Parallel and Asynchronous Programming with .NET

Dr. Dietrich Birngruber, Part 2

Contents

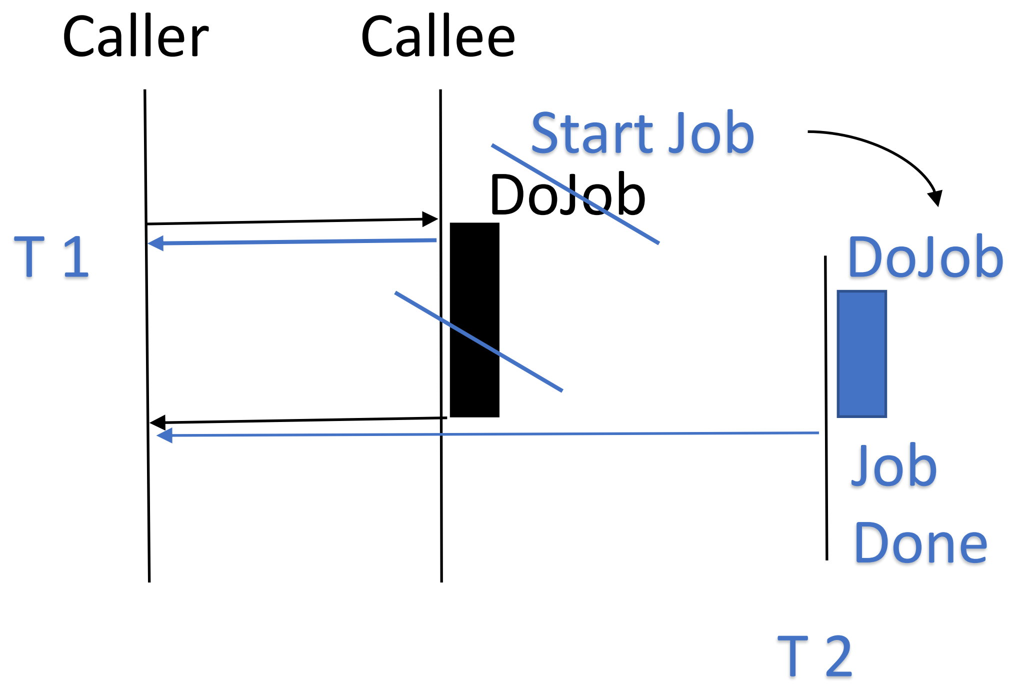
- Asynchronous or parallel programming?
- Threading basics in .NET
- Asynchronous programming patterns in .NET

Ups, my app has a problem ....

- IO operations take some time
  - E.g. loading / saving a file, downloading a file
- UI is frozen, while it waits for a download / file load to complete
- What can we do?
  - Ignore it?
  - Inform user with a message box and then freeze UI
  - ???

Synchronous → Parallel ('Offloading')

- We need to transform a synchronous call to ***a non blocking*** call
- A method call that returns immediately to the caller
- Called method continues doing its stuff in the background (perhaps)
- Called method typically executes a *callback* when job is done
- Callback contains result *or* exception



