Dot Net Technologies (Windows and Web API)

HOME

Text

Popular Posts

(no title)
C# Threading Interview Questions & Answers Introduction and Concepts Thread Life Cycle What is the T hread ? ...

(no title)
C # E xperienced 125 I nterview Q uestions and ANS 1. What is C#? C# (pronounced "C ...

(no title)
C# Experienced Interview and Questions Dispose vs Final? Dispose is a method for realse from the memory for an object. Finalize is u...

(no title)
WPF Interview Questions and Answers How to define a button USING XAML? To define a button in WPD using XAML use the...

(no title)
SQL Server Connection Pooling What is the meaning of SQL Connection pooling ? The server hosts more than 50 databases that serve an AS...

(no title)
LINQ interview question and answers What is a Lambda expression? A Lambda expression is nothing but an Anonymous Function, can conta...

(no title)
C#, Socket Programming Interview Questions and Answers What Is a Socket? A server application normally listens to a specific port ...

(no title)
Basic SQL Interview Questions Question : What type of joins have you used? Question : How can you combine two tables/views togethe...

(no title)
GC Generations In C# The heap is organized into generations so it can handle long-lived and short-lived objects. Garbage...

(no title)
Optimizing Connectivity (SQL Server) To optimize performance, you can modify the settings based on the bandwidth of the network connecti...

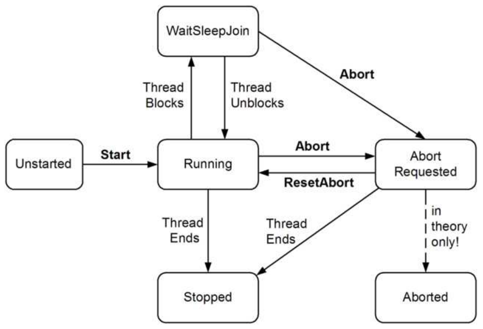
Blogger templates

POSTED BY PRASAD KM | 06:11 CATEGORIES: THREADING

C# Threading Interview Questions & Answers

Introduction and Concepts

Thread Life Cycle



What is the Thread ?

C# supports parallel execution of code through multithreading. A thread is an independent execution path, able to run simultaneously with other threads.

```
 class ThreadTest
{
static void Main()
{
Thread t = new Thread (WriteY); // Kick off a new thread
t.Start(); // running WriteY()
// Simultaneously, do something on the main thread.
for (int i = 0; i < 1000; i++) Console.Write ("x");
}
static void WriteY()
{
for (int i = 0; i < 1000; i++) Console.Write ("y");
}
}
```

The CLR assigns each thread its own memory stack so that local variables are kept separate. In the next example, we define a method with a local variable, then call the method simultaneously on the main thread and a newly created

thread:

```
static void Main()
{
new Thread (Go).Start(); // Call Go() on a new thread
Go(); // Call Go() on the main thread
}
static void Go()
```

```
{
// Declare and use a local variable - 'cycles'
for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

Out Put

```
??????????
```

What is the difference Between Join and Sleep ?

You can wait for another thread to end by calling its Join method. For example:

```
static void Main()
{
Thread t = new Thread (Go);
t.Start();
t.Join();
Console.WriteLine ("Thread t has ended!");
}
static void Go()
{
for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

This prints "y" 1,000 times, followed by "Thread t has ended!" immediately afterward. You can include a timeout when calling Join, either in milliseconds or as a TimeSpan. It then returns true if the thread ended or false if it timed out.

What is the difference Between Threads vs Processes ?

A thread is analogous to the operating system process in which your
application runs. Just as processes run in parallel on a computer,
threads run in parallel within a single process. Processes are fully
isolated from each other; threads have just a limited degree of
isolation. In particular, threads share (heap) memory with other threads running in the same application. This, in part, is
why threading is useful: one thread can fetch data in the background, for instance, while another thread can display the
data as it arrives.

What is the difference Between Passing Data to a Thread?

The easiest way to pass arguments to a thread's target method is to execute a lambda expression that calls the method
with the desired arguments:

```
static void Main()
{
Thread t = new Thread ( () => Print ("Hello from t!") );
t.Start();
}
static void Print (string message)
{
Console.WriteLine (message);
}
```

With this approach, you can pass in any number of arguments to the method. You can even wrap the entire
implementation in a multi-statement lambda:

```
new Thread (() =>
{
Console.WriteLine ("I'm running on another thread!");
Console.WriteLine ("This is so easy!");
}).Start();
```

You can do the same thing almost as easily in C# 2.0 with anonymous methods:

```
new Thread (delegate()
{
...
}).Start();
```

Another technique is to pass an argument into Thread's Start method:

What is the Naming Threads ?

