THREADING

Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

PROZESSE

- Ein Prozess ist eine Instanz eines Programmes
- Wird vom Betriebssystem ausgeführt
- Jeder Prozess hat seinen eigenen Speicherbereich und läuft unabhängig von anderen Prozessen
- Prozesse können Child-Prozesse erstellen
- Der erstellende Prozess wird Parent-Prozess genannt
- Parent-Prozess kann z.B. auf Fehler in Child-Prozessen reagieren
- Jeder Prozess beginnt mit einem primären Thread
- Zusätzliche Threads können erstellt werden

PROZESSE IN C#

- Prozesse können in C# mit der System.Diagnostics.Process-Klasse erstellt und verwaltet werden
- Mit dieser Klasse können neue Prozesse gestartet oder auf laufende Prozesse zugegriffen werden etc.
- Beispiel: Ein neuer Prozess wird gestartet, der das notepad.exe ausführt, was den Notepad-Editor öffnet
- Anschließend wird gewartet, bis der Prozess beendet ist, bevor eine Ausgabe in die Konsole geschrieben wird

```
static void Main()
{
    var process = Process.Start("notepad.exe");
    process.WaitForExit();
    Console.WriteLine("Notepad wurde beendet");
}
```

THREADS

- Ein Thread ist eine Ausführungseinheit innerhalb eines Prozesses
- Threads teilen sich den Speicherbereich eines Prozesses und können auf gemeinsame Ressourcen zugreifen
- Sie ermöglichen die gleichzeitige Ausführung von Aufgaben innerhalb eines Prozesses
- Threads können die Leistung und Reaktionsfähigkeit von Anwendungen verbessern
- Threads werden durch das Betriebssystem verwaltet
- Einige Programmiersprachen wie Java, bieten eine Laufzeitumgebunggesteuerte Thread-Implementierung die als Green-Thread bekannt ist
- Green-Threads werden nicht vom Betriebssystem verwaltet

