JavaScript

ECMA

Bjarte Wang-Kileng

HVL

August 29, 2025

Outline

Outline

Blocking and non-blocking operations

- All JavaScript code is run in the same thread.
    - If JavaScript hangs, page will not respond. JavaScript code is blocking.
- I/O operations are non-blocking in the browser.
    - Data can be fetched in parallel with Ajax.
    - JavaScript files can be opened in parallel.

Threads and JavaScript

- Older browsers shared one thread between all windows and panes.
  - If code from one web page blocked, all panes and windows would freeze.
- Newer browsers use separate threads for different windows and panes.
- Multiple JavaScript threads are possible through *WebWorkers*, introduced in 2009, and *WebAssembly*.
  - WebAssembly threads are mapped to WebWorker threads.

Outline

WebWorker example

- Main thread:

```
if (window.Worker) {
    const myWorker = new Worker('./worker.js');

    // Sending message to worker
    myWorker.postMessage("Message to worker");

    // Callback to run on message from worker
    myWorker.addEventListener('message',
        (e) => {console.log(e.data)}
    );
} else {
    console.log('Your browser does not support web workers.');
}
```

- Worker thread:

```
// Callback in Worker to run on message to Worker
self.addEventListener('message', (e) => {
    console.log(`Got message: ${e.data}`);

    // Sending message to main thread
    self.postMessage('Mesage to main thread');
})
```

Modules and Workers

- Worker created as module can import modules:

```
const worker = new Worker('./worker.js', { type: 'module' });
```

- Require a rather modern browser.

Sharing data between threads

- Through messages.
  - Between main thread and web workers.
- Shared memory.
  - Between main thread and workers.
  - Between workers.
- Main thread can act as mediator of messages between workers.
  - Main thread receives and forward message.

Using shared memory

Not subject for the exam

- **SharedArrayBuffer** is a fixed-length raw binary data buffer.
- Buffer can be shared between workers and main thread.
- Create buffer and share with WebWorker:

```
// Create a 16 byte data buffer
const buffer = new SharedArrayBuffer(16);

// Share buffer with worker
worker.postMessage(buffer);
```

- Store a two byte unsigned integer at index 2 in buffer:

```
const view = new DataView(buffer);
view.setUint16(2,65535); // Max unsigned 16-bit integer
```

- Using view index to add two byte unsigned integer:

```
const view = new Uint16Array(buffer);
view[3] = 65535;  // View index 3 is index 3*2=6 of buffer
```

Working with **SharedArrayBuffer**

Not subject for the exam

- Support disabled January 2018 due to the Spectre vulnerability.
- Re-enabled in 2020 in most browsers using a new secure standard.
- Document must be in a secure context:
  - Webserver must set the COOP and COEP response headers.

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

  - Unless URL to localhost, must use https.

- Check if browser is in a secure context:

```
if (crossOriginIsolated) {
    const buffer = new SharedArrayBuffer(16);
}
```

Working with shared data

Not subject for the exam

- Modifications of shared data is a critical region (DAT103).
- Atomics object has methods for atomic operations and synchronization.

Examples of atomic operations

Not subject for the exam

- Add *number* to value stored at *index* in *view*, using a critical region:

```
const before_value = Atomics.add(view, index, number);
```

- Replace value at *index* with *number*, using a critical region:

```
const replaced_value = Atomics.replace(view, index, number);
```

- Store *number* at position *index*, using a critical region:

```
const stored_value = Atomics.store(view, index, number);
```

- Fetch value at position *index*, using a critical region:

```
const value = Atomics.load(view, index);
```

Atomics blocking construct

Not subject for the exam

- Wait if value at *index* is *value*.

```
const status = Atomics.wait(view,index,value);
```

  - Changing the number at *index* will **not** wake already waiting workers.
  - Main thread can not be put on hold.
  - Return value tells if number at *index* is *value* at wake up.

- Wake up workers put on wait:
  - Tell new arriving worker not to wait by changing value at *index*:

```
Atomics.store(view,index,value-1);
```

  - Wake up workers that wait due to value at *index*:

```
const woken_workers = Atomics.notify(view,index);
console.log(`Number of woken workers: ${woken_workers}`);
```

DOM and Workers

- Only the main JavaScript thread can access the DOM.
- WebWorkers can handle Ajax and do calculations, but only the main thread can update the HTML code.

Outline

Asynchronous code

- Asynchronous code is independent of the main program flow.
- Asynchronous code is run when an event occurs.
  - A button was clicked.
  - Some text was entered in an input element.
  - A file has finished to load.
  - A server request got a response.

