Parallele Programmierung
GUI und Threading

Vorlesung 7

Prof. Dr. Luc Bläser

Letzte Vorlesung - Quiz

```
int count = 0;
foreach (var number in array) {
  if (IsPrime(number)) {
    count++;
  }
}
```

Wie kann man dies einfach parallelisieren?
Auf was muss man aufpassen?

Inhalt Heute

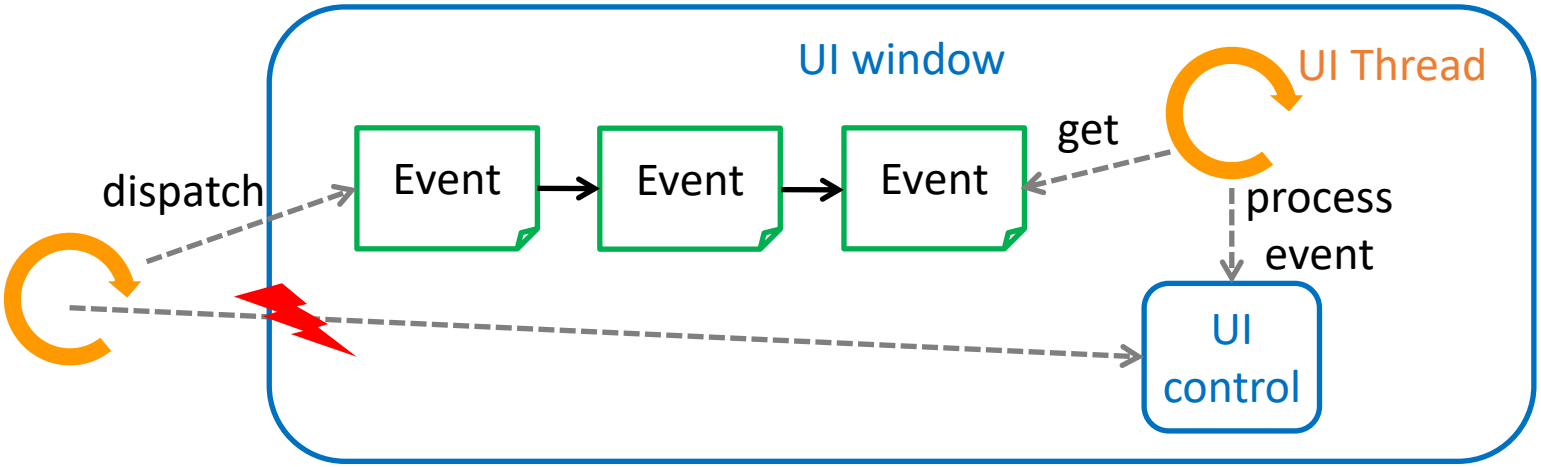
- GUI und Threading
  - Java AWT/Swing/SWT/Android
  - .NET WPF/WinForms
- C# async/await
  - Ausführungsmodell
  - Besonderheiten

Lernziele

- Thread Model in gängigen GUI-Frameworks verstehen
- Reaktionsfähige GUIs mit asynchroner Ausführung entwickeln können
- Das C# async/await Modell kennen und deren Nutzen für GUIs verstehen

Threading und GUI

- GUI Frameworks erlauben nur Single-Threading
  - Nur spezieller UI-Thread darf UI-Komponenten zugreifen
- UI Thread
  - Loop zur Ausführung der Ereignisse aus einer Queue



GUI Single Thread Modell

Wieso basieren GUI-Frameworks auf einem Single-Thread Modell?

