Information Processing Techniques

Threading using C# and .NET

WEEK 08

MURTAZA MUNAWAR FAZAL

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.

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

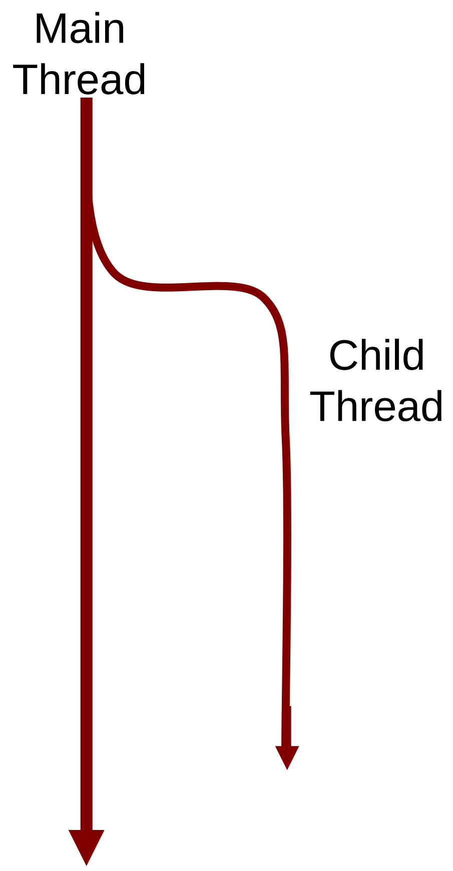
Threading: Advantages and Dangers

Reasons to use Threading

- Timers
- UI Responsiveness
- Multiple processors

Dangers

- Race Conditions
- Deadlock



Race Condition

```
if (x == 5) // t0 and t1
{
    y = x * 2; // The "Act"
   // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
}
```

Spawning a new Thread

IO

Delegates

System.Thread

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

Thread Class

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

- Some of them are :

- CurrentThread
- IsAlive
- IsBackground
- Name
- Priority
- ThreadState

Thread Constructor

Takes a ThreadStart delegate

ThreadStart(void () target)

- Takes a method which returns void and has no params:
  - void MyMethod()

Thread begins execution when Start() is called

Thread executes method passed to ThreadStart

Thread Properties

IsBackground

- get or set
- Once all foreground threads are finished, the runtime calls Abort on all background threads
- Default is **false** (or foreground)

Priority

- ThreadPriority Enum
- Lowest, BelowNormal, Normal, AboveNormal, Highest

ThreadState

- New Thread: ThreadState.Unstarted
- Started Thread: ThreadState.Running
- Sleep called: ThreadState.WaitSleepJoin
- Suspend called: ThreadState.Suspended
- See ThreadState Enumeration

Thread Methods

Start()

- Begins execution of the thread
- Once a thread is finished, it cannot be restarted

Suspend()

- Suspends the thread
- If the thread is already suspended, there is no effect

Resume()

- Resumes a suspended thread

Interrupt()

- Resumes a thread that is in a WaitSleepJoin state
- If the thread is not in WaitSleepJoin state it will be interrupted next time it is blocked

Thread Methods (Continued)

Abort()

- Attempts to abort the thread
- Throws a ThreadAbortException

Join()

- Blocks the calling thread until the owning thread terminates

Sleep()

- Suspends the current thread for the specified amount of time

Exceptions Between threads

Must be handled in the thread it was thrown in

Exceptions not handled in the thread are considered unhandled and will terminate that thread

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
```

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 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.
    }
```

ResetAbort / Join

- 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.

Synchronization Classes and Constructs

- lock keyword
- Mutex class
- Monitor class
- Interlocked class

The **lock** Keyword

Marks a statement as a critical section

Is a Monitor underneath

Example:

```
lock(lockObject)
{
   // critical section
}
```

lockObject must be a reference-type instance

Typically you will lock on:

- 'this'
  - locks the current instance
- typeof(MyClass)
  - global lock for the given type
- A collection instance
  - locks access to a specific collection

The **lock** Mutex

lock defines one Mutex for each object locked on

Blocks until the current thread is finished

Mutex Class

Stands for **Mut**ual-**Ex**clusion

Blocking synchronization object

WaitOne()
- Begins the critical section

ReleaseMutex()
- Ends critical section
- Call ReleaseMutex() in a finally block

Monitor Class (static)

Enter(object obj)
- Begins a critical section
- Blocks if another thread has the same lock

Exit(object obj)
- Ends a critical section
- Releases the lock

TryEnter(object obj)
- Returns **true** if it obtains the lock
- Returns **false** if it can’t obtain the lock
- Avoids blocking if you can’t obtain the lock

Wait(object obj)
- Releases the lock on an object and blocks until it reaquires the lock

Pulse(object obj)
- Signals the next waiting thread that the lock may be free

PulseAll(object obj)
- Signals all waiting threads that the lock may be free

Thread Synchronization / Monitors

- 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.

