# I.

### What is os process ?

An instance of a computer program that is being executed, and also a group of the following resources:

- Machine code
- Memory - call stack, local stack, heap
- Operating system specific resources
- Security attributes - process owner, set of permissions
- Processor state(context)

### What is thread ?

A thread is an independent execution path, able to run simultaneously with other threads. Can be managed independently by a scheduler, which is typically a part of the operating system,

In most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time

### Advantages of multithreading

- Maintaining a responsive user interface

By running time-consuming tasks on a parallel “worker” thread, the main UI thread is free to continue processing keyboard and mouse events.

- Making efficient use of an otherwise blocked CPU

Multithreading is useful when a thread is awaiting a response from another computer or piece of hardware. While one thread is blocked while performing the task, other threads can take advantage of the otherwise unburdened computer.

- Parallel programming

Code that performs intensive calculations can execute faster on multicore or multiprocessor computers if the workload is shared among multiple threads in a “divide-and-conquer” strategy.

- Speculative execution

On multicore machines, you can sometimes improve performance by predicting something that might need to be done, and then doing it ahead of time.

- Allowing requests to be processed simultaneously

On a server, client requests can arrive concurrently and so need to be handled in parallel (the .NET Framework creates threads for this automatically if you use ASP.NET, WCF, Web Services, or Remoting).

### Disadvantages of multithreading

- Tricky interaction between threads (typically via shared data)
- Long development cycles and an ongoing susceptibility to intermittent and nonreproducible bugs. For this reason, it pays to keep interaction to a minimum, and to stick to simple and proven designs wherever possible

- Threading also incurs a resource and CPU cost in scheduling and switching threads (when there are more active threads than CPU cores) — and there's also a creation/tear-down cost

## II.

### Thread execution on single and multi processor systems

Multithreading is managed internally by a thread scheduler, a function the CLR typically delegates to the operating system. A thread scheduler ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked (for instance, on an exclusive lock or on user input) do not consume CPU time.

**On a single-processor computer**, a thread scheduler performs *time-slicing* — rapidly switching execution between each of the active threads. Under Windows, a time-slice is typically in the tens-of-milliseconds region — much larger than the CPU overhead in actually switching context between one thread and another (which is typically in the few-microseconds region).

**On a multi-processor computer**, multithreading is implemented with a mixture of time-slicing and genuine concurrency, where different threads run code simultaneously on different CPUs. It's almost certain there will still be some time-slicing, because of the operating system's need to service its own threads — as well as those of other applications.

A thread is said to be *preempted* when its execution is interrupted due to an external factor such as time-slicing. In most situations, a thread has no control over when and where it's preempted.

### How to pass data into a thread

- using *ParameterizedThreadStart(object obj)* delegate when instancing new thread and then passing necessary data object
- using lambdas, anonymous etc. methods with hardcoded parameter
- using some common shared variable

### How exceptions work in thread

Any *try/catch/finally* blocks in scope when a thread is created are of no relevance to the thread when it starts executing. You should handle possible exceptions manually (if you want to) in thread target running method instead, like in a classic one-threaded applications.

An unhandled exception causes the whole application to shut down! Except this ones:

- A ThreadAbortException is thrown in a thread because Abort was called.
- An AppDomainUnloadedException is thrown in a thread because the application domain in which the thread is executing is being unloaded.

If any of these exceptions (*ThreadAbort* and *AppDomainUnloaded*) are unhandled in threads created by the common language runtime, the exception terminates the thread (like any other exception), but the common language runtime *does not allow the exception to proceed further*.

**How to “interrupt” thread**

You can user thread.Interrupt method. Interrupts a thread that is in the WaitSleepJoin thread state.

If this thread is not currently blocked in a wait, sleep, or join state, it will be interrupted when it next begins to block.

ThreadInterruptedException is thrown in the interrupted thread, but not until the thread blocks. If the thread never blocks, the exception is never thrown, and thus the thread might complete without ever being interrupted.

**How to abort thread**

You can use *thread.Abort* method. Raises a *ThreadAbortException* in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.

*ThreadAbortException* is a special exception that can be caught by application code, but is re-thrown at the end of the catch, finally blocks in order to terminate the thread.

If *Abort* is called on a thread that has not been started, the thread will abort when *Start* is called. If *Abort* is called on a thread that is blocked or is sleeping, the thread is interrupted and then aborted.

After *Abort* is invoked on a thread, the state of the thread includes *AbortRequested*. After the thread has terminated as a result of a successful call to *Abort*, the state of the thread is changed to *Stopped.*

**How to resume aborted thread**

When this method is invoked on a thread, the system throws a *ThreadAbortException* in the thread to abort it. *ThreadAbortException* is a special exception that can be caught by application code, but is re-thrown at the end of the catch block unless ***ResetAbort*** is called. ***ResetAbort*** cancels the request to abort, and prevents the *ThreadAbortException* from terminating the thread. Unexecuted finally blocks are executed before the thread is aborted.

## III.

### What does Thread.Yield method do ?

*public static bool Yield()* - method causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.  Can be used to test multi-threaded code in order to find uncertain raise conditions

### What does [ThreadStaticAttribute] do ?

- Each thread sees a separate copy of static field
- Attribute Only for static fields
- Field initializers — they execute **only *once*** on the thread that's running when the static constructor executes
-

### How to store thread specific data ?

*ThreadLocal<T>*

- It provides thread-local storage for both static and instance fields
- Allows you to specify default values
- Lazily evaluated: the factory function evaluates on the first call (for each thread)

*Thread class methods: GetData and SetData*

- public sealed class LocalDataStoreSlot
    - Encapsulates a memory slot to store local data
    - The common language runtime allocates a multi-slot data store array to each process when it is created

- public static void SetData(LocalDataStoreSlot slot, object data)
- public static object GetData(LocalDataStoreSlot slot)

### What is thread pool advantages ?

- You do not have to create, manage, schedule, and terminate your thread, the thread pool class do all of this for you.
- There is no worries about creating too many threads and hence affecting system performance. Thread pool size is constrained by the .NET runtime. The number of threads you can use at the same time is limited.
- You need to write less code, because the .NET framework manages your thread internally with a set of well tested, and bug free routines.

**Where is it better to use dedicated thread ?**

- You require a foreground thread. (thread pool consists of background threads)
- You require a thread to have a particular priority. (thread pool sets Normal priority for each thread)
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task

**Where is thread pool used implicitly ?**

- Via the Task Parallel Library (from Framework 4.0)
- PLINQ
- Via BackgroundWorker
- WCF, ASP.NET, and ASMX Web Services application servers
- System.Timers.Timer and System.Threading.Timer

**Worker threads vs I/O completion threads**

- Both are normal clr threads
- Two sub-pools for each type
- I/O completions threads needed to handle native I/O callbacks
- To avoid a situation where high demand on worker threads exhausts all the threads available to dispatch native I/O callbacks