Asynchronous code does not use threads

- Asynchronous code has nothing to do with threads!
- Asynchronous code is run by the thread when idle.
- Blocking asynchronous code will block the thread.
- The main thread, and WebWorkers can include asynchronous code.
- Asynchronous code does not start new thread.
  - Code is run by the owner thread.
  - Asynchronous code can of course create WebWorkers to do work.

Asynchronous JavaScript

- A function can be attached to an event, a *callback*.
  - The callbacks are examples of asynchronous code.
  - When the event occur, the callback is run by the browser.
- The browser does not block while waiting for the event.
  - Web applications become more responsive.
- Asynchronous code is (usually) run by the main JavaScript thread.
  - A callback that hangs will block all JavaScript, except WebWorkers.
- Next waiting callback is run when the main JavaScript thread is idle.

Asynchronous operations in JavaScript

- Script tag with `async`, `defer` or `type='module'`.
  - Code is downloaded in separate I/O thread.
  - Code is run by the main JavaScript thread.
- Event handlers are run when event occur.
  - Ajax data is downloaded in separate I/O thread.
  - Handling received data is done by thread that sent the request.
- Promises.
  - Promises has nothing to do with threads!
  - All promise code is managed by the thread that created the Promise.
- Writing to the browser console:

```
console.log("Writing to the browser console");
```

Note on scripts with defer (or `type='module'`)

- Run in the order they occur.
- Run after the DOM is built.
- Page loading is fastest if script tag in head, using *defer*, `type='module'` or *async*.

Promises

- Much new JavaScript features are implemented using promises, e.g. the **Fetch API** for Ajax, and the *import* function for dynamic imports.
- Promises allow us to run code and continue immediately before the answer is ready.
- When the answer is ready, a callback can be run.
  - Asynchronous code.
- A useful promise is resolved or rejected by asynchronous code.
- Observe that all code is run by the same JavaScript thread.

Promise example

```
const myPromise = new Promise(
    (resolve,reject) => {
        // JavaScript code to calculate OK, and result or error
        if (OK) {
            resolve(result);
        } else {
            reject(error);
        }
    }
)

myPromise.then(
    (result) => {console.log(result)}
).catch(
    (error) =>  {console.log(error)}
);
```

- The above promise resolves or rejects synchronously.
- A rather useless promise!

A meaningful promise example

```
const myPromise = new Promise(
    (resolve,reject) => {
        someElement.addEventListener("some-event",(e) => {
            // JavaScript code to calculate OK, and result or error
            if (OK) {
                resolve(result);
            } else {
                reject(error);
            }
        });
    }
)

myPromise.then(
    (result) => {console.log(result)}
).catch(
    (error) =>  {console.log(error)}
);
```

- The above promise resolves and rejects on an event.
- The promise is resolved and rejected by asynchronous code.

Using promises

- Constructor is given a function with two arguments.
- The first argument is a function for returning a successful answer.
  - This function is created and supplied by the browser.
- The second argument is a function to inform about errors.
  - This function is created and supplied by the browser.
- The first use of *resolve* or *reject* will determine the value.
  - Unlike events like **click**, a promise “happens” only once.
  - Observe though that code after *resolve* and *reject* is run.
- A syntax with *async* and *await* can make the code easier to read.
  - The only possible return value from an *async* function is a promise.

Demo with async and await

```
function myPromise(input) {
    return new Promise(
        (resolve,reject) => {
            someElement.addEventListener("some-event",(e) => {
                // JavaScript here must calculate OK, and result or error
                if (OK) {
                    resolve(result);
                } else {
                    reject(error);
                }
            }
        }
    )
}

async function usePromise(value) {
    try {
        const result = await myPromise(value);
        console.log(result);
    }  catch (error) {
        console.log(error);
    }
}
```

When to use promises

- Many new JavaScript features return promises.
- Usually, we do not create promises ourselves.
- Only useful if *reject* and *resolve* are run by asynchronous code.
  - Resolve when file is loaded.
  - Resolve on response from Ajax request.
  - Resolve when event occur.
  - Resolve on message from WebWorker.
- Usually not useful for encapsulating DOM events, as Promise only resolves once.

A comment on the browser console

- The code below may not work as expected:

```
console.log("Write something");
console.log("Write something more");
doOtherStuff();
```

- The first use of the *log* will write and occupy the browser console.
- The second use of *log* is put on wait as the console is busy.
  - Logging to the console is an asynchronous operation.
- "Write something more" can appear after *doOtherStuff* has finished.