The C # 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.

Interlocked Class (static)

Provides **atomic** operations for variables

CompareExchange(ref int dest, int source, int compare)

- Replaces *dest* with *source* if *dest* == *compare*
- Overloaded

Exchange(ref int dest, int source)

- places *source* into *dest*
- returns the original value of *dest*

Increment(ref int value)

- increments value

Decrement(ref int value)

- decrements value

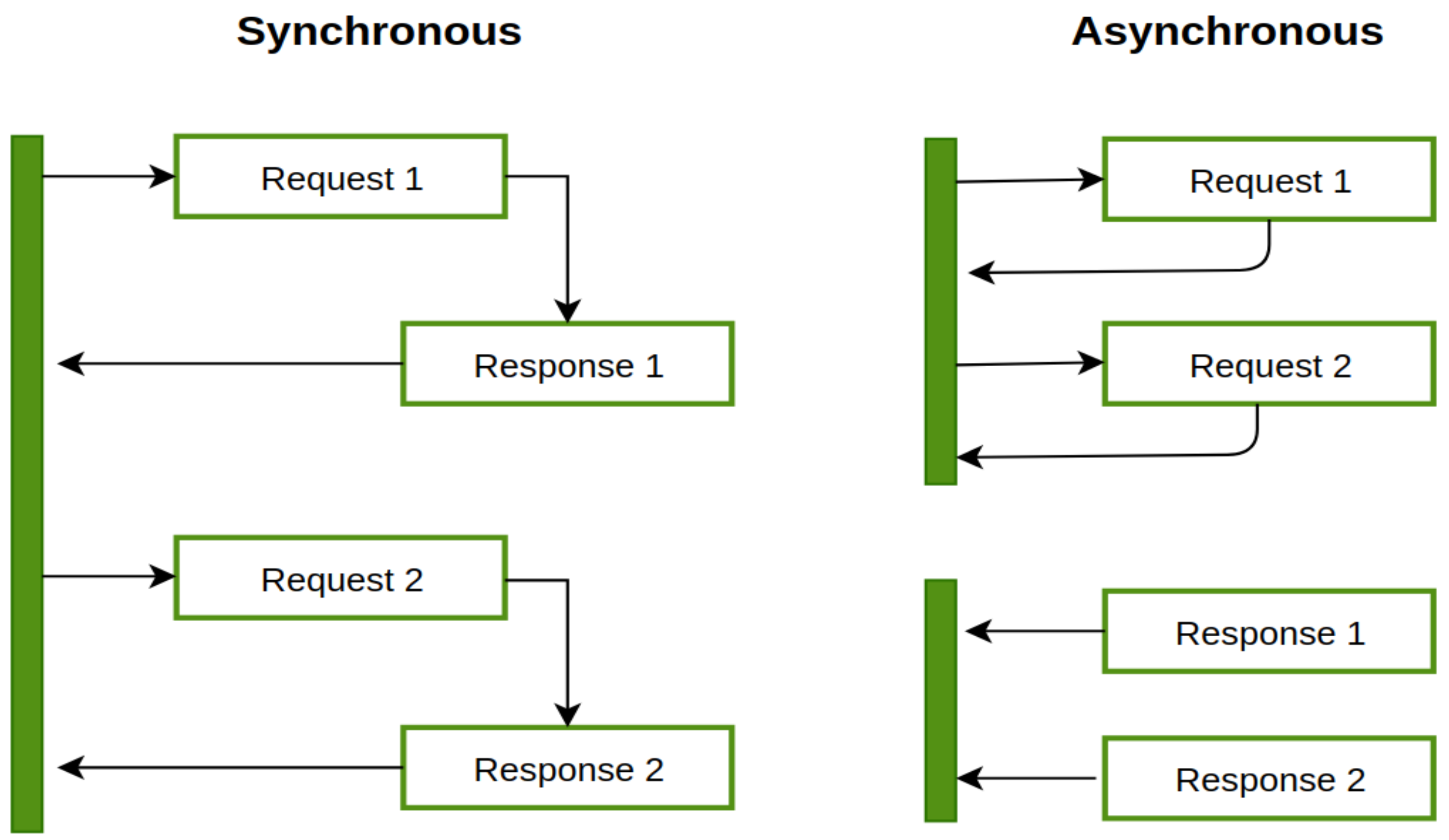
