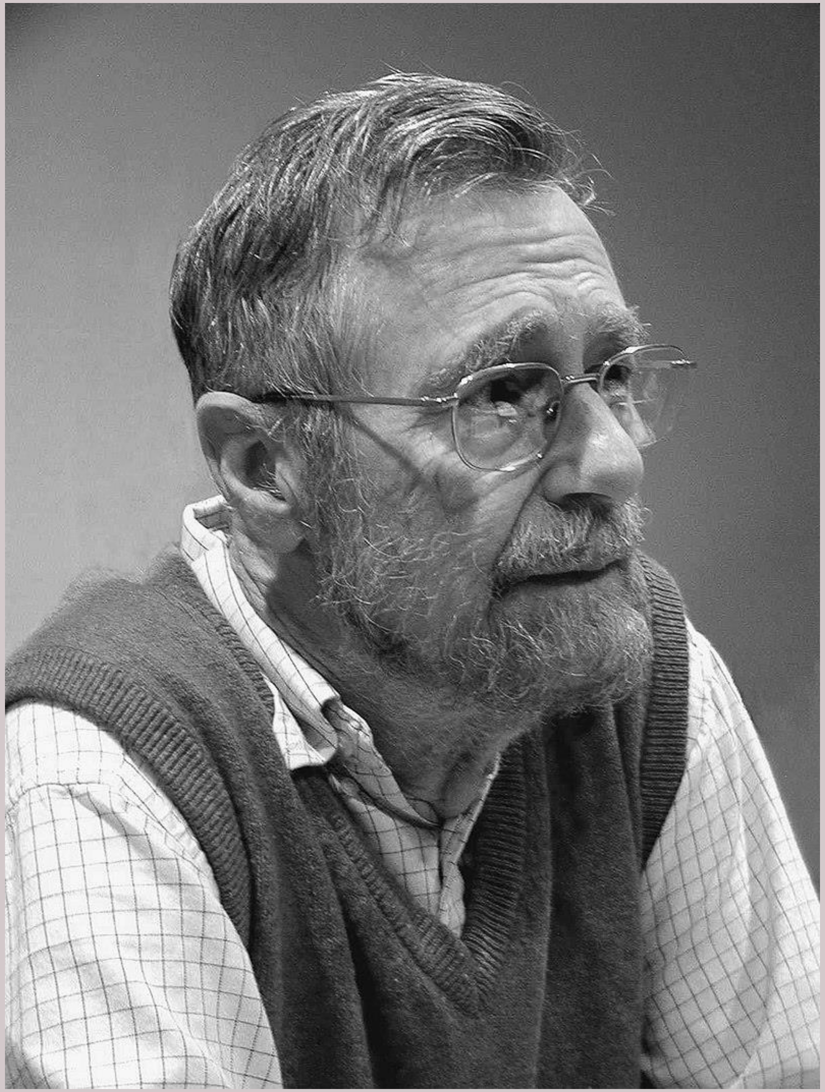
ASYNCHRONOUS PROGRAMMING



“Simplicity is a great virtue,
but it requires hard work to achieve it
and education to appreciate it.

And to make matters worse:
complexity sells better.”

(E. W. Dijkstra)

OVERVIEW

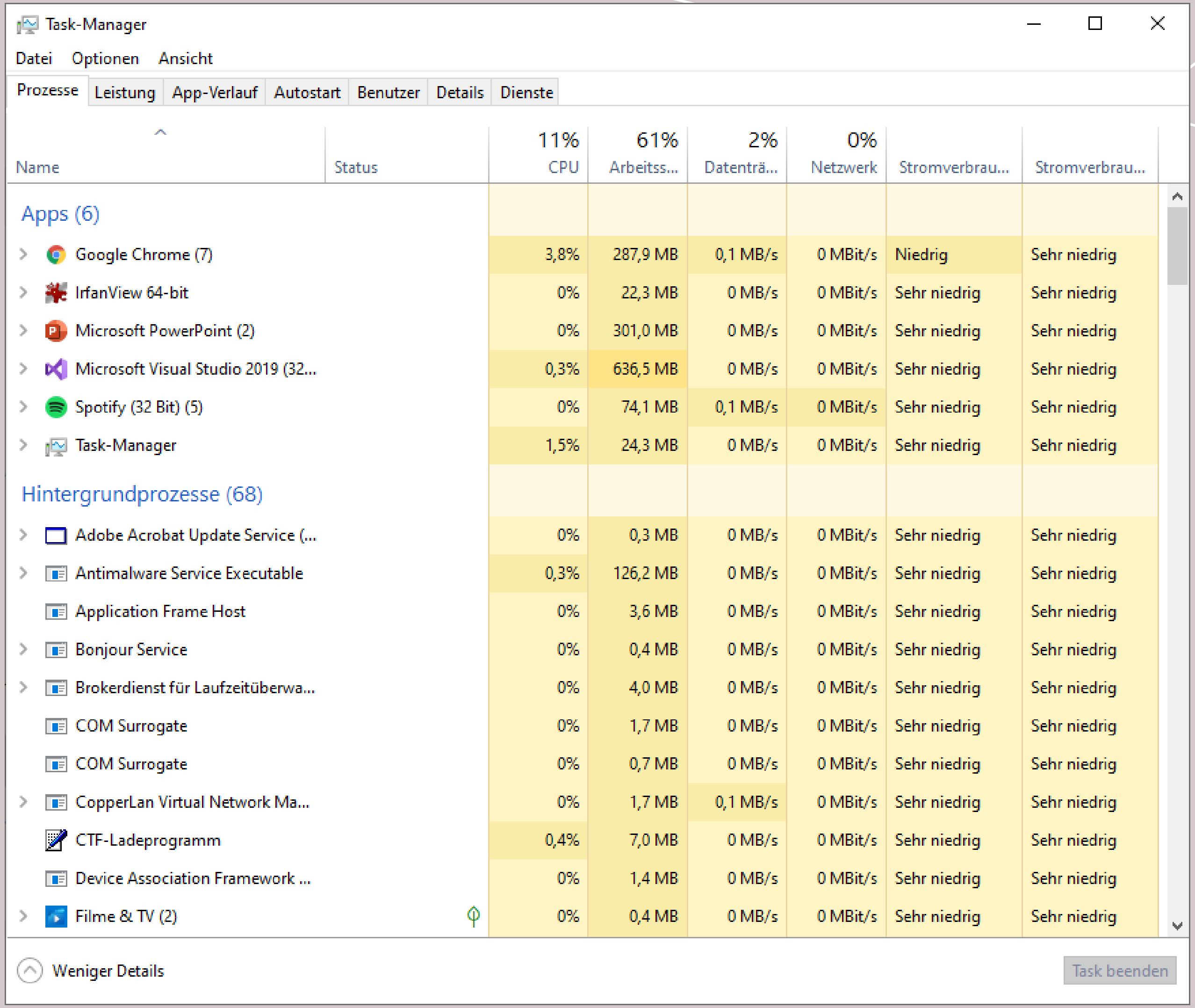
- Asynchronous Programming
- Process vs. Thread
- Concurrency and typical problems
- Synchronization strategies
- Task-based programming
- Async/Await Pattern
- Example Project

