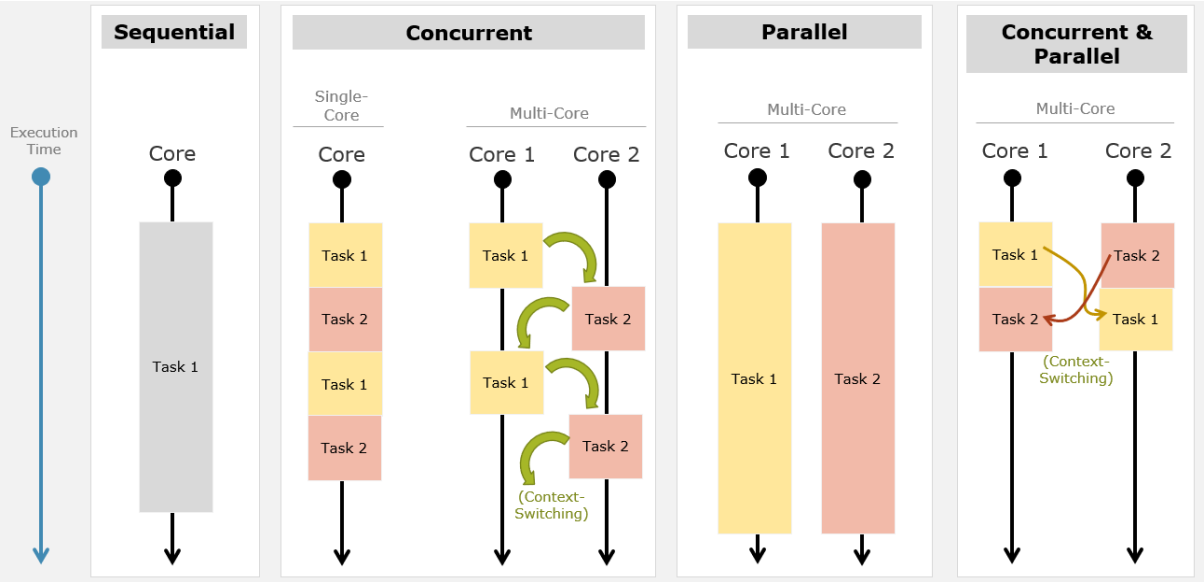
C# - Ultimate Guide - Beginner to Advanced | Master class

Section 31 – Threading

Introduction to Concurrent Execution



Concurrent

Concurrent execution is concept where the programmer can divide the program into independent parts (threads) and can execute them in order-independent manner or in partial order, without affecting the outcome.

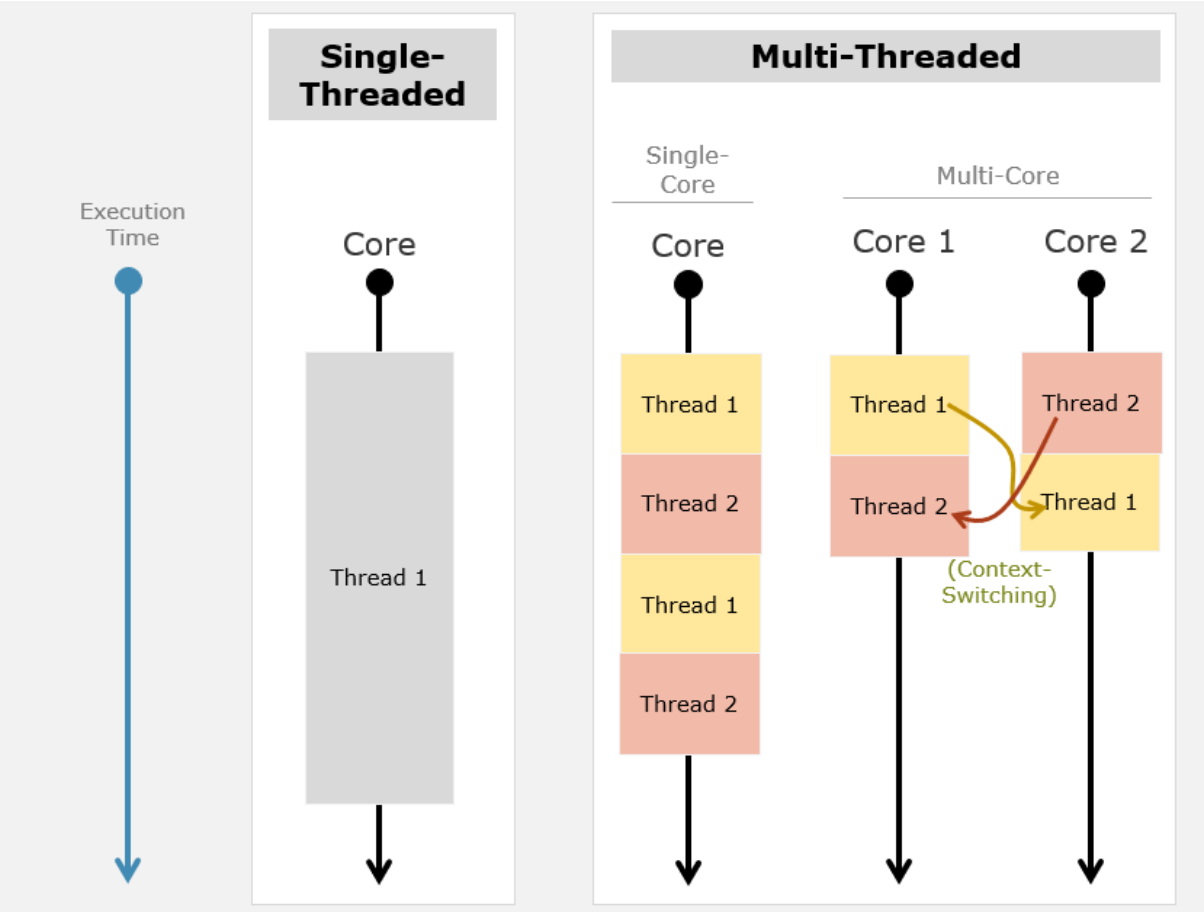
Parallel

Parallel execution is a concept that enables a program to execute multiple tasks simultaneously, typically by utilizing multiple CPUs or processing cores.

Concurrent & Parallel

It is a combination of both concurrent execution and parallel execution. The threads may execute in parallel (based on time-slicing algorithm of the O/S).

Introduction to Threading



Thread is a light-weight unit of execution within a process.

- Multi-Threading refers to the ability to create and run more than one independent threads concurrently within a single process.
- It enables concurrent program execution.
- By default, all the code written in the entire C# application executes under a default thread called "Main Thread".

Benefits of Threading

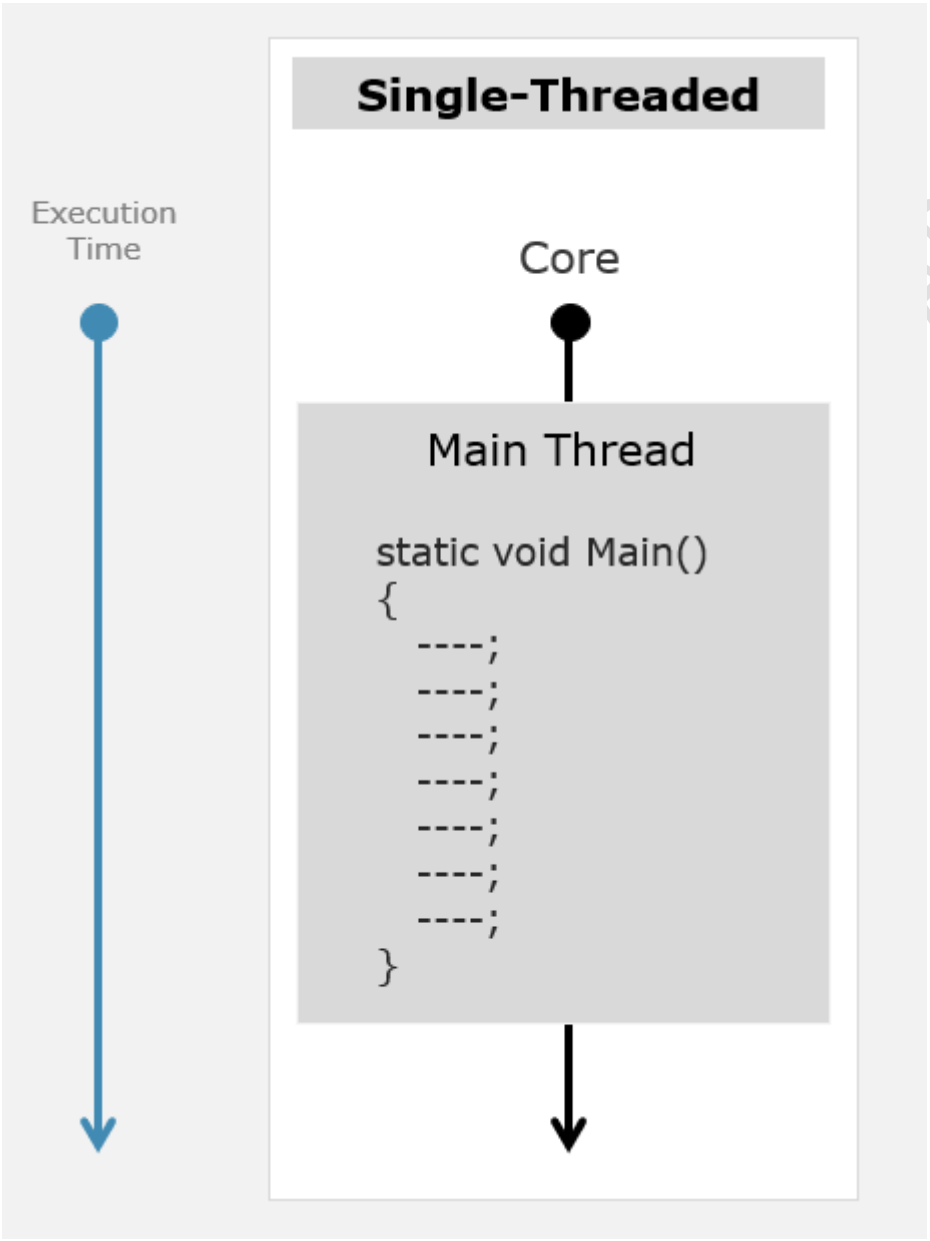
Improved Performance

Multithreading can help to increase the overall performance of a program by allowing it take full utilization of multiple cores of the processor.

Improved UI Responsiveness

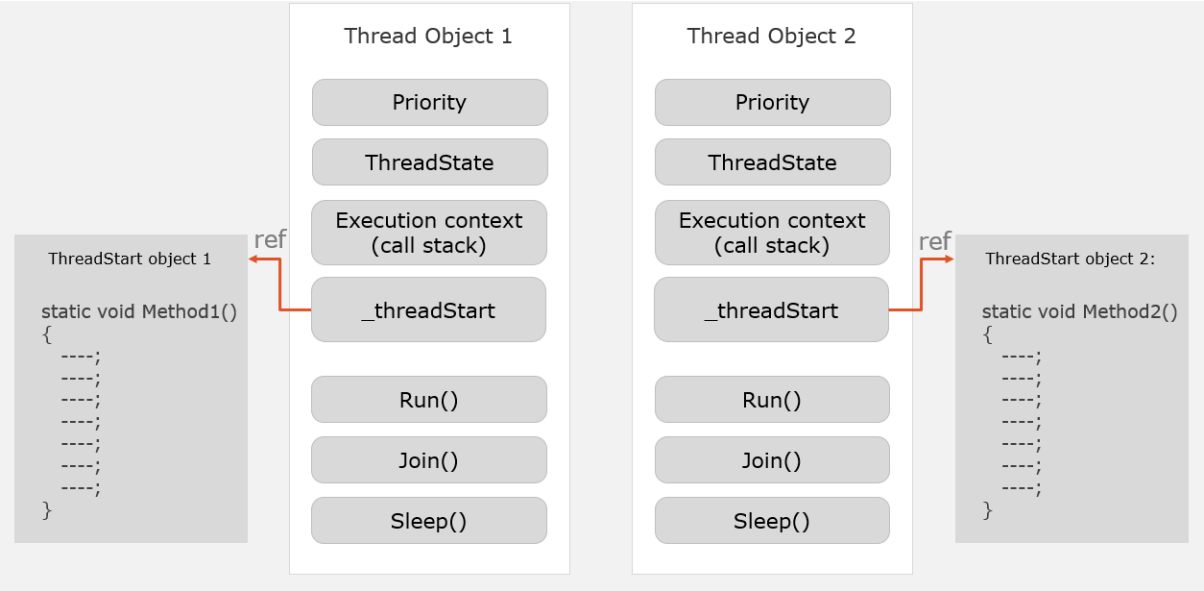
Multithreading can help to improve the responsiveness of a program by allowing it to execute tasks in background, without making the UI hang / stuck until the task gets fully completed.

Main Thread



By default, all the code written in the entire C# application executes under a default thread called "Main Thread".

