LUND UNIVERSITY

Concurrency and async

EDAF90 Web Programming

Per Andersson

Concurrency

- today concurrency is the key to performance
- parallellism is great for
  - speed
  - energy efficiency
- but adds challenges for the programmer
  - race conditions
  - locks and deadlock

Race Condition

- the program behaviour depends on timing
- dependant concurrent activities

```
let balance = 1000;
```

Lets do two transactions in parellel:

```
let tmp = balance;
\\ tmp is now 1000
tmp = tmp + 1000;
balance = tmp;
```

```
let tmp = balance;
\\ tmp is now 1000
tmp = tmp - 100;
balance = tmp;
```

`balance` is either 2000 or 900. One transaction is lost.

Strategies for Concurrency Problems

Ignorance is bliss:

- works surprisingly well
- conflicts are rare in many applications
- last write wins

Serialise access to shared data

- locks and transactions: traditional approach for threads
- single thread

Functional approach:

- pure functions
- immutable data structures

Locks

- to address this, locks was introduced
- *semaphores* is the commonly used low level lock
- only one thread can own the lock
- all other needs to wait
- not a problem for performance, only a tiny part of the program is serialized
- for this course: you only need to know what a lock is
- locks are covered in: EDAP10 - Concurrent Programming

Deadlock

- using locks can lead to deadlocks
- classical example: dining philosopers
- can only occur when:
  - a thread holds a resource while requesting another
  - there is a cycle in the dependency graph
- deadlock analysis is covered in: EDAP10 - Concurrent Programming

Pure Functions and Immutable Data Structures

Pure Function

- output only depends on input
- stateless

Immutable Data Structures

- data can not change over time
- ensure consistent data
- component state in react is based on this principle

Common in web frameworks, e.g. react component state

Polling and Busy Wait

- Polling:
  - ask for a resource without locking
- can be ok if it is unfrequent
- busy wait
  - repeatedly ask for a resource in a loop that do nothing else
  - kills performance
  - starves the other threads
  - nothing happens

JavaScript

JavaScript

- single threaded according to the specification
- the code runs from start to finish:
  - can not be interrupted
  - can not hand over execution to another function/thread
- advantages:
  - no need for locks
- but:
  - any longer computations blocks the entire application and GUI
  - you should break down your app into small functions
  - `alert()` is blocking.
  - asynchronous events can cause race conditions (read/write data from server)

Call Back Functions

Call Back Functions

- no need for polling
- are central to JavaScript programming
- many APIs are based on call backs
- most of the application code are the call back functions
- called when events occur
- you can not control the order in which your call back functions are called

Current Thread Loop

Current Thread Loop manage the execution of callback functions

- also called *Event Loop*
- a queue of functions to execute
- functions in the queue are executed one by one, in sequence
- remember, JavaScript is single threaded and execution can no be interrupted
- DOM and network events and JavaScript code will add new call back functions to the execution queue

SetTimeout

You can add your own functions to the execution queue:

- `const id = setTimeout(foo, delay, arg1, arg2, ...)`
- `clearTimeout(id)`
- `const id = setInterval(foo, period, arg1, arg2, ...)`
- `clearInterval(id)`
- **`this`** defaults to `global`
- use a timeout of 0 to add the function directly to the execution queue
- do this to break long computations

SetTimeout

```
class MyTimer {
  counter = 0;
  tick() {
    console.log(this.counter++);
  }
}
const obj = new MyTimer();
setTimeout(obj.tick, 1000);
```

Use `bind` if you use **`this`**

```
setInterval(obj.tick.bind(obj), 1000);
```

Execution Order

Using callback functions, it can be hard to follow the order of execution.

```
setTimeout(console.log, 1193, 'three');
setTimeout(console.log, 1058, 'four');
setTimeout(console.log, 1234, 'five');
setTimeout(console.log, 0, 'two');
console.log('one');
```

Pyramid of DOOM

Nesting callbacks are common, but leads to callback hell

```
fetchFile(url, function(error, file1) {
  if (error) {
    handleError(error);
  } else {
    fetchUrl(file1.nextUrl, function(error, file2) {
      if (error) {
        handleError(error);
      } else {
        fetchFile(file2.nextUrl, function(error, file3) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all files are loaded
          }
```