UI Thread Confinement

- Synchronisationskosten
  - Locking in allen Komponenten und Methoden relativ teuer
- Deadlock-Risiko
  - Bei zyklischen geschachtelten Aufrufen (z.B. MVC)

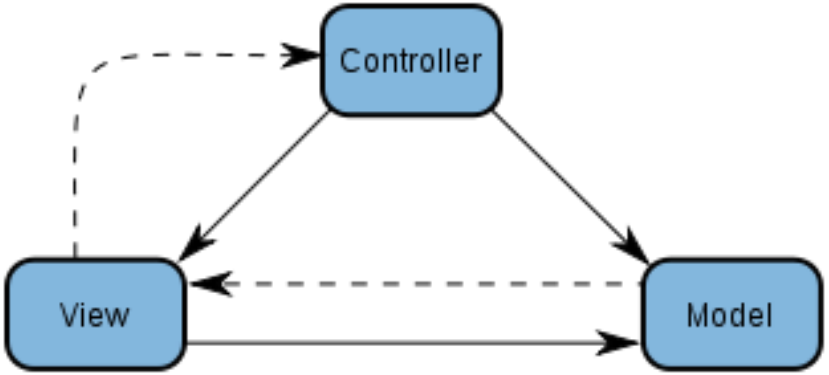


Bild © Wikipedia.org

UI Thread Model: Diskussion

Welche Einschränkungen ergeben sich aus dem UI-Thread Modell?

GUI Implikationen

- Keine lange Operationen in UI Events
  - Blockiert sonst UI
  - Betrifft UI-Listener, paint(), update()
- Kein Zugriff auf UI-Elemente durch fremde Threads
  - Sonst Race Condition
  - In AWT/Swing unentdeckt
  - In SWT, Android & .NET Exception («Invalid Thread Access»)
  - In JavaScript: Nur Single Thread hat DOM-Zugriff

UI Interaktion aus Threads

- Andere Threads dürfen UI Komponenten nicht direkt zugreifen
  - UI Operationen müssen als Events in die UI Event Queue eingereiht werden

Swing: Dispatching an UI Thread

- Komponentenzugriffe an UI Thread delegieren
  - Als Aufgabe in Event Queue einreihen
  - UI Thread führt die Aufgabe später aus
- Benutzung der Klasse `SwingUtilities`
  - `static void invokeLater(Runnable doRun)`
    - Asynchron
  - `static void invokeAndWait(Runnable doRun)`
    - Synchron

Beispiel: UI Dispatching

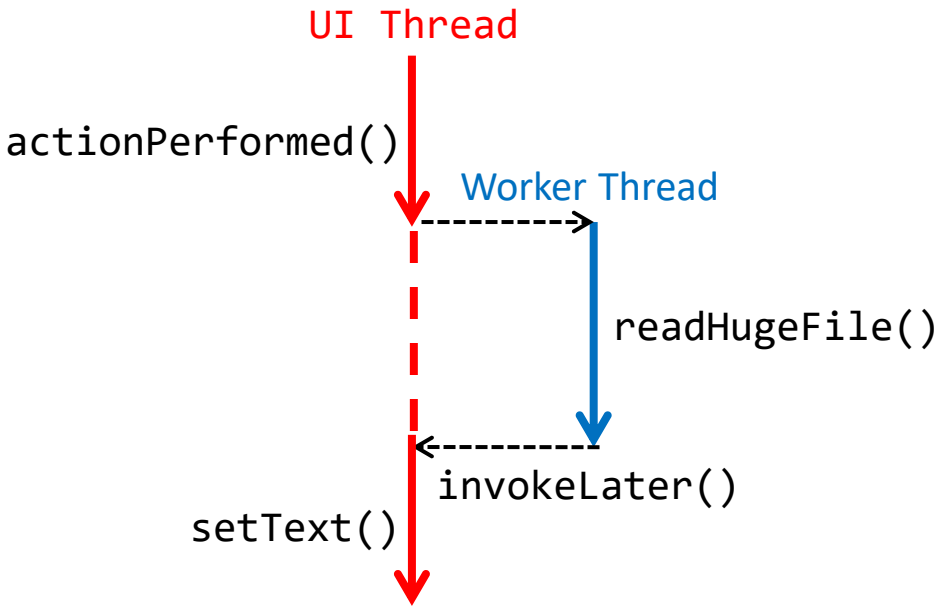
```
button.addActionListener(event -> {
  new Thread(() -> {
    var text = readHugeFile();
    SwingUtilities.invokeLater(() -> {
      textArea.setText(text);
    });
  }).start();
});
```

Was wird von welchem Thread ausgeführt und wieso?