Thread Class



The 'System.Threading.Thread' class is represents a thread (execution unit that has a set of statements to execute) in the application

A thread represents light-weight unit of execution that contains its own set of instructions to execute.

Constructors

1. Thread(ThreadStart threadStart)
2. Thread(ParameterizedThreadStart parameterizedThreadStart)

Properties

- string Name { get; set; }
- ThreadPriority Priority { get; set; } //Lowest | BelowNormal | Normal | AboveNormal | Highest
- bool IsAlive { get; }
- bool IsBackground { get; set; }
- ThreadState ThreadState { get; } //Unstarted | Running | WaitSleepJoin | Aborted etc.
- int ManagedThreadId { get; }
- static Thread CurrentThread { get; }

Methods

- public void Start()
- public void Join()
- public static void Sleep(int millisecondsTimeout)
- public void Interrupt()

Note: There few methods called Abort(), Suspend() and Resume() in 'Thread' class in earlier versions of .NET.

But these methods are obsolete (deprecated) and not recommended to use in real world applications, because they can cause thread instability, deadlocks and other synchronization issues etc.

'Thread' Class - Properties

string Name { get; set; }

Represents a custom name of the thread for easier identification. Default is null.

ThreadPriority Priority { get; set; }

Represents a value that determines the relative importance of the thread compared to other threads in the system.

Options in 'ThreadPriority' enum: Lowest | BelowNormal | Normal | AboveNormal | Highest

Note: Actual thread execution speed / completion time depends on various factors such as amount of code of the thread, OS's time-slicing algorithm etc.

bool IsAlive { get; }

Represents a boolean value that indicates whether the thread is currently running (started or not). It becomes true when the Start() method called and becomes false when the thread execution is completed.

bool IsBackground { get; set; }

Represents a boolean value that indicates whether the thread is a background thread or not.

true: The thread will not prevent the application from terminating when, even when the thread is not completed.

false: The thread will prevent the application from terminating when, even when the thread is not completed.

ThreadState ThreadState { get; }

Returns an enum value that represents the current state of the thread.

Options in 'ThreadState' enum:

1. **Unstarted:** Indicates that the thread has been created but has not yet been started. It is the default value of a new thread.
2. **Running:** Indicates that the thread is currently running and executing the code.
3. **WaitSleepJoin:** Indicates that the thread is currently in a wait, sleep or join state; and is not executing the code.
4. **Suspended:** Indicates that the thread is suspended by using the "Thread.Suspend()" method. But the Suspend() method is deprecated in .net core.
5. **Aborted:** Indicates that the thread is aborted (terminated) by using the "Thread.Abort()" method. But the Abort() method is deprecated in .net core.
6. **Stopped:** Indicates that the thread execution has been completed.

int ManagedThreadId { get; }

Returns an integer value that represents the unique identifier (auto-generated) of the thread within the current application.

static Thread CurrentThread { get; }

It is a static property that returns a reference to the thread object, which is executing the current working method.

'Thread' Class - Methods

public void Start()

Starts executing the thread.

That means, it invokes the 'ThreadStart' delegate or 'ParameterizedThreadStart' delegate that is stored as a private field.

It turns the thread object's state from "Unstarted" to "Running" state.

public void Join()

This method blocks the calling thread (the thread that invokes the Join method) until the thread completes.

That means, the calling thread waits until the thread completes its execution.

It turns the thread from "Running" state to "WaitSleepJoin" state for specified duration (in milliseconds).

public static void Sleep(int millisecondsTimeout)

Pauses the thread for specified number of milliseconds.

That means, it turns the thread from "Running" state to "WaitSleepJoin" state for specified duration (in milliseconds).

While in sleep period, the thread is not executing any code and is not consuming CPU resources.

public void Interrupt()

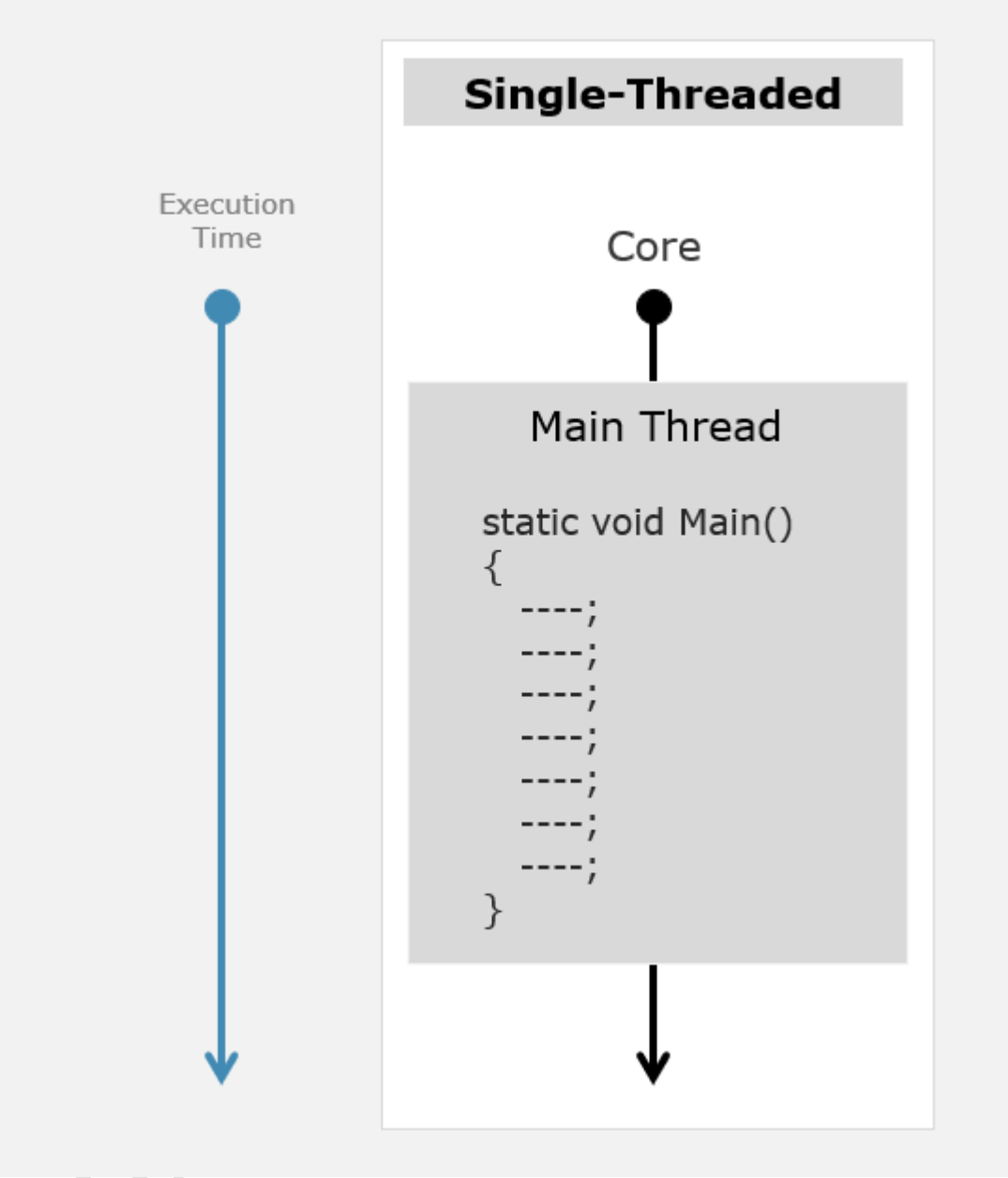
This method interrupts the thread by throwing a "ThreadInterruptedException".

If the thread is blocked state (sleeping or waiting for another signal), it will be woken up and will throw a ThreadInterruptedException immediately when it resumes execution. However, if the thread is in Running state, it immediately throws ThreadInterruptedException.

The ThreadInterruptedException can be gracefully handled by the thread method.

Main Thread

Single Threaded App



By default, all the code written in the entire C# application executes under a default thread called "Main Thread".

'Thread' Class

The 'System.Threading.Thread' class is represents a thread (execution unit that has a set of statements to execute) in the application

A thread represents light-weight unit of execution that contains its own set of instructions to execute.

Constructors

1. Thread(ThreadStart threadStart)
2. Thread(ParameterizedThreadStart parameterizedThreadStart)

Properties

- string Name { get; set; }
- ThreadPriority Priority { get; set; } //Lowest | BelowNormal | Normal | AboveNormal | Highest
- bool IsActive { get; }
- bool IsBackground { get; set; }
- ThreadState ThreadState { get; } //Unstarted | Running | WaitSleepJoin | Aborted etc.
- int ManagedThreadId { get; }
- static Thread CurrentThread { get; }

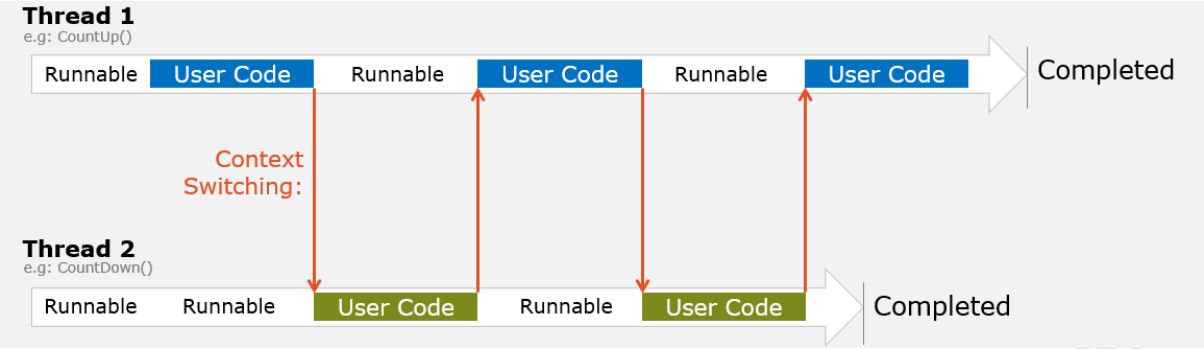
Methods

- public void Start()
- public void Join()
- public static void Sleep(int millisecondsTimeout)
- public void Interrupt()

Note: There few methods called Abort(), Suspend() and Resume() in 'Thread' class in earlier versions of .NET.

But these methods are obsolete (deprecated) and not recommended to use in real world applications, because they can cause thread instability, deadlocks and other synchronization issues etc.

Multi-Threaded App



'Thread' Class - Methods

public void Start()

Starts executing the thread.

That means, it invokes the 'ThreadStart' delegate or 'ParameterizedThreadStart' delegate that is stored as a private field.

It turns the thread object's state from "Unstarted" to "Running" state.

public static void Sleep(int millisecondsTimeout)

Pauses the thread for specified number of milliseconds.

That means, it turns the thread from "Running" state to "WaitSleepJoin" state for specified duration (in milliseconds).

While in sleep period, the thread is not executing any code and is not consuming CPU resources.

public static void Sleep(int millisecondsTimeout)

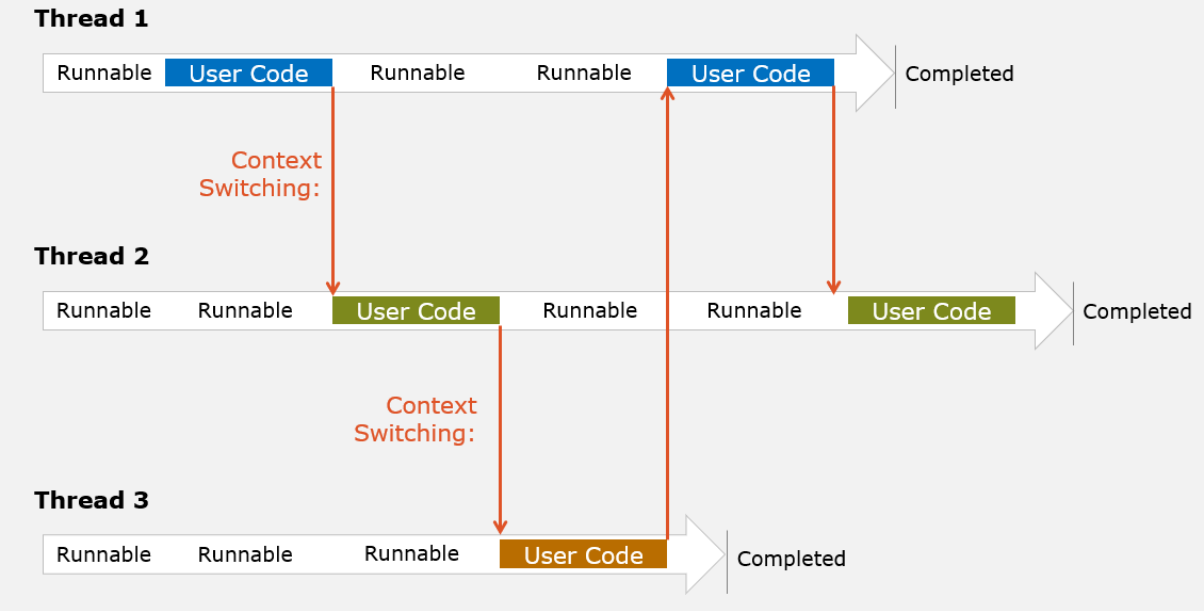
Pauses the thread for specified number of milliseconds.

