PARALLEL TASKS IN C#/.NET

AGENDA

- Task Basics
- The Default Task Scheduler
- Passing data to asynchronous work
- The Task-Based Asynchronous Pattern (TAP)

TASK BASICS

- A task is an isolated, logical unit of work – a sequential operation fit for **parallelization**
  - A task is NOT a thread
  - A task is a sequential operation
  - `System.Threading.Tasks`
- Also: Use parallel tasks when you have several distinct **asynchronous** operations (see later under TAP)
- Tasks are provided in the Task Parallel Library (TPL)
  - Part of Microsoft Parallel Extensions for .NET (along with PLINQ)
  - TPL dynamically scales the degree of parallelism to most efficiently use all processors available.
  - TPL assists in the partitioning of work and scheduling of tasks in the .NET thread pool.

STARTING TASKS

Task parallelism at its simplest:

```
// Good ole sequential code
public void DoAll()
{
  DoLeft();
  DoRight();
}
```

```
// Using Parallel.Invoke()
public void DoAll()
{
  // Parallel.Invoke() automatically creates tasks
  // for the arguments, then awaits for completion
  Parallel.Invoke(DoLeft, DoRight);
}
```

```
// Using Task.Run() – from .NET 4.5
public void DoAll()
{
  Task t1 = Task.Run((Action) DoLeft);
  Task t2 = Task.Run((Action) DoRight);
  Task.WaitAll(t1, t2); // Wait for both tasks to
complete
}
```

```
// Using TaskFactory
public void DoAll()
{
  Task t1 = Task.Factory.StartNew(DoLeft);
  Task t2 = Task.Factory.StartNew(DoRight);
  Task.WaitAll(t1, t2); // Wait for both tasks to
complete
}
```

Tasks do not necessarily begin executing on creation. They are placed in a *work queue* from which a *task scheduler* remove and schedule them for execution when e.g. a core is available

WAITING FOR TASK COMPLETION

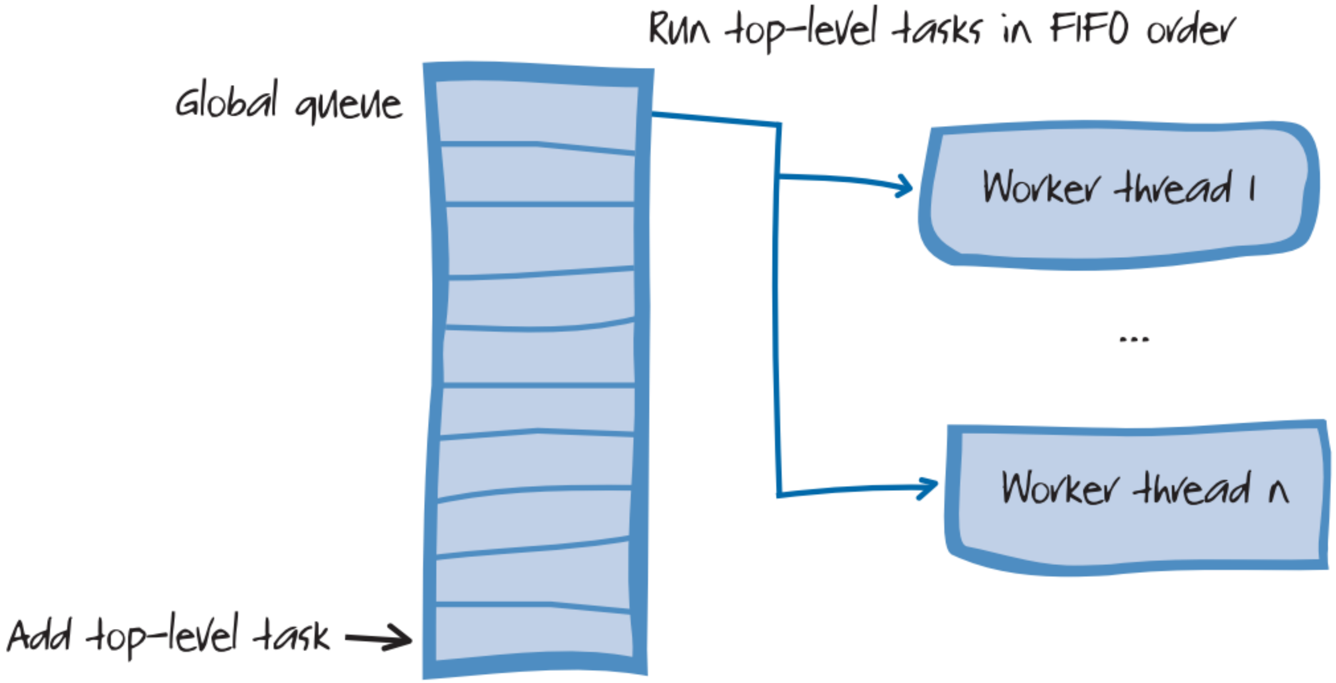
```
// Using WaitAll() or WaitAny()
public void DoAllUsingWait()
{
  Task t1 = Task.Run((Action)DoLeft);
  Task t2 = Task.Run((Action)DoRight);

  Task.Wait(t1);
  Task.Wait(t2);

  // --- OR ---

  Task.WaitAll(t1, t2); // Wait for both tasks to complete

  // --- OR ---

  Task.WaitAny(t1, t2); // Wait for the first task to complete
}
```