UI Dispatching Analyse

```
button.addActionListener(event -> {
  new Thread(() -> {
    var text = readHugeFile();
    SwingUtilities.invokeLater(() -> {
      textArea.setText(text);
    });
  }).start();
});
```

Was eignet sich anstatt des Thread.start()?



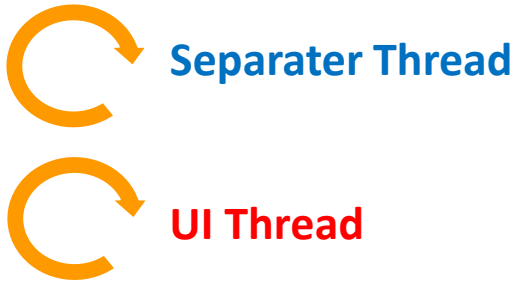
Sauberer Swing GUI Setup

```
public static void main(String[] args) {
  var frame = new JFrame("Test");
  var button = new JButton();
  button.setText("Click");
  frame.getContentPane().add(button, ..);
  // ...
  SwingUtilities.invokeLater(() -> {
    frame.pack();
    frame.setVisible(true);
  });
}
```

`pack()` & `setVisible()` im UI Thread ausführen

Swing Background Worker

- Hilfsklasse für Hintergrund-Arbeiten
  - Zeitaufwendige Operationen als Task in Thread Pool
    - **doInBackground()**
  - UI-Zugriffe durch EventDispatchThread
    - **done()**




```
public abstract class SwingWorker<Result, Temp> {
  protected abstract Result doInBackground();

  protected void done();
}
```

läuft in separatem Thread (keine UI-Zugriffe)

Durch UI Thread (UI Zugriffe)

Background Worker Beispiel

```
class BackgroundCalculator extends SwingWorker<Integer, Void> {
  @Override
  public Integer doInBackground() {
    return longComputation();
  }

  @Override
  protected void done() {
    try {
      int result = get();
      label.setText("Result: " + result);
    } catch (InterruptedException | ExecutionException e) {
      …
    }
  }
  …
}
```

Resultat von Background

Keine Zwischen-Resultate

Separater Thread

Resultat von doInBackground()

UI Thread

```
new BackgroundCalculator().execute();
```

Andere Java GUI Frameworks

- Exception bei falschen Thread-Zugriff auf GUI
- SWT
  - Pendant zu SwingUtilities
    - `widget.getDisplay().asyncExec(Runnable)`
- Android
  - Dispatch-Möglichkeiten
    - `post(Runnable)` auf UI-Komponente aufrufen
    - `AsyncTask` analog zu SwingWorker
    - `Activity.runOnUiThread(Runnable)`

Android AsyncTask

```
class BackgroundCalculator extends AsyncTask<Void, Void, Integer> {
  @Override
  public Integer doInBackground(Void... input) {
    return longComputation();
  }

  @Override
  protected void onPostExecute(Integer result) {
    view.setText("Result: " + result);
  }
  …
}
```

Kein Input

Resultat von Background

Keine Zwischen-Resultate

Separater Thread

UI Thread

Resultat von doInBackground()

```
new BackgroundCalculator().execute();
```

.NET UI Threading Modell

- Gleiches Prinzip wie in Java
  - Single Thread Apartment (Confinement)
- UI Thread
  - Aufrufer von `Application.Run()`
  - Bei WPF implizit der Main Thread
  - Oder modales Show
- UI Event Dispatching
  - WPF: `control.Dispatcher.InvokeAsync(action)`
  - WinForm: `control.BeginInvoke(delegate)`
  - `InvokeAsync/BeginInvoke` sind asynchron, `Invoke` synchron

Analog: Nicht blockierendes WPF

```
public class MainWindow : Window {
  private void startCalculationButton_Click(…) {
    calculationResultLabel.Content = "(computing)";
    int number;
    if (int.TryParse(numberTextBox.Text, out number)) {
      Task.Run(() => {
        int result = LongCalculation(number);
        Dispatcher.InvokeAsync(() => {
          resultLabel.Content = result;
        });
      });
    }
  }
}
```

