# Many Flavors of Concurrency: Asynchronous and Event-Driven Programming

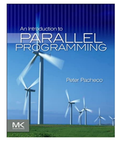
CS4405 – Analysis of Concurrent and Distributed Programs

Burcu Kulahcioglu Ozkan

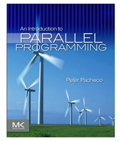
TUDelft

## Many flavours of concurrency

- Concurrent programming
  - Multiple tasks can be in progress at any instant
- Parallel programming
  - Utilizing more than one processors for running the program
- Asynchronous programming
  - Programming with non-blocking requests/method calls
- Event-driven programming
  - The flow of the execution is determined by the (possibly concurrent) events
- Distributed programming
  - Multiple computers run as a single system



*Many authors consider shared-memory programs to be "parallel" and distributed-memory programs to be "distributed"*



## Single-threaded asynchronous programming

## Asynchronous programming

What does the following JS code print to the console?

```
   function myfunc() {
      console.log(' Here ')
    }

    setTimeout(myfunc, 2)

    console.log(' I am ')
```

Asynchronous programming refers to a style of structuring a program whereby a *call* to some unit of functionality triggers an action that is allowed to continue outside of the ongoing flow of the program.

[https://nodesource.com/blog/why-asynchronous/]

Asynchronous programming allows us to execute a block of code without blocking the caller thread

## Synchronous vs asynchronous

```
function myfunc(s) {
  console.log(s)
}

myfunc('A‘)

myfunc(‘B‘)


console.log('C')
```

```
function myfunc(s) {
  console.log(s)
}

setTimeout(myfunc, 2, 'A‘)

setTimeout(myfunc, 2, ‘B‘)

console.log('C')
```

## Asynchrony using callbacks

- Callback is any reference to executable code that is passed as an argument to other code; that other code is expected to call back (execute) the code at a given time. This execution may be immediate as in a synchronous callback, or it might happen at a later point in time as in an asynchronous callback

- Do not block, call it back when needed.

- Programming languages support callbacks in different ways, e.g., lambda expressions, blocks, function pointers.

## Timeout callbacks:

```
console.log(' Let\'s start! \n' )

setTimeout(() => {
  console.log(' Here ')
}, 2)

setTimeout(() => {
  console.log(' I am ')
}, 2)
```

Sets a timer and executes the callback function after the timer expires

What does the JS code above print?

## User event callbacks:

```
document.getElementById("demo").onclick = function() {
    myFunction()
};

function myFunction() {
  document.getElementById("demo").innerHTML = "CLICKED!";
}
```

Calls the callback function whenever the event occurs

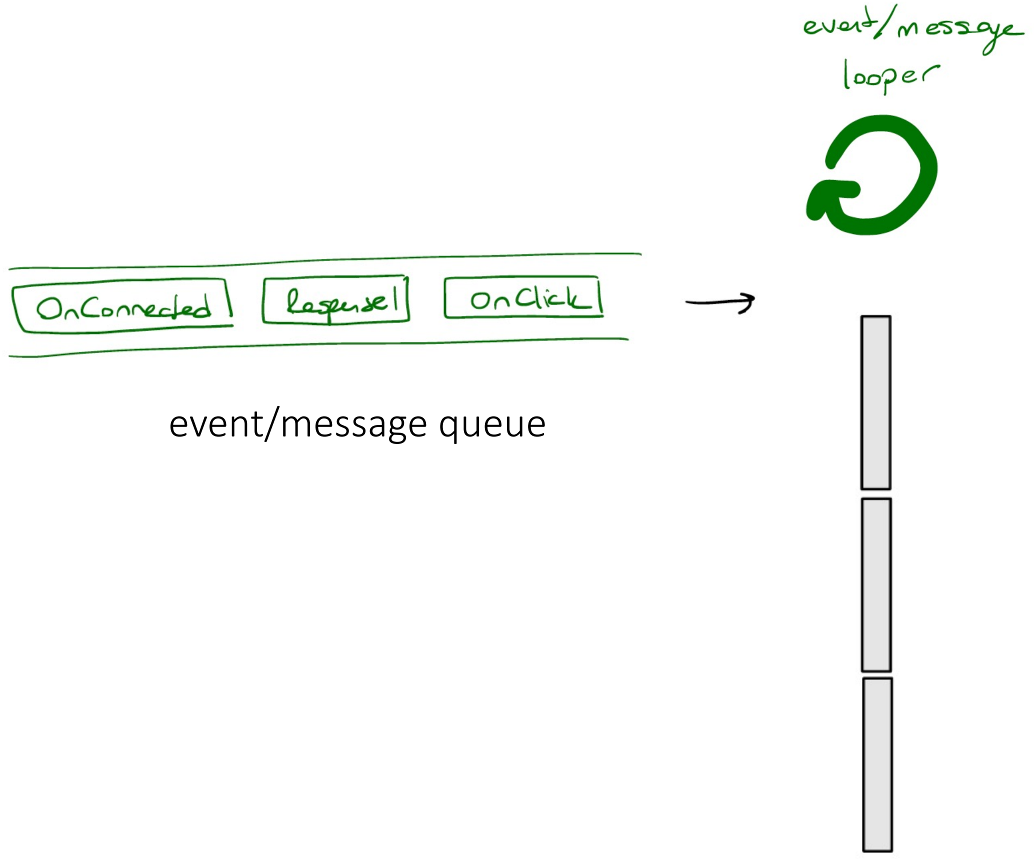
## Network event callbacks

```
const req = new XMLHttpRequest(),
method = "GET",
url = "https://cs4405.github.io/";

req.open(method, url, true);

// set up the callback
req.onreadystatechange = function () {
  if(req.readyState === XMLHttpRequest.DONE) {
    var status = req.status;
    if (status === 0 || (status >= 200 && status < 400)) {
      // The request has been completed successfully
      console.log(' Here ')
    } else {
      console.log(' Error on call 1 ')
    }
  }
};

req.send();

console.log(' I am ')
```