ASYNCHRONOUS PROGRAMMING

- “Simultaneous“ execution of code
  - Time-sharing on a single CPU core
  - Actual parallel execution on multiple cores, CPUs or systems

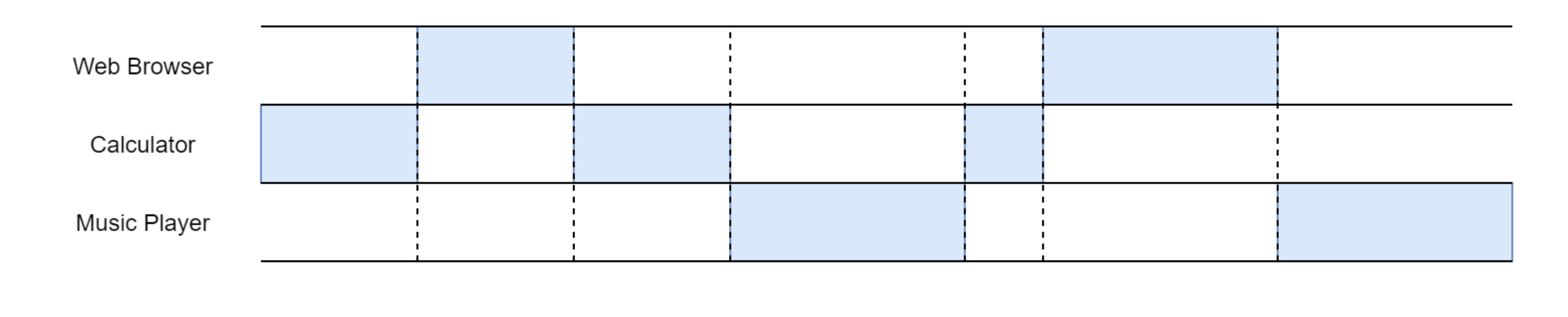
- Run multiple programs simultaneously on a single system
  - Handled by the operating system

- Parallel execution within a single program
  - Handled by the programming environment
  - Improved responsiveness and performance



WINDOWS PROCESSES

“TIME SLICING”



SCHEDULING

- A Scheduler manages parallel execution of work

- Assigns available resources (e.g. processors) to pieces of work (e.g. process or thread):
  - Preemptive: work can be interrupted (most used)
  - Non-preemptive: scheduler can only assign new work after previous work has finished

- Scheduling algorithms:
  - FIFO
  - Round-Robin
  - Scheduling based on priorities

PROCESS, THREAD & TASK

- Process:
  - Self-contained program or background process
  - Managed by operating system

- Thread:
  - Parallel execution of code within a programming language
  - Managed by the programming environment (e.g. .NET Runtime)

- Task:
  - “Piece of work” that can be executed by a thread

MULTI-THREADING

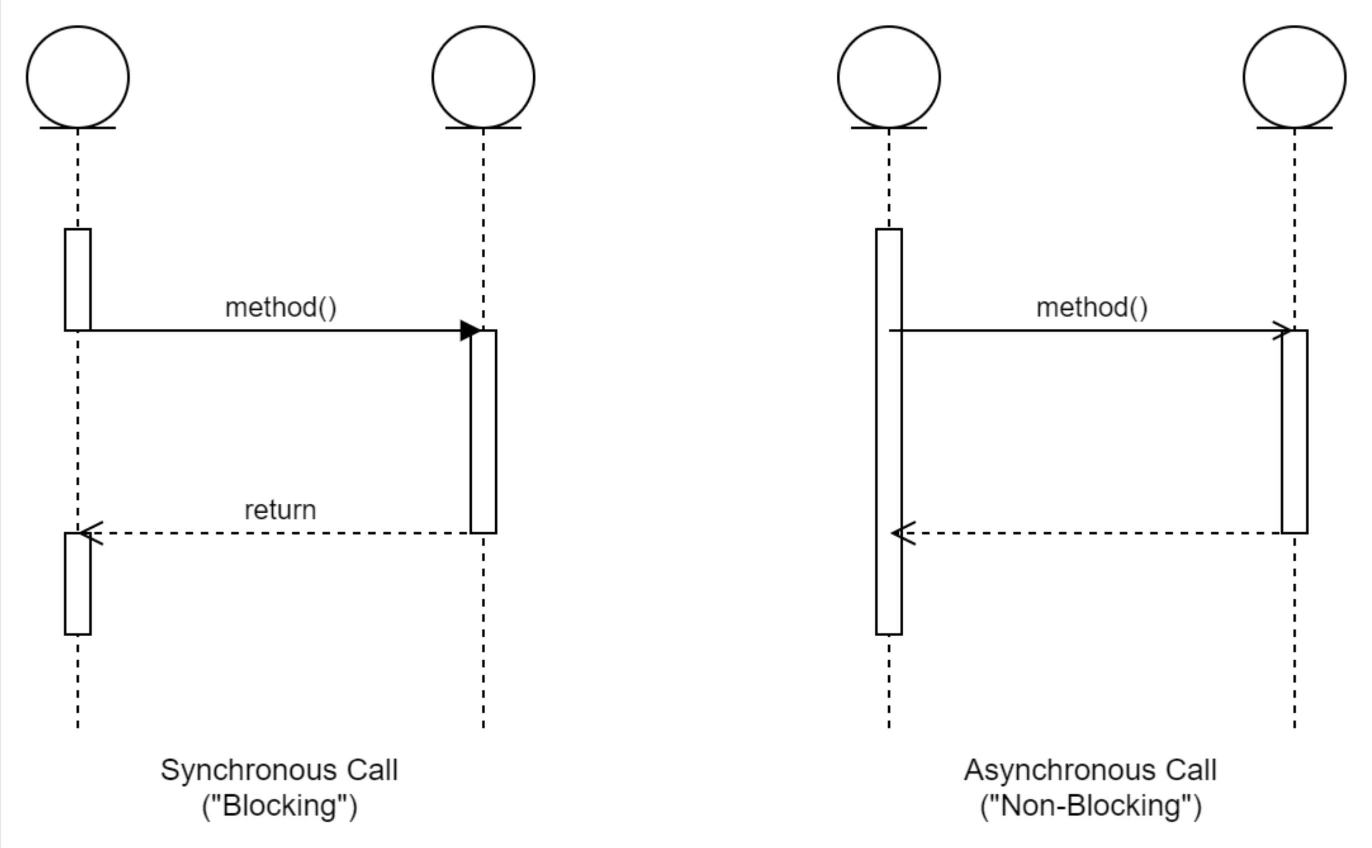
MULTI-THREADING

- Threads are a lightweight construct within a programming language
  - “Green threads”: completely managed by the programming environment
  - “Native threads”: provided and partially managed by the operating system (can also be executed on several processors!)

- Threads can read and write shared resources

- In many cases, synchronization between threads is required!