In anderen Thread outsourcen

UI Update wieder als UI Event dispatchen

Unleseserlich

Nicht-Blockierende UIs

- Klassisch: Zerstückelung der Logik
  - Kette von Dispatch (UI Thread/fremder Thread)
  - Hilfsklasse BackgroundWorker
- Leserlicher Code mit C# async/await
  - Logik in einem Guss (eine Methode)
  - Zerstückelung in mehrere UI-Events hinter Kulissen

Asynchronität mit Async/Await

Async Methode

```
public async Task<int> LongOperationAsync() { … }

…
Task<int> task = LongOperationAsync();
OtherWork();
int result = await task;
…
```

Warte auf Beendigung der Async Methode

Async und Await

- Schlüsselwort async für Methode
  - Aufrufer ist nicht zwingend während der gesamten Ausführung der async Methode blockiert
- Schlüsselwort await für Tasks
  - Warte auf Ende eines TPL Tasks
  - Liefert Resultat des Tasks, falls mit Rückgabe

Async Rückgabetypen

- `void`: «fire-and-forget»
- `Task`: Keine Rückgabe, erlaubt Warten auf Ende
- `Task<T>`: Für Methode mit Rückgabetyp T
- Keine `ref` oder `out` Parameter

Beispiel: Async Methode

Suffix „Async“ als Namenskonvention

Rückgabetyp string

```
async Task<string> ConcatWebSitesAsync(string url1, string url2) {
  HttpClient client = new HttpClient();
  Task<string> download1 = client.GetStringAsync(url1);
  Task<string> download2 = client.GetStringAsync(url2);
  string site1 = await download1;
  string site2 = await download2;
  return site1 + site2;
}
```

async Task<string> GetStringAsync(string url)

Direkte Rückgabe des String

Spezielle Regeln

- async Methode
  - Muss await enthalten => Sonst Compiler-Warnung
- await Anweisung
  - Nur in async Methode erlaubt => Sonst Compiler-Fehler

Grund: Spezielles Ausführungsmodell

Ausführungsmodell

- Async Methode läuft teilweise synchron, teilweise asynchron
  - Aufrufer führt Methode solange synchron aus, bis ein blockierendes await anliegt
  - Danach läuft die Methode asynchron

```
async Task<int> GetSiteLengthAsync(string url) {
  HttpClient client = new HttpClient();
  Task<string> task = client.GetStringAsync(url);
  string site1 = await task;
  return site1.Length;
}
```

Synchron
(Aufrufer Thread)

Asynchron
(evtl. anderer Thread)

Mechanismus

- Compiler zerlegt Methode in Abschnitte
  - Erster Abschnitt vor Await: Synchron durch Aufrufer
  - Abschnitt nach Await: Läuft später nach Task-Ende (Continuation)