## Example with multiple callbacks: What does it print?

```
const req1 = new XMLHttpRequest(), method1 = "GET",
url1 = "https://cs4405.github.io/slides/lecture-3.pdf";
req1.open(method1, url1, true);

req1.onreadystatechange = function () {
    if(req1.readyState === XMLHttpRequest.DONE) {
      if (req1.status === 0 || (req1.status >= 200 && req1.status < 400)) {
        console.log(' Call 1 resolved')
      }
}};

const req2 = new XMLHttpRequest(), method2 = "GET",
url2 = "http://cs4405.github.io/";
req2.open(method2, url2, true);

req2.onreadystatechange = function () {
    if(req2.readyState === XMLHttpRequest.DONE) {
      if (req2.status === 0 || (req2.status >= 200 && req2.status < 400)) {
        console.log(' Call 2 resolved')
      }
}};

req1.send();
req2.send();
```

# Single threaded event loop:



## Nested callbacks, aka, “callback hell”

```
window.addEventListener('load', () => {
  document.getElementById('button').addEventListener('click', () => {
    setTimeout(() => {
      items.forEach(item => {
         // some code here
      })
    }, 2000)
  })
})
```

```
firstFunction(args, function() {
  . . .
  secondFunction(args, function() {
    . . .
    thirdFunction(args, function() {
      . . .
      // And so on…
    });
  });
});
```

## Structured asynchrony: Promises (ES2015)

- A promise is commonly defined as **a proxy for a value that will eventually become available**
  - Once a promise has been called, it will start in **pending state**.
  - The caller is not blocked: it continues its execution
  - Promise can be resolved in: resolved state or rejected state

```
const checkIfItsDone = () => {
  p.then(ok => { console.log(ok) })
   .catch(err => { console.error(err) })
}

const fetchPromise = fetch("https://. . .");
   fetchPromise.then(response => {
   console.log(response);
});
```

## Structured asynchrony: Chaining promises

- A promise can be returned to another promise, creating a chain of promises.

```
firstFunction(args, function() {
  . . .
  secondFunction(args, function() {
    . . .
    thirdFunction(args, function() {
      . . .
      // And so on…
    });
  });
});
```

callbacks

```
const chainAll = () => {
  first.then(ok => { second (..) })
       .then(ok => { third (..) })
       .catch(err => { console.error(err) })
}
```

promises

Implemented in **JavaScript, Node. js, Scala, Java, C++, etc**

## Another example for chaining promises

```
const status = response => {
 if (response.status >= 200 && response.status < 300) {
   return Promise.resolve(response)
  }
 return Promise.reject(new Error(response.statusText))
}
const json = response => response.json()
fetch('/todos.json’)
  .then(status)
  .then(json)
  .then(data => { console.log('Request succeeded with JSON response', data) })
  .catch(error => { console.log('Request failed', error) })
```

Example from: https://flaviocopes.com/javascript-callbacks/

## Structured asynchrony: Async/await (ES2017)

- A higher level abstraction built on promises

```
const doSomethingAsync = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}

const doSomething = async () => {
  console.log(await doSomethingAsync())
}

console.log('Before')
doSomething()
console.log('After')
```

Implemented in C# 5.0, C++, Python 3.5, F#, Kotlin 1.1, Rust 1.39, JavaScript ES2017, Scala (beta), etc

# Structured asynchrony: Async/await

- Simpler to read code, similar to synchronous calls

```
const getFirstUserData = () => {
  return fetch('/users.json')
    .then(response => response.json())
    .then(users => users[0])
    .then(user => fetch(`/users/${user.name}`))
    .then(userResponse => userResponse.json())
}
getFirstUserData()
```

promises

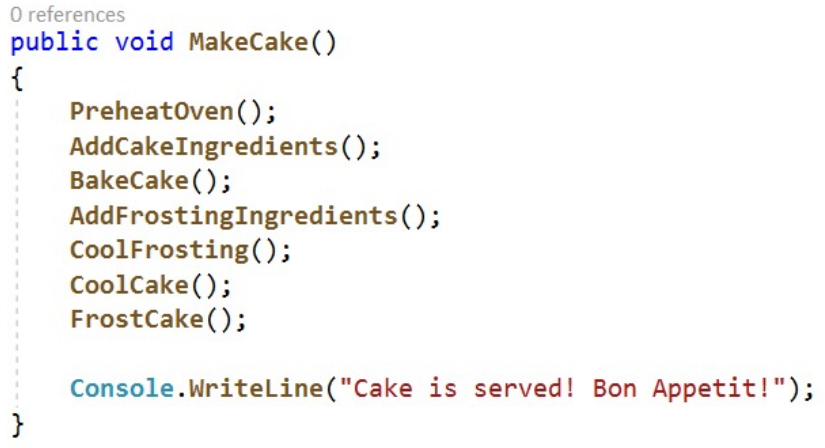
```
const getFirstUserData = async () => {
  const response = await fetch('/users.json')
  const users = await response.json()
  user = users[0]
  const userResponse = await fetch(`/users/${user.name}`)
  const userData = await userResponse.json()
}
getFirstUserData()
```