BLOCKING VS NON-BLOCKING CALLS



BENEFITS OF ASYNCHRONOUS PROGRAMMING

- Improved Responsiveness:
    - While waiting for an external result, the user can do something else
    - Avoid blocking while waiting (e.g. user interface waiting for a service call)

- Improved Performance:
    - Parallel execution of long running tasks can greatly improve response times
    - Parallel computation of subproblems for certain algorithms

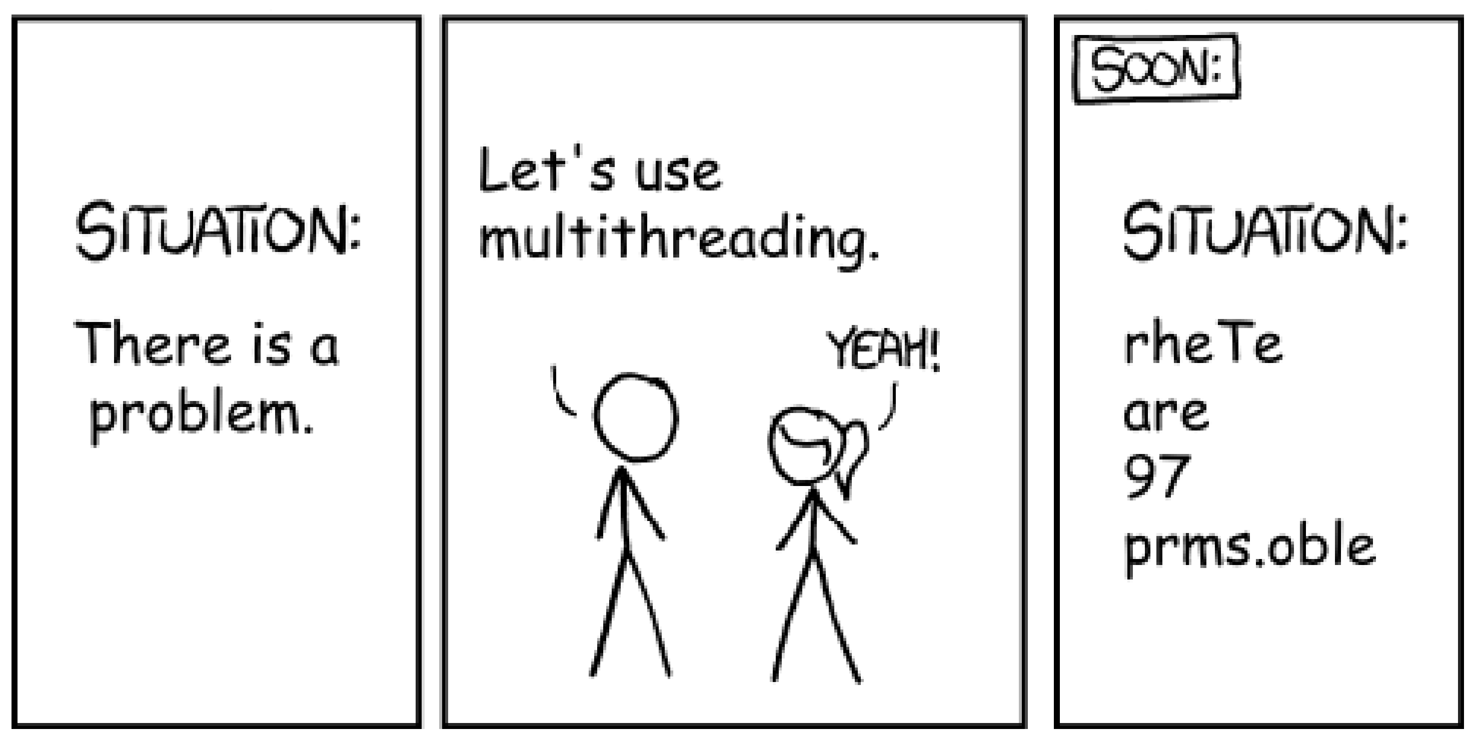
- Improved Resource Usage:
    - Threads can do something else while waiting (e.g. server waiting for a database request)

CHALLENGES OF ASYNCHRONOUS PROGRAMMING

- Several problems can occur with parallel execution of code

- Asynchronous bugs are often:
  - Hard to find
  - Hard to reproduce (“Heisenbug”)
  - Hard to solve

- Actual appearance of problems strongly depends on execution circumstances
  - System environment, workload, certain rare execution sequence patterns
  - Worst case: problem occurs for the first time in production


SITUATION:
There is a problem.
Let's use multithreading.
YEAH!
SOON:
SITUATION:
rheTe
are
97
prms.oble

THREADS IN C#

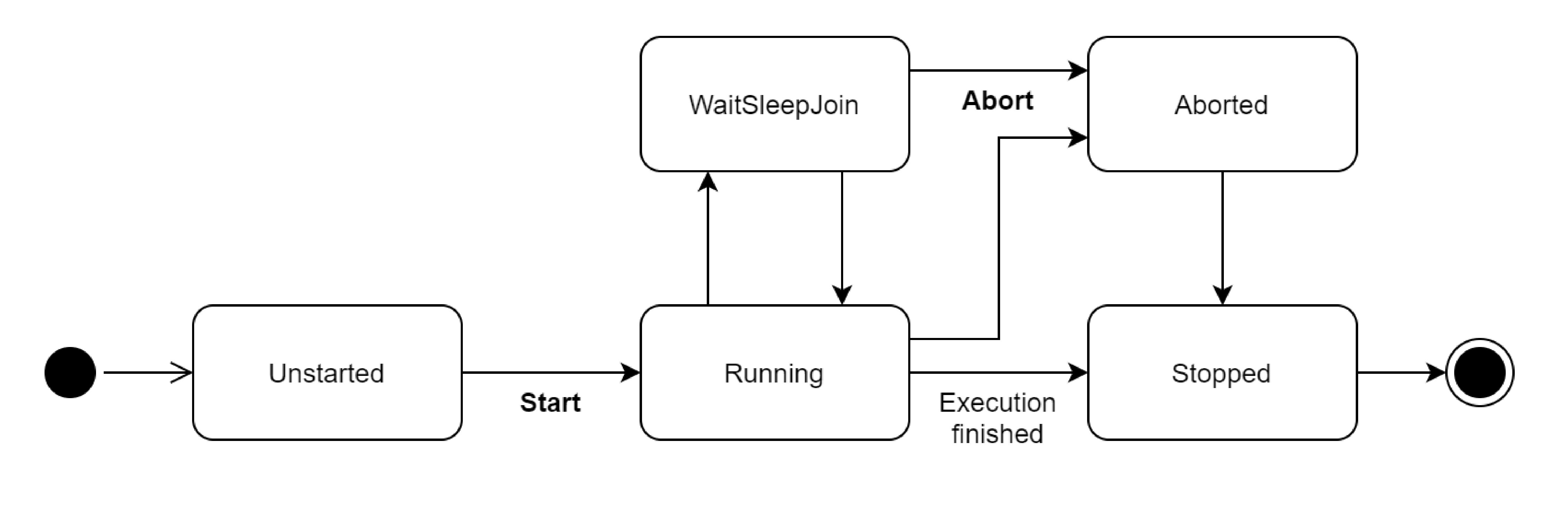
- Create a new thread: **Thread t = new Thread(MyMethod);**
- Get the current thread: **Thread.CurrentThread**
- Some common attributes:
  - **ManagedThreadId** (unique ID)
  - **ThreadState** (current state)
  - **IsBackground** (background threads do not prevent application from exiting)
  - **Priority** (threads with higher priority are executed more often and get more execution time)

THREADS IN C#

- Some common methods:
  - **Start** (begin execution)
  - **Abort** (stop the thread)
  - **Join** (wait for thread to finish by blocking the calling thread)
  - **Sleep** (wait for a certain amount of time)
  - **Interrupt** (wake up a thread which is sleeping or waiting by throwing an exception)

- Good code uses these methods only in very rare cases and for a good reason!
- Most of the time, programming can be entirely done on a higher level of abstraction (without manually dealing with threads at all!)