That means, it turns the thread from "Running" state to "WaitSleepJoin" state for specified duration (in milliseconds).

While in sleep period, the thread is not executing any code and is not consuming CPU resources.

Thread.Join() method

Multi-Threaded App



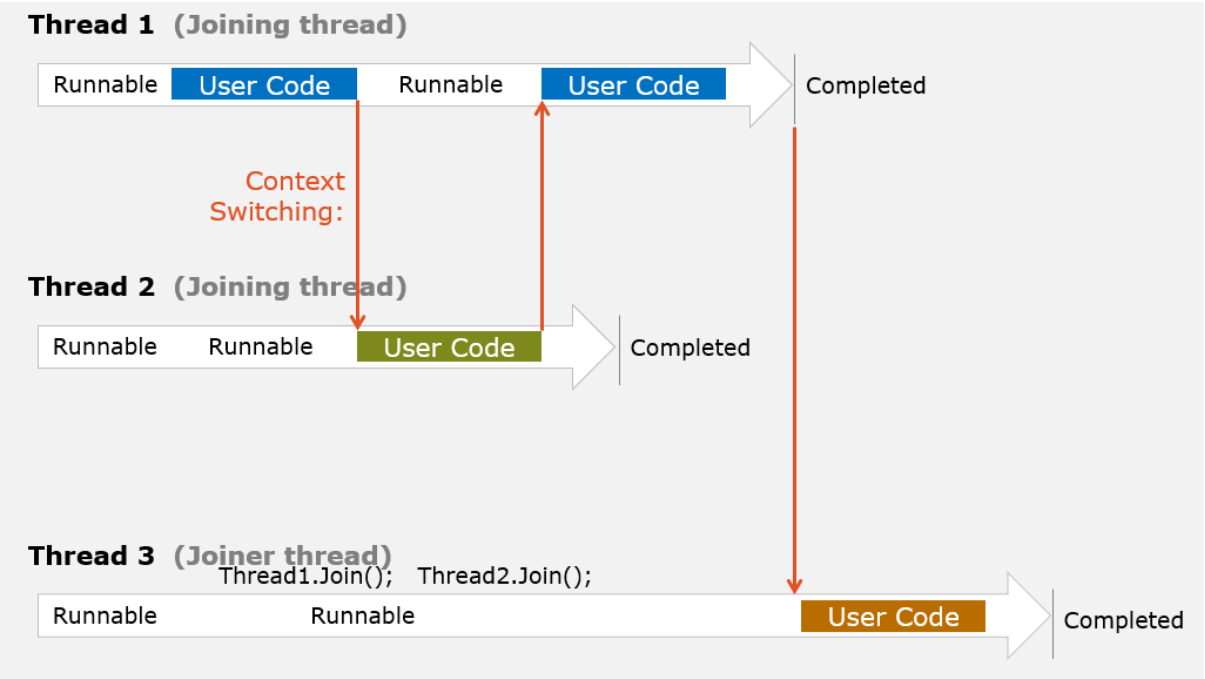
Join() method - 'Thread' Class

public void Join()

This method blocks the calling thread (the thread that invokes the Join method) until the thread completes.

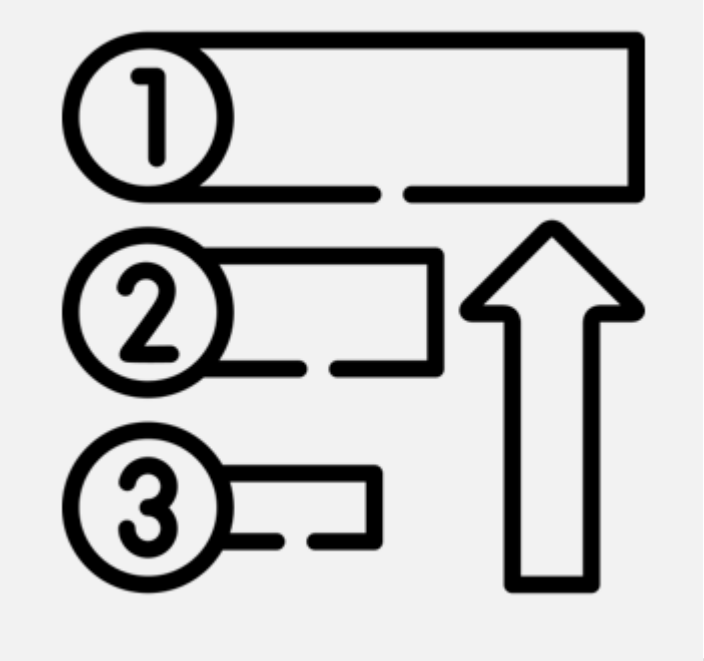
That means, the calling thread waits until the thread completes its execution.

It turns the thread from "Running" state to "WaitSleepJoin" state for specified duration (in milliseconds).



Thread Priority

Lowest | BelowNormal | Normal | AboveNormal | Highest



'Thread' Class - Properties

ThreadPriority Priority { get; set; }

Represents a value that determines the relative importance of the thread compared to other threads in the system.

Options in 'ThreadPriority' enum: Lowest | BelowNormal | Normal | AboveNormal | Highest

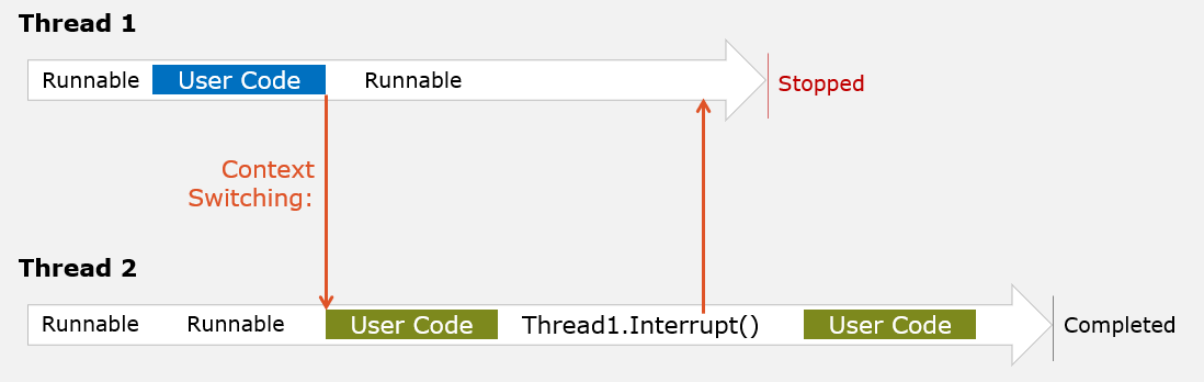
Note: Actual thread execution speed / completion time depends on various factors such as amount of code of the thread, OS's time-slicing algorithm etc.

int ManagedThreadId { get; }

Returns an integer value that represents the unique identifier (auto-generated) of the thread within the current application.

Thread - Interrupt

'Thread' Class – “Interrupt” method



public void Interrupt()

•This method interrupts the thread by throwing a "ThreadInterruptedException".

•If the thread is blocked state (sleeping or waiting for another signal), it will be woken up and will throw a ThreadInterruptedException immediately. However, if the thread is in Running state, it immediately throws ThreadInterruptedException.

•The ThreadInterruptedException can be gracefully handled by the thread method.

Thread State

'Thread' Class – ‘ThreadState’ property

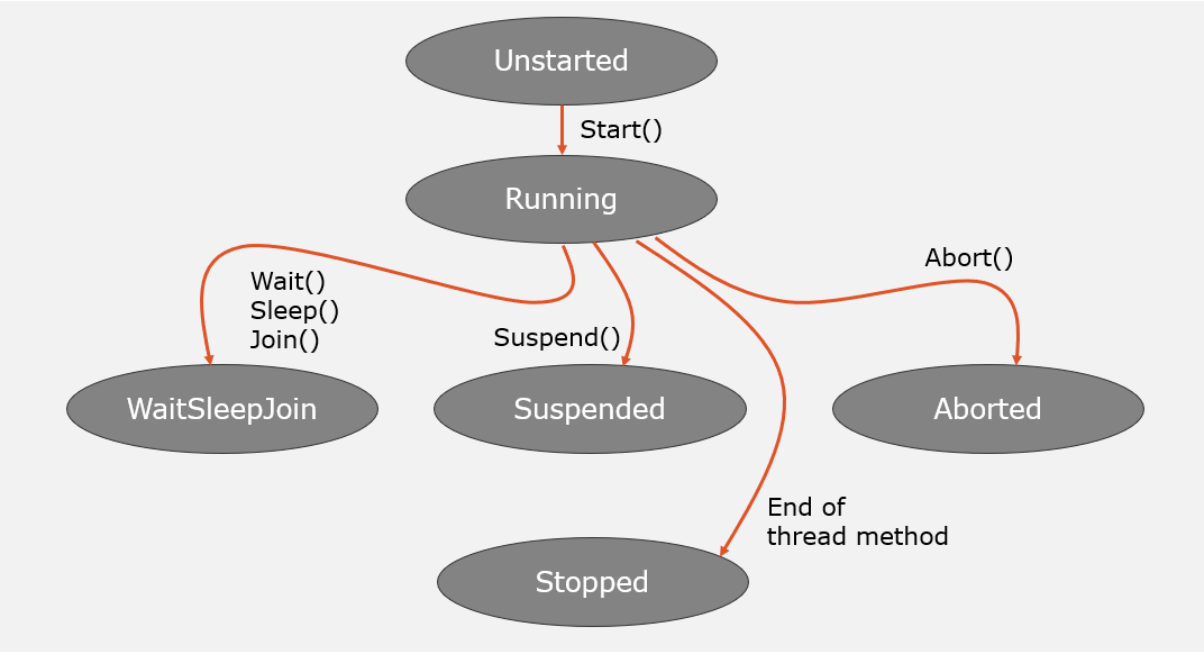
ThreadState ThreadState { get; }

Returns an enum value that represents the current state of the thread.

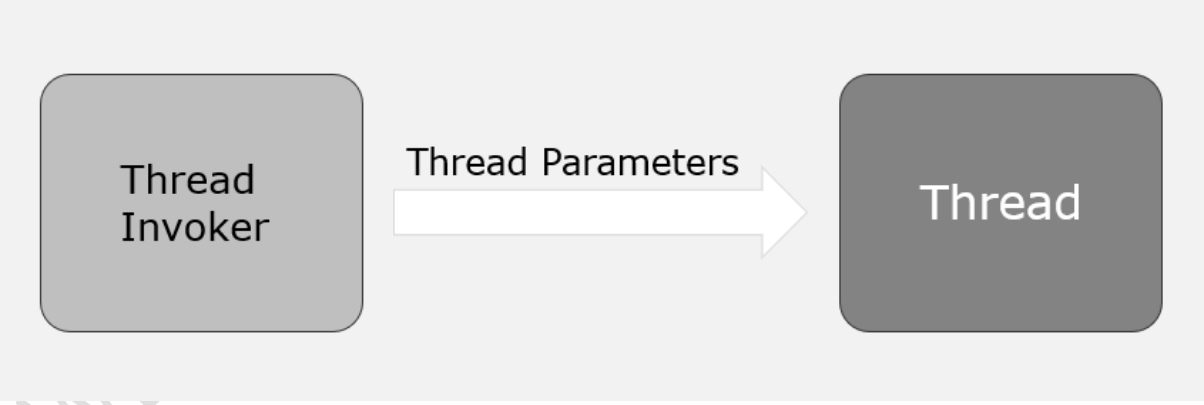
Options in 'ThreadState' enum:

1. **Unstarted:** Indicates that the thread has been created but has not yet been started. It is the default value of a new thread.
2. **Running:** Indicates that the thread is currently running and executing the code.
3. **WaitSleepJoin:** Indicates that the thread is currently in a wait, sleep or join state; and is not executing the code.
4. **Suspended:** Indicates that the thread is suspended by using the "Thread.Suspend()" method. But the Suspend() method is deprecated in .net core.
5. **Aborted:** Indicates that the thread is aborted (terminated) by using the "Thread.Abort()" method. But the Abort() method is deprecated in .net core.
6. **Stopped:** Indicates that the thread execution has been completed.