async/await

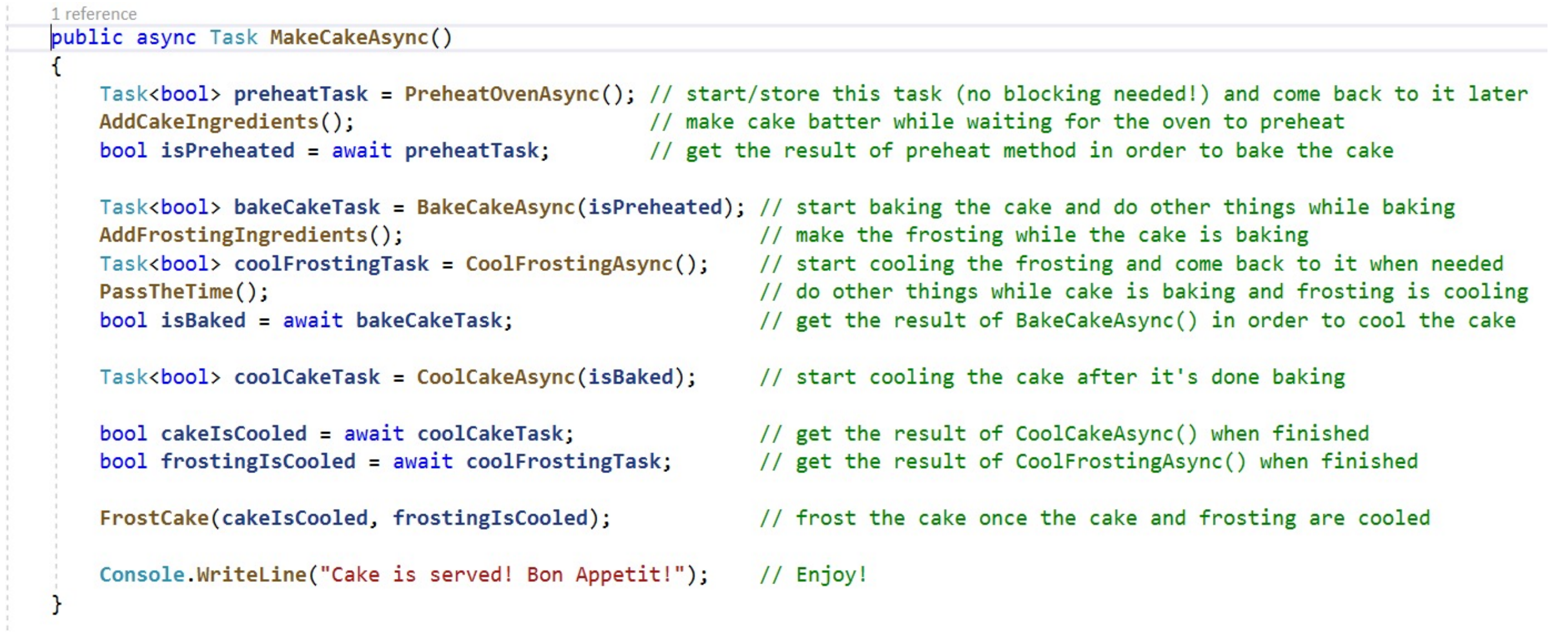
Example from: https://flaviocopes.com/javascript-async-await/

## Why use asynchrony?

- Increase responsiveness: Less blocking
- Increase scalability: Programs can serve for multiple requests concurrently



```
0 references
public void MakeCake()
{
    PreheatOven();
    AddCakeIngredients();
    BakeCake();
    AddFrostingIngredients();
    CoolFrosting();
    CoolCake();
    FrostCake();

    Console.WriteLine("Cake is served! Bon Appetit!");
}
```



```
1 reference
public async Task MakeCakeAsync()
{
    Task<bool> preheatTask = PreheatOvenAsync(); // start/store this task (no blocking needed!) and come back to it later
    AddCakeIngredients();                        // make cake batter while waiting for the oven to preheat
    bool isPreheated = await preheatTask;        // get the result of preheat method in order to bake the cake

    Task<bool> bakeCakeTask = BakeCakeAsync(isPreheated); // start baking the cake and do other things while baking
    AddFrostingIngredients();                             // make the frosting while the cake is baking
    Task<bool> coolFrostingTask = CoolFrostingAsync();    // start cooling the frosting and come back to it when needed
    PassTheTime();                                        // do other things while cake is baking and frosting is cooling
    bool isBaked = await bakeCakeTask;                    // get the result of BakeCakeAsync() in order to cool the cake

    Task<bool> coolCakeTask = CoolCakeAsync(isBaked);     // start cooling the cake after it's done baking

    bool cakeIsCooled = await coolCakeTask;               // get the result of CoolCakeAsync() when finished
    bool frostingIsCooled = await coolFrostingTask;       // get the result of CoolFrostingAsync() when finished

    FrostCake(cakeIsCooled, frostingIsCooled);            // frost the cake once the cake and frosting are cooled

    Console.WriteLine("Cake is served! Bon Appetit!");    // Enjoy!
}
```

Example from: https://devblogs.microsoft.com/visualstudio/how-do-i-think-about-async-code/

## Asynchronous programming pros and cons:

- ✓ No low-level race conditions
- ✓ Control over task switching

- x Difficult to implement correctly: Complex control flow
- x Need to be careful with the load for better performance

Example from: https://devblogs.microsoft.com/visualstudio/how-do-i-think-about-async-code/

## Concurrency errors due to asynchrony: Data races

- Caused by the nondeterminism in the event dispatch, network responses, CPU speed, etc.
- The traditional definition of a data race (for multithreaded programs) does not apply directly

- [Petrov. et. al, 2012] identifies several types of data races in web applications and proposes WEBRACER dynamic race detector for web applications
  - Based on a definition of a happens-before relation for JS and HTML features

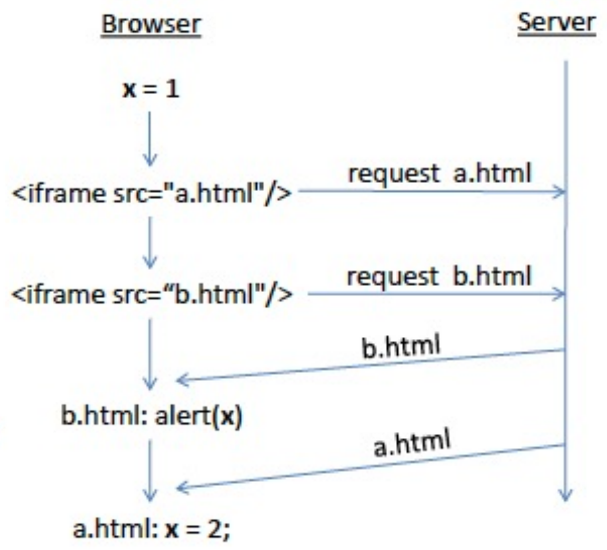
- Let $A, A' \in \{read, write\} \times OpId$ be memory accesses to some logical location (JavaScript variable or an HTML elements in the DOM) $m$ in an execution. A race exists between $A$ and $A'$ if:
  - $op(A) \neq op(A')$
  - $A$ and $A'$ are not related with happens-before relation
  - one of the accesses $A$ and $A'$ is a write