Parallel, Asynchronous

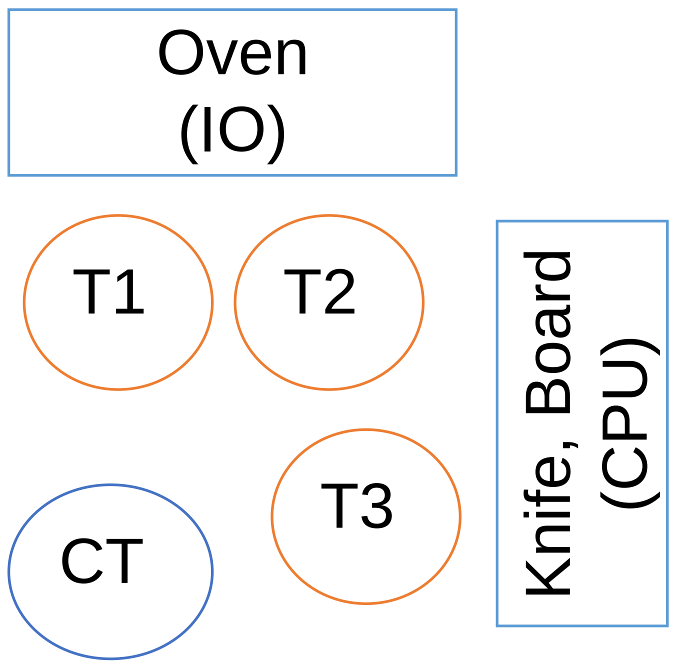
Let's make a soup

A simple metaphor (don't try this at home ;-)). Making our soup requires four steps:

1. Heat water in a pot until it is boiling - and add salt
2. Cook rice in another pot
3. Chop vegetables into small pieces
4. Everything goes together into the pot with hot water – let it cook for 42 seconds

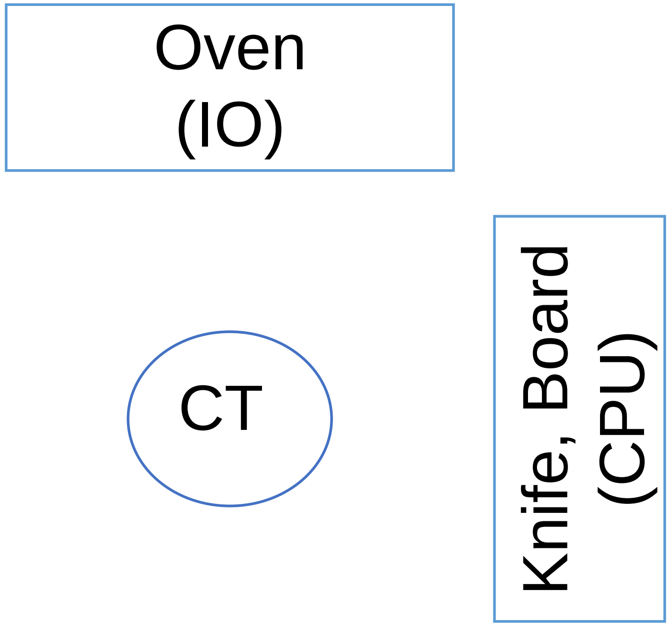
Parallel: Let’s make a soup

- One coordinator thread (CT); it creates 3 worker threads and asks them from time to time, if they are finished
- Worker T1: Puts hot water in pot 1, puts the pot on the oven, and actively waits until water is boiling (external event)
- Worker T2: Puts water + rice in pot 2, puts pot 2 on the oven and actively waits until rice is done (external event)
- Worker T3: Cuts vegetables
- When T1, T2, T3 are all done, the coordinator puts all into pot 1 and waits for 42 seconds



Async: Let’s make a soup

- One coordinator thread (CT); it starts two asynchronous operations – each async operation interacts with the oven, and the CT cuts the vegetables
- Async OP1: CT puts hot water in pot 1, puts the pot on the oven, CT returns to do next step. When water is boiling (external event) - CT is called back
- Async OP2: CT puts water + rice in pot 2, puts pot 2 on the oven CT returns to do next step; external callback
- CT: Cuts vegetables
- When OP1 and OP2 are completed, the coordinator puts all into pot 1 and (a)waits a timer to complete after 42 seconds



Asynchronous / Parallel – Difference?