C# THREAD STATES & LIFECYCLE



DEMO: THREADS IN C#

- Thread creation
- Examples for behavior and effects of parallel execution

EXERCISE: SHARED DATA

- Implement two threads that access a shared variable **int data** like this:
  - while **data** is less than 100…
    1. Print the current value to the console
    2. Increase data by 1

- What happens and why?

```
int data = 0;

new Thread(() =>
{
    while (data < 100)
    {
        Console.WriteLine(data);
        data++;
    }
}).Start();

new Thread(() =>
{
    while (data < 100)
    {
        Console.WriteLine(data);
        data++;
    }
}).Start();
```

- Order of execution unpredictable
- At any time, a thread can be interrupted
- A variety of strange phenomena can occur!

- Even **data++** can be interrupted
  (only a shortcut for **data = data + 1**)

Data	Thread 1	Thread 2	Console
24	Console.WriteLine(data)		24
		Console.WriteLine(data)	24
		data + 1 = 25	
25		data = 25	
		Console.WriteLine(data)	25
		data + 1 = 26	
	data + 1 = 26		
26	data = 26		
	Console.WriteLine(data)		26
	data + 1 = 27		
27	data = 27		
	Console.WriteLine(data)		27
26		data = 26	
		Console.WriteLine(data)	26

CONCURRENCY

CONCURRENCY PROBLEMS

- Shared Resources introduce several problems

- **Race Condition**: result depends on order of execution
  - *Dirty Read:* local data is not up-to-date anymore
  - *Dirty Write:* out-of-date data is written back

DEMO: CONCURRENCY PROBLEMS

- Modified Collections (Flawed)

- Bank Account (Flawed)

SYNCHRONIZATION STRATEGIES

- Use thread-safe types (e.g. ConcurrentDictionary)
- Define “Critical Sections“ of code with restricted access
- Lock / Monitor:
  - Exclusive access to a section of code
  - Thread can request an exclusive lock to an object (“Enter”)
  - Other threads must wait until the lock is released again (“Exit”)

```csharp
int data = 0;

object lockObject = new object();

new Thread(() =>
{
    while (data < 100)
    {
        lock (lockObject)
        {
            Console.WriteLine(data);
            data++;
        }
    }
}).Start();
```

- Only one thread can enter the locked section
- Other threads must wait until the lock is released

Data	Thread 1	Thread 2	Console
24	**enter lock**		
	Console.WriteLine(data)		24
	data + 1 = 25		
25	data = 25		
	exit lock		
		enter lock	
		Console.WriteLine(data)	25
		data + 1 = 26	
26		data = 26	
		Console.WriteLine(data)	26
		data + 1 = 27	
		data = 27	
		exit lock	
	enter lock		
	Console.WriteLine(data)		27

BEST PRACTICES

- Avoid locking on public references
  - Behavior can get out of control if additional locks are added from outside

- Try not to lock whenever possible
  (e.g. if an attribute must not change during execution, just create a local copy)

- Thread-safe types only work with one operation
  - If the synchronization should involve more operations, a lock is required
    (e.g. “check size & conditionally add”)
  - Use regular types with a lock in the first place, so you don’t have to change your code later

DEMO: CONCURRENCY PROBLEMS

- Modified Collections (Fixed)
- Bank Account (Fixed)

EXERCISE: SHARED DATA FIXED

- Implement a thread-safe class **CountTo100** with:
  - an attribute **int number**
  - a method **void NextNumber()** that increases it up to 100 and then decreases it back to 0
- Implement two threads which:
  - each call **NextNumber()** in parallel for 100 times
- Wait for both threads to finish (Join) and print the number in the end (should be 0)

DEMO: PRODUCER-CONSUMER

- Classic example for concurrency (E. W. Dijkstra)
- One or several threads that produce data
- One or several threads that read and delete that data
- Data must only be read once!

EXERCISE: WAREHOUSE

- Implement thread-safe classes **Factory**, **Warehouse** and **Product**
- Factory and Warehouse can hold products up to a configurable capacity
- Create several threads that concurrently:
  - Add new products to the factory (up to its capacity)
  - Move products from factory to warehouse (only if warehouse has capacity left!)
  - Remove products from the warehouse

EXERCISE: WAREHOUSE HINTS

- In order to move a product from Factory to Warehouse, a lock needs to be acquired on both!
  (in order to e.g. remove from Factory only if the Warehouse has capacity left)

- This lock should have an internal visibility and can (also) be acquired from the accessing Thread

