Day 50/60 - Understanding Mutex vs Semaphore in C#

When dealing with multithreading, Mutex and Semaphore help synchronize access to shared resources, but they work differently.

Mutex (Exclusive Lock for a Single Thread)

Ensures only one thread can access a resource at a time.
Can be used across processes (not just within the same application).
More overhead than a lock due to kernel-level operations.

Example:

```
Mutex mutex = new Mutex();

void AccessResource()
{
 mutex.WaitOne(); // Acquire lock
 try
 {
 Console.WriteLine("Resource accessed.");
 }
 finally
 {
 mutex.ReleaseMutex(); // Release lock
 }
}
```

Use Mutex when only one thread (or process) should access a resource at a time.

Semaphore (Allows Limited Concurrent Access)

Controls multiple threads accessing a resource at the same time.
Can limit the number of simultaneous accesses.

Example: Allowing Two Threads at a Time

```
SemaphoreSlim semaphore = new SemaphoreSlim(2); // Max 2 threads

async Task AccessResource()
{
 await semaphore.WaitAsync(); // Acquire slot
 try
 {
 Console.WriteLine("Thread entered.");
 await Task.Delay(1000); // Simulating work
 }
 finally
 {
 semaphore.Release(); // Release slot
 }
}
```

Use Semaphore when you want multiple but controlled concurrent access to a resource.

When to Use What?

Use Mutex for exclusive resource access across threads or processes.
Use Semaphore for limiting concurrent access to a shared resource.

Choosing the right synchronization mechanism improves thread safety and application performance!

#dotnet #csharp #multithreading #mutex #semaphore