Asynchronous

- Non blocking operations
- **IO bound** operations
  "caller is waiting for IO" – no, returns to caller with a "resumption point" to continue with
- **Scales** better – e.g. processing server requests
- "Cooperative on one thread"

Parallel

- **CPU bound** operations
  e.g., sort large object sets in memory, calculate 3D rendering, calculate xy
- Works well, when you can separate tasks into independent pieces of work – which can be executed in parallel
- Does **not scale well** – so number of parallel tasks / thread should be "small"

Synchronous Solution: Download Web Pages

```
public static int SumPageSizes(IList<Uri> urlList) {
 int total = 0;
 try {
  WebClient webClient = new WebClient();
  foreach (Uri uri in urlList) {
   byte[] data = webClient.DownloadData(uri); // long blocking call
   total += data.Length;
  }
 } catch (Exception ex) {// No exception is expected.
  Console.WriteLine("Synchronous foreach has thrown an exception.\n{0}", ex);
 }
 return total;
}
```

Demo

First Idea: We use class Thread
simulate download with Thread.Sleep;
for-loop with 10.000 iterations (URLs);
start program without debugger;

Question: is this parallel or async?
(Async Demo02)

Threading Basics in .NET

Multi-Threading using class Thread

Thread Synchronisation Essentials

Pros and Cons of Multiple Threads in .NET

Multi-Threading using class Thread

- Class System.Threading.Thread exists since .NET 1.0
- „Lightweight process“ inside a .NET AppDomain
- Basic idea: write long running synchronous code, and execute it „(quasi-)parallel“

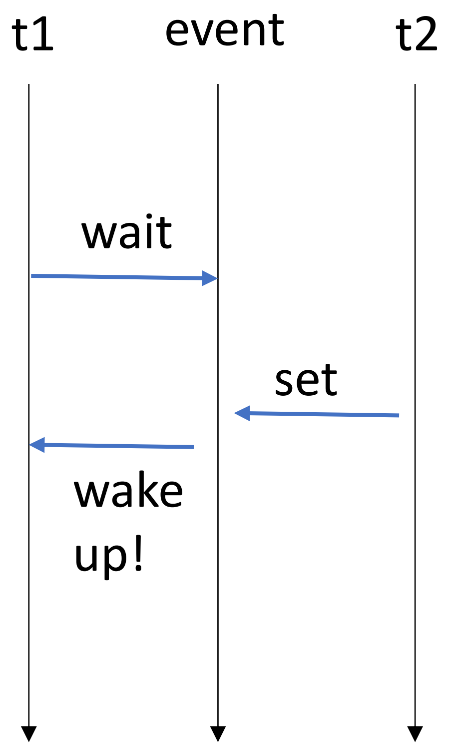
```
Thread t1 = new Thread ( new ThreadStart(RunServer) );
t1.Start(); //RunServer() executes (quasi-)parallel
…
```

Thread Synchronization Essentials - Locking

- When accessing any writable shared field
  - lock(syncObj) { .. } == Monitor.Enter / Monitor.Exit
    - *Exclusive* locking (i.e. only one thread executes the block at a time)
  - ReadWriteLockSlim
    - Faster than exclusive locks, if there are many readers with only occasional updates
  - Semaphore(Slim)
    - Local / system wide
    - Limits the number of threads that can access a resource concurrently
  - Mutex
    - Exclusive locking across process boundaries

Thread Synchronization Essentials - Signaling

- When others wait for another Thread
  - t1.Join(..) – wait until Thread t1 completes
- ManualResetEvent / **ManualResetEventSlim**
  - manualEvent.Wait(..); //Thread t1 waits until event is signaled
  - manualEvent.Set(); //Thread t2 sets event → t1 continues
  - manualEvent.Reset(); //reset to non-signaled
- **AutoResetEvent**
  - resets automatically to non-signaled after releasing a waiting thread
  - autoResetEvent.WaitOne(…) //calling Thread waits ….
- **CountDownEvent**
  - Special event which unblocks its waiting threads after it has been signaled a certain number of times



Pros and Cons of Multiple Threads

- Threads are wonderful for small number of very long running operations
- Background / foreground Threads
- Threads with different priorities

- 1 Thread consumes ~ 1 MB memory (.net 4.x)
  - Loop spans 10.000 Threads → potential OutOfMemoryExceptions
- Managing large number of threads consumes significant processor time
  - one process has most of the current threads → threads in other processes are scheduled less frequently
- Think twice before you create a Thread instance
  - Execute a stress-test without a VS.NET debugger attached!
- ThreadPool: reduces startup-time
  - but don’t consume a pooled Thread for a long time!
  - No infinite threads are in the pool