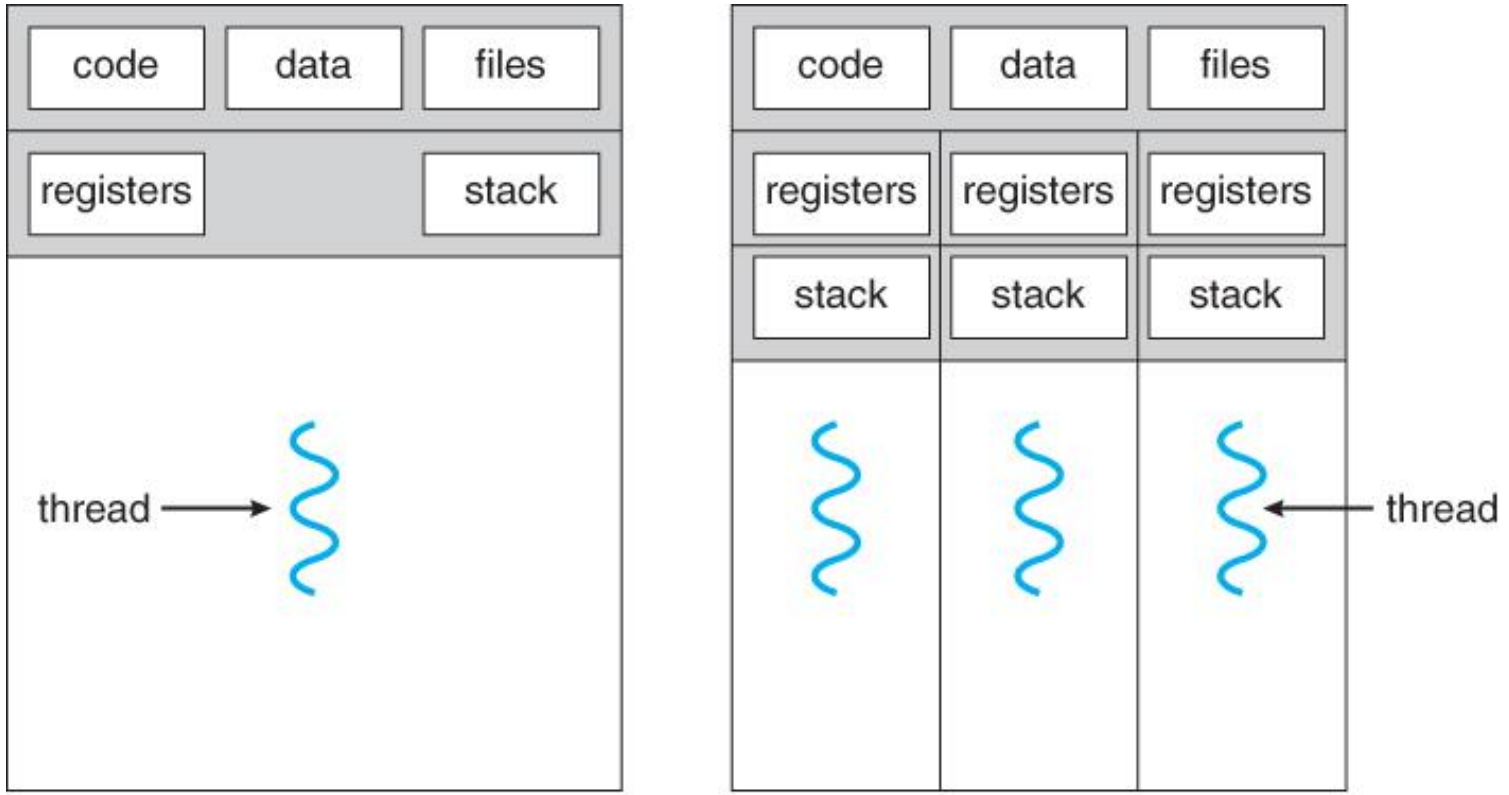
THREADS IN C#

- Threads können in C# mit der System.Threading.Thread-Klasse erstellt und verwaltet werden
- Mit dieser Klasse können neue Threads gestartet oder auf laufende Threads zugegriffen werden etc.
- Beispiel: Ein neuer Thread wird gestartet, der 5 Sekunden lang nichts tut und anschließend eine Ausgabe in die Konsole schreibt

```
static void Main()
{
    Thread newThread = new(Work);
    newThread.Start();
    Console.ReadLine();
}

static void Work()
{
    // Separater Thread
    Thread.Sleep(5000);
    Console.WriteLine("Thread ist fertig");
}
```