'Thread' Life Cycle



Thread Parameters

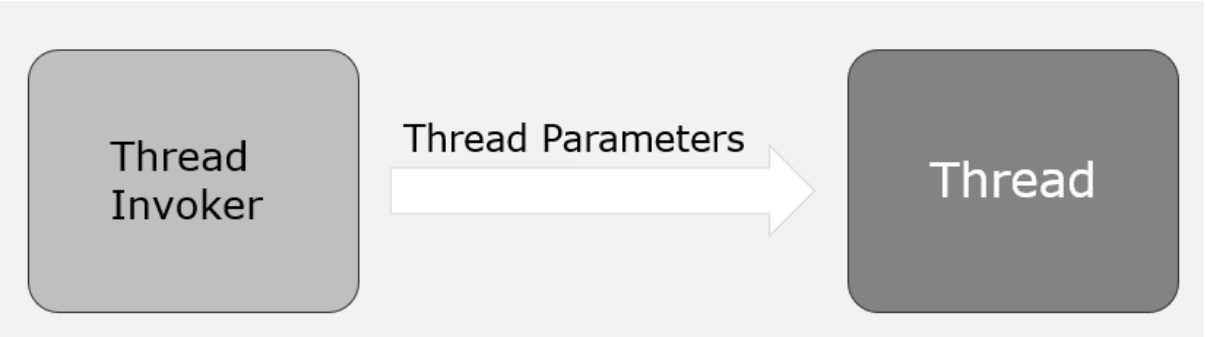


() => {

(arguments);

}

ParameterizedThreadStart



public delegate void ParameterizedThreadStart(object? obj)

public delegate void ThreadStart()

Create ParameterizedThreadStart delegate object:

ParameterizedThreadStart threadStart = new ParameterizedThreadStart(ThreadMethod);

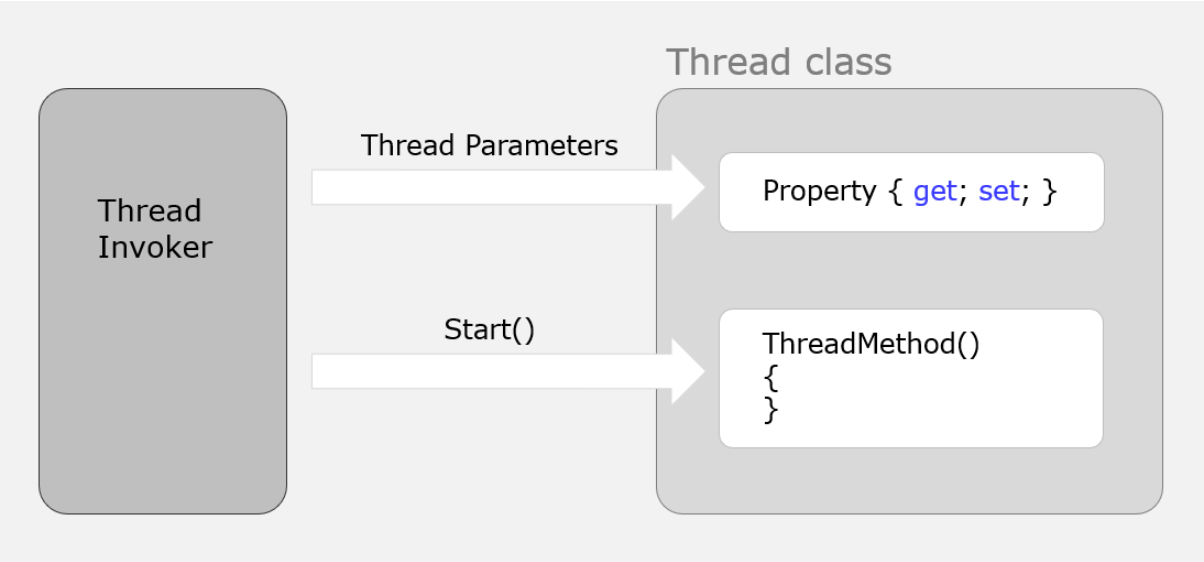
Create Thread object:

Thread thread = new Thread(threadStart);

Start the thread and supply arguments:

thread.Start(arguments);

Custom Thread Object

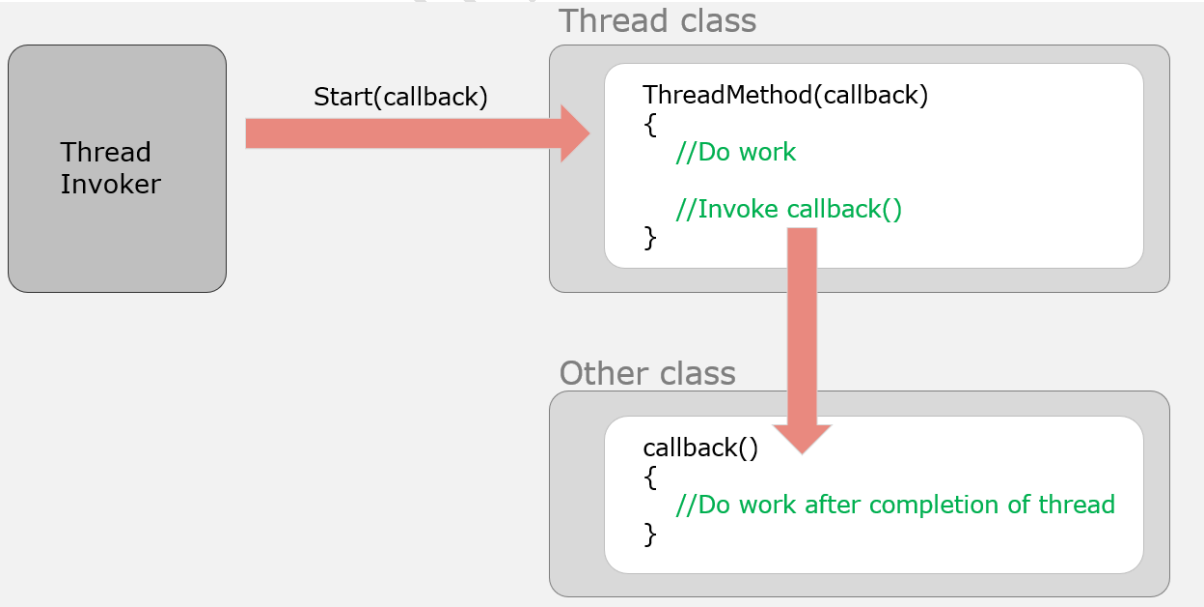


Callback

Callback is a method to be executed by a thread after completion of thread execution.

Generally, the callback method is defined by the thread invoker.

Callback - How they work?



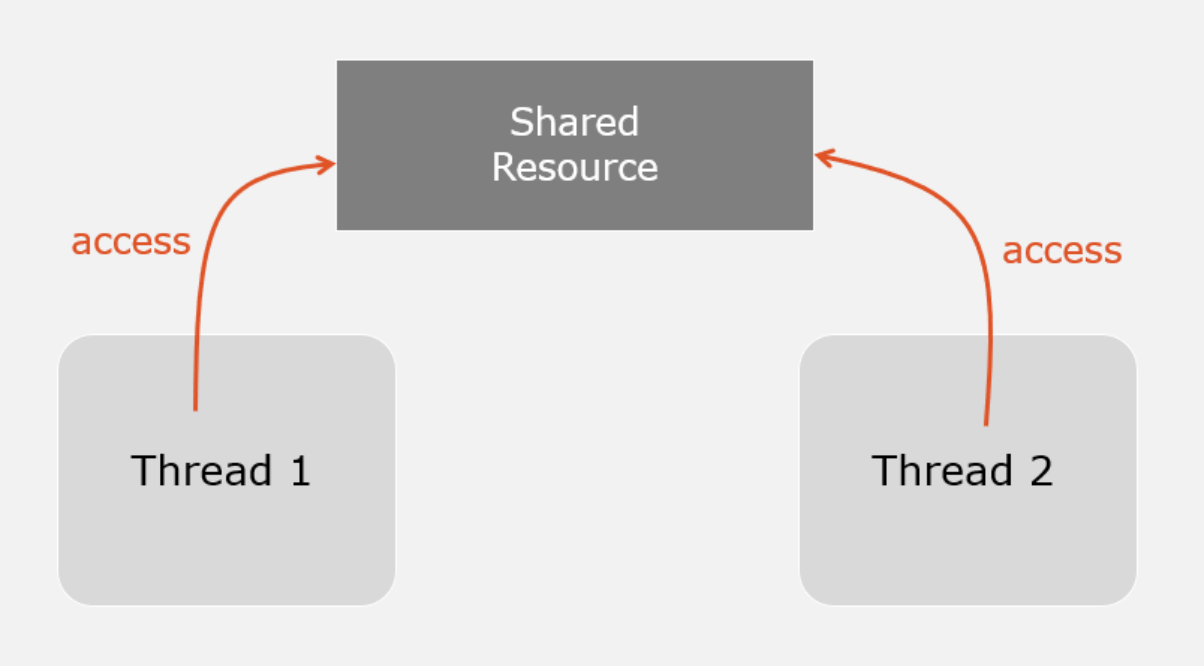
Benefits of Callback

Callbacks are essential for enabling communication between different threads.

They provide a way for a thread to signal the completion of a task or notify other threads about certain events.

e.g: File processing, Networking etc.

Shared Resources



Problems in Shared Resources

When two or more threads attempt to access and modify the same shared resource simultaneously, it is termed as "Shared Resource".

Without proper synchronization, this can lead to various issues and unexpected behavior in the program.

Race condition

A race condition occurs when multiple threads try to update shared data simultaneously, and the final outcome depends on the order in which the threads are executed. The result becomes unpredictable and can lead to data corruption or incorrect computations.

Data Inconsistency

When multiple threads read and modify shared data without synchronization, it can lead to data inconsistency. One thread may read the data while another is in the process of modifying it, leading to incorrect results or unexpected behavior.

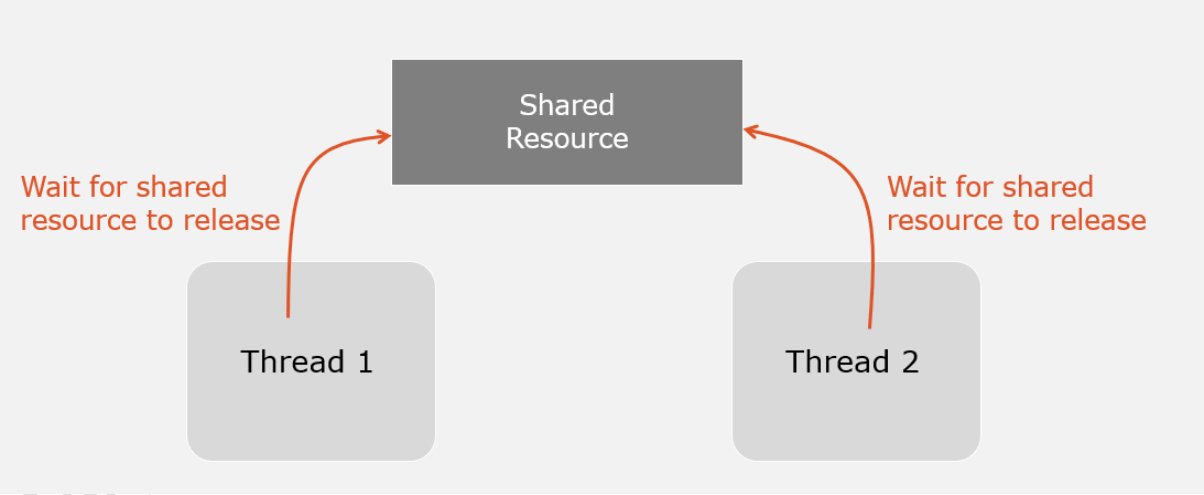
Deadlock

A deadlock occurs when two or more threads are blocked indefinitely because each thread is waiting for a resource that is held by another thread. This situation results in a standstill, where no thread can proceed, and the program becomes unresponsive.

Starvation

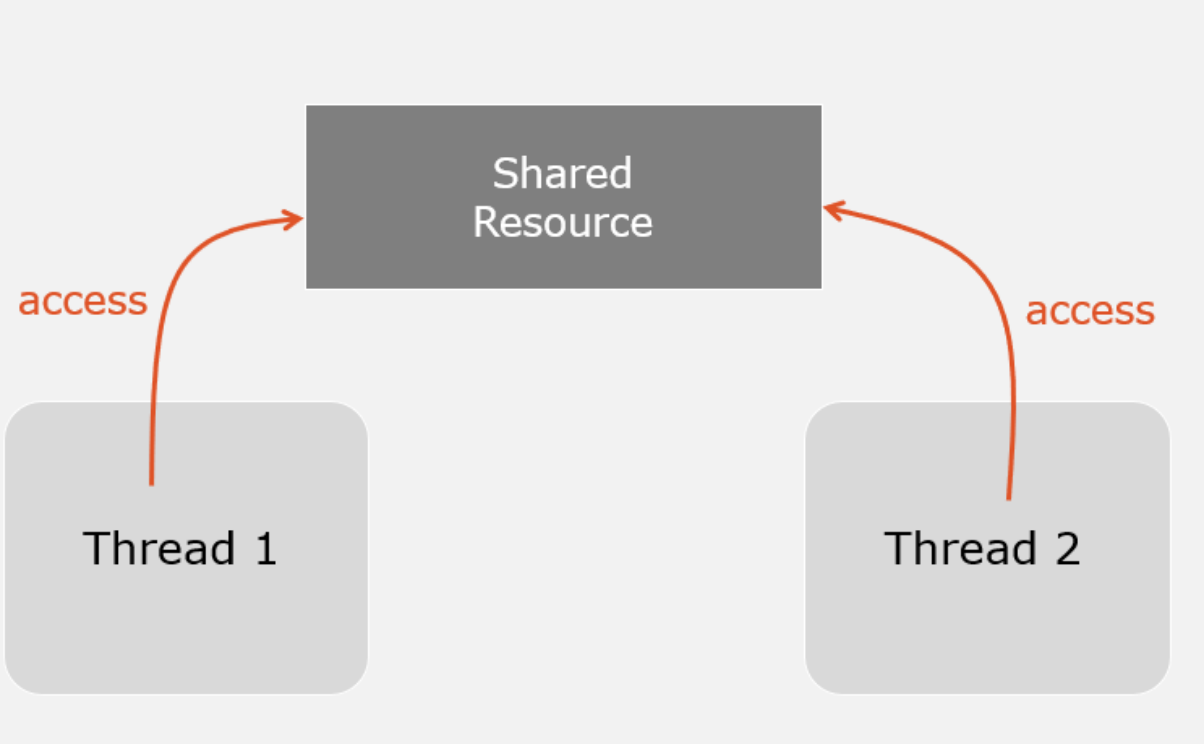
Starvation happens when a thread is perpetually denied access to a shared resource because other high-priority threads consistently acquire the resource first. The starved thread cannot make progress and may result in reduced performance.