Promise

Promise

- wraps a value that eventually will be produced
- can have the states: *pending*, *fulfilled*, or *rejected*
- a promise is *settled* when *fulfilled* or *rejected*
- can have exactly one result or error

```
const promise1 = new Promise(resolutionFunction);
function resolutionFunction(resolve, reject) {
  ... when settled ...
  resolve(fulfilledValue);
     or
  reject(rejectedValue);
}
);
```

Example

```
const promise1 = new Promise(
  (resolve, reject) => setTimeout(resolve, 10*1000, 1))
);
const promise2 = new Promise(
  (resolve, reject) => reject('things went bad')
);
```

- the resolution function is executed directly
- commonly have asynchronous calls, `resolve()` will be called later
- replace callback functions in modern APIs

Reject

- **`throw`** an error in the resolution function is the same as calling reject
- recommendation: pass an `Error` object when rejecting a `Promise`
- remember: run time errors throws exceptions

```
const promise1 = new Promise(
  (resolve, reject) => {
    const ref = undefined;
    resolve(ref.field);
});
const promise2 = new Promise(
  (resolve, reject) => reject('things went bad')
);
```

.then

- you can not read the state, or the result directly
- `.then()` gives your the result when a promise is settled
- parameters are two callback functions:
  - success handler
  - reject handelr
- called asynchronously

```
myPromise.then(
  value => handleSuccess(value),
  errorValue => handleFailure(errorValue),
);
```

chaining

- `.then()` always returns a promise
- the promise wraps the return value of `then` handler:
  - a promise, a copy is passed on
  - other value is wrapped inside a new resolved promise
  - don not return anything, a resolved promise with the value **`undefined`**
  - throws an exception: the exception is wrapped into a rejected promise
- if a handler is missing, the promise passed down the chain

```
fetchFile(url).then(
  file1 => fetchFile(file1.nextUrl)
).then(
  file2 => fetchFile(file2.nextUrl)
).then(
  console.log,
  error => failedToLoadFiles(error)
);
```

chaining — closure

```
const allFiles = {};
fetchFile("one").then(
  file1 => {
    allFiles.file1 = file1;
    return fetchFile(file1.nextUrl);
  }
).then(
  file2 => {
    allFiles.file2 = file2;
    return fetchFile(file2.nextUrl)
  }
).then(
  file3 => {
    allFiles.file3 = file3;
    console.log(JSON.stringify(allFiles));
  },
  error => failedToLoadFiles(error)
);
```

chaining — pass on

```
fetchFile("one").then(
  file1 => fetchFile(file1.nextUrl)
             .then(file2 => ({file1, file2}))
).then(
  ({file1, file2}) => fetchFile(file2.nextUrl)
                      .then(file3 => ({file1, file2, file3}))
).then(
  allFiles => console.log(JSON.stringify(allFiles)),
  error => failedToLoadFiles(error)
);
```

catch, finally

- `.catch()` handler for rejected promises
- `.finally()`
  - handler has no parameter
  - pass on the input parameter

```
myPromise.finally(
  _ => { done = true; }
).catch(
 error => {
   bellyUp(error);
   return "default value";
 }
).then(console.log);
```

PromiseAll

Promise.all

- takes an iterable of promises as parameter
- return a promise that will settle when the input promises settles
- will contain all values of the fulfilled promises
- will be rejected as soon as any of the input promises is rejected

Parallell fetch

```
const promises = [
  fetchFile("one"),
  fetchFile("two"),
  fetchFile("three")
];
Promise.all(promises).then(
  console.log
)
```

Asynchronous Functions

async function `foo(){ /*body */}`

- the body starts to execute directly
- returns a `Promise` object, same semantic as `.then()`
- the execution continues with the code after the function call

```
async function one() {
  return 1;
}
const inc = async x => x=1;

one()
.then(inc)
.then(console.log);
```

await

- can only be used inside **`async`** functions
- waits for a promise to settle
- returns the resolved value
- If the promise is rejected, the await expression throws the rejected value.

```
async function fetchAll(url) {
  file1 = await fetchFile(url),
  file2 = await fetchFile(file1.nextUrl),
  file3 = await fetchFile(file2.nextUrl)
  return {file1, file2, file3};
}
fetchAll("one")
.then(console.log);
```