SINGLE VS. MULTITHREAD



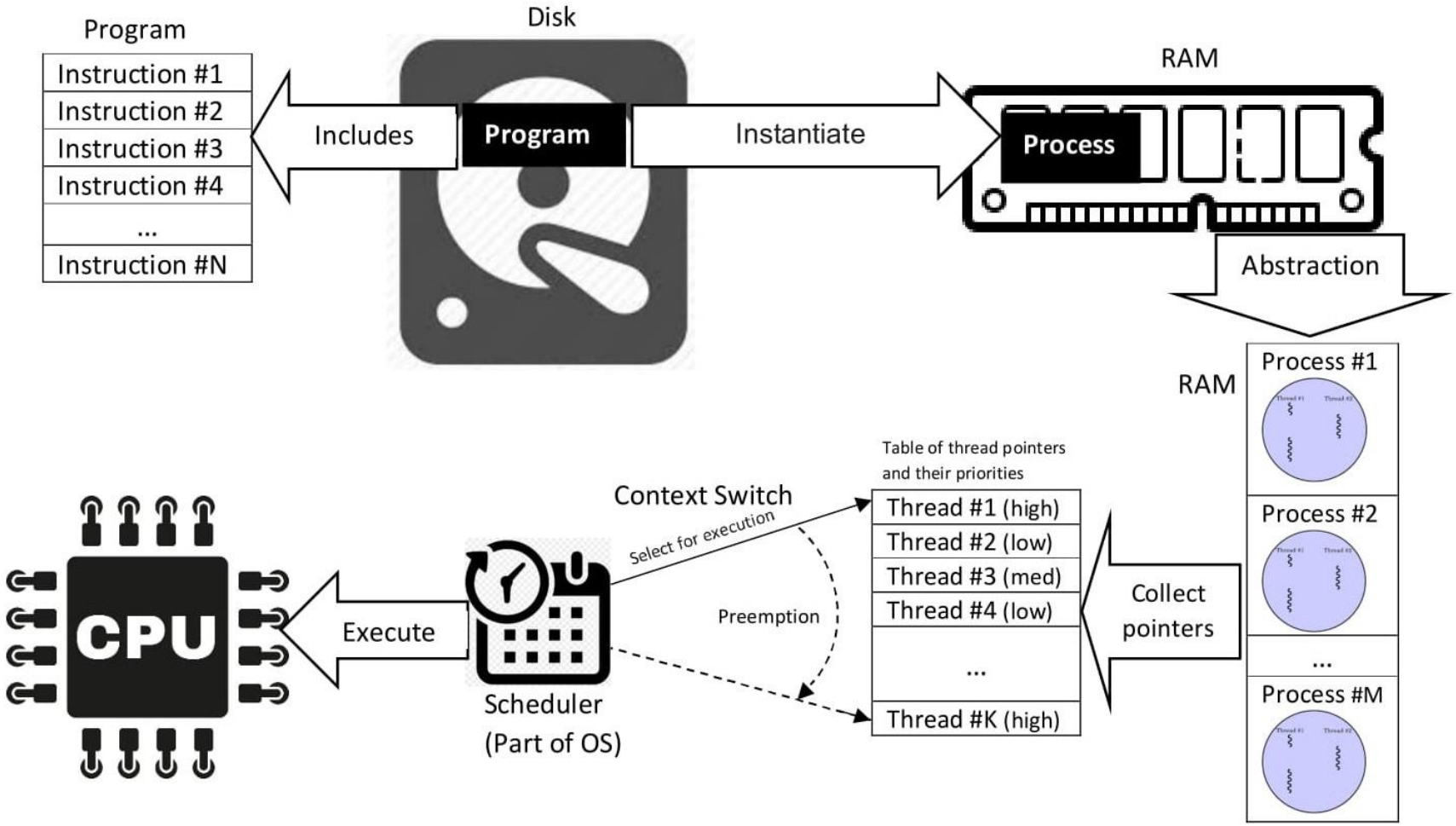
single-threaded process

multithreaded process

CONTEXT SWITCH

- Ein Context Switch ist der Wechsel der CPU von einem Prozess oder Thread zu einem anderen
- Der Zustand des vorherigen Prozesses / Threads wird gespeichert
- Der nächste Prozess / Thread wird geladen und ausgeführt
- Im Context sind Informationen zur Wiederaufnahme der Ausführung gespeichert
- Beispiele wären der CPU-Register und der Stack
- Ermöglicht die scheinbar parallele Ausführung von Aufgaben auf einer CPU
- Das Betriebssystem teilt die Rechenzeit in Zeitfenster auf
- Prozessor führt z.B. Prozess 1 für 5 ms auf, dann Prozess 2 für 5 ms, dann wieder Prozess 1 usw.

CONTEXT SWITCH



PROBLEME

- Threads verhalten sich nicht Deterministisch
- Die Reihenfolge in der die Threads gestartet und abgearbeitet werden kann nicht vorhergesagt werden

- Wenn zwei oder mehr Threads gleichzeitig auf eine gemeinsame Variable zugreifen und diese ändern, kann dies zu unerwarteten Ergebnissen führen

- Deadlock: Ein Deadlock tritt auf, wenn zwei oder mehr Threads in ihrer Ausführung eingefroren sind, weil sie aufeinander warten

THREADSICHERHEIT

- In Anwendungen können Threads gleichzeitig auf gemeinsame Ressourcen zugreifen
- Dies kann zu unerwarteten Ergebnissen führen, wenn nicht ordnungsgemäß synchronisiert wird
- Ein Mutex (Mutual Exclusion Object) ist ein Synchronisationsmechanismus, der verwendet wird, um den Zugriff auf eine gemeinsame Ressource zu kontrollieren
- Sorgt dafür, das nur ein Thread gleichzeitig auf die Ressource zugreifen kann

LOCK

- Das lock-Schlüsselwort in C# wird verwendet, um den Zugriff auf einen Codeblock zu synchronisieren, sodass nur ein Thread gleichzeitig darauf zugreifen kann
- Der lock sollte so kurz wie möglich gehalten werden
- Verwendet ein dediziertes Objekt das gelocked wird