Deadlock



Thread Synchronization

Shared Resources



Problems:

Race conditions

Data Inconsistency

Deadlocks

Starvation

Thread Synchronization

Thread synchronization is the process of coordinating the execution of multiple threads to ensure that they do not interfere with each other while accessing shared resources.

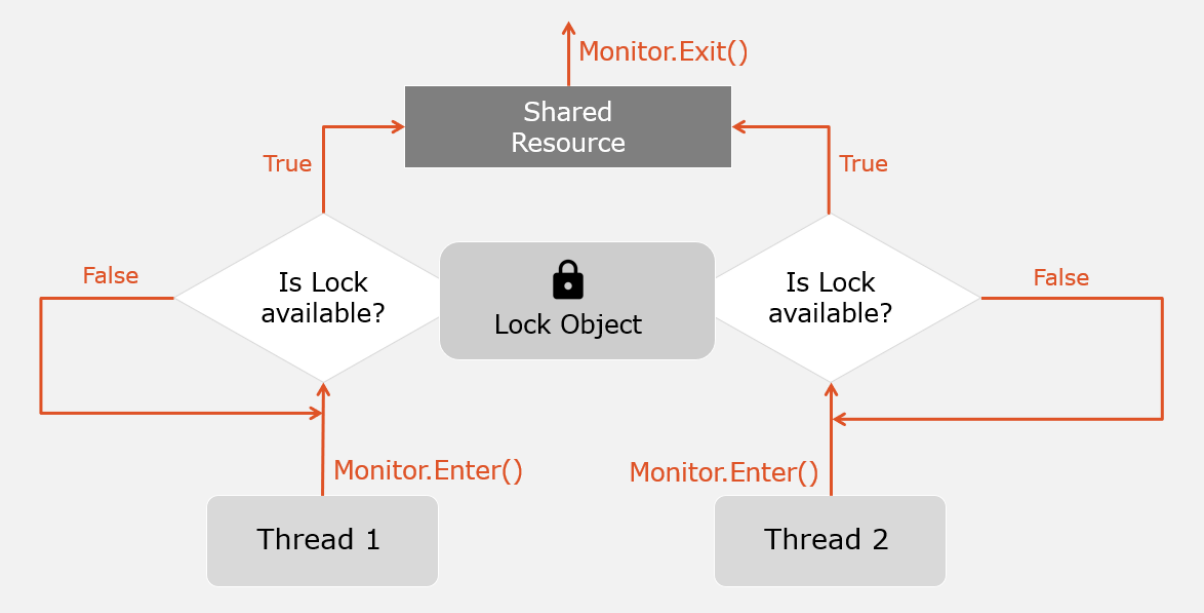
This is essential if two or more threads are accessing a shared resource such as a variable / property or any other external resource.

Thread Synchronization Mechanisms

- Monitor
- Locks
- Mutex
- Semaphores

Monitor

Thread Synchronization with 'Monitor' class



'Monitor' class

The 'Monitor' class is a synchronization primitive (synchronization mechanism) that enables thread synchronization based on a lock object.

It ensures that only one thread can execute the code within the critical section at a time, thus preventing race conditions and deadlocks.

Two threads that use the same lock object can access a common shared resource one-at-a-time.

'lock' statement

'Monitor' [vs] 'lock'

Monitor

Monitor.Enter(lockObject);

…

Monitor.Exit(lockObject);

'lock' statement

lock (lockObject)

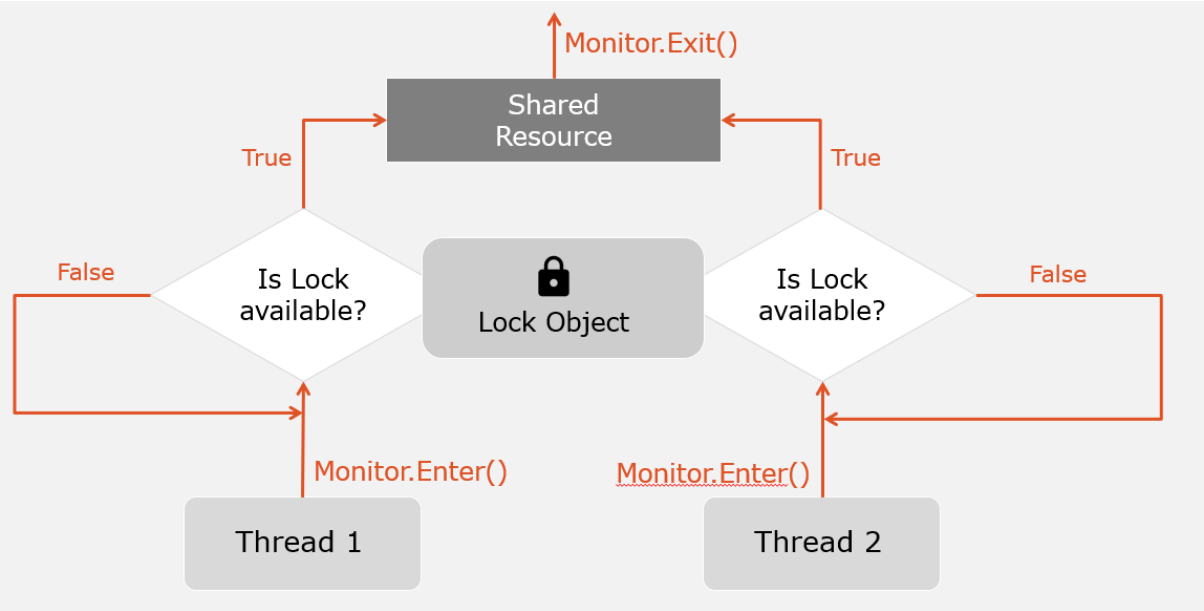
{

…

}

A shortcut syntax to use Monitor.Enter() and Monitor.Exit().

'lock' statement

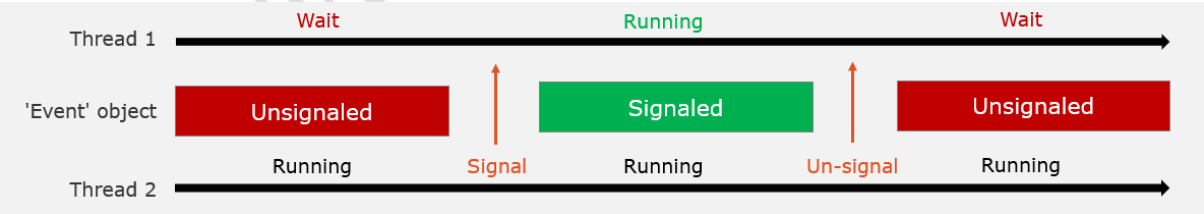


ManualResetEvent

Thread Signaling

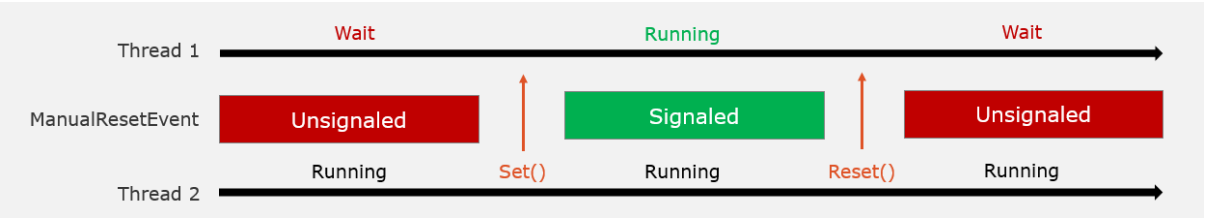
Thread Signaling is a technique to make a thread (referred as waiting thread) wait for a signal from another thread (referred as signaling thread) when a specific condition is met.

Implemented using Events (ManualResetEvent, AutoResetEvent) and Monitor.

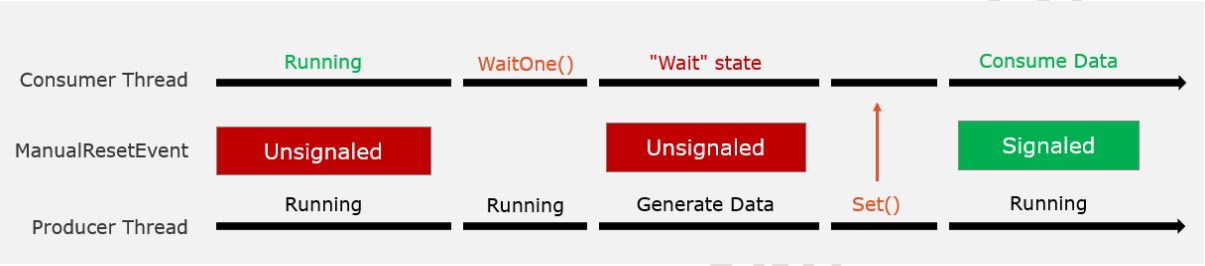


ManualResetEvent is a synchronization primitive that is used to enable signaling mechanism between threads.

It makes a thread to wait until another thread signals it to be executed.



ManualResetEvent - Scenario



ManualResetEvent - Operations

Set()

This operation sets the state of the event to signaled.

When the event is signaled, any one of the threads that are waiting for it will be allowed to proceed.

Reset()

This operation resets the state of the event to non-signaled.

When the event is not signaled, any threads that subsequently wait for it will be blocked until the event becomes signaled again.

WaitOne()

This operation makes the thread wait for the event to become signaled.

ManualResetEvent - Scenarios

Producer-Consumer Pattern

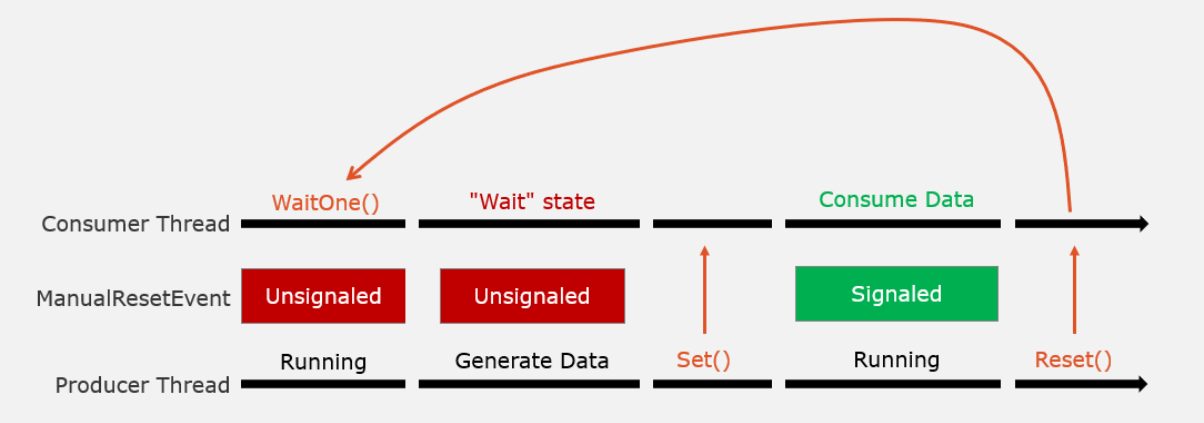
One or more producer threads generate data that can be consumed by one or more consumer threads.

The consumer threads can wait until the producer thread signals the event indicating that the data is ready to be consumed.

Waiting for External Events

The threads can wait for specific events to occur such as input from the user, response from a database server etc.

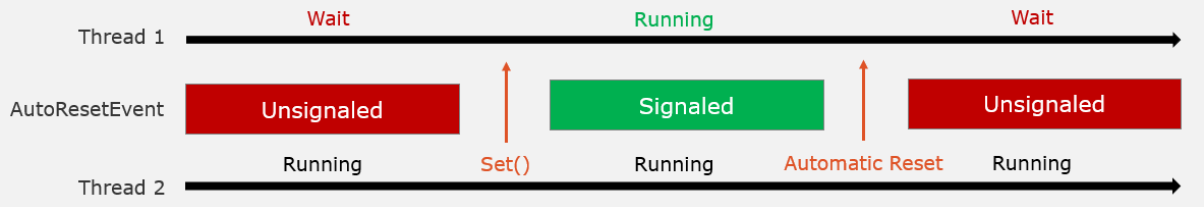
ManualResetEvent - Scenario 2



AutoResetEvent

AutoResetEvent is a synchronization primitive that is used to enable signaling mechanism between threads.