[Petrov. et. al, 2012] Race detection for web applications
B. Petrov, M. T. Vechev, M. Sridharan, J. Dolby. PLDI 2012

## Data races in JS programs

- Data race on a variable

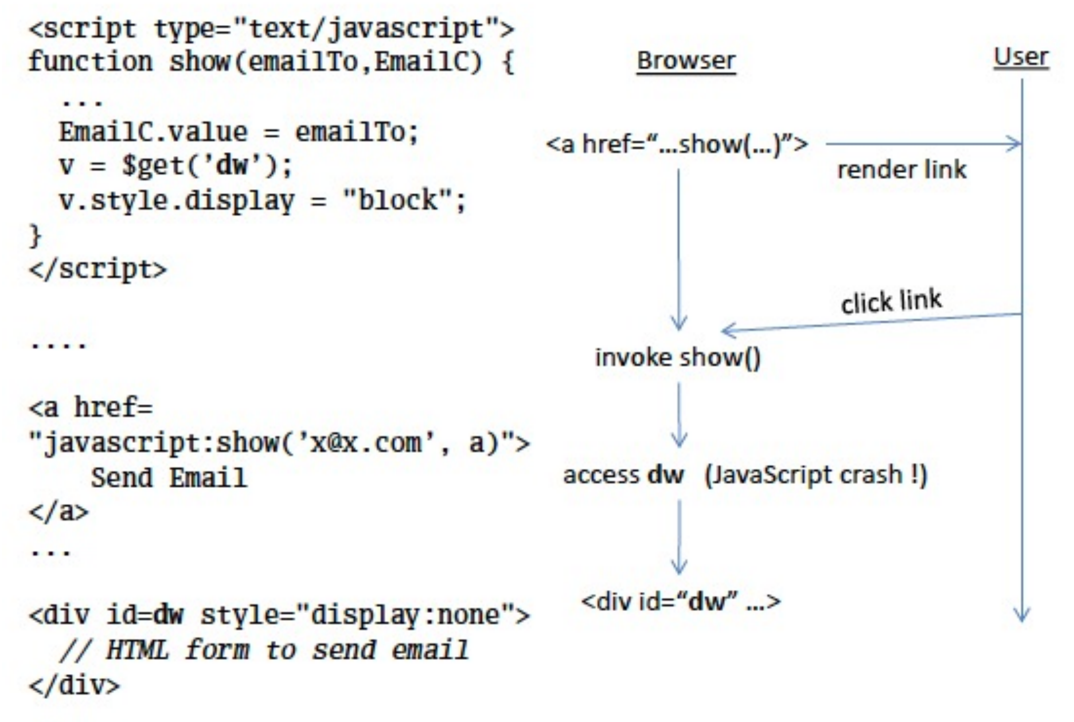
```
<script>x = 1;</script>
<iframe src="a.html" />
<iframe src="b.html" />

<!-- a.html -->
<script>x = 2;</script>

<!-- b.html -->
<script>alert(x);</script>
```



```
<input type="text" id="depart" />
...
<script type="text/javascript">
// add a hint to the box
document.getElementById("depart").value =
  "City of Departure";
// code to remove hint when user clicks
...
</script>
```

# Data races in JS programs

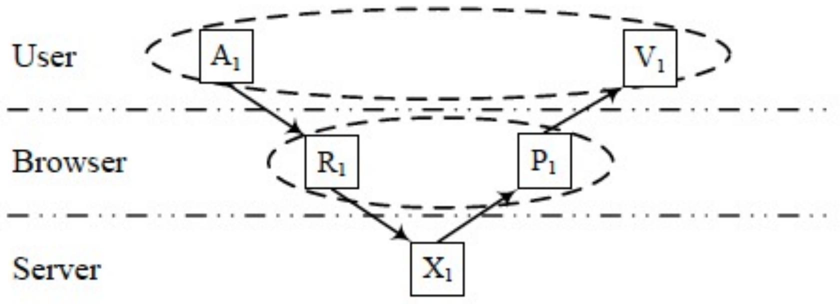
- Data race on an HTML element (the access of an element may occur before its creation)



```
<script type="text/javascript">
function show(emailTo,EmailC) {
  ...
  EmailC.value = emailTo;
  v = $get('dw');
  v.style.display = "block";
}
</script>

....

<a href=
"javascript:show('x@x.com', a)">
    Send Email
</a>
...

<div id=dw style="display:none">
  // HTML form to send email
</div>
```

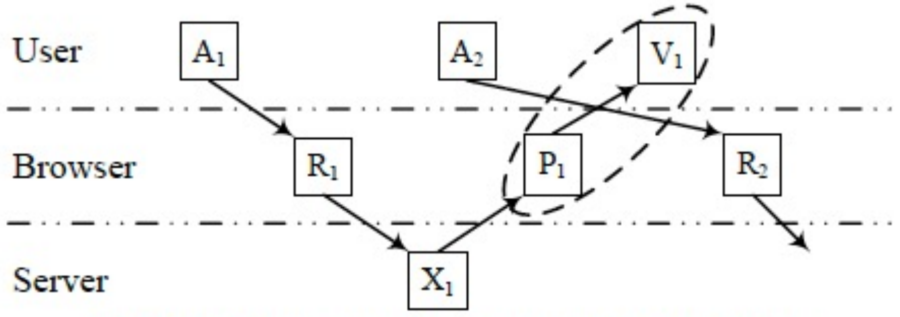
- More: Data races on functions, event dispatches