Tasks defer exception handling. The caller of `Wait*()` gets the exception. This allows "sequential-style" exception handing

THE DEFAULT TASK SCHEDULER

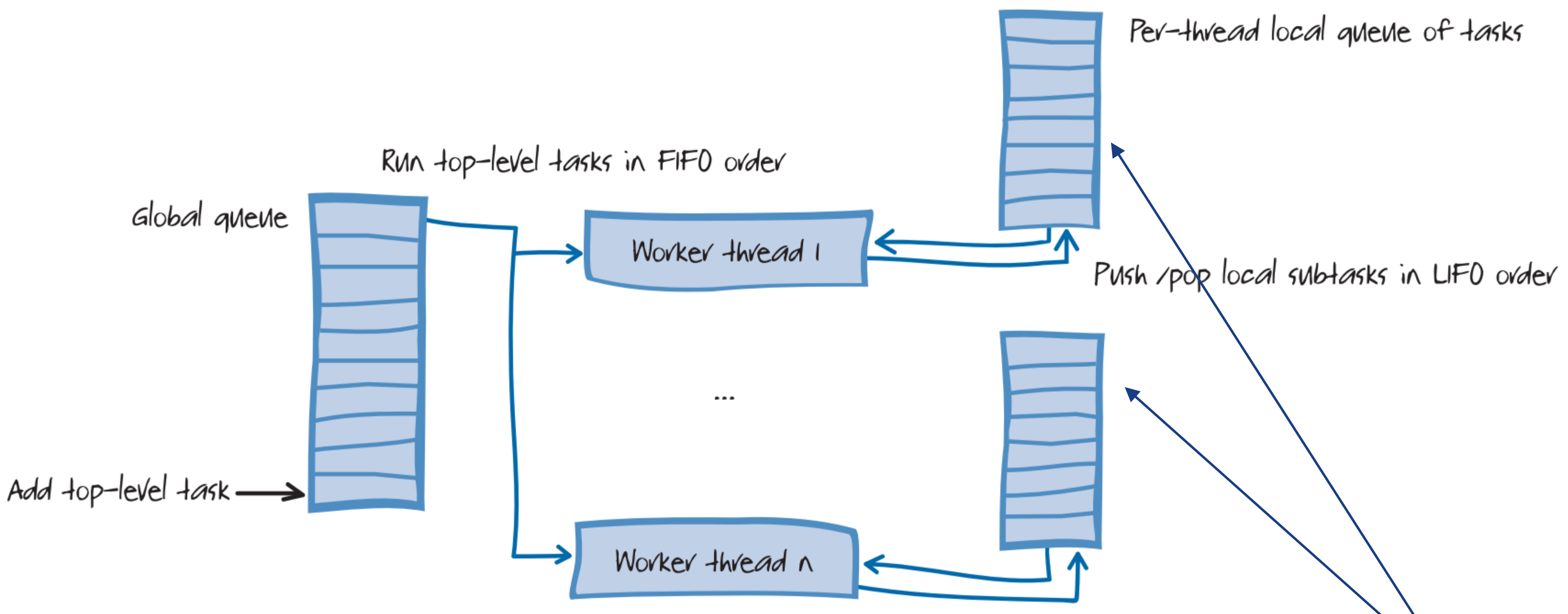
- TPL uses a task scheduler to schedule the tasks
  - Default task scheduler
  - Custom task scheduler (not covered)
- Tasks are generally very fine-grained work items
- TPL uses worker threads to execute tasks
  - Worker threads are managed by the .NET ThreadPool class
  - At least 1 thread per CPU core
  - Tasks (work items) queue on the threads (execution contexts)
  - "Millions" of tasks may exist for "a few threads"