It makes a thread to wait until another thread signals it to be executed.



ManualResetEvent [vs] AutoResetEvent

ManualResetEvent

When signaled using Set() method, it stays signaled until the Reset() method is called to make it unsignaled.

Its behavior is something similar to regular main gate, which allows all persons to pass in.

AutoResetEvent

When signaled using Set() method, it automatically gets unsignaled after releasing one waiting thread.

Its behavior is something similar to turnstile gate, which allows only one person to pass in.

AutoResetEvent - Operations

Set()

This operation sets the state of the event to signaled.

When the event is signaled, any one of the threads that are waiting for it will be allowed to proceed.

Reset()

This operation resets the state of the event to non-signaled.

When the event is not signaled, any threads that subsequently wait for it will be blocked until the event becomes signaled again.

WaitOne()

This operation makes the thread wait for the event to become signaled.

AutoResetEvent - Scenarios

Producer-Consumer Pattern

One or more producer threads generate data that can be consumed by one or more consumer threads.

So the consumer threads can wait until the producer thread signals the event indicating that the data is ready to be consumed.

Waiting for External Events

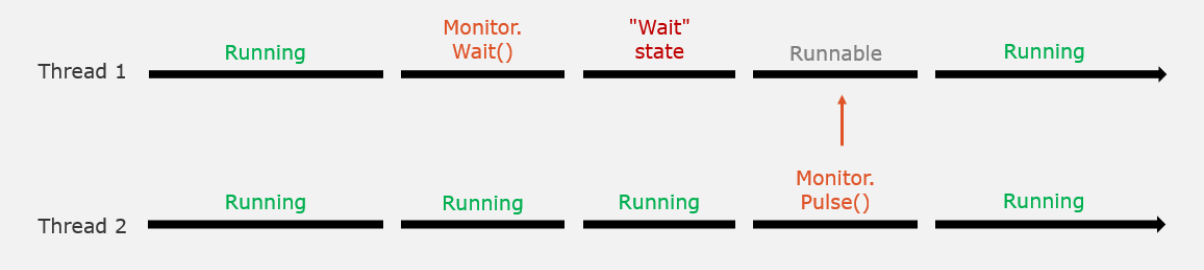
The threads can wait for specific events to occur such as input from the user, response from a database server etc.

Wait and Pulse

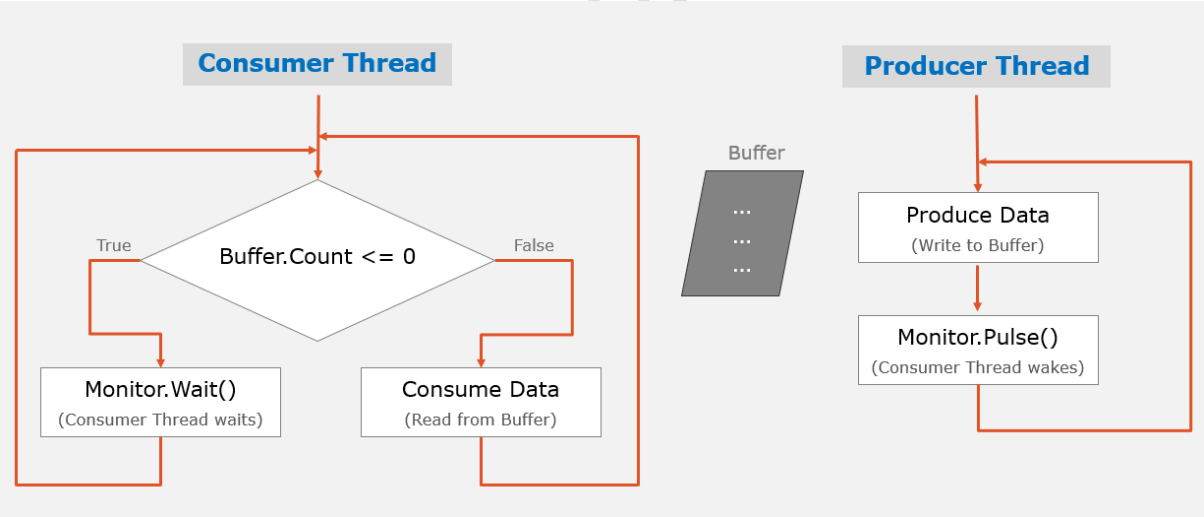
Monitor.Wait() and Monitor.Pulse()

The 'Monitor.Wait()' and 'Monitor.Pulse()' methods are used for implementation of signaling mechanism among threads.

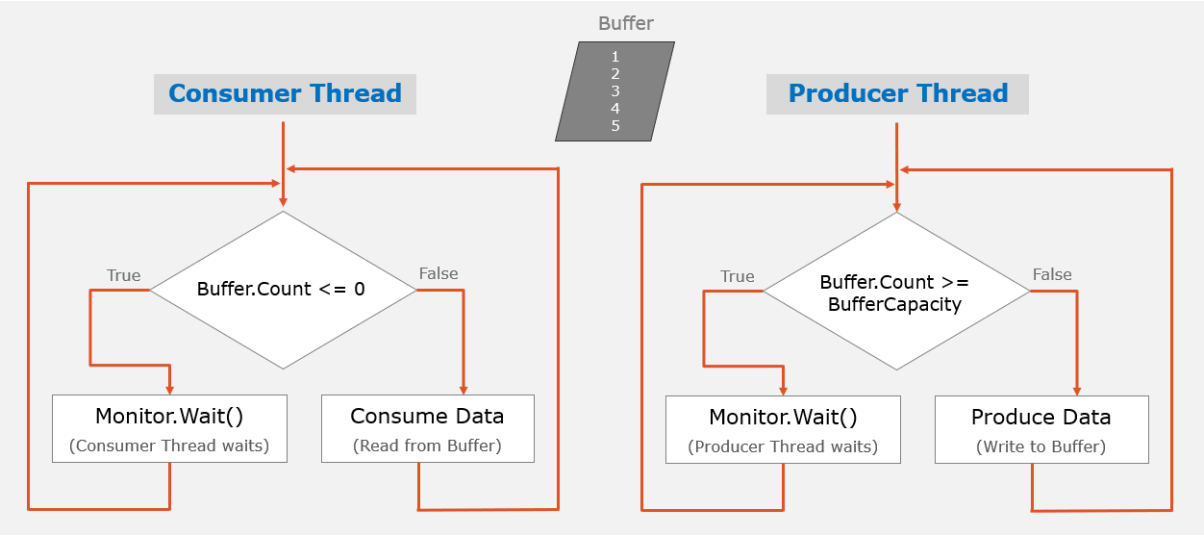
It makes a thread to wait until another thread signals it to be executed.



Monitor.Wait() and Monitor.Pulse() - Scenario



Monitor.Wait() and Monitor.Pulse() - Scenario 2



Wait()

This operation puts the current thread into a "waiting state" to wait for a signal from another thread.

Pulse()

This operation wakes up one waiting thread, allowing it to proceed.

The Wait() method should be called before the Pulse() method.

PulseAll()

This operation wakes up all waiting threads, allowing them to proceed.

Monitor [vs] ResetEvents (ManualResetEvent and AutoResetEvent)

1. Implementation Level

Reset Events:

(ManualResetEvent / AutoResetEvent)

The ResetEvents work based on implementation at OS level (based on kernel-level events).

Monitor:

The Monitor's Wait() and Pulse() methods work based on implementation at CLR level.

2. State of Signal

Reset Events:

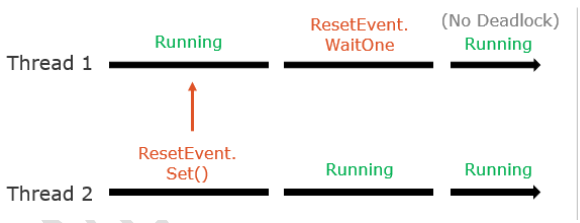
The ResetEvents maintain the state of signal (signaled / unsignaled).

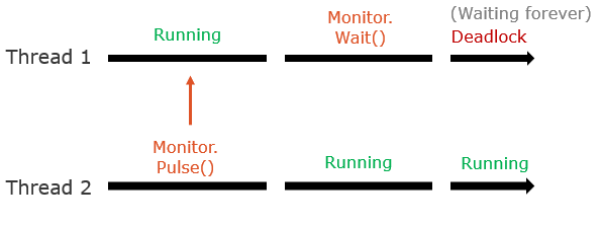
The Set() method can release a thread which is already in "waiting" state; or which enters into "waiting" state in future.

Monitor:

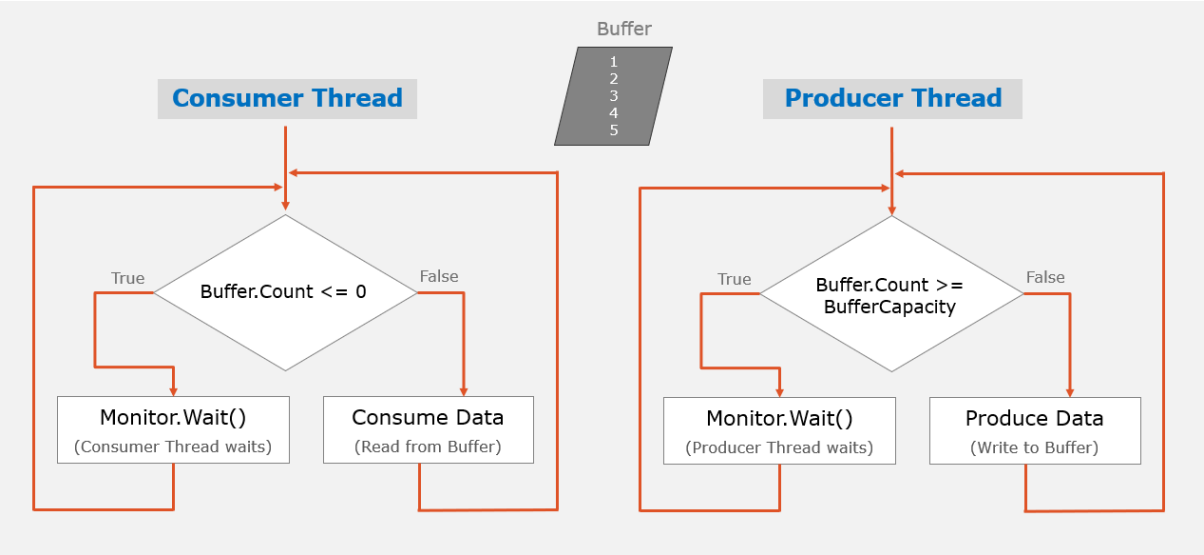
The Monitor's Wait() and Pulse() methods do not maintain any state of the signal (signaled / signaled).

The Pulse() method just releases a thread which is already in "waiting" state; but not the threads that enter into "waiting" state in future.

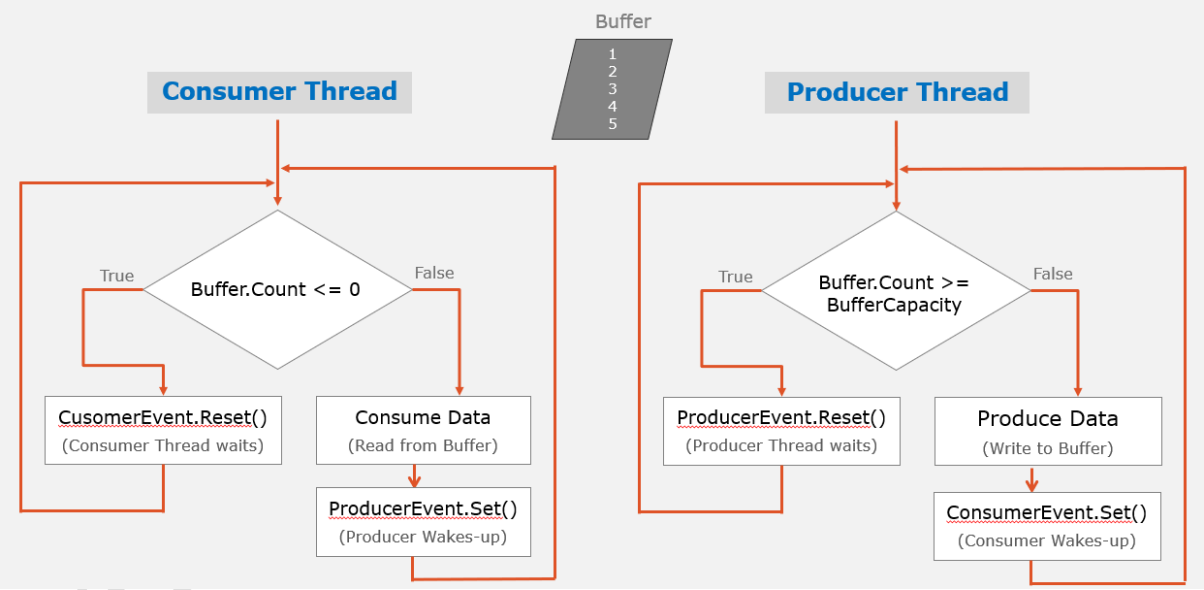




Monitor.Wait() and Monitor.Pulse() - Scenario



Monitor with ManualResetEvent



Concurrent Collections

'Concurrent collections' are specialized data structures designed for multi-threaded access that ensure thread-safe concurrent access.