.NET Asynchronous Programming patterns

Asynchronous Programming Model (APM)

Event-based Asynchronous Pattern (EAP)

Task Asynchronous Pattern (TAP)

Overview of Asynchronous Programming Patterns

- Asynchronous Programming Model (APM)
  - Since .NET 1.0
  - Begin / End or IAsyncResult pattern
- Event-based Asynchronous Pattern (EAP)
  - Since .NET 2.0
  - Was designed for WinForms / UI applications;
  - Uses events as callbacks
- Task Asynchronous Pattern (TAP)
  - Since .NET 4.0
  - Task classes

Asynchronous Programming Model (APM)

- APM has naming conventions:
  - *IAsyncResult Begin*OperationName (input-arguments, *AsyncCallback userCallback*, *object stateObject*)
  - Result-Type *End*OperationName (*IAsyncResult* resultHandle);
- APM uses IAsyncResult,
  - caller has a „handle“ when operation completes – i.e. when caller is notified
- APM implementation uses a delegate to invoke synchronous method asynchronously

Demo

IAsyncResult / APM Demo
(Async Demo03)

Demo blocks calling Thread

APM Exceptions and Cancellation

- Exception is thrown in the EndXY method
  - When EndXY is called in the callback method and there is no try-catch → who catches Exception?
- Cancellation of the running operation? We only have IAsyncResult ☹

```
interface IAsyncResult
{
  object AsyncState { get; }
  WaitHandle AsyncWaitHandle { get; }
  bool CompletedSynchronously { get; }
  bool IsCompleted { get; }
}
```

APM Deficiencies

- One logical operation is scattered
- Complex to write BeginXY / EndXY methods
- Complex to call / use async methods with APM
- Complex for caller to continue when APM method finished
- Not recommended for new development
- BUT still used in BCL and assemblies prior .NET 4.0
    - TAP wrapper exists for APM

Event-based Asynchronous Pattern (EAP)

- Simpler to use than APM – especially for UI development
- Introduced together with class BackgroundWorker in .NET 2.0
- Uses events to inform caller that async operation finished
  - Optional: callback on the same thread as caller started the operation
- Naming conventions / coding patterns:

```
event AsyncCompletedEventHandler MethodNameCompleted;
void MethodNameAsync(input-args [, object userState ]);
```

- For return values subclass AsyncCompletedEventArgs
  - Return value should be a **read only** property called **Result**

Optionally with EAP You Can

- Allow more async calls simultaneously
  - Object userState is used to distinguish calls
  - Remember userState and return it to caller in completed event
- Cancellation of an asynchronous task
  - Special cancel method with naming pattern

void *MethodName***AsyncCancel (object userState)**

- Progress reporting
  - Add „ProgressChanged“ event which raises ProgressChangedEventArgs

event EventHandler<**ProgressChangedEventArgs** > *MethodName***ProgressChanged**

Event-based Asynchronous Pattern (EAP)

```
// some signature samples
event DownloadDataCompletedEventHandler DownloadDataCompleted;
event DownloadProgressChangedEventHandler DownloadDataProgressChanged;

void DownloadDataAsync(Uri uri, object userState);
void DownloadDataAsync(Uri uri); // supports only one call at a time
void DownloadDataAsyncCancel(object userState); //cancel a task identified by userState

// usage samples
client.DownloadDataCompleted += new DownloadDataCompletedEventHandler(OnCompleted);
client.DownloadDataAsync(aUri, myUserState1);
client.DownloadDataAsync(anotherUri, anotherUri); // user state is used in OnCompleted
```

Demo

Usage of EAP – Download Example; Usage of WebClient (AsyncConsoleApp-Demo 04)

