){
            Thread.Sleep(1000);
            Console.WriteLine("ID of thread in Add(): {0}",
                Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine($"{p.value01} + {p.value02} = {p.value01 + p.value02}");
            // Tell other thread we are done.
            waitHandle.Set();
        }
    }
```

Working with the ParameterizedThreadStart Delegate

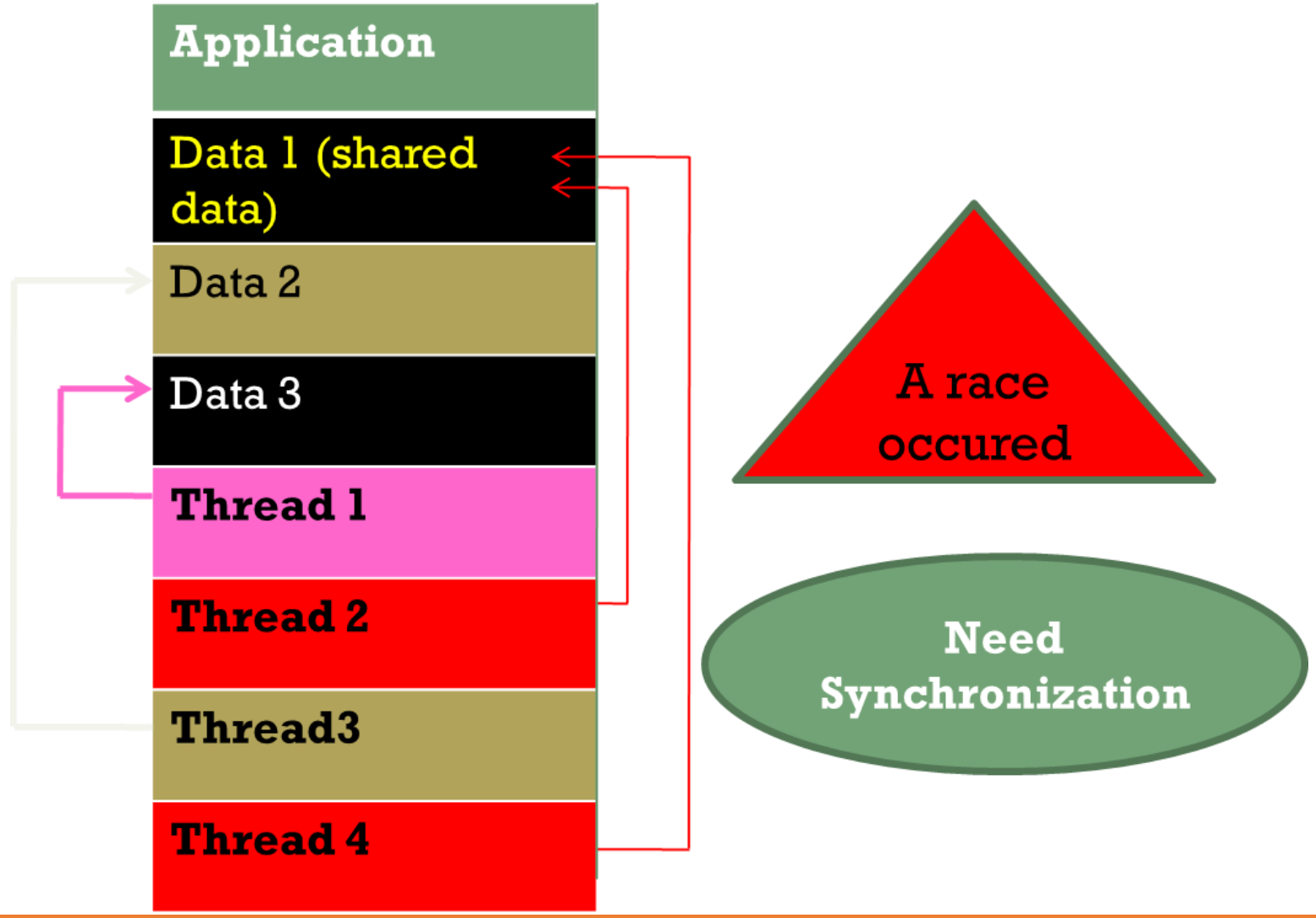
```
static void Main(string[] args){
  Console.WriteLine("ID of thread in Main(): {0}",
  Thread.CurrentThread.ManagedThreadId);
    // Make an MyParams object to pass to the secondary thread.
    MyParams p = new MyParams { value01 = 5, value02 = 15 };
    Thread t = new Thread(new ParameterizedThreadStart(AddNumber));
    //Set to background thread
    t.IsBackground = true;
    t.Start(p);
    //Wait for the wait handle to complete
    waitHandle.WaitOne();
    Console.WriteLine("Main thread: Done.");
    Console.ReadLine();
  }//End Main
}//End Program
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Progr
ID of thread in Main(): 1
ID of thread in Add(): 4
5 + 15 = 20
Main thread: Done.
```

Foreground Threads and Background Threads

- **Foreground threads** have the ability to prevent the current application from terminating. TheCLR will not shut down an application (which is to say, unload the hosting AppDomain) until all foreground threads have ended
- **Background threads** (sometimes called **daemon threads**) are viewed by the CLR as expendable paths of execution that can be ignored at any point in time (even if they are currently laboring over some unit of work)
  - Thus, if all foreground threads have terminated, any and all background threads are automatically killed when the application domain unloads

The Issue of Concurrency: Race Conditions