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

static void Main()
{
    Thread thread1 = new Thread(Work);
    Thread thread2 = new Thread(Work);
    …
}

static void Work()
{
    lock(_lock)
    {
        // Zugriff auf gemeinsame Ressource
    }
}
```

BEGRIFFLICHKEITEN

- Nebenläufigkeit
- Zwei oder mehr Befehle werden zur gleichen Zeit abgearbeitet

- Asynchrone Verarbeitung
- Die Abarbeitung eines Befehls kann unterbrochen werden, um einen anderen zu verarbeiten
- Die Verarbeitung des ersten Befehls wird anschließend fortgesetzt
- Dies kann nebenläufig erfolgen, muss es aber nicht

ASYNCHRONE VERARBEITUNG

- Asynchrone Programmierung ermöglicht es, Operationen ohne Blockierung des ausführenden Threads durchzuführen
- Besonders bei I/O-gebundenen Aufgaben relevant
- Verbesserte Reaktionsfähigkeit, insbesondere in UI-Anwendungen
- Kann komplexe Operationen, durch Konstrukte wie async und await vereinfachen

KLASSE TASK

- Die Klasse Task dient dazu, asynchrone Operationen in einem Programm zu repräsentieren und zu steuern
- Threads müssen nicht vom Entwickler selbst erstellt werden
- Für eine explizite Aufgabe wird ein Task definiert
- Die Erstellung der Threads erfolgt automatisch zur Laufzeit
- Nach Beendigung der Aufgabe wird das Ergebnis im Task gespeichert

TASK SYNTAX

- Objekt vom Typ Task erstellen
- Action Delegaten als aufzuführende Tätigkeit übergeben
- Die Methode (oder der Lambda-Ausdruck) werden als Hintergrund-Thread gestartet
- Das heißt, die Tasks werden beendet, wenn das Hauptprogramm beendet wird

```
Task task = new Task(() => {
    Console.WriteLine("Hallo");
});

task.Start();
```

TASK SYNTAX

- Task verzögert starten

```
Task task = new Task(() =>
{
    Console.WriteLine("Hallo");
});

task.Start();
```

- Task sofort starten

```
Task task1 = Task.Factory.StartNew(() => { });

// oder (bevorzugt!)

Task task2 = Task.Run(() => { });
```

TASK SYNTAX

- Auf Beendigung eines einzelnen Tasks warten

```
Task task = Task.Run(() =>
{
    Console.WriteLine("Hallo");
});

task.Wait();
```

- Auf mehrere Tasks warten durch statische WaitAll Methode der Klasse Task

```
Task task1 = Task.Run(() => Console.WriteLine("Hallo2"));
Task task2 = Task.Run(() => Console.WriteLine("Hallo2"));

Task.WaitAll(task1, task2);
```

TASK SYNTAX

- Auf Beendigung eines der Tasks warten
- Rückgabewert ist der Index des fertiggestellten Tasks

```
Task task1 = Task.Run(() => Console.WriteLine("Hallo2"));
Task task2 = Task.Run(() => Console.WriteLine("Hallo2"));

int index = Task.WaitAny(task1, task2);
```

TASK SYNTAX

- Tasks können Rückgabewerte haben
- Werden in der Eigenschaft Result gespeichert

```
Task<int> task1 = Task.Run(() => {
    Thread.Sleep(5000);
    return 15;
});

task1.Wait();

Console.WriteLine(task1.Result);
```

TASKS UND EXCEPTIONS

- Unbehandelte Ausnahmen, welche in einem Task auftreten, werden an den aufrufenden Thread zurückgeliefert
- Die Ausnahmen werden z.B. beim Aufruf der Methode Wait oder der Eigenschaft Result zurückgegeben
- Try-Catch Blöcke können zum Abfangen verwendet werden

```
Task task1 = Task.Run(() => {
    throw new Exception();
});

task1.Wait(); // Löst eine AggregateException aus
```

TASKS UND EXCEPTIONS

- Da ein Task auf mehrere weitere Tasks warten kann, liefert er im Falle einer Ausnahme eine AggregateException zurück
- Diese beinhaltet alle Ausnahmen in einem einzelnen Objekt
- Über die Eigenschaft InnerExceptions kann auf die tatächlich aufgetretenen Ausnahmen zugegriffen werden

```
Task task1 = Task.Run(() =>
{
    throw new Exception("Fehler");
});

try
{
    task1.Wait(); // Löst eine AggregateException aus
} catch(AggregateException ex) {
    foreach(var innerException in ex.InnerExceptions) {
        Console.WriteLine(innerException.Message);
    }
}
```

TASKS ABBRECHEN

- Tasks können innerhalb der Ausführung mit return beendet werden
- Um Tasks von außen abzubrechen, benötigen wir ein Objekt vom Typ CancellationTokenSource
- Innerhalb des CancellationTokenSource Objektes befindet sich ein CancellationToken, der die Abbruchbenachrichtung weiterleitet

```
CancellationTokenSource cancellationTokenSource =
new CancellationTokenSource();