Each thread has a Name property that you can set for the benefit of debugging.
This is particularly useful in Visual Studio, since the thread's name is displayed
in the Threads Window and Debug Location toolbar. You can set a thread's
name just once; attempts to change it later will throw an exception.
The static Thread.CurrentThread property gives you the currently executing
thread. In the following example, we set the main thread's name:

```
class ThreadNaming
{
static void Main()
{
Thread.CurrentThread.Name = "main";
Thread worker = new Thread (Go);
worker.Name = "worker";
worker.Start();
Go();
}
static void Go()
{
Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
}
}
```

What is the Thread Priority?

A thread's Priority property determines how much execution time it gets relative to other active threads in the
operating system, on the following scale:

enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }

What is the Asynchronous delegates?ThreadPool.QueueUserWorkItem doesn't provide an easy mechanism for getting return values back from a thread after it has finished executing. Asynchronous delegate invocations (asynchronous delegates for short) solve this, allowing any number of typed arguments to be passed in both directions. Furthermore, unhandled exceptions on asynchronous delegates are conveniently rethrown on the original thread (or more accurately, the thread that calls EndInvoke), and so they don't need explicit handling.

Here's how you start a worker task via an asynchronous delegate:

1. Instantiate a delegate targeting the method you want to run in parallel (typically one of the predefined Func
delegates).
2. Call BeginInvoke on the delegate, saving its IAsyncResult return value.
BeginInvoke returns immediately to the caller. You can then perform other activities while the pooled
thread is working.
3. When you need the results, call EndInvoke on the delegate, passing in the saved IAsyncResult object.

What is the difference Between Monitor.Enter and Monitor.Exit

C#'s lock statement is in fact a syntactic shortcut for a call to the methods Monitor.Enter and Monitor.Exit, with a try/finally block. Here's (a simplified version of) what's actually happening within the Go method of the preceding
example:

```
Monitor.Enter (_locker);
try
{
if (_val2 != 0) Console.WriteLine (_val1 / _val2);
_val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

Calling Monitor.Exit without first calling Monitor.Enter on the same object throws an exception.

What is the meaning of Deadlocks

A deadlock happens when two threads each wait for a resource held by the other, so neither can proceed. The easiest way to illustrate this is with two locks:

```
object locker1 = new object();
object locker2 = new object();
new Thread (() => {
lock (locker1)
{
Thread.Sleep (1000);
lock (locker2); // Deadlock
}
}).Start();
lock (locker2)
{
Thread.Sleep (1000);
lock (locker1); // Deadlock
}
```

More elaborate deadlocking chains can be created with three or more threads.
Deadlocking is one of the hardest problems in multithreading—especially when there are many interrelated objects. Fundamentally, the hard problem is that you can't be sure what locks your caller has taken out.
So, you might innocently lock private field a within your class x, unaware that your caller (or caller's caller) has already locked field b within class y. Meanwhile, another thread is doing the reverse—creating a deadlock. Ironically, the problem is exacerbated by (good) object-oriented design patterns, because such patterns create call chains that are not determined until runtime. The popular advice, "lock objects in a consistent order to avoid deadlocks," although helpful in our initial example, is hard to apply to the scenario just described. A better strategy is to be wary of locking around calling methods in objects that may have references back to your own object. Also, consider whether you really need to lock around calling methods in other classes (often you do—as we'll see later—but sometimes there are other options). Relying more on declarative and data parallelism, immutable types, and nonblocking synchronization constructs, can lessen the need for locking.

What is the meaning of Mutex

A Mutex is like a C# lock, but it can work across multiple processes. In other words, Mutex can be computer-wide as
well as application-wide.
With a Mutex class, you call the WaitOne method to lock and ReleaseMutex to unlock.
Closing or disposing a Mutex automatically releases it. Just as with the lock statement, a Mutex can be released only from the same thread that obtained it. A common use for a cross-process Mutex is to ensure that only one instance of a program can run at a time.
Here's how it's done:

```
class OneAtATimePlease
{
static void Main()
{
// Naming a Mutex makes it available computer-wide. Use a name that's
// unique to your company and application (e.g., include your URL).
using (var mutex = new Mutex (false, "oreilly.com OneAtATimeDemo"))
{
// Wait a few seconds if contended, in case another instance
// of the program is still in the process of shutting down.
if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
{
Console.WriteLine ("Another instance of the app is running. Bye!");
return;
}
RunProgram();
}
}
static void RunProgram()
{
Console.WriteLine ("Running. Press Enter to exit");
Console.ReadLine();
}
}
```

What is the meaning Semaphore ?

A semaphore is like a nightclub: it has a certain capacity, enforced by a bouncer. Once it's full, no more people can enter, and a queue builds up outside. Then, for each person that leaves, one person enters from the head of the queue. The constructor requires a minimum of two arguments: the number of places currently available in the nightclub and the club's total capacity.

A semaphore with a capacity of one is similar to a Mutex or lock, except that the semaphore has no “owner”—it's thread-agnostic. Any thread can call Release on a Semaphore, whereas with Mutex and lock, only the thread that obtained the lock can release it.

Semaphores can be useful in limiting concurrency—preventing too many threads from executing a particular piece of code at once. In the following example, five threads try to enter a nightclub that allows only three **threads in at once:**

```
class TheClub // No door lists!
{
static SemaphoreSlim _sem = new SemaphoreSlim (3); // Capacity of 3
static void Main()
{
for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
}
static void Enter (object id)
{
Console.WriteLine (id + " wants to enter");
_sem.Wait();
Console.WriteLine (id + " is in!"); // Only three threads
Thread.Sleep (1000 * (int) id); // can be here at
Console.WriteLine (id + " is leaving"); // a time.
_sem.Release();
}
}
```

Out Put

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

If the Sleep statement was instead performing intensive disk I/O, the Semaphore would improve overall performance
by limiting excessive concurrent hard-drive activity.
A Semaphore, if named, can span processes in the same way as a Mutex.

What is synchronization in respect to multi-threading in C#?

With respect to multi-threading, synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one Java thread to modify a shared variable while another thread is in the process of using or updating same shared variable. This usually leads to erroneous behavior or program.

Explain different way of using thread?

A Java thread could be implemented by using Runnable interface or by extending the Thread class. The Runnable is more advantageous, when you are going for multiple inheritance.

What is the difference between Thread.start() & Thread.run() method?

Thread.start() method (native method) of Thread class actually does the job of running the Thread.run() method in a thread. If we directly call Thread.run() method it will executed in same thread, so does not solve the purpose of creating a new thread.

Why do we need run() & start() method both. Can we achieve it with only run method?

We need run() & start() method both because JVM needs to create a separate thread which can not be differentiated from a normal method call. So this job is done by start method native implementation which has to be explicitly called. Another advantage of having these two methods is we can have any object run as a thread if it implements Runnable interface. This is to avoid Java's multiple inheritance problems which will make it difficult to inherit another class with Thread.

What is ThreadLocal class? How can it be used?

Below are some key points about ThreadLocal variables

- A thread-local variable effectively provides a separate copy of its value for each thread that uses it.
- ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread
- In case when multiple threads access a ThreadLocal instance, separate copy of Threadlocal variable is maintained for each thread.
- Common use is seen in DAO pattern where the DAO class can be singleton but the Database connection can be maintained separately for each thread. (Per Thread Singleton)

ThreadLocal variable are difficult to understand and I have found below reference links very useful in getting better understanding on them

When InvalidMonitorStateException is thrown? Why?

This exception is thrown when you try to call wait()/notify()/notifyAll() any of these methods for an Object from a point in your program where u are NOT having a lock on that object.(i.e. u r not executing any synchronized block/method of that object and still trying to call wait()/notify()/notifyAll()) wait(), notify() and notifyAll() all throw IllegalMonitorStateException. since This exception is a subclass of RuntimeException so we r not bound to catch it (although u may if u want to). and being a RuntimeException this exception is not mentioned in the signature of wait(), notify(), notifyAll() methods.

What is the difference between sleep(), suspend() and wait() ?

Thread.sleep() takes the current thread to a "Not Runnable" state for specified amount of time. The thread holds the monitors it has acquired. For example, if a thread is running a synchronized block or method and sleep method is called then no other thread will be able to enter this block or method. The sleeping thread can wake up when some other thread calls t.interrupt on it. Note that sleep is a static method, that means it always affects the current thread (the one executing sleep method). A common mistake is trying to call t2.sleep() where t2 is a different thread; even then, it is the current thread that will sleep, not the t2 thread. thread.suspend() is deprecated method. Its possible to send other threads into suspended state by making a suspend method call. In suspended state a thread keeps all its monitors and can not be interrupted. This may cause deadlocks therefore it has been deprecated. object.wait() call also takes the current thread into a "Not Runnable" state, just like sleep(), but with a slight change. Wait method is invoked on a lock object, not thread.

Here is the sequence of operations you can think

- A thread T1 is already running a synchronized block with a lock on object - lets say "lockObject"
- Another thread T2 comes to execute the synchronized block and find that its already acquired.
- Now T2 calls lockObject.wait() method for waiting on lock to be release by T1 thread.
- T1 thread finishes all its synchronized block work.
- T1 thread calls lockObject.notifyAll() to notify all waiting threads that its done using the lock.
- Since T2 thread is first in the queue of waiting it acquires the lock and starts processing.

What happens when I make a static method as synchronized?

Synchronized static methods have a lock on the class "Class", so when a thread enters a synchronized static method, the class itself gets locked by the thread monitor and no other thread can enter any static synchronized methods on that class. This is unlike instance methods, as multiple threads can access "same synchronized instance methods" at same time for different instances.

Can a thread call a non-synchronized instance method of an Object when a synchronized method is being executed ?

Yes, a Non synchronized method can always be called without any problem. In fact Java does not do any check for a non-synchronized method. The Lock object check is performed only for synchronized methods/blocks. In case the method is not declared synchronized Jave will call even if you are playing with shared data. So you have to be careful while doing such thing. The decision of declaring a method as synchronized has to be based on critical section access. If your method does not access a critical section (shared resource or data structure) it need not be declared synchronized. Below is the example which demonstrates this, The Common class has two methods synchronizedMethod1() and method1() MyThread class is calling both the methods in separate threads,

view plainprint?