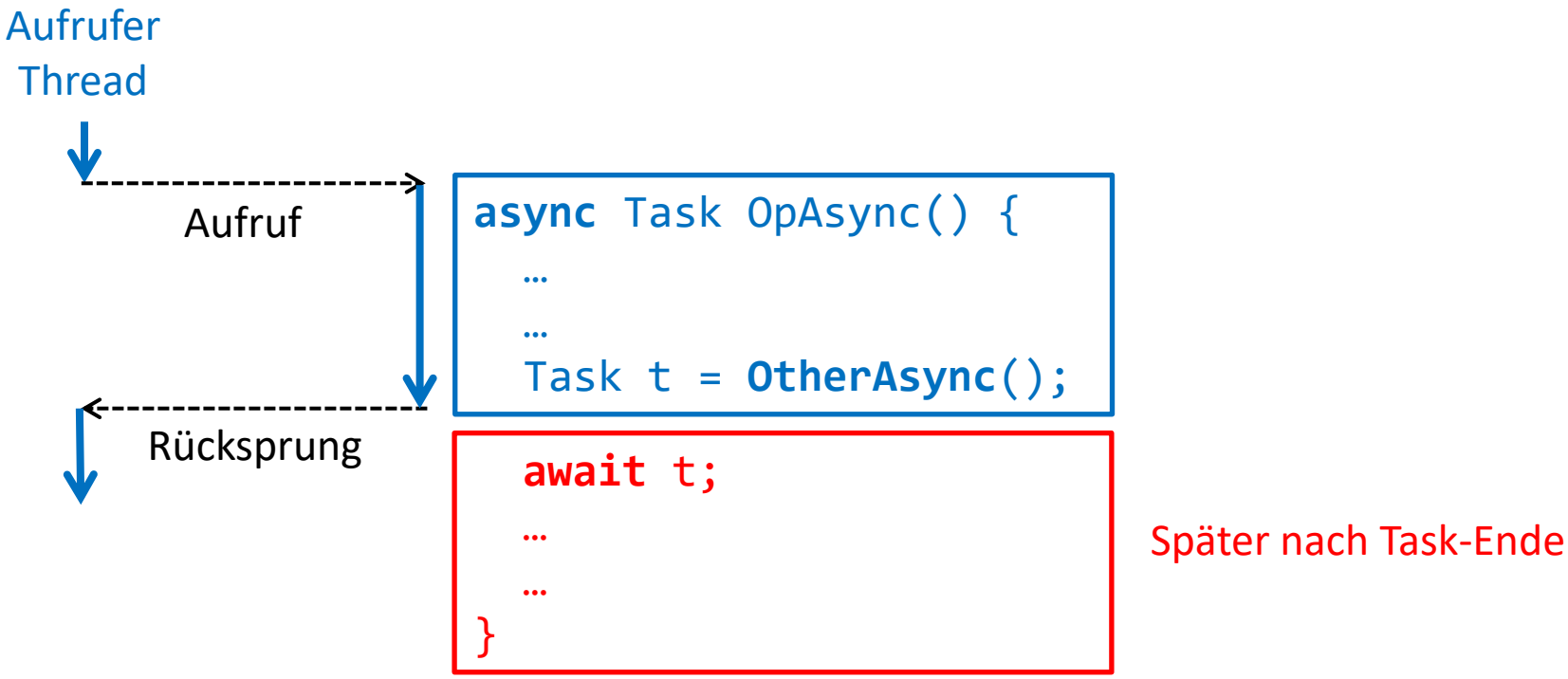
Synchron durch Aufrufer

```
async Task OpAsync() {
  …
  …
  Task t = OtherAsync();
  await t;
  …
  …
}
```