# Concurrency errors due to asynchrony: Atomicity violation



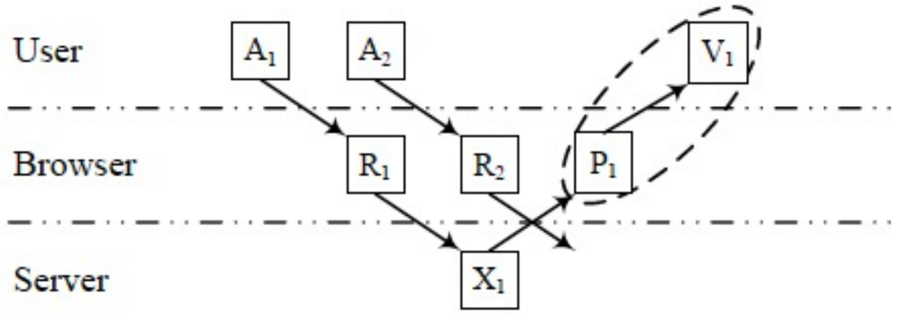
(a) Synchronous request

Atomicity between what the user triggers and what he sees



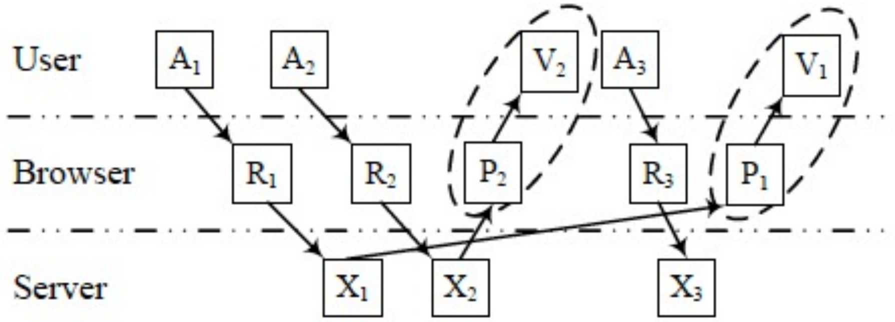
(b) What you operate on is not what you see

If P1 updates some data that is used by R2 , R2 is operates on the new data instead of what the user sees



(c) Atomicity violation

Assume an object is tested to be not null in R1. An interleaving R2 may et the object to null and lead to an exception in P1



(d) User confusion

When the user performs A3 according to what he sees, he might get confused and think that he is operating on the result of A1 .

## Some concurrency analysis works for single-threaded asynchrony *(tbc)*

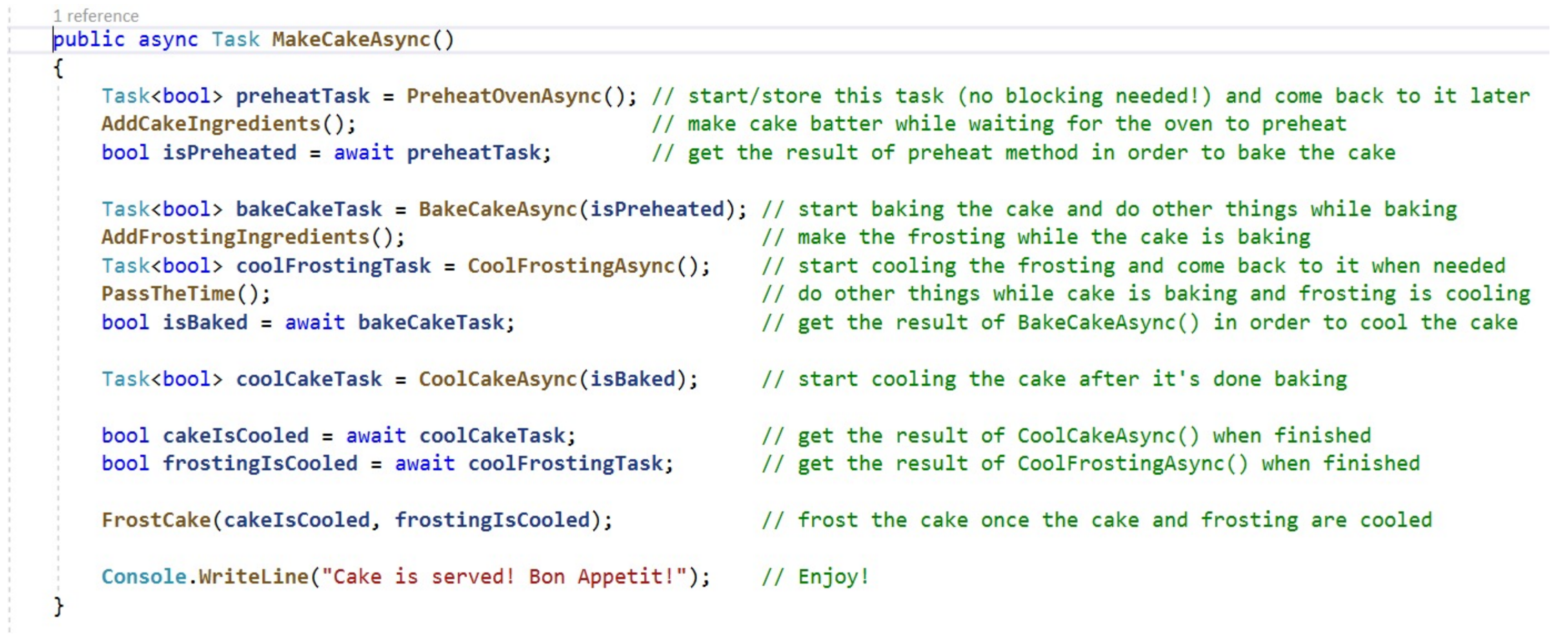
- Statically locating web application bugs caused by asynchronous calls. Y. Zheng, T., X. Zhang. WWW 2011
- Race detection for web applications B. Petrov, M. T. Vechev, M. Sridharan, J. Dolby. PLDI 2012
- I Know It When I See It: Observable Races in JavaScript Applications E. Mutlu, S. Tasiran, B. Livshits. Dyla@PLDI 2014
- A comprehensive study on real world concurrency bugs in Node.js. J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, J. Wei. ASE 2017
- Deferrability Analysis for JavaScript J. Kloos, R. Majumdar, F. McCabe. Haifa Verification Conference 2017
- Detecting atomicity violations for event-driven Node.js applications. X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, T. Huang. ICSE 2019
- Enabling Additional Parallelism in Asynchronous JavaScript Applications. E. Arteca, F. Tip, M. Schäfer. ECOOP 2021
- Race Detection for Event-Driven Node.js Applications. X. Chang, W. Dou, J. Wei, T. Huang, J. Xie, Y. Deng, J. Yang, J. Yang. ASE 2021.