EVEN MORE CONCURRENCY PROBLEMS

- Using locks can solve race conditions ☺
- …but can lead to other problems ☹

- **Deadlock**: circular dependency situation

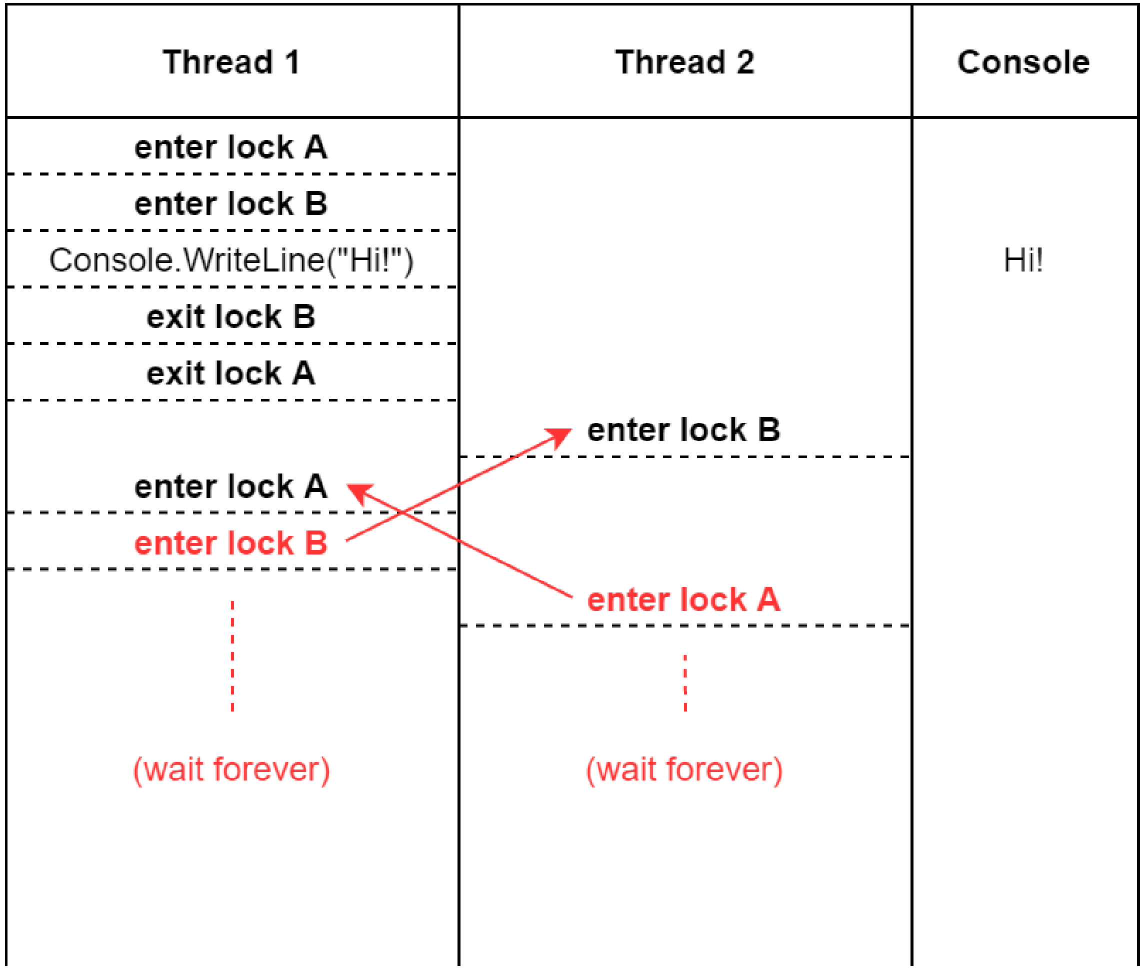
- **Livelock**: infinite loop of state changes

```csharp
object lockA = new object();
object lockB = new object();

new Thread(() =>
{
    while (true)
    {
        lock (lockA)
        {
            lock (lockB)
            {
                Console.WriteLine("Hi!");
            }
        }
    }
}).Start();

new Thread(() =>
{
    while (true)
    {
        lock (lockB)
        {
            lock (lockA)
            {
                Console.WriteLine("Hi!");
            }
        }
    }
}).Start();
```

- In previous examples, all threads lock in the same order
- If the order is different, deadlocks can easily occur!

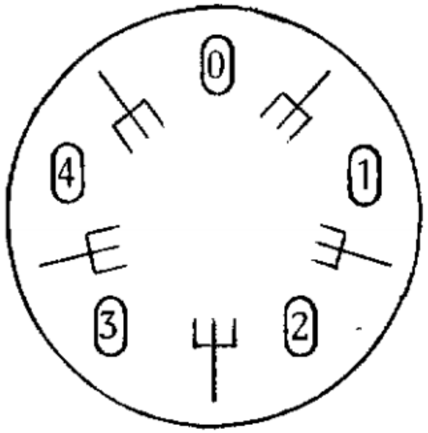

Thread 1
Thread 2
Console
enter lock A
enter lock B
Console.WriteLine("Hi!")
Hi!
exit lock B
exit lock A
enter lock B
enter lock A
enter lock B
enter lock A
(wait forever)
(wait forever)

DEMO: DINING PHILOSOPHERS

- Another classic example for concurrency (again, by E. W. Dijkstra)

- N Philosophers sit around a table
- Each philosopher alternately **eats** and **thinks**
- In order to eat, a philosopher needs two chopsticks
- However, they only have N chopsticks – one between two philosophers

Five philosophers, numbered from 0 through 4 are living in a house where the table laid for them, each philosopher having his own place at the table:



Their only problem - besides those of philosophy - is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

DINING PHILOSOPHERS

- In a continuous loop…
  1. **Think()** for some time
  2. **Eat()** for some time

- **Eat()** requires these steps:
  1. Take both left and right chopstick
  2. Use them to eat
  3. Put chopsticks back

DINING PHILOSOPHERS

- Naïve implementation of **Eat()**:
  - Wait until left chopstick is free and take it
  - Wait until right chopstick is free and take it
  - Eat
  - Put chopsticks back

- Can lead to a circular wait!

DINING PHILOSOPHERS

- Possible solutions to avoid the deadlock:
  - Use a central lock (only one philosopher can eat at a given time)
  - Use a central lock only during the process of taking the chopsticks
  - Introduce a hierarchical order over the chopsticks (Dijkstra’s solution)

EXERCISE: TRANSFERS BETWEEN MULTIPLE BANK ACCOUNTS

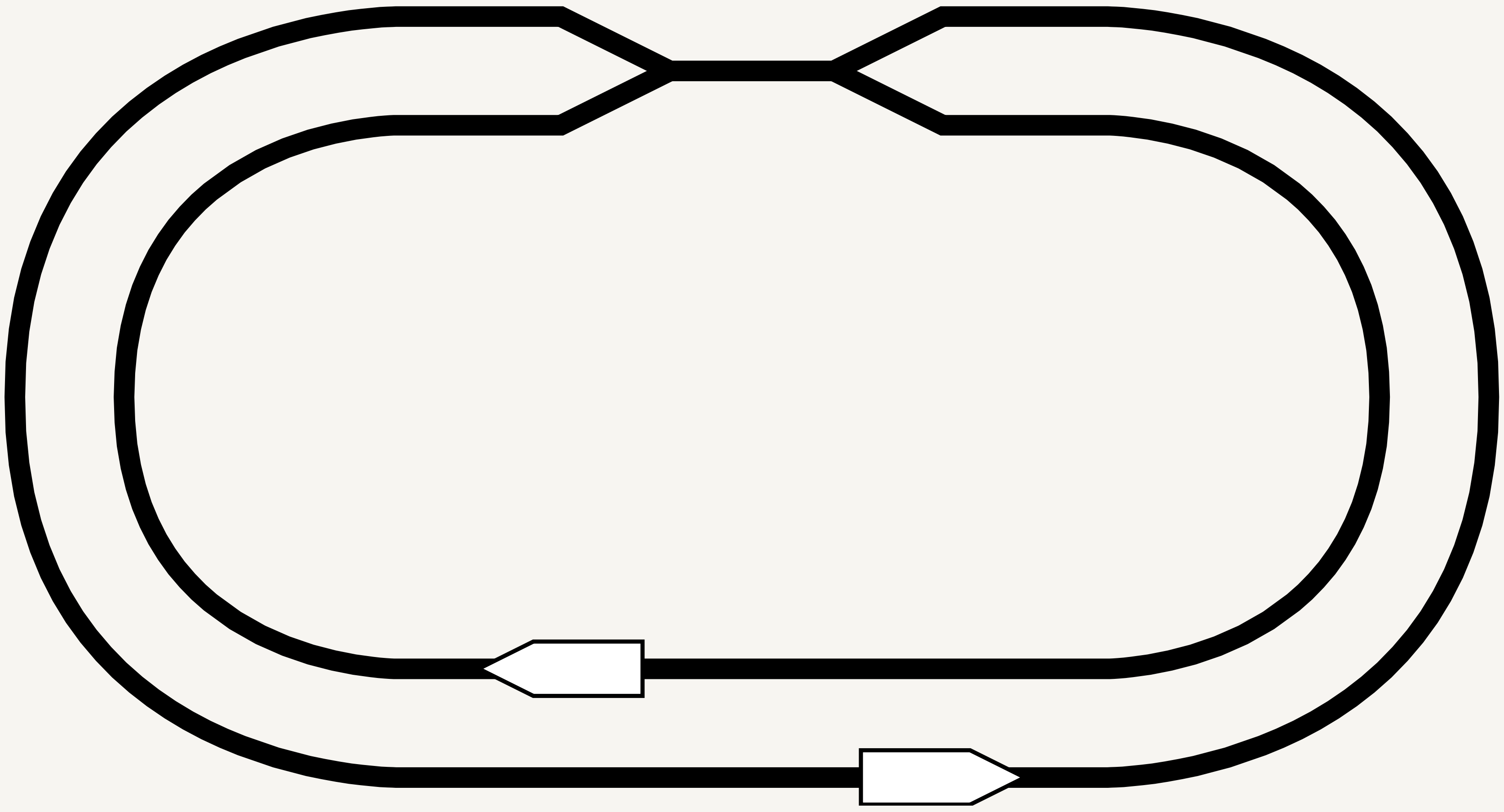
- Implement a thread-safe class **BankAccount** with:
  - A property **decimal Balance**
  - A method **TransferTo(BankAccount otherAccount, decimal amount)**
    - During a transfer, no other thread must be allowed to modify an account!
    - Ensure that several transfers can take place at the same time!
- Create several threads that randomly transfer money between accounts
- Keep track of transfers and check if the numbers match up in the end