Benefit: The developer NEED NOT explicitly implement thread synchronization mechanisms such as 'lock' or 'Monitor'.

Most Common Concurrent Collections

ConcurrentStack<T>

Thread-safe LIFO (Last-In-First-Out) stack.

Alternative to System.Threading.Collections.Generic.Stack<T>.

ConcurrentQueue<T>

Thread-safe FIFO (First-In-First-Out) queue.

Alternative to System.Threading.Collections.Generic.Queue<T>.

ConcurrentBag<T>

Thread-safe unordered collection.

Alternative to System.Collections.Generic.HashSet<T>.

ConcurrentDictionary<TKey, TValue>

Thread-safe key-value store.

Alternative to System.Threading.Collections.Generic.Dictionary<TKey, TValue>.

Benefits of Concurrent Collections

Implicit Thread Safety

Avoid the need for explicit locks and thread synchronization.

They provide built-in thread synchronization.

Reduced code complexity

Since there is no need to explicit thread synchronization, the multi-threading code becomes simpler and easy to maintain.

Manual Thread Synchronization [vs] Concurrent Collections

Fine-Grained Control

When you use the lock statement, you have fine-grained control over the synchronization of critical sections of your code. This means you can precisely specify which parts of your code should be protected from concurrent access, allowing for more efficient resource utilization.

Compatibility

Not all data structures and scenarios can be handled using thread-safe collections. In some cases, you may need to work with custom data structures or external resources that require explicit locking.

Non-Collection Scenarios

Thread synchronization is not limited to collections. You may encounter scenarios where you need to protect shared resources other than collections, such as database connections, files, or network resources. In these cases, the lock statement is essential.

Performance

While thread-safe collections are convenient, they can improve the performance as they are more scalable in scenarios where you have a high number of concurrent read and write operations, as they are optimized for such situations.

ConcurrentQueue

ConcurrentQueue<T> (System.Collections.Concurrent) is a thread-safe implementation of “System.Collections.Generic.Queue<T>” to create a FIFO (First-In-First-Out) queue.

It's designed specifically for scenarios where multiple threads need to enqueue and dequeue items concurrently without the need for explicit locking mechanisms.

ConcurrentQueue - Methods

void Enqueue(T item)

It adds an element at the rear-end of the queue.

e.g: queue.Enqueue(item);

bool TryDequeue(out T item)

Removes and retrieves an element at the front-end of the queue.

e.g: bool isSuccess = queue.TryDequeue(out item);

bool TryPeek(out T item)

Retrieves an element at the front-end of the queue, without removing it.

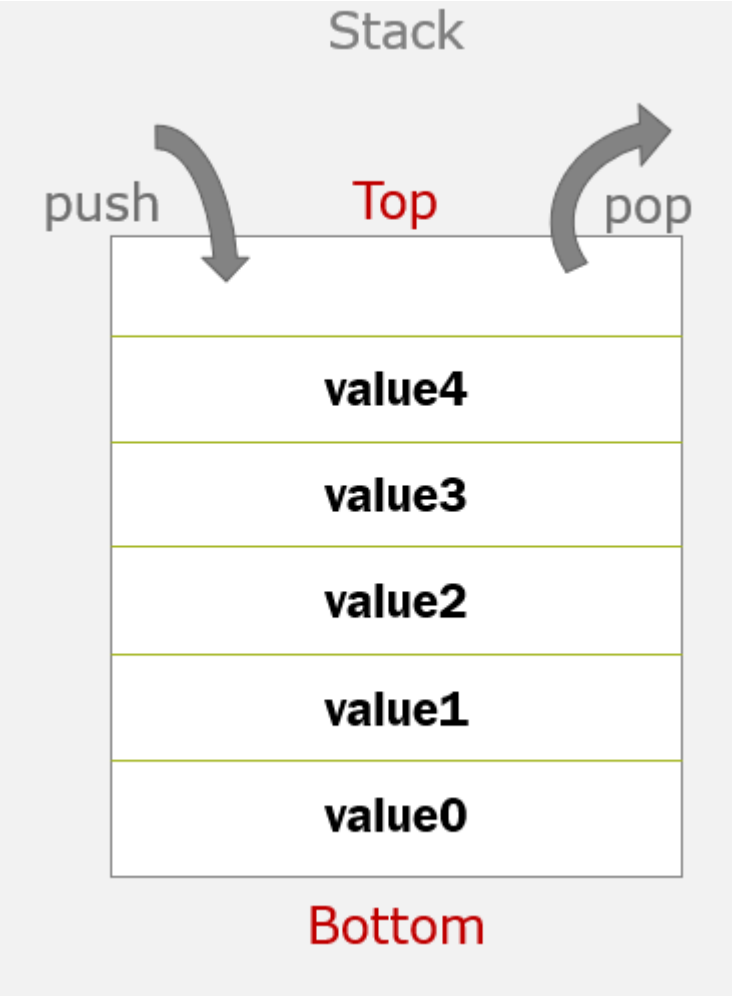
e.g: bool isSuccess = queue.TryPeek(out item);

Other Concurrent Collections

ConcurrentStack

ConcurrentStack<T> (System.Collections.Concurrent) is a thread-safe implementation of "System.Collections.Generic.Stack<T>" to create a LIFO (Last-In-First-Out) stack.

It is designed specifically for scenarios where multiple threads need to push and pop elements onto and from the stack concurrently without the need for explicit locking mechanisms.



ConcurrentStack - Methods

void Push(T item)

It adds an element at the top of the stack.

e.g: stack.Push(item);

bool TryPop(out T item)

Removes and retrieves an element at the top of the stack.

e.g: bool isSuccess = stack.TryPop(out item);

bool TryPeek(out T item)

Retrieves (peeks) an element at the top of the stack, without removing it.

e.g: bool isSuccess = stack.TryPeek(out item);

ConcurrentBag

ConcurrentBag<T> (System.Collections.Concurrent) is a thread-safe alternative to "System.Collections.Generic.HashSet<T>" to create an un-ordered collection of elements.

It is designed specifically for scenarios where multiple threads need to add and remove elements into and from the bag concurrently without the need for explicit locking mechanisms.

Collection

value0

value1

value3

value4

ConcurrentBag - Methods

void Add(T item)

It adds an element into the bag.

e.g: bag.Add(item);

bool TryTake(out T item)

Removes and retrieves an arbitrary (unordered) element from the bag.

e.g: bool isSuccess = bag.TryTake(out item);

bool TryPeek(out T item)

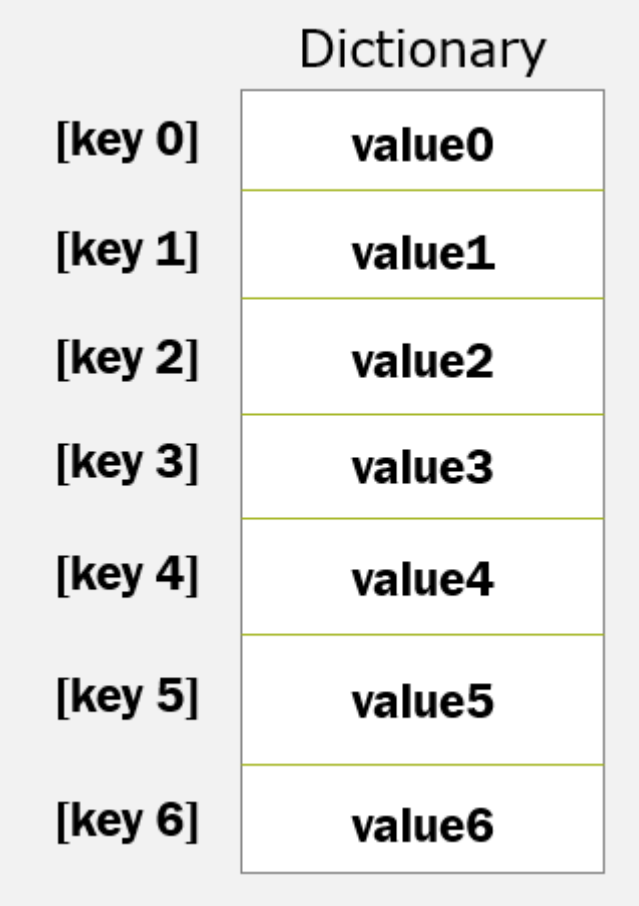
Retrieves (peeks) an arbitrary (unordered) element from the bag, without removing it.

e.g: bool isSuccess = bag.TryPeek(out item);

ConcurrentDictionary

ConcurrentDictionary<TKey, TValue> (System.Collections.Concurrent) is a thread-safe alternative to "System.Collections.Generic.Dictionary<TKey, TValue>" to create a collection of key-value pairs.

It is designed specifically for scenarios where multiple threads need to add and remove elements into and from the dictionary concurrently without the need for explicit locking mechanisms.



ConcurrentDictionary - Methods

bool TryAdd(TKey key, TValue value)

It adds an element (key-value pair) into the dictionary; returns a boolean value indicating its success.

e.g: bool isSuccess = dictionary.TryAdd(key, value);

bool TryGetValue(TKey key, out TValue value)

Retrieves the value at the specified key from the dictionary; returns a boolean value indicating its success.

e.g: bool isSuccess = dictionary.TryGetValue(key, out value);

ConcurrentDictionary - Methods

bool TryRemove(TKey key, out TValue value)

It removes the element (key-value pair) from the dictionary; and retrieves the same value; returns a boolean value indicating its success.

e.g: bool isSuccess = dictionary.TryRemove(key, out value);

bool TryUpdate(TKey key, TValue value, TValue comparisonValue)

•Updates the value associated with the key to newValue if the existing value with key is equal to comparisonValue; returns a boolean value indicating its success.

•e.g: bool isSuccess = dictionary.TryUpdate(key, newValue, oldValue);

bool TryRemove(TKey key, out TValue value)

It removes the element (key-value pair) from the dictionary; and retrieves the same value; returns a boolean value indicating its success.

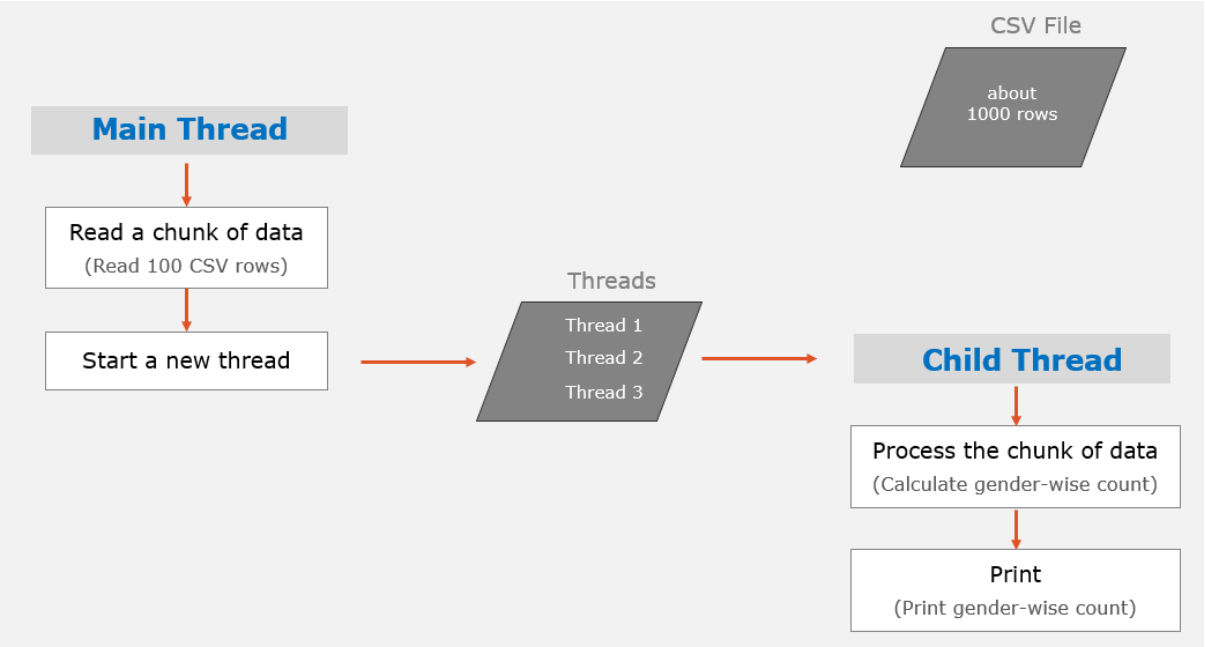
e.g: bool isSuccess = dictionary.TryRemove(key, out value);

bool TryUpdate(TKey key, TValue value, TValue comparisonValue)

Updates the value associated with the key to newValue if the existing value with key is equal to comparisonValue; returns a boolean value indicating its success.

