Multithreading in C# - Mert Iscan, Nicole Steinhauer, Christian Vasiu, Dennis Wagner

Einführung in Multithreading

- · Was ist Multithreading?
  - Parallele Verarbeitung von Abläufen innerhalb eines Programms durch die Nutzung mehrerer Threads.
- Wie funktioniert es?
  - Ein Programm wird in **mehrere Threads** unterteilt, die unabhängig oder parallel arbeiten. Diese Threads teilen sich denselben Speicherbereich und können gleichzeitig auf einer oder mehreren CPU-Kernen laufen.
- Anwendungsbereiche?
  - Webseiten: Gleichzeitige Verarbeitung mehrerer Benutzeranfragen
  - Apps: Reaktionsfähige Benutzeroberflächen durch Hintergrundprozesse

Threads in C#

· Erstellen eines Threads (durch Thread-Klasse)

· Starten eines Threads (thread.Start())

· Thread Priorität setzen (thread.Priority = ThreadPriority.Highest)

Probleme und Herausforderungen

- Race Conditions – Wenn mehrere Threads gleichzeitig auf eine geteilte Ressource zugreifen und unerwartete Ergebnisse entstehen.
- Deadlocks – Zwei oder mehr Threads blockieren sich gegenseitig, weil sie auf Ressourcen warten, die von anderen Threads gehalten werden.
- Thread-Synchronisation – Ohne Synchronisationsmechanismen (lock, Mutex, Semaphore) kann es zu inkonsistenten Daten kommen.
- Erhöhter Speicher- und CPU-Verbrauch – Zu viele Threads können das System überlasten und die Leistung verschlechtern.
- Schwierige Fehlersuche und Debugging – Fehler im Multithreading sind oft schwer reproduzierbar, da sie von der Thread-Reihenfolge abhängen.

Lösungen von Synchronisationsprobleme

Um Probleme vorzubeugen kann man „lock“, „Mutex“ oder Semaphore verwenden.

```
private readonly object _lock = new object();

public void UpdateResource()
{
    lock (_lock)
    {
        // Nur ein Thread kann diesen Codeblock gleichzeitig ausführen.
    }
}
```

```
Mutex mutex = new Mutex();

public void UpdateResource()
{
    mutex.WaitOne(); // Warten, bis der Mutex verfügbar ist
    try
    {
        // Kritischer Code
    }
    finally
    {
        mutex.ReleaseMutex(); // Mutex freigeben
    }
}
```

```
SemaphoreSlim semaphore = new SemaphoreSlim(3); // Maximal 3 Threads gleichzeitig

public async Task AccessResource()
{
    await semaphore.WaitAsync();
    try
    {
        // Kritischer Code
    }
    finally
    {
        semaphore.Release();
    }
}
```

Moderne Alternativen

async:

- Asynchrone Programmierung zur Vermeidung der Blockierung des Haupt-Threads
- Wird mit async-Methode und await verwendet
- Ideal für IO-gebundene Aufgaben (z. B. Dateizugriffe, Netzwerkoperationen)
- Erlaubt die gleichzeitige Ausführung ohne mehrere Threads zu verwenden

Parallel:

- Parallele Ausführung von Schleifen oder Methoden
- Nutzt mehrere Threads für gleichzeitige Ausführung
- Ideal für CPU-gebundene Aufgaben (z. B. komplexe Berechnungen)
- Wird mit Parallel.For, Parallel.ForEach genutzt

Vergleich der Alternativen

- Thread
    - Erstellt einen eigenen Thread manuell
    - Langsam & schwergewichtig, da direkter Zugriff auf Threads
    - Gut für dauerhafte Hintergrundprozesse
    - Keine Unterstützung für async/await
    - Muss manuell synchronisiert werden
- Task
    - Verwendet Thread-Pool (effizienter als Thread)
    - Besser für asynchrone Operationen (async/await möglich)
    - Automatische Verwaltung von Threads
    - Unterstützt Fehlerbehandlung mit try/catch
    - Bessere Performance als manuelle Threads
- Parallel
    - Optimiert für parallele Schleifen (Parallel.For, Parallel.Invoke)
    - Nutzt alle CPU-Kerne effizient
    - Perfekt für Datenverarbeitung & große Berechnungen
    - Schlecht für IO-Operationen (z. B. Datei lesen)
    - Wenig Kontrolle über einzelne Tasks