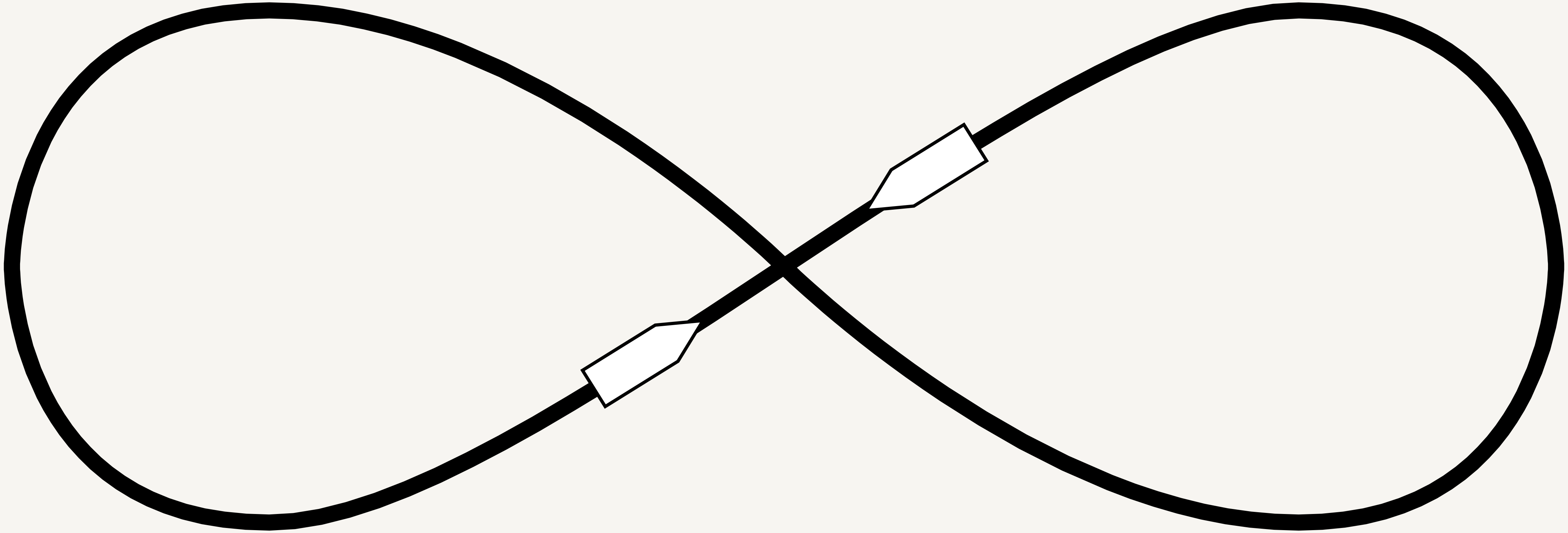
DEADLOCK DETECTION AND RESOLUTION

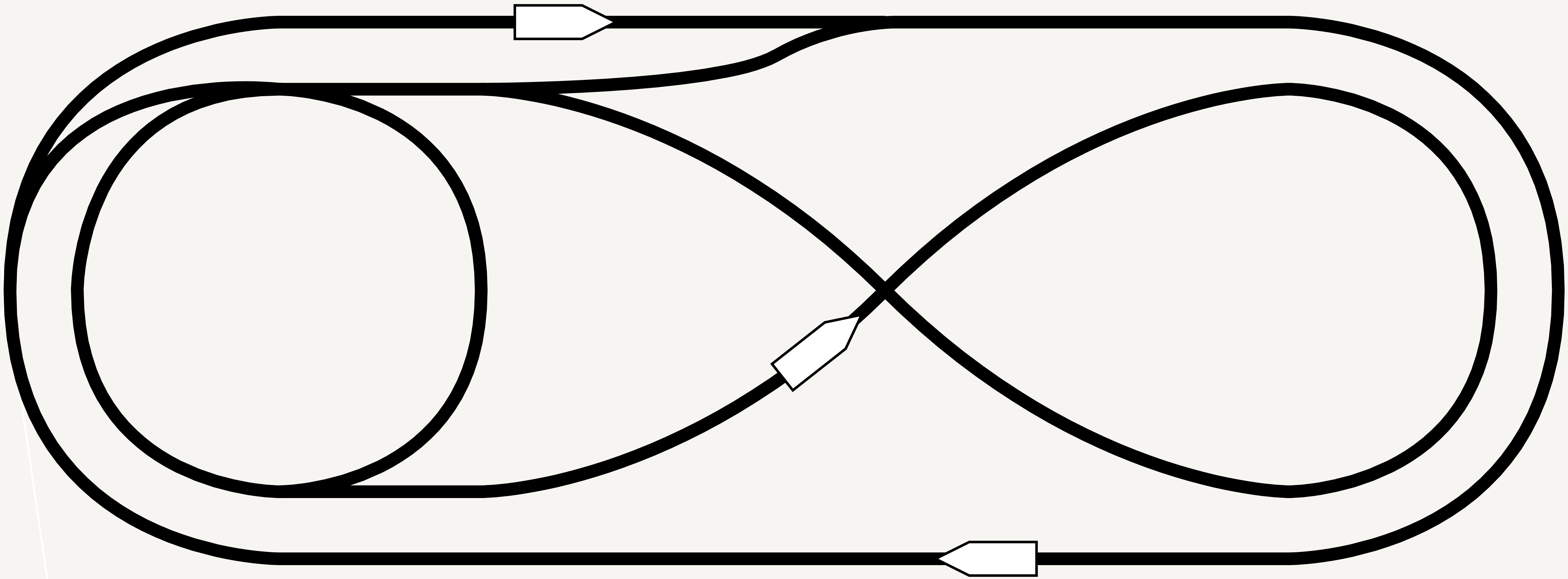
- Deadlocks can not only be prevented, but also detected and resolved
- However, the effort typically is very large, and resolution requires lots of problem-specific logic
- “Ostrich Algorithm”:
  - Put your head in the sand and hope nothing happens
  - Taken by most common operating systems

