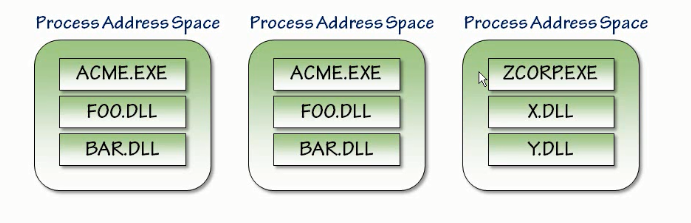
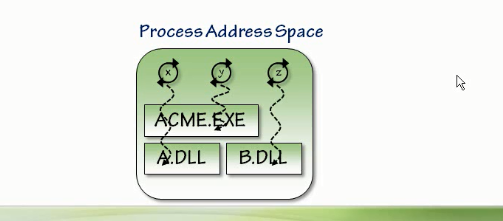
**Threading**

Operating systems use processes to separate the different applications that they are executing. Threads are the basic unit to which an operating system allocates processor time, and more than one thread can be executing code inside that process. Each thread maintains exception handlers, a scheduling priority, and a set of structures the system uses to save the thread context until it is scheduled. The thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack, in the address space of the thread's host process.

The .NET Framework further subdivides an operating system process into lightweight managed subprocesses, called application domains, represented by [System.AppDomain](https://docs.microsoft.com/en-us/dotnet/api/system.appdomain). One or more managed threads (represented by [System.Threading.Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread)) can run in one or any number of application domains within the same managed process. Although each application domain is started with a single thread, code in that application domain can create additional application domains and additional threads. The result is that a managed thread can move freely between application domains inside the same managed process; you might have only one thread moving among several application domains.



Thread Started within a process.



**Exceptions in Managed Threads**

* A [ThreadAbortException](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadabortexception) is thrown in a thread because [Abort](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.abort) was called.
* An [AppDomainUnloadedException](https://docs.microsoft.com/en-us/dotnet/api/system.appdomainunloadedexception) is thrown in a thread because the application domain in which the thread is executing is being unloaded.
* The common language runtime or a host process terminates the thread by throwing an internal exception.

If any of these exceptions are unhandled in threads created by the common language runtime, the exception terminates the thread, but the common language runtime does not allow the exception to proceed further.

(child thread exception cannot be caught in the main Thread)

If these exceptions are unhandled in the main thread, or in threads that entered the runtime from unmanaged code, they proceed normally, resulting in termination of the application.

**Creating threads and passing data at start time**

When an operating-system process is created, the operating system injects a thread to execute code in that process, including any original application domain. From that point on, application domains can be created and destroyed without any operating system threads necessarily being created or destroyed. If the code being executed is managed code, then a [Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread) object for the thread executing in the current application domain can be obtained by retrieving the static [CurrentThread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.currentthread) property of type [Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread).

**Creating a thread**

Creating a new [Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread) object creates a new managed thread. The [Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread) class has constructors that take a [ThreadStart](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadstart) delegate or a [ParameterizedThreadStart](https://docs.microsoft.com/en-us/dotnet/api/system.threading.parameterizedthreadstart) delegate; the delegate wraps the method that is invoked by the new thread when you call the [Start](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.start) method. Calling [Start](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.start) more than once causes a [ThreadStateException](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadstateexception) to be thrown.

Thread InstanceCaller = new Thread(

new ThreadStart(serverObject.InstanceMethod));

// Start the thread.

InstanceCaller.Start();

**Pausing and interrupting threads**

## **The Thread.Sleep method**

Calling the [Thread.Sleep](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.sleep) method causes the current thread to immediately block for the number of milliseconds or the time interval you pass to the method, and yields the remainder of its time slice to another thread. Once that interval elapses, the sleeping thread resumes execution.

One thread cannot call [Thread.Sleep](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.sleep) on another thread. [Thread.Sleep](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.sleep) is a static method that always causes the current thread to sleep.

Calling [Thread.Sleep](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.sleep) with a value of [Timeout.Infinite](https://docs.microsoft.com/en-us/dotnet/api/system.threading.timeout.infinite) causes a thread to sleep until it is interrupted by another thread that calls the [Thread.Interrupt](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.interrupt) method on the sleeping thread, or until it is terminated by a call to its [Thread.Abort](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.abort) method.