```
public class Printer{
    // Lock token.
    private object threadLock = new object();
    public void PrintNumbers() {
        //use lock tokent
        lock (threadLock)
            //Monitor.Enter(threadLock); //or using Monitor
            try {
                Console.WriteLine("{0} is executing PrintNumbers()", Thread.CurrentThread.Name);
                // Print out numbers.
                for (int i = 1; i <= 5; i++) {
                    Random r = new Random();
                    Thread.Sleep(500 * r.Next(5));
                    Console.Write($"{i,3}{(i == 5 ? "" : ",")}");
                }
                Console.WriteLine();
            }
            catch (Exception ex) {
                Console.WriteLine(ex.Message);
            }
            finally {
                //Monitor.Exit(threadLock);  //or using Monitor
            }
    }
}
```

The Issue of Concurrency: Race Conditions

```
class Program {
    static void Main(string[] args){
        Console.WriteLine("*******Demo Synchronizing Threads*******\n");
        Printer p = new Printer();
        // Make 05 threads that are all pointing to the same
        // method on the same object.
        Thread[] threads = new Thread[5];
        for (int i = 0; i < 5; i++) {
            threads[i] = new Thread(new ThreadStart(p.PrintNumbers)){
                Name = $"Worker thread #{i+1:D2}"
            };
        }
        // Now start each one.
        foreach (Thread t in threads) {
            t.Start();
        }
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Programming\DemoSynchronized\bin\Debu
*******Demo Synchronizing Threads*******

Worker thread #01 is executing PrintNumbers()
  1,  2,  3,  4,  5
Worker thread #03 is executing PrintNumbers()
  1,  2,  3,  4,  5
Worker thread #02 is executing PrintNumbers()
  1,  2,  3,  4,  5
Worker thread #04 is executing PrintNumbers()
  1,  2,  3,  4,  5
Worker thread #05 is executing PrintNumbers()
  1,  2,  3,  4,  5
```

Working with the Timer Callbacks

- Many applications have the need to call a specific method during regular intervals of time:
  - Display the current time on a status bar via a given helper function.
  - Perform noncritical background tasks such as checking for new e-mail messages
- Use the **System.Threading.Timer** type in conjunction with a related delegate named **TimerCallback**.

Working with the Timer Callbacks

```
class Program{
    static void PrintTime(object state){
        Console.WriteLine("Time is: {0}. Param is {1}",
            DateTime.Now.ToLongTimeString(), state.ToString());
    }
    static void Main(string[] args)
    {
        Console.WriteLine("***** Working with Timer type *****");
        // Create the delegate for the Timer type.
        TimerCallback timeCB = new TimerCallback(PrintTime);
        // Establish timer settings.
        var _ = new Timer(
            timeCB,              // The TimerCallback delegate object.
            "Hello from Main", // Any info to pass into the called method
            0,                   // Amount of time to wait before starting (in milliseconds).
            1000);               // Interval of time between calls (in milliseconds).