EXAMPLE: TRACKS & TRAINS

- Several trains drive to certain destinations on several connected track segments
- Trains can move to adjacent tracks in any direction
- Each track can only be occupied by one train







DEADLOCK DETECTION AND RESOLUTION

- There is no easy and general way to deal with complex deadlock/livelock situations

- Both **Prevention** and **Resolution** can require very complex additional logic
  - Centralized forward planning
  - Problem-specific communication between threads
  - “Rollback” of past actions that must not conflict with existing logic

- Luckily, most real-life problems are rather simple ☺

MANUAL UPDATE LOOP

- Concept from real-time systems and game programming
- “Don’t use asynchronous programming at all”
- Manually update every concurrent component in a defined order

MANUAL UPDATE LOOP

- Implement a method **Update()** in every component
- Implement a central loop which sequentially updates every component
- Benefits:
  - Defined update order and frequency
  - No code will ever get interrupted
  - Atomic execution of each component’s update method
- Drawbacks: requires manual work and is not very flexible

TASKS

TASKS

- Creating a new Thread is rather lightweight and cheap
  - however, it does take *some* effort

- For small pieces of work, the effort of creating a new thread might not be worth it

- Solution:
  - Use a fixed set of threads (“thread pool”)
  - Create only pieces of work (“tasks”) instead of actual threads
  - Threads in the pool are dynamically assigned to tasks
  - Threads are continually re-used for different tasks

TASKS IN C#

- Create a new task: **Task t = Task.Run(MyMethod);**

- Some common methods:
  - **t.Wait()** (wait to complete execution)
  - **Task.WaitAll(myTaskArray)** (wait for all tasks to complete)
  - **Task.WaitAny(myTaskArray)** (wait for any task to complete)
  - **Task.Delay(1000)** (creates a task that completes after a certain amount of time)

- Tasks can also have a return type: **Task<int> t = Task.Run(MyIntMethod);**
  - **t.Result** (wait for a task with a return type and get the result)

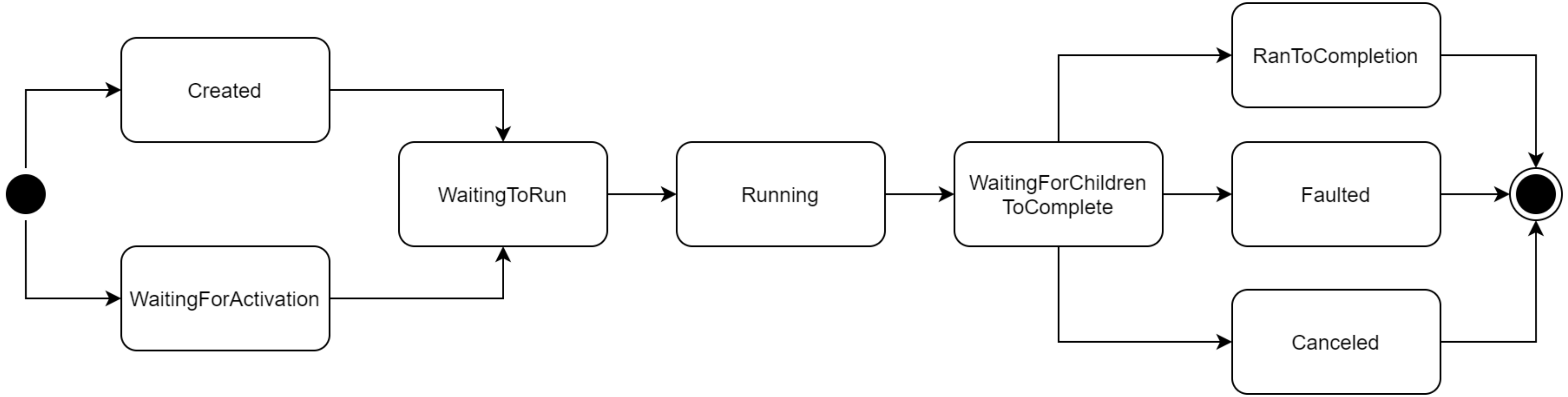
CANCEL A TASK

- Tasks can be cancelled before regular end of execution

- Manually implement cancellation (e.g. with a boolean variable)

- Use a **CancellationToken**
  - Created with a **CancellationTokenSource**
  - Cancellation can be requested with **source.Cancel();**
  - **token.ThrowIfCancellationRequested();** throws an exception if source has been cancelled

C# TASK STATES & LIFECYCLE



DEMO: CREATE TASKS IN C#

- Task creation
- Tasks with return type
- Wait for tasks to complete execution
- Cancel tasks

EXERCISE: GUESS MY LUCKY NUMBER

- Create a random number from 0 to N (e.g. N == 10.000.000)
- Create several tasks that try to guess that number (hint: use different random generators!)
- When the number has been guessed, cancel all other tasks
- Repeatedly guess numbers and measure the time it takes until the right number is found

- How does the time depend on N and the number of tasks?

SYNCHRONIZATION CONTEXT

- Defines certain synchronization requirements for certain scenarios (e.g. “only one thread can manipulate the UI at a given time”)

- Defines a so-called “synchronization model”

- Implementations exist e.g. for:
  - UI frameworks (UWP, WPF, Windows Forms)
  - Communication frameworks (WCF, ASP.NET)
  - Also custom implementations possible

EXAMPLE: UI DISPATCHER

- UI elements can only be manipulated by a single thread
  - Ensures sequential processing of user actions in a defined order
  - Avoids unpredictable results, inconsistent states and complex synchronization

- Computationally expensive or blocking code will completely block the UI!

- Solution:
  - Create a task that will run in another thread
  - Upon completion, explicitly invoke the UI thread to update UI elements

1 reference
```csharp
private void Button_Click(object sender, RoutedEventArgs e)
{
    Task.Run(() =>
    {
        // do some background work (e.g. call a service)

        // update UI
        textBlock.Text = "Updated!";
    });
}
```

Exception User-Unhandled

System.Exception: 'Eine Schnittstelle, die für einen anderen Thread marshalled war, wurde von der Anwendung aufgerufen. (Exception from HRESULT: 0x8001010E (RPC_E_WRONG_THREAD))'

View Details | Copy Details | Start Live Share session...