```
public class Common {

```

```
public synchronized void synchronizedMethod1() {
System.out.println("synchronizedMethod1 called");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("synchronizedMethod1 done");
}
public void method1() {
System.out.println("Method 1 called");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Method 1 done");
}
}
```

view plainprint?

```
public class MyThread extends Thread {
private int id = 0;
private Common common;

public MyThread(String name, int no, Common object) {
super(name);
common = object;
id = no;
}

public void run() {
System.out.println("Running Thread" + this.getName());
try {
if (id == 0) {
common.synchronizedMethod1();
} else {
common.method1();
}
} catch (Exception e) {
e.printStackTrace();
}
}

public static void main(String[] args) {
Common c = new Common();
MyThread t1 = new MyThread("MyThread-1", 0, c);
MyThread t2 = new MyThread("MyThread-2", 1, c);
t1.start();
t2.start();
}
}
```

Here is the output of the program

view plainprint?

```
Running ThreadMyThread-1
synchronizedMethod1 called
Running ThreadMyThread-2
Method 1 called
synchronizedMethod1 done
Method 1 done
```

This shows that method1() - is called even though the synchronizedMethod1() was in execution.

Can two threads call two different synchronized instance methods of an Object?

No. If a object has synchronized instance methods then the Object itself is used a lock object for controlling the synchronization. Therefore all other instance methods need to wait until previous method call is completed. See the below sample code which demonstrate it very clearly. The Class Common has 2 methods called synchronizedMethod1() and synchronizedMethod2()

MyThread class is calling both the methods

view plainprint?

```
public class Common {
public synchronized void synchronizedMethod1() {
System.out.println("synchronizedMethod1 called");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("synchronizedMethod1 done");
}

public synchronized void synchronizedMethod2() {
System.out.println("synchronizedMethod2 called");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("synchronizedMethod2 done");
}
}
```

view plainprint?

```
public class MyThread extends Thread {
private int id = 0;
private Common common;

public MyThread(String name, int no, Common object) {
super(name);
common = object;
id = no;
}

public void run() {
System.out.println("Running Thread" + this.getName());
try {
if (id == 0) {
common.synchronizedMethod1();
} else {
common.synchronizedMethod2();
}
} catch (Exception e) {
e.printStackTrace();
}
}