        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_10_Concurrency_Programming\DemoTimerCallback\bin\De
Time is: 5:16:30 PM. Param is Hello from Main
Time is: 5:16:31 PM. Param is Hello from Main
Time is: 5:16:32 PM. Param is Hello from Main
Time is: 5:16:33 PM. Param is Hello from Main
Time is: 5:16:34 PM. Param is Hello from Main
Time is: 5:16:35 PM. Param is Hello from Main
Time is: 5:16:36 PM. Param is Hello from Main
Time is: 5:16:37 PM. Param is Hello from Main
Time is: 5:16:38 PM. Param is Hello from Main
Time is: 5:16:39 PM. Param is Hello from Main
```

Working with the ThreadPool

- 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
- The thread pool manages threads efficiently by minimizing the number of threads that must be created, started, and stopped
- By using the thread pool, we can focus on our business problem rather than the application's threading infrastructure
- The ThreadPool class has several static methods including the QueueUserWorkItem that is responsible for calling a thread pool worker thread when it is available. If no worker thread is available in the thread pool, it waits until the thread becomes available

Working with the ThreadPool

```
public class Printer{
    private object threadLock = new object();
    public void PrintNumbers(){
        Monitor.Enter(threadLock);
        try {
            Console.WriteLine("->{0} is executing PrintNumbers()", Thread.CurrentThread.ManagedThreadId);
            // Print out numbers.
            for (int i = 1; i <= 5; i++){
                Random r = new Random();
                Thread.Sleep(500 * r.Next(5));
                Console.Write($"{i,3}{(i == 5 ? "" : ",")}");
            }
            Console.WriteLine();
        }
        catch (Exception ex){
            Console.WriteLine(ex.Message);
        }
        finally{
            Monitor.Exit(threadLock);
        }
    }
}
```

Working with the ThreadPool

```
class Program {
    static void PrintTheNumbers(object state){
        Printer task = (Printer)state;
        task.PrintNumbers();
    }
    static void Main(string[] args){
        Console.WriteLine("***** Demo The CoreCLR Thread Pool *****");
        Console.WriteLine("Main thread started. ThreadID = {0}",
            Thread.CurrentThread.ManagedThreadId);
        Printer p = new Printer();
        WaitCallback workItem = new WaitCallback(PrintTheNumbers);
        // Queue the method 10 times
        for (int i = 0; i < 10; i++){
            ThreadPool.QueueUserWorkItem(workItem, p);
        }
        Console.WriteLine("All tasks queued.");
        Console.ReadLine();
    }
}
```

```
Select Microsoft Visual Studio Debug Console
***** Demo The CoreCLR Thread Pool *****
Main thread started. ThreadID = 1
All tasks queued.
->6 is executing PrintNumbers()
  1,  2,  3,  4,  5
->5 is executing PrintNumbers()
  1,  2,  3,  4,  5
->4 is executing PrintNumbers()
  1,  2,  3,  4,  5
->7 is executing PrintNumbers()
  1,  2,  3,  4,  5
->8 is executing PrintNumbers()
  1,  2,  3,  4,  5
->9 is executing PrintNumbers()
  1,  2,  3,  4,  5
->10 is executing PrintNumbers()
  1,  2,  3,  4,  5
->11 is executing PrintNumbers()
  1,  2,  3,  4,  5
->12 is executing PrintNumbers()
  1,  2,  3,  4,  5
->13 is executing PrintNumbers()
  1,  2,  3,  4,  5
```

Summary

- Concepts were introduced:
  - Overview Concurrency Programming
  - Overview about MultiThreading
  - Explain about Synchronization: lock and Monitor
  - Explain and Demo about The Issue of Concurrency: Race Conditions
  - Explain about ThreadPool and TimerCallback
  - Demo MultiThreading application with C#
  - Demo Synchronization in MultiThreading application
  - Demo ThreadPool and TimerCallback