Asynchron nach Task-Ende

Async Methodenaufruf

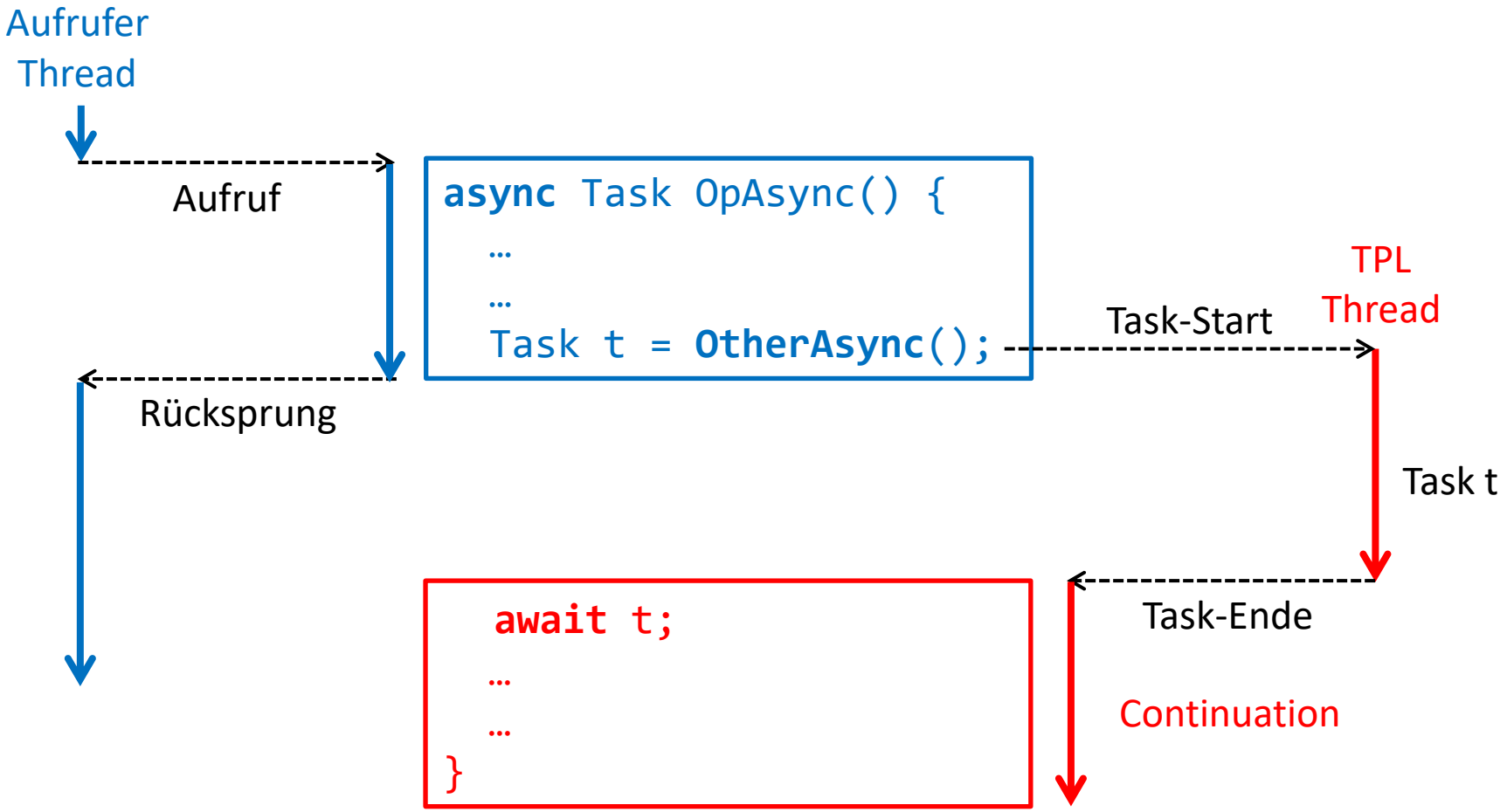
- Methode läuft synchron bis zu blockierenden Await
  - Bei Warten auf anderen Thread bzw. externes I/O
- Bei blockierendem Await Rücksprung zum Aufrufer