public static void main(String[] args) {
Common c = new Common();
MyThread t1 = new MyThread("MyThread-1", 0, c);
MyThread t2 = new MyThread("MyThread-2", 1, c);
t1.start();
t2.start();
}
}
```

What is a deadlock?

Deadlock is a situation where two or more threads are blocked forever, waiting for each other. This may occur when two threads, each having a lock on one resource, attempt to acquire a lock on the other's resource. Each thread would wait indefinitely for the other to release the lock, unless one of the user processes is terminated. In terms of Java API, thread deadlock can occur in following conditions:

- When two threads call Thread.join() on each other.
- When two threads use nested synchronized blocks to lock two objects and the blocks lock the same objects in different order.

What is Starvation? and What is a Livelock?

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

LiveLock

Livelock occurs when all threads are blocked, or are otherwise unable to proceed due to unavailability of required resources, and the non-existence of any unblocked thread to make those resources available. In terms of Java API, thread livelock can occur in following conditions:

- When all the threads in a program execute Object.wait(0) on an object with zero parameter. The program is live-locked and cannot proceed until one or more threads call Object.notify() or Object.notifyAll() on the relevant objects. Because all the threads are blocked, neither call can be made.
- When all the threads in a program are stuck in infinite loops.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked. Starvation occurs when one thread cannot access the CPU because one or more other threads are monopolizing the CPU. In Java, thread starvation can be caused by setting thread priorities inappropriately. A lower-priority thread can be starved by higher-priority threads if the higher-priority threads do not yield control of the CPU from time to time.

How to find a deadlock has occurred in Java? How to detect a Deadlock in Java?

Earlier versions of Java had no mechanism to handle/detect deadlock. Since JDK 1.5 there are some powerful methods added in the java.lang.management package to diagnose and detect deadlocks. The java.lang.management.ThreadMXBean interface is management interface for the thread system of the Java virtual machine. It has two methods which can leveraged to detect deadlock in a Java application.

- findMonitorDeadlockedThreads() - This method can be used to detect cycles of threads that are in deadlock waiting to acquire object monitors. It returns an array of thread IDs that are deadlocked waiting on monitor.
- findDeadlockedThreads() - It returns an array of thread IDs that are deadlocked waiting on monitor or ownable synchronizers.

What is immutable object? How does it help in writing concurrent application?

An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code. Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state. Examples of immutable objects from the JDK include String and Integer. Immutable objects greatly simplify your multi threaded program, since they are

- Simple to construct, test, and use.
- Automatically thread-safe and have no synchronization issues.

To create a object immutable You need to make the class final and all its member final so that once objects gets crated no one can modify its state. You can achieve same functionality by making member as non final but private and not modifying them except in constructor.

How will you take thread dump in Java? How will you analyze Thread dump?

A Thread Dump is a complete list of active threads. A java thread dump is a way of finding out what each thread in the JVM is doing at a particular point of time. This is especially useful when your java application seems to have some performance issues. Thread dump will help you to find out which thread is causing this. There are several ways to take thread dumps from a JVM. It is highly recommended to take more than 1 thread dump and analyze the results based on it. Follow below steps to take thread dump of a java process

- Step 1

On UNIX, Linux and Mac OSX Environment run below command:

```
ps -el | grep java
```

On Windows:

Press Ctrl+Shift+Esc to open the task manager and find the PID of the java process

- Step 2:

  Use jstack command to print the Java stack traces for a given Java process PID

  ```
  jstack [PID]
  ```

What is a thread leak? What does it mean in C#?

Thread leak is when a application does not release references to a thread object properly. Due to this some Threads do not get garbage collected and the number of unused threads grow with time. Thread leak can often cause serious issues on a Java application since over a period of time too many threads will be created but not released and may cause applications to respond slow or hang.

How can I trace whether the application has a thread leak?

If an application has thread leak then with time it will have too many unused threads. Try to find out what type of threads is leaking out. This can be done using following ways

- Give unique and descriptive names to the threads created in application. - Add log entry in all thread at various entry and exit points in threads.
- Change debugging config levels (debug, info, error etc) and analyze log messages.
- When you find the class that is leaking out threads check how new threads are instantiated and how they're closed.
- Make sure the thread is Guaranteed to close properly by doing following - Handling all Exceptions properly.
- Make sure the thread is Guaranteed to close properly by doing following
  - Handling all Exceptions properly.
  - releasing all resources (e.g. connections, files etc) before it closes.

What is thread pool? Why should we use thread pools?

A thread pool is a collection of threads on which task can be scheduled. Instead of creating a new thread for each task, you can have one of the threads from the thread pool pulled out of the pool and assigned to the task. When the thread is finished with the task, it adds itself back to the pool and waits for another assignment. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Below are key reasons to use a Thread Pool

- Using thread pools minimizes the JVM overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and de-allocating many thread objects creates a significant memory management overhead.
- You have control over the maximum number of tasks that are being processed in parallel (= number of threads in the pool).

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Can we synchronize the run method? If yes then what will be the behavior?

Yes, the run method of a runnable class can be synchronized. If you make run method synchronized then the lock on runnable object will be occupied before executing the run method. In case we start multiple threads using the same runnable object in the constructor of the Thread then it would work. But until the 1st thread ends the 2nd thread cannot start and until the 2nd thread ends the next cannot start as all the threads depend on lock on same object g+1 +5 Recommend this on Google

NO COMMENTS

Newer Post Home Older Post

0 comments:

Post a Comment

Subscribe to: Post Comments (Atom)

Copyright © 2012 Dot Net Technologies (Windows and Web API) | Powered by Blogger
Design by Web2feel | Blogger Template by NewBloggerThemes.com