Exception Settings

Solution: invoke UI thread for changes to UI elements
(UWP: RunAsync, WPF: BeginInvoke)

```
1 reference
private void Button_Click(object sender, RoutedEventArgs e)
{
    Task.Run(() =>
    {
        // do some background work (e.g. call a service)

        // update UI
        Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () => { textBlock.Text = "Updated!"; });
    });
}
```

- Additional code which has nothing to do with actual functionality (“Boilerplate code”)
- When using MVVM pattern, separation between UI and ViewModel becomes harder

ASYNC/AWAIT

ASYNC/AWAIT

- Pattern for asynchronous programming
- Based on Tasks
- Dealing with blocking calls
- Waiting for results without blocking threads
- Improved responsiveness and performance

ASYNC/AWAIT

- Programming style very close to synchronous code
- Simplified task synchronization integrated directly in the programming language
- Simplified handling after task completion
  (e.g. most of the time no need to write callback methods or invoke a dispatcher)

ASYNC/AWAIT KEYWORDS

- Keyword **async**:
  - Marks a *method* as being asynchronous
  - Indicates that the method can contain one or more tasks that need to be waited on

- Keyword **await**:
  - Marks a *task* as required to be waited on
  - Suspends method execution and returns control to the caller
  - Does not block the calling thread! (unlike task.Wait())

ASYNC/AWAIT EXECUTION

- An **async** method first runs <u>synchronously</u> until the first **await**
- When an await is reached, the task behind it is executed <u>asynchronously</u>
- Control is returned to the calling thread (which can do other things in the meantime)
- After the task has finished, the result is returned (if any) and the method continues

- If not specified otherwise, the method will continue within the same SynchronizationContext from which it has been originally called from (i.e. no need to explicitly invoke e.g. the UI thread!)

```
1 reference
private async void Button_Click(object sender, RoutedEventArgs e)
{
    string updatedText = await Task.Run(() =>
    {
        // do some background work (e.g. call a service)

        return "Updated!";
    });

    // update UI (will be executed by UI thread again!)
    textBlock.Text = updatedText;
}
```

ASYNC/AWAIT RETURN TYPES

- For the calling thread to correctly process async methods, special return types are required
- **Task** corresponds to **void** (i.e. the method returns nothing)
- **Task<Type>** corresponds to a method returning **Type** (e.g. **Task<int>**)
  (sometimes also called a “promise” to eventually return something)
- Special case: **void**
  - Should only be used to write async event handler or commands
  - Can’t be awaited
  - Different behavior in exception handling

DEMO: ASYNC/AWAIT

- Create async methods

- Await tasks

EXERCISE: CSV FILE READER

- A comma-separated values file can contain multiple lines of data, each holding several values that are separated by a semicolon (‘;’)
- Define classes **CSVReader** and **CSVEntry** for reading such files
- **CSVEntry** contains an array of string values
- **CSVReader** provides a method **async Task<List<CSVEntry>> ReadFile(string fileName)**

EXERCISE: CSV FILE READER HINTS

- **Read a file line by line: File.ReadLines(filePath)**

- **Split a string along a character: string[] lineSplit = line.Split(';');**

ASYNC/AWAIT WITH LOCKS

- Locks inside a task (awaited or not) – no problem! ☺

- Lock around an awaited task – bad ☹
  - Lock is held by the calling thread, not the task's thread!
  - Inside the task, anything could happen (new tasks, new locks…)
  - After task completion, execution can be specified to be done not by the calling thread
  - Can lead to confusing scenarios, out-of-control behavior and deadlocks
  - Will not even compile!

ASYNC/AWAIT EXCEPTION HANDLING

- An async method can throw exceptions:
  - From the synchronous part
  - From within a task

- Only if an async method is being awaited, exceptions can be caught from the caller!
  - Otherwise, it depends on the return type:
    - For **Task** and **Task<Type>**, the exception will get lost
    - For **void**, the exception will be treated as unhandled

- If an exception is thrown from an awaited task, it can be handled “from outside” the task
  - However, in debugging mode it will show up as “unhandled” (though it will be handled later)

DEMO: ASYNC/AWAIT

- Async/Await with locks

- Async/Await exception handling

MIXING ASYNC/AWAIT WITH TASK.WAIT() / TASK.RESULT?

- Bad idea!
  - Use one or the other
  - Don’t mix the approaches

- For new code, better only use the Async/Await pattern

- When mixing or integrating with old code, take care of deadlocks!

- Button_Click is called by UI thread
- Calls Do() and waits for completion
- Wait() blocks the UI thread!
- UI thread will execute until the await
- Task is executed by another thread
- Upon completion, execution should be done by UI thread again

- Deadlock!

  (UI thread waits for Do() to complete, Do() waits for UI thread to be free again)

```
1 reference
private void Button_Click(object sender, RoutedEventArgs e)
{
    Do().Wait();
}

1 reference
private async Task Do()
{
    await Task.Delay(1000);
}
```

CONFIGURE AWAIT

- After an awaited task has completed, the original thread will continue execution by default

- However, it is also possible to explicitly continue with the task’s thread using this method:
  - task.configureAwait(boolean continueOnCapturedContext)
  - continueOnCapturedContext = false

- Setting this to false can yield better performance and prevent deadlocks

- Usage highly controversial (opinions range from “always use it” to “never use it”)

CONFIGURE AWAIT

- Performance Improvements?
  - Can indeed in some cases save some thread switches
  - However, benefit is not very large
  - In many cases, the original thread needs to continue (e.g. when manipulating UI elements)

- Deadlock Prevention?
  - Can indeed prevent deadlocks which result from mixing task.Wait()/Result with Async/Await
  - However, those should not be mixed anyway!
  - There may be some justified usages in combination with legacy code,
    but in general the problem should better be solved in a different way

ASYNC/AWAIT PROGRAMMING STYLE

To do this..	Instead of this:	Use this:
Retrieve the result of a background task	Task.Wait or Task.Result	await
Wait for any task to complete	Task.WaitAny	await Task.WhenAny
Retrieve the results of multiple tasks	Task.WaitAll	await Task.WhenAll
Wait a period of time	Thread.Sleep	await Task.Delay

TASK.WHENALL(…)

- Task.WhenAll(…) creates a task which returns an array of all the tasks' individual results
- It will wait for all tasks to complete
  - If an exception occurs in any task, it will be thrown
- Getting the results can be done like this: Type[] results = await Task.WhenAll(…)

MANUALLY AWAIT MULTIPLE TASKS

- As an alternative to Task.WhenAll(…), multiple tasks can also be awaited manually
- Useful for example when awaiting several tasks with different return types)

```
Task<int> task0 = Task.Run(() => { return 3; });
Task<string> task1 = Task.Run(() => { return "Hi!"; });

int result0 = await task0;
string result1 = await task1;
```

TASK.WHENANY(…)

- Task.WhenAny(…) creates a task which returns another task (Task<Task<Type>>)

- It will return the first task that completes
  - If an exception occurs in that first task, it will be thrown
  - Exceptions in other tasks will not be thrown (but will be shown as unhandled when debugging)

- Getting the actual result can be done like this: await await Task.WhenAny(…)

DEMO: ASYNC/AWAIT

- Task.WhenAny(…)

- Task.WhenAll(…)

EXERCISE: FRUIT STAND

- A fruit stand should be implemented based on these classes:
  - **Search**, with a method **Task<List<FruitSearchResult>> Search(string name)**
  - **Product**, with a method **Task<List<FruitInformation>> GetInformation(List<int> fruitIDs)**
  - **Price**, with a method **Task<List<FruitPriceInformation>> GetPrice(List<int> fruitIDs)**
  - Data is provided as .csv files, on each method call the file should be loaded again

- Implement a class **Client** which calls all 3 methods and combines the results

THANKS FOR YOUR ATTENTION!

QUESTIONS?

DISCUSSION!