MethodImpl Attribute

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

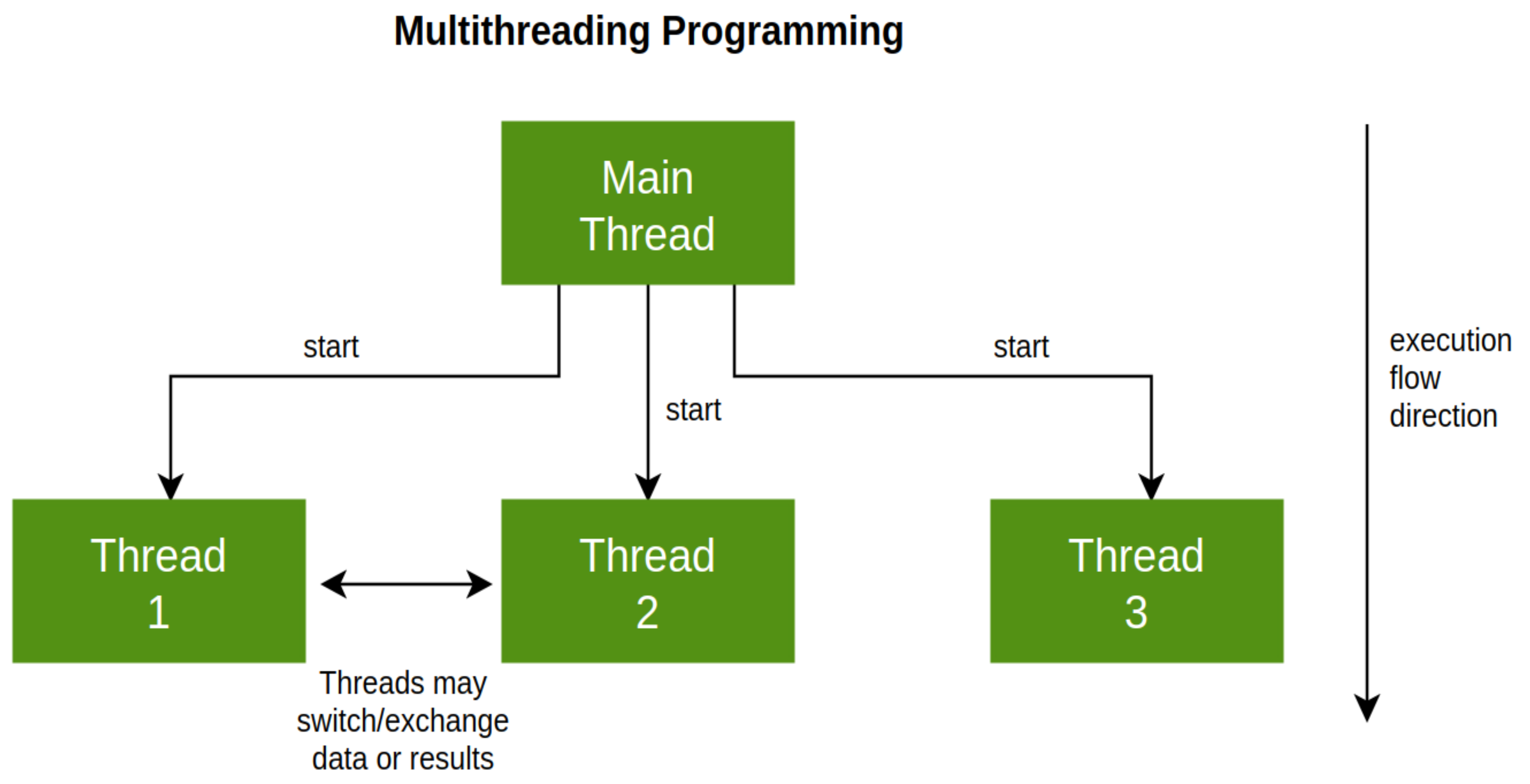
```
[MethodImpl] (MethodImplOptions.Synchronized)]
Byte[] TransformData(byte[] buffer)
{
……
}
```

Only one thread at a time can enter the method.

Sync v/s ASync



Multithreading



Reading on ASync

- Asynchronous programming in C# | Microsoft Docs (https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/)

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.