AsyncCompletedEventArgs

```
public class AsyncCompletedEventArgs : EventArgs
  public AsyncCompletedEventArgs();
  public AsyncCompletedEventArgs(Exception error, bool cancelled, object
userState);

  public bool Cancelled { get; }
  public Exception Error { get; }
  public object UserState { get; }

  protected void RaiseExceptionIfNecessary();
}
```

EAP Exceptions and Cancellation

- Exceptions are returned in the AsyncCompletedEventArgs
  - If there is an exception, then
    Result property value is null (if async operation is a function)
    Error property has value
- Cancellation
  - **Property Cancelled** of AsyncCompletedEventArgs is set to true – so caller knows operation was cancelled
  - Request cancellation by calling cancel method
    void *MethodName***AsyncCancel**(object userState);

Demo

Develop Async Method using EAP
(Demo 05)

Pros and Cons of Event-Based Async Pattern

- Simpler to use than APM – especially for UI developers, because they know events
- More flexible than APM
  - Progress reporting
  - Cancellation
  - Error propagation

- Logic is still scattered
  - Start, operation, continuation
- Still complex to develop async methods
  - Serveral naming conventions & events
  - Sub classing AsyncCompletedEventArgs → Reuse based on Result type
- Did you *check Error* property?
- How often did you register for XyCompleted event?
- Was it *your completed* event?

Task Asynchronous Pattern (TAP)

- Introduced with .NET 4.0
  - Deprecates APM and EAP
- Based on Task and Task<Result> classes
- Uses single method to represent start and completion of an asynchronous operation
  - Asynchronous method returns Task or Task<Result>
- Task represents an operation and has a status
  - Operation can be executed synchronously, or asynchronously
- No need for “object userState” – caller has a Task object

Demo

Calling TAP methods
1- Download Example (Demo 06)
2- For loop iterating 10.000 times (Demo 07)

Some TAP Classes

- Task, Task<Result>
  - Represents an operation / function which might be executed asynchronously
- TaskCompletionSource<Result> + TaskFactory
  - For creating / writing async operations using TAP
- CancellationTokenSource + CancellationToken
  - For requesting a cancellation (caller)
  - For checking, if running operation should be cancelled (callee)
- Basis for Task Parallel Library (TPL)

Task-Continuation – some thoughts

- On completion of an asynchronous operation, a second operation is invoked and data is passed on
  - e.g. using callback operations or event handlers
- Synchronous vs. TAP continuation example:
  - Int x = DoMethod1(…); //this statement is long running
    DoMethod2(x, …); //continue when method-1 completed
  - vs.
    Task t1 = DoMethod1Async(…);
    Task t2 = t1.ContinueWith(cT => DoMethod2(cT.Result, …));

With Task.ContinueWith(…) we can

- pass data from the antecedent to the continuation
- specify the precise conditions under which the continuation will be invoked or not invoked
- cancel a continuation either before it starts or cooperatively as it is running
- provide hints about how the continuation should be scheduled
  - A new Task? → newly scheduled (default)
  - Execute on the same Thread as the completed Task? (for short statements)
  - Execute on the same Thread as the caller of the first Task? (for UI updates)
  - …

With Task.ContinueWith(…) we can

- invoke multiple continuations from the same antecedent
- invoke one continuation when all or any one of multiple antecedents complete
- chain continuations one after another to any arbitrary length
- use a continuation to handle exceptions thrown by the antecedent

- Take a look at enum **TaskContinuationOptions**
  - E.g. continuation updates a bool field only → TaskContinuationOptions.**ExecuteSynchronously**

Cancelling and Progress Reporting

- Callee checks if to cancel operation using CancellationToken

```
cancelToken.ThrowIfCancellationRequested();
```

- Caller requests cancellation using CancellationTokenSource

```
Task<IList<string>> task = client.DownloadAsync (adrs, cancellationTokenSource.Token);
 // .....
cancellationTokenSource.Cancel();
```

- Progress reporting using IProgress<T>

```
Task<IList<string>> DownloadAsync(string[] adrs, CancellationToken cancel, IProgress<int> progress)
{
   … if (progress != null) { progress.Report(i); } …
}
```

Demo

Cancelling and Progress Reporting – Download HTML (Demo 08)

(Parallel) Task vs. Thread

- A Task is more efficient and allows a more scalable use of system resources
  - Behind the scenes, tasks are queued to the ThreadPool;
  - ThreadPool has been enhanced
  - Task has less overhead than a dedicated user Thread
- A Task gives you more programmatic control than you get with a thread or work item
  - Waiting, cancellation, continuation, exception handling, status, scheduling, ...