TASK SCHEDULING – FIRST APPROACH



Many cores -> fine-grained tasks -> contention problems on work queue

THE DEFAULT TASK SCHEDULER



Local "work-stealing" task queues

INLINING EXECUTION OF PENDING TASKS

- The default task scheduler may *inline* waiting tasks
- When `task1` awaits `task2`, and `task2` is not started at the time `task1` awaits it, the scheduler can execute `task2` immediately on `task1`'s thread
- Happens only when...
  - `task1` calls `Task.Wait(task2)` or `Task.WaitAll()`, and
  - The local work queue of the thread at which `task1` is executing also contains `task2`

PASSING DATA USING CLOSURES

- Closures are easy!

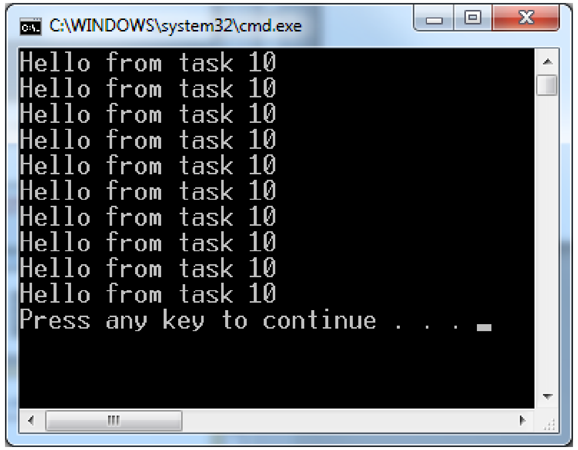
```
public void DoWork() {
  int data1 = 42;
  string data2 = "The Answer to the Ultimate Question of " +
    "Life, the Universe, and Everything";

  Task.Run(()=>
  {
    Console.WriteLine(data2 + ": " + data1);
  });
}
```

Delegate refers to state outside its scope (`data1` and `data2`), so compiler creates a ”closure” for `data1` and `data2` to make them accessible to the delegate

PASSING DATA USING CLOSURES

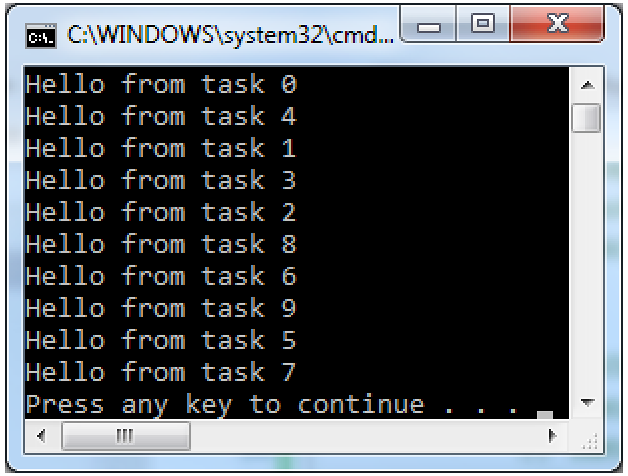
- Closures are also error-prone!
  - Copy the variable to be captured to a local variable before capturing it.

```
for(int i=0; i< 10; i++)
  Task.Run(()=> {
    Console.WriteLine(" Hello from task " + i);
  });
```



Capture value of **i** in local variable `localI`

```
for(int i=0; i< 10; i++)
{
  var localI = i;
  Task.Run(()=> {
    Console.WriteLine(" Hello from task " + localI);
  });
```



PASSING DATA USING STATE OBJECTS

- Passing objects containing state to task:

```
int taskNo = 42;

Task.Factory.StartNew((state) =>
  {
    Console.WriteLine(" Hello from task " + (int) state);
  }
  , taskNo);
```

taskNo is the value of state when task is run.
taskNo is captured properly
Passing objects can *not* be done with Task.Run()!