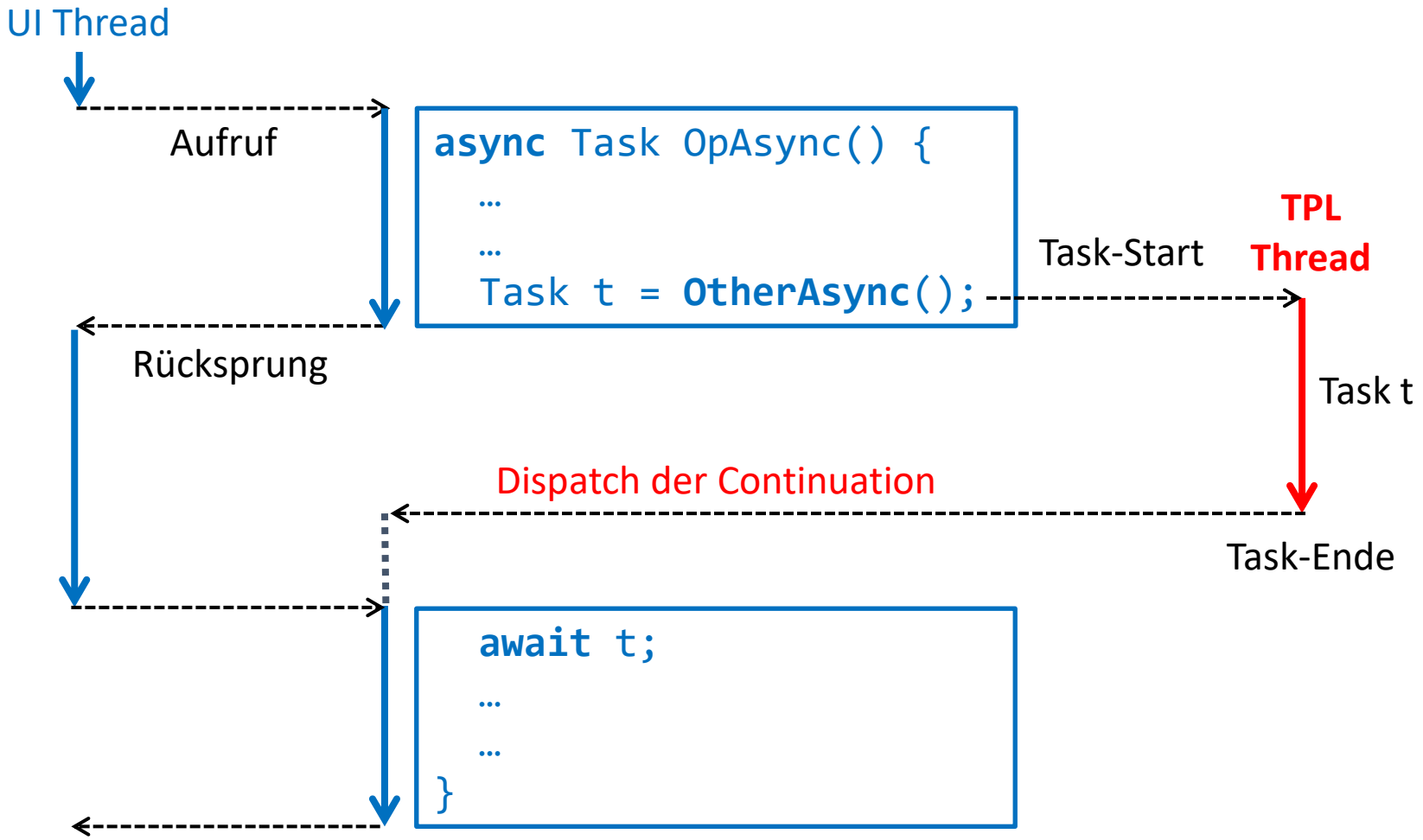
Verschiedene Ausführungen

- Fall 1: Aufrufer ist ein «normaler» Thread
  - Meiste Threads: Console, TPL etc.
  - Abschnitt wird durch TPL-Worker-Thread ausgeführt
- Fall 2: Aufrufer ist UI-Thread
  - Abschnitt wird an UI dispatched
  - Wird als Event vom UI-Thread ausgeführt