and more …

## Multi-threaded asynchronous programming

## Multithreaded asynchronous programming

- Asynchronous method calls + multithreading
- Asynchronous calls run in parallel to the main thread



```
1 reference
public async Task MakeCakeAsync()
{
    Task<bool> preheatTask = PreheatOvenAsync(); // start/store this task (no blocking needed!) and come back to it later
    AddCakeIngredients();                        // make cake batter while waiting for the oven to preheat
    bool isPreheated = await preheatTask;        // get the result of preheat method in order to bake the cake

    Task<bool> bakeCakeTask = BakeCakeAsync(isPreheated); // start baking the cake and do other things while baking
    AddFrostingIngredients();                             // make the frosting while the cake is baking
    Task<bool> coolFrostingTask = CoolFrostingAsync();    // start cooling the frosting and come back to it when needed
    PassTheTime();                                        // do other things while cake is baking and frosting is cooling
    bool isBaked = await bakeCakeTask;                    // get the result of BakeCakeAsync() in order to cool the cake

    Task<bool> coolCakeTask = CoolCakeAsync(isBaked);     // start cooling the cake after it's done baking

    bool cakeIsCooled = await coolCakeTask;               // get the result of CoolCakeAsync() when finished
    bool frostingIsCooled = await coolFrostingTask;       // get the result of CoolFrostingAsync() when finished

    FrostCake(cakeIsCooled, frostingIsCooled);            // frost the cake once the cake and frosting are cooled

    Console.WriteLine("Cake is served! Bon Appetit!");    // Enjoy!
}
```

C# Async/Await: Asynchronous and multithreaded

## What does the following JS code print on the console?

```
setTimeout(() => {
    console.log('Here I am');
}, 2)

let i = 1;
while (i < 1000) {
    console.log(i);
    i++;
 }
```

Single-threaded asynchrony: Asynchronous task does not interleave with the caller task

## What does the following C# code print on the console?

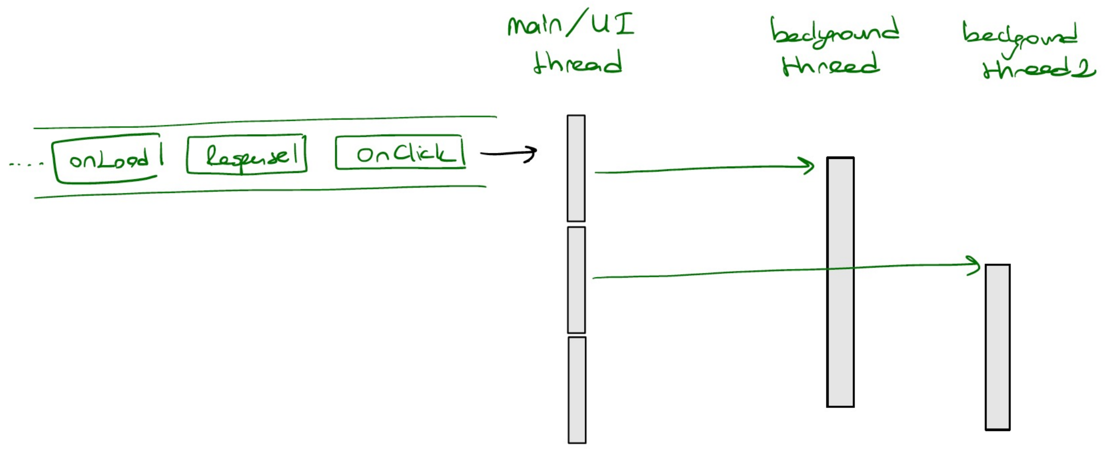
```
public static void Main()
{
  myFunction();
  int i = 1;
  while (i < 1000) {
    Console.WriteLine(i);
    i++;
  }
 }
```

```
async public static void myFunction() {
  await Task.Run(() => {
      Task.Delay(200).Wait();
      Console.WriteLine(" Log from async!");
    });
}
```

Multi-threaded asynchrony: Asynchronous task can interleave with the caller task

## Multithreaded asynchronous programming

- Inherits the challenges of multithreaded concurrency
  - + more complex execution flow



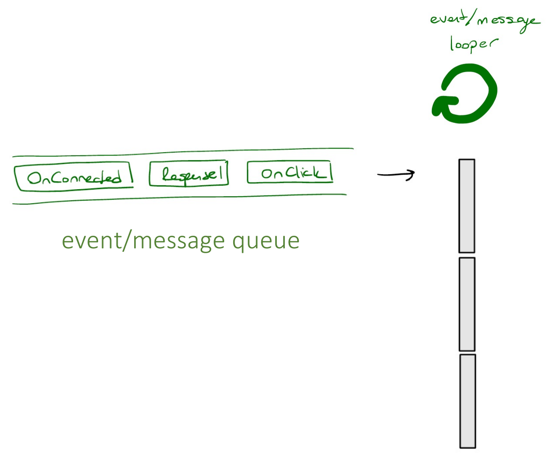
Prone to:

- Data races
- Order violation
- Atomicity violation
- Deadlocks

## Event-driven programming

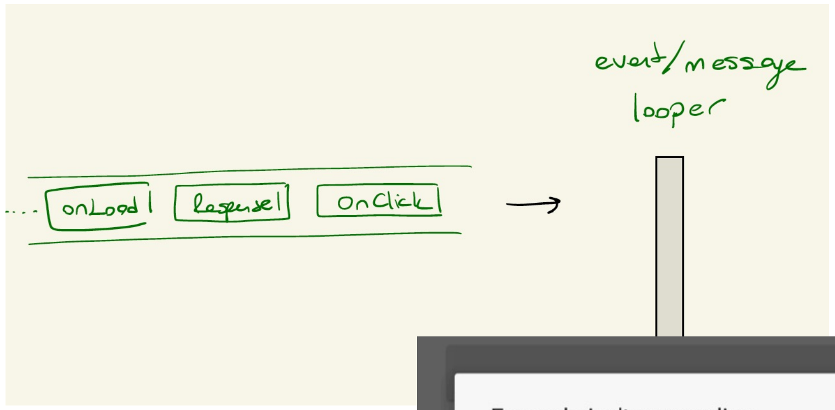
# Event-driven programming

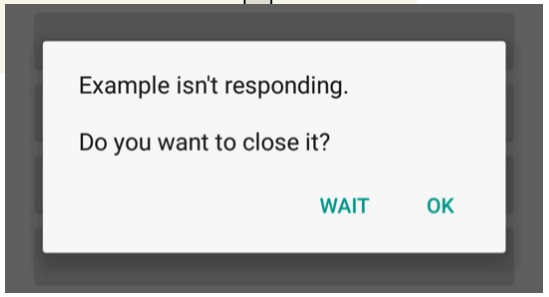
- The program is written as a set of event handlers
- It listens for events The flow of execution is determined by the invocation of the events
- Generally, a main loop that listens for events (e.g., user interactions, notifications) and then triggers the callback function of the invoked events
- Ex: Javascript web applications, mobile applications



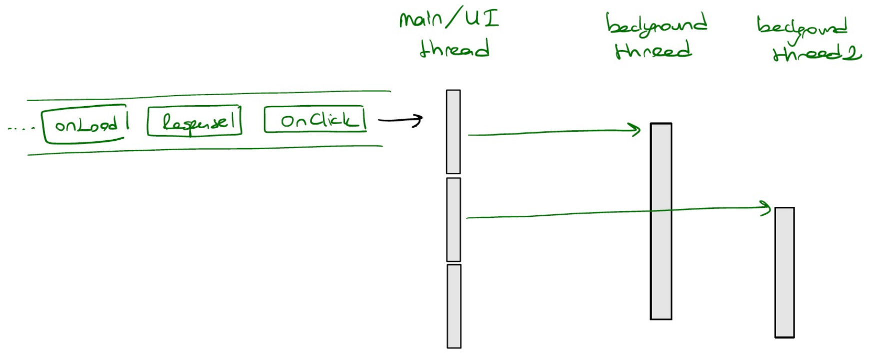
## Event-driven programming: Main thread + background threads

- Do not block the UI thread, asynchronously call IO operations, delegate long-running tasks to the background threads

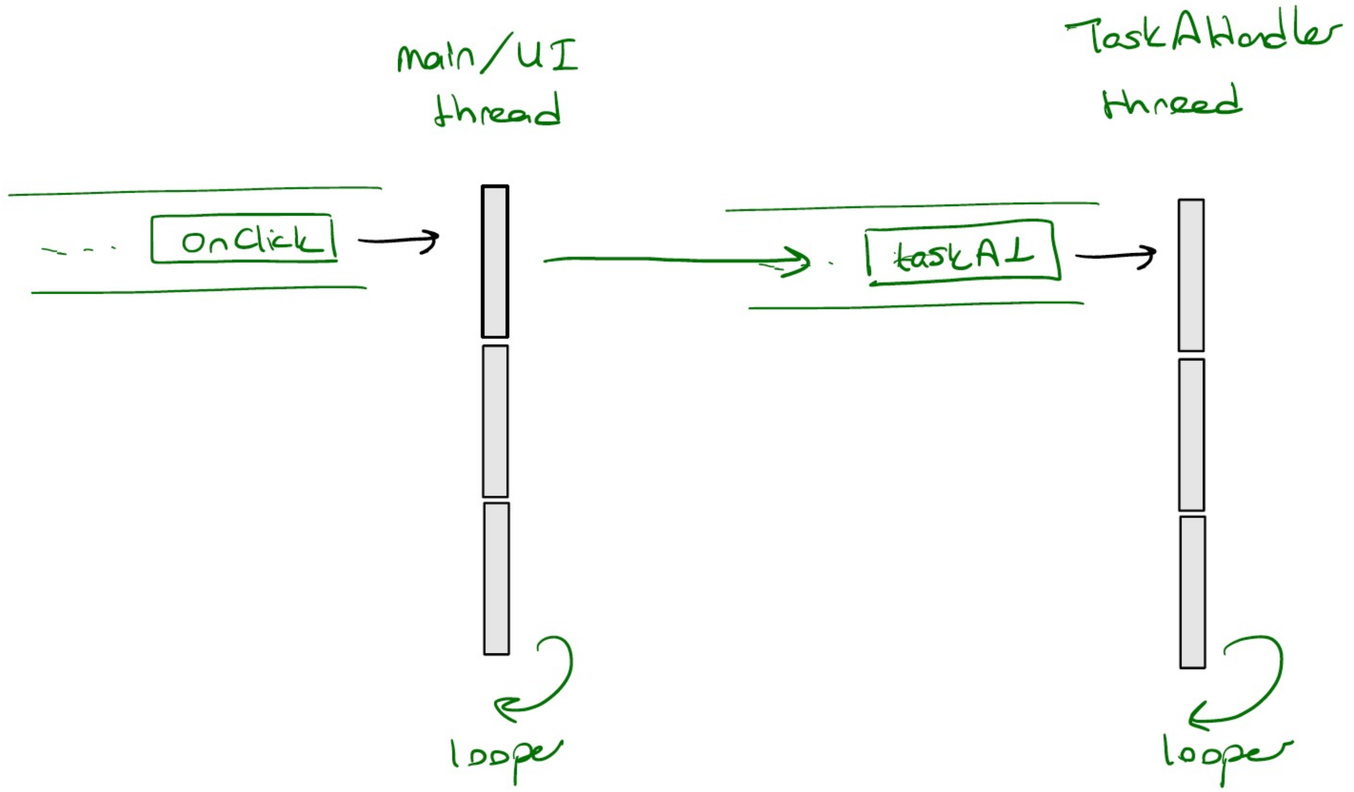



Example isn't responding.