PASSING DATA USING STATE OBJECTS

- Passing objects containing state *and* methods:

```
class Work
{
  public int Data1;
  public string Data2;
  public void Run() {
    Console.WriteLine(Data1 + ": " + Data2);
  }
}

public static void Main()
{
  Work w = new Work();
  w.Data1 = 42;
  w.Data2 = "The Answer to the Ultimate Question of...";
  Task.Run(w.Run);
}
```

https://learn.microsoft.com/en-us/dotnet/api/system.action?view=net-6.0

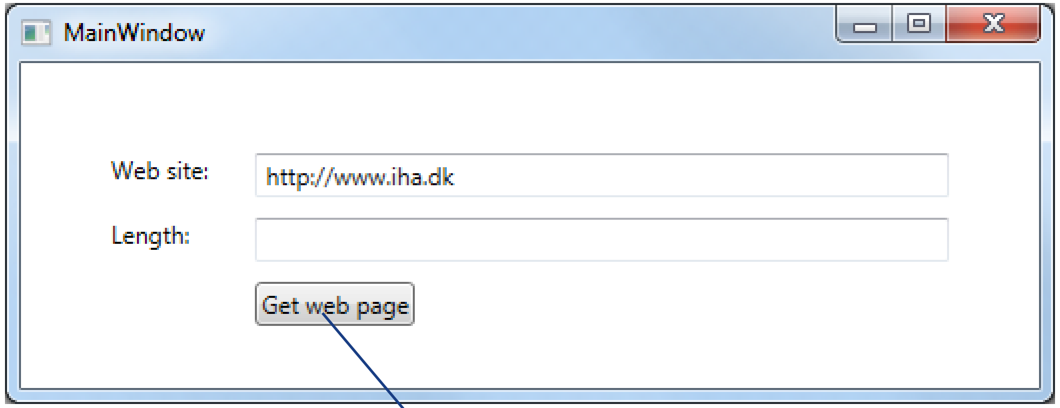
TASK-BASED ASYNCHRNOUS PATTERN (TAP)

- .Net 4.5 exposed asynchronous versions of many operations following the Task-based Asynchronous Pattern (TAP)

- TAP is used with the `async` modifier and can include `await` expressions

- Allows calling thread (e.g. UI thread) to remain responsive while heavy operations execute

TAP



GetStringAsync() actually returns a Task<string>, this is unwrapped by await

```
private async void btnGetHtml_Click(object sender, RoutedEventArgs e)
{
    tbxLength.Text = "Fetching...";
    string url = tbxUrl.Text;
    HttpClient client = new HttpClient();
    string text = await client.GetStringAsync(url);
    tbxLength.Text = text.Length.ToString();
}
```

The boxed code is wrapped as a new Task which will scheduled on the same thread when the operation returns

TAP - AWAIT

- `await` allows you to await the completion of an asynchronous operation
  - The rest of the code (from `await` onwards) is executed when the awaited operation returns.
  - The awaited operation is executed in the hardware, the drivers, and perhaps another thread
  - Use it for time-consuming IO (Input/Output)
- Awaiting is accomplished *without* blocking the calling thread (often the UI thread)
  - Instead, the awaited method returns immediately after `await`
  - If a result is immediately available, the caller continues
  - If not, a continuation is scheduled to execute on the caller thread when the awaited operation has been completed

`ConfigureAwait`

- `ConfigureAwait(false)`
  - Avoid queuing the task callback
  - Code after ‘await’ does not necessarily run in the same context
  - Improves performance

- `ConfigureAwait(true)`
  - == default behavior

AARHUS
UNIVERSITY