Fall 1: Kein UI Thread



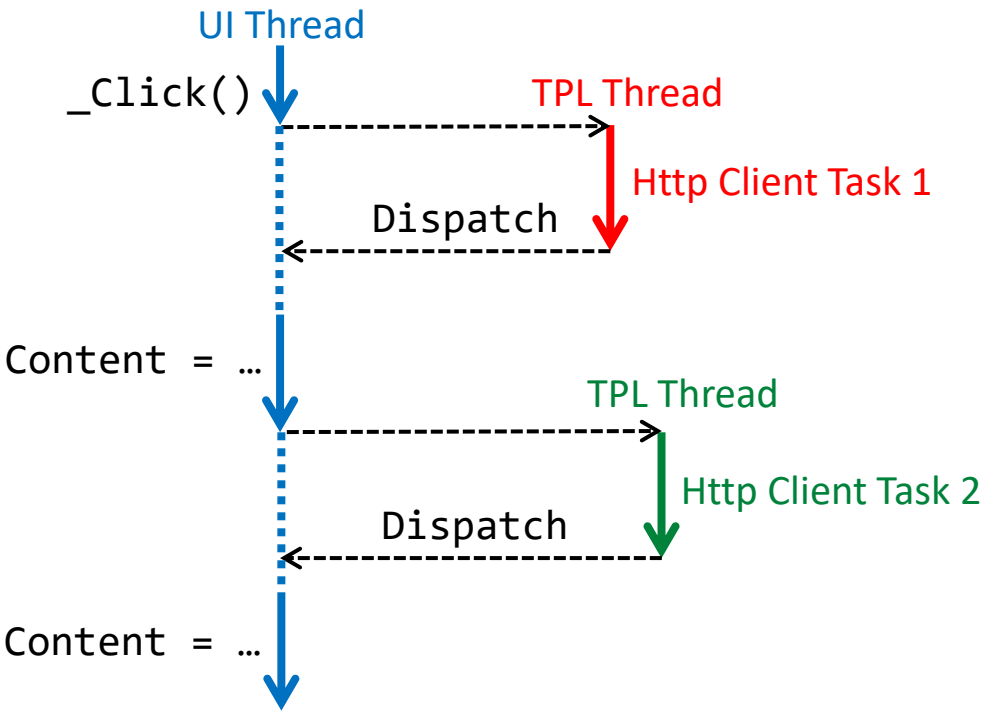
Fall 2: UI Thread



Async/Await Sequenz

```
async void startDownload_Click(…) {
   HttpClient client = new HttpClient();
   foreach (var url in collection) {
      var data = await client.GetStringAsync(url);
      textArea.Content += data;
   }
}
```

API bietet diverse async Methoden



Häufiger Fehler

- async Methoden sind nicht per se asynchron

```
public async Task<bool> IsPrimeAsync(long number) {
  for (long i = 2; i * i <= number; i++) {
    if (number % i == 0) { return false; }
  }
  return true;
}
```

Läuft vollständig synchron

Aufrufer ist während gesamter Ausführung blockiert

Compiler-Warnung: kein await in async Methode

Workaround

- Teure Operation explizit als Task ausführen

```
public Task<bool> IsPrimeAsync(long number) {
  return Task.Run(() => {
     for (long i = 2; i * i <= number; i++) {
       if (number % i == 0) { return false; }
     }
     return true;
  });
}
```

Sonderheit

- Thread-Wechsel innerhalb Methodenaufruf (Fall 1)

```
public async Task DownloadAsync() {
  Console.WriteLine($"BEFORE {Thread.CurrentThread.ManagedThreadId}");
  HttpClient client = new HttpClient();
  Task<string> task = client.GetStringAsync("...");
  string result = await task;
  Console.WriteLine($"AFTER {Thread.CurrentThread.ManagedThreadId}");
}
```

Partielle Nebenläufigkeit berücksichtigen

```
BEFORE 10
…
AFTER 14
```

Design-Kritik

- Eine Async-Methode => Zwei Aufruffälle
  - Fall 1 (kein UI-Thread) und Fall 2 (UI-Thread)
- Viraler Effekt: async/await-Aufrufer wird auch async
  - Komplexe Methodensignaturen mit Tasks
  - Kosten wegen interner Methodenzerstückelung
- Synchrone/asynchrone Verwendung sollte meist orthogonal sein
  - «Caller» sollte entscheiden, nicht «Callee»

Rückblick: Lernziele

- Thread Model in gängigen GUI-Frameworks verstehen
- Reaktionsfähige GUIs mit asynchroner Ausführung entwickeln können
- Das C# async/await Modell kennen und deren Nutzen für GUIs verstehen

Weitere Lektüre (fakultativ)

- B. Goetz et al. Java Concurrency in Practice, Addison Wesley, 2006
  - Kapitel 9: GUI Applications
- Java Tutorial: Concurrency in Swing
  - http://docs.oracle.com/javase/tutorial/uiswing/concurrency
- C# async/await
  - MSDN: S. Cleary: Best Practices in Asynchronous Programming, MSDN Magazine, Mar. 2013
  - MSDN: S. Toub: Async Performance: Understanding the Costs of Async/Await, MSDN Magazine, Oct. 2011