CancellationToken cancellationToken =
cancellationTokenSource.Token;
```

TASKS ABBRECHEN

- Abbruch kann über die Methode Cancel des CancellationTokenSource-Objektes eingeleitet werden
- Die IsCancellationRequested Eigenschaft des CancellationTokens gibt Auskunft darüber, ob ein Abbruch gewünscht ist
- Die Methode ThrowIfCancellationRequested des CancellationTokens verwenden, um zu überprüfen, ob eine Abbruchanforderung vorliegt und eine OperationCanceledException auszulösen
- Der Task kann darauf mit return oder einer Exception wie z.B. OperationCanceledException reagieren

```
static void Main()
{
    var cts = new CancellationTokenSource();

    Task task1 = Task.Run(() =>
    {
        DoSomething(cts.Token);
    }, cts.Token);
    Thread.Sleep(2000);
    cts.Cancel();
}

static void DoSomething(CancellationToken cts)
{
    while (!cts.IsCancellationRequested)
    {
        Console.WriteLine("Task is running");
        Thread.Sleep(1000);
    }
}
```

TASKS FORTSETZEN

- Tasks können durch andere Tasks fortgesetzt werden
- Besteht eine Aufgabe aus mehreren aufeinanderfolgenden Teilen, kann es notwending sein nach Beendigung eine weiter Aufgabe zu starten
- Die Fortsetzung startet, sobald der ursprüngliche Task beendet wurde
- Das Ergebnis des ersten Tasks wird via prevResult an den zweiten Task übergeben

```
Task<int> task1 = Task.Run(() => {
    Console.WriteLine("Task 1 wird ausgeführt");
    Thread.Sleep(3000);
    Console.WriteLine("Task 1 ist fertig");
    return 42;
});

task1.ContinueWith((prevResult) => {
    Console.WriteLine("Ergebnis des ersten Tasks : "
+ prevResult.Result);
    Console.WriteLine("Task 2 wird ausgeführt");
    Thread.Sleep(3000);
    Console.WriteLine("Task 2 ist fertig");
});

Thread.Sleep(10000);
```

TASKS FORTSETZEN

- Mit der Überladung von ContinueWith kann festgelegt werden, unter welchen Umständen der zweite Task, also die Fortsetzung stattfinden soll
- Optionen sind z.B. NotOnCancled, NotOnFaulted, OnlyOnCancled, OnlyOnFaulted etc.

```
Task<int> task1 = Task.Run(() => {
    Thread.Sleep(1000);
    return 42;
});

task1.ContinueWith((prevResult) => {
    Thread.Sleep(1000);
}, TaskContinuationOptions.NotOnCanceled);

Thread.Sleep(10000);
```

CHILD TASKS

- Tasks innerhalb eines anderen Tasks können als Child-Task angelegt werden
- Der äußere Task (Parent-Task) endet erst, wenn alle Child-Tasks beendet wurden

```
Task task1 = Task.Factory.StartNew(() =>
{
    Task task2 = new Task(() => {
        Thread.Sleep(5000);
    }, TaskCreationOptions.AttachedToParent);

    Task task3 = new Task(()=> {
        Thread.Sleep(10000);
    }, TaskCreationOptions.AttachedToParent);

    task2.Start();
    task3.Start();
});

task1.Wait();
Console.WriteLine("Fertig");
```

THREADSICHERE COLLECTIONS

- Der Namensraum System.Collections.Concurrent stellt mehrere threadsichere Auflistungsklassen bereit
- Erlauben sicheren und parallelen Zugriff durch mehrere Threads auf eine Auflistung
- Werden anstelle der entsprechenden generischen bzw. nicht generischen Collections verwendet
- Die threadsicheren Collections stellen gleiche bzw. ähnliche Methoden zum Manipulieren der Daten in der Collection zur Verfügung
- Um Nebenläufigkeit zu gewährleisten, unterscheiden sich die Methoden teilweise von den nicht threadsicheren Collections

THREADSICHERE COLLECTIONS

Klasse	Beschreibung
`ConcurrentDictionary<TKey, TValue>`	Wörterbuch von Schlüssel-Wert-Paaren
`ConcurrentQueue<T>`	FIFO queue
`ConcurrentStack<T>`	LIFO stack
`ConcurrentBag<T>`	Ungeordnete Liste von Elementen
`BlockingCollection<T>`	Liste mit Sperr- und Begrenzungsfunktionen
`Partitioner`	Partitionierungsstrategien für Arrays und Listen