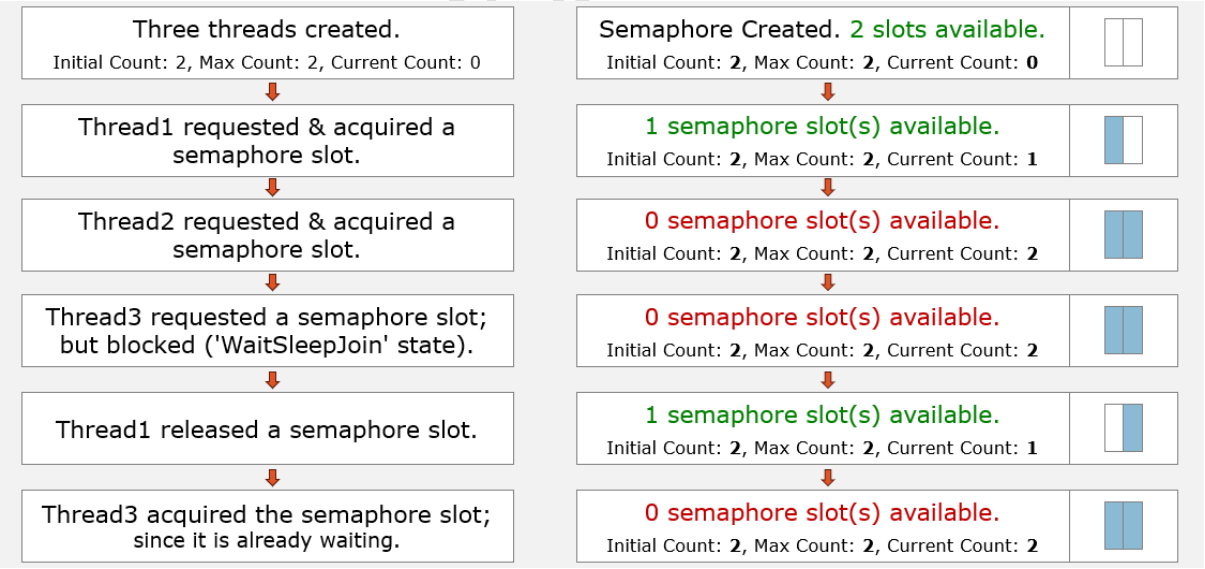
e.g: bool isSuccess = dictionary.TryUpdate(key, newValue, oldValue);

CSV with Threads



Semaphore

Semaphore is a synchronization primitive to control the number of concurrent threads that can access one or more shared resources.



Semaphore - Methods

WaitOne()

This method is used acquire a semaphore lock (slot).

If a semaphore slot is available, the current thread acquires the semaphore lock and decreases the available semaphore locks count.

If a semaphore slot is not available, the current thread will be blocked (kept under 'WaitSleepJoin' state) until a slot becomes available.

WaitOne(int millisecondsTimeout)

This method is used acquire a semaphore lock (slot).

If a semaphore slot is available, the thread acquires the semaphore lock and decreases the available semaphore locks count; and it returns '*true*'.

If a semaphore slot is not available, the thread will be blocked (kept under 'WaitSleepJoin' state) until the specified timeout elapsed; after that it returns '*false*' if still semaphore lock is not available.

Release()

This method is used release a semaphore lock (slot).

This method is used when a thread is finished using a shared resource and wants to release it. So it increments the available semaphore locks count.

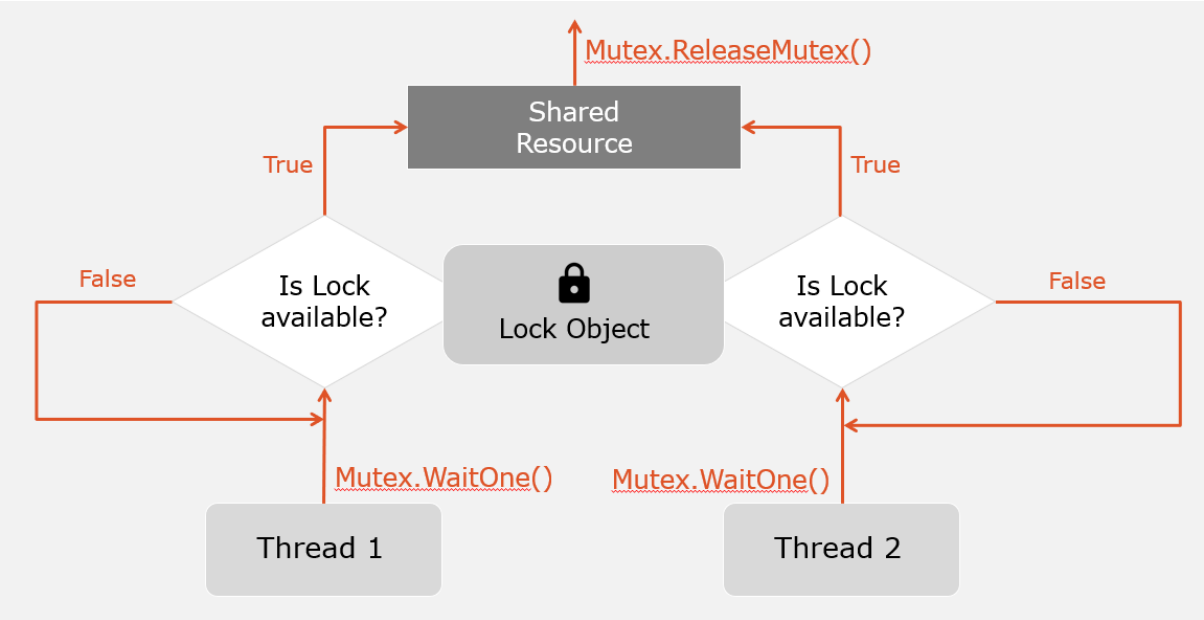
Mutex

The 'Mutex' class is a synchronization primitive that enables thread synchronization.

It ensures that only one thread can execute the code within the critical section at a time; thus preventing race conditions and deadlocks.

It is an alternative to 'Monitor'.

Thread Synchronization with 'Mutex' class



WaitOne()

This method is used acquire the mutex ownership (lock).

If the mutex is not currently owned by any other thread or process, the calling thread immediately gains ownership.

If the mutex is currently owned by another thread or process, the current thread will be blocked until the mutex becomes available.

ReleaseMutex()

This method is used release ownership of mutex.

After releasing mutex, it allows other threads or processes waiting for it to acquire ownership of mutex.

The WaitOne() should be called at the beginning of critical section; ReleaseMutex() at the end of critical section that accesses any shared resources.

Monitor [vs] Mutex

Lock object

Monitor requires you to create an explicit lock object, because the methods of 'Monitor' class are static methods.

Mutex allows you to create an object first; and it itself acts as a lock object, without needing you to create an explicit lock object.

Inter-process Communication

Monitor can only be accessed within the single process.

Mutex can be accessed across multiple processes (referred IPC - Inter Process Communication) using named pipes or sockets.

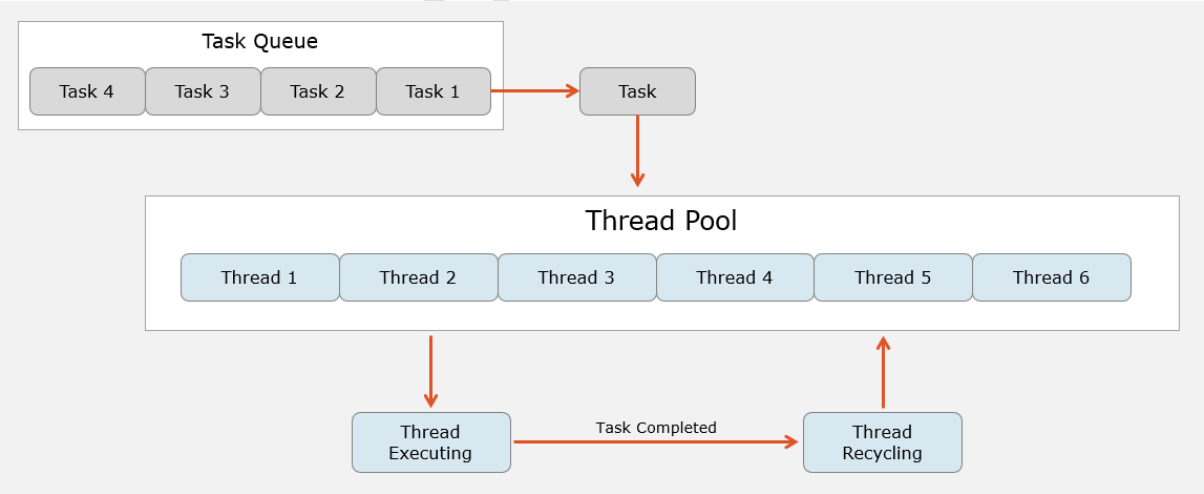
Note: Monitor is recommended because of its faster performance and light-weight nature compared to Mutex, unless you need IPC.

Thread Pool

The Thread Pool is a collection of worker threads that are pre-allocated and managed by the .NET runtime.

These worker threads are available for executing tasks on a first-come, first-served basis.

The number of threads in the pool is typically determined by the system and can be adjusted based on system resources and application demands.



How thread pool works?

Task Queuing

When you have a task that needs to be executed asynchronously, you can submit it to the Thread Pool.

The Thread Pool maintains a queue of tasks, and as soon as a worker thread becomes available, it dequeues a task and executes it. It operates on a "first-come, first-served" basis, meaning that, the tasks are executed in the order they are added to the queue.

This queuing mechanism ensures that tasks are executed efficiently without the overhead of creating and destroying threads for each task.

If there are more tasks queued than there are available worker threads, and the system allows it, the Thread Pool can create additional worker threads to handle the load.

Conversely, if there are idle worker threads, the Thread Pool may retire them to conserve system resources.

Task Completion

After a worker thread completes the execution of a task, it becomes available to pick up another task from the queue. Thread Pool itself does not maintain a separate queue for completed tasks.

The completed task is considered done, and any resources associated with it are released.

Advantages of Thread Pool

Improved Performance

Thread pooling efficiently utilizes available CPU resources.

Because, the thread pool's queuing mechanism ensures that tasks are executed efficiently without the overhead of creating and destroying threads for each task.

Scalability

Thread pooling easily adapts to increased number of tasks.

Because, the Thread Pool can create additional worker threads to handle the load.

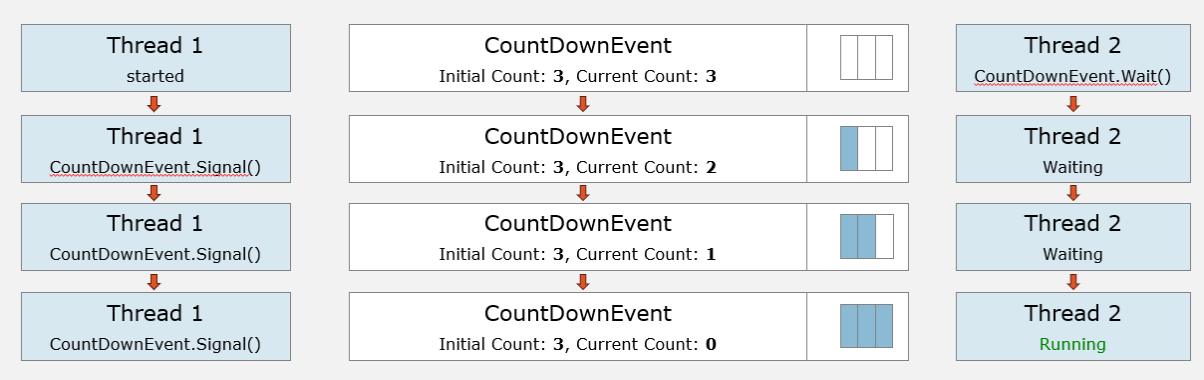
CountDownEvent

CountDownEvent is a synchronization primitive that allows one or more threads to wait until a specified number of signals (counts) have occurred before they proceed.

First, you will specify the number of signals needed to unblock waiting threads.

Then, the calling (waiting) thread can wait until the specified number of signals have occurred.

CountDownEvent - How it works



Wait()

Blocks the calling thread until the current count reaches zero.

If the count is already zero, this method does not block the thread.

Signal()

Decrements the current count by one.

If the count becomes zero, any threads waiting in the Wait() method will be unblocked and allowed to proceed.

If the count is already zero, this method does not block the calling thread.

int InitialCount (property)

Represents the value of count that should occur before releasing the waiting thread(s).

int CurrentCount (property)

Gets the current count of the CountDownEvent.

Reset()

Resets the count to the initial count specified when the CountDownEvent was created.

This method can be used to reuse the CountDownEvent for a new set of signals.

Optionally, you can supply a specific value of initial count.