Task + ContinueWith → async + await

- VS 2012 / .NET 4.5 introduced new keywords

```
await
async
```

- Writing and consuming async operations are part of the *language* AND the *framework*
  - Language: keywords + compiler
  - Framework: Task classes
- Compiler does a lot of work
- So, what is it?

Using Task<> *without* await (async)

```
Task<byte[]> downloadTask = new WebClient().DownloadDataTaskAsync(tgwUri);
downloadTask.ContinueWith(task =>
 {
   if (task.Exception != null)
   {
        Console.WriteLine("Downloaded {0} bytes. (ASYNC)", task.Result.Length);
   } else
   { … }
 } );
```

Using Task<> *with* await (async)

```
//calling asynchronous method using TAP and new await keyword
Task<byte[]> downloadTask = new WebClient().DownloadDataTaskAsync(tgwUri);
byte[] result = await downloadTask;
Console.WriteLine("Downloaded {0} bytes. (ASYNC)", result.Length);

//or even shorter:
byte[] result = await new WebClient().DownloadDataTaskAsync(uri);
Console.WriteLine("Downloaded {0} bytes. (ASYNC)", result.Length);
```

Demo

Consume + Develop a "Asynchronous Download" method using TAP with await and async (Demo 09, Demo 10)

TAP + UI Thread = ☺

- Use **TaskScheduler** to declare in which thread the continuation should be executed
- WPF creates a SynchronizationContext for us → provides associated *TaskScheduler* in the *ContinueWith* method

```
Task<byte[]> downloadTask = client.DownloadPageAsync(new Uri(…));
downloadTask.ContinueWith( aTask =>
  {
    if (aTask.Exception == null) {
      statusTextBlock.Text = aTask.Result.Length + " bytes downloaded.";
    }
  } ,
  TaskScheduler.FromCurrentSynchronizationContext()
);
```

TAP + UI Thread + async/await = ☺ ☺

- C# compiler uses UI SynchronizationContext automatically
  - Generates ContinueWith method
- Async WPF event handlers can call async methods

```
byte[] data = await client.DownloadPageAsync(new Uri(…));
statusTextBlock.Text = data.Length + " bytes downloaded.";
```

await Inside a Loop

```
int totalLength = 0;
foreach (string uri in uris)
{
  string html = await (new WebClient().DownloadStringTaskAsync(new Uri (uri)));
  totalLength += html.Length;
}
Console.WriteLine(totalLength);
```

- Logically, execution *exits the method* and *returns to the caller* upon reaching the *await* statement
  - Statements after *await* are part of the continuation (also Console.WriteLine(..))
- When task completes, the continuation kicks off and execution jumps back into the middle of the loop - right where it left off
- Like “yield return” statement

await and Exceptions

- If the task completes with an exception, the exception gets re-thrown onto whoever awaits it.

```
string html;
try
{
    html = await (new WebClient().DownloadStringTaskAsync(new Uri("http://no.x/")));
}
catch (Exception ex)
{
    html = "Error! " + ex;
}

Console.WriteLine(html);
```

Summary

- Asynchronous operation = non blocking method call
  Parallel <> Async
- Threading Basics
- APM: Asynchronous Programming Model
- EAP: Event-based Asynchronous Pattern
- TAP: Task Asynchronous Pattern
- Language + Framework: async, await + Task

- What‘s next? more TAP, more concurrent / parallel programming