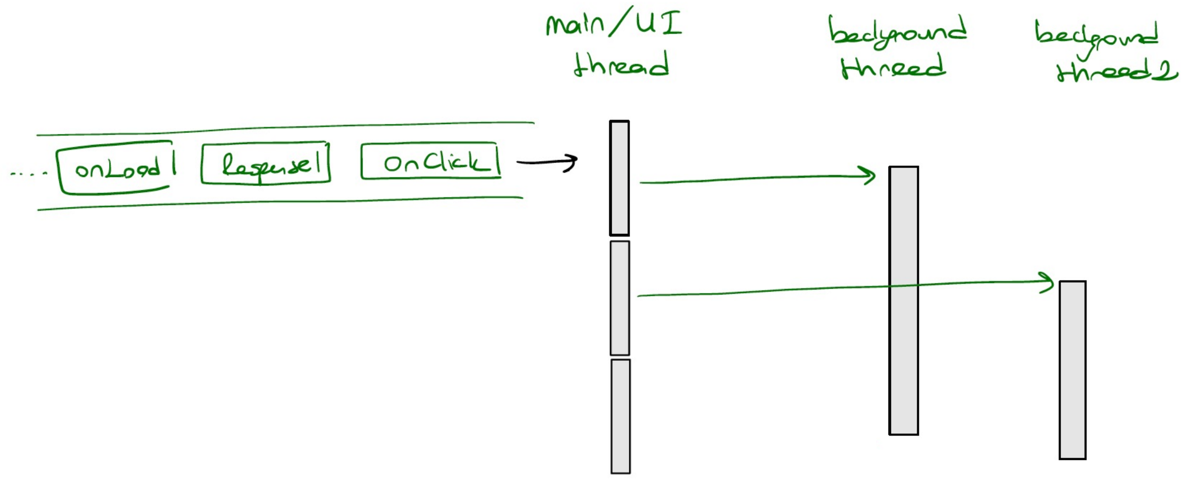
Do you want to close it?

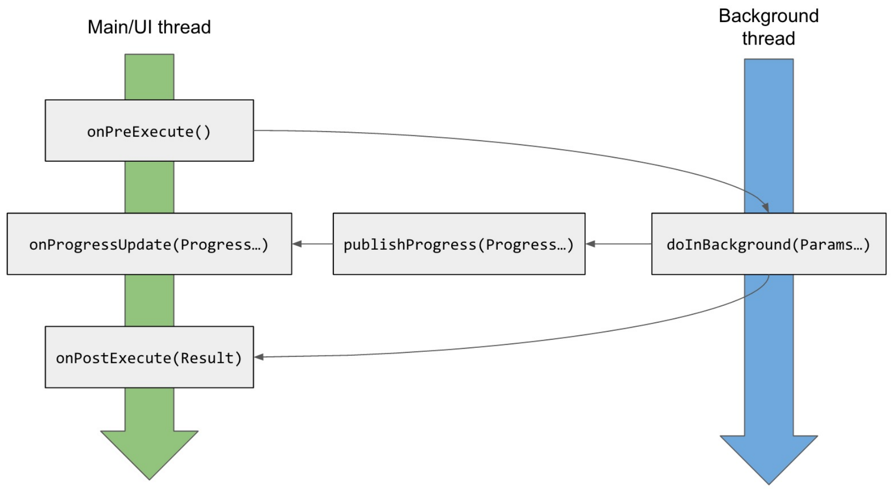


## Event-driven programming: Multiple looper threads



# Event-driven programming: Asynchronous task posting to UI thread





e.g. `AsyncTask` callbacks, or `onrunOnUIThread` method in Android API

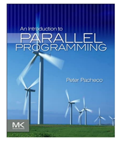
Image credit: https://androidkennel.org/

# Some concurrency analysis works for multi-threaded asynchrony *(tbc)*

- Race detection for Android applications. P. Maiya, A. Kanade, R. Majumdar. PLDI 2014:
- Race detection for event-driven mobile applications. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, J. Flinn. PLDI 2014:
- Retrofitting concurrency for Android applications through refactoring. Y. Lin, C. Radoi, D. Dig. SIGSOFT FSE 2014:
- Systematic Asynchrony Bug Exploration for Android Apps. B. Kulahcioglu Ozkan, M. Emmi, S. Tasiran. CAV 2015:
- Systematic testing of asynchronous reactive systems. Ankush Desai, Shaz Qadeer, Sanjit A. Seshia. ESEC/SIGSOFT FSE 2015
- Effectively Manifesting Concurrency Bugs in Android Apps. Q. Li, Y. Jiang, T. Gu, C. Xu, J. Ma, X. Ma, J. Lu. APSEC 2016:
- Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency. A. Bouajjani, M. Emmi, C. Enea, B. Kulahcioglu Ozkan, S. Tasiran. ESOP 2017:
- Static deadlock detection for asynchronous C# programs. A. Santhiar, A. Kanade. PLDI 2017:
- Static detection of event-based races in Android apps. Y. Hu, I. Neamtiu. ASPLOS 2018:

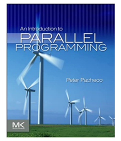
and more …

## Revisit: Many flavours of concurrency



- Concurrent programming
  - Multiple tasks can be in progress at any instant
- Parallel programming
  - Utilizing more than one processors for running the program
- Asynchronous programming
  - Programming with non-blocking requests/method calls
- Event-driven programming
  - The flow of the execution is determined by the (possibly concurrent) events
- Distributed programming
  - Multiple computers run as a single system

*Many authors consider shared-memory programs to be "parallel" and distributed-memory programs to be "distributed"*



https://www.reddit.com/r/ProgrammerHumor/comments/st360l/multi_mess/

## Takeaways

- Know your execution model
- Know your programming model
  - The semantics of concurrency constructs
- Do not reinvent the wheel if not necessary
- Beware of asynchrony and concurrency ☺