FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

**Fachhochschule Aachen**
**Campus Jülich**

# Parallelism in JavaScript

## Current state and limitations in the communication between threads

**Bachelorthesis**

**Thorben Below**
**Matr. No.: 3255003**

**Faculty of**
**Medical Engineering and Applied Mathematics**
**Applied Mathematics and Computer Science BASc**

**Aachen, August 2022**

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

My name _____

signature

Diese Arbeit wurde betreut von:

1. Prüfer: Prof. Dr. Karola Merkel
2. Prüfer: Dipl.-Inf Stephan Oehlert

# Contents

# 1 Introduction

## 1.1 Context and motivation

Moore's Law states that the number of transistors doubles about every two years, though the cost of computers is halved [1]. While this has been true for a long time, semiconductor advancement has slowed industry-wide since around 2010. Looking at the development of CPU architectures over the last decades in general shows that the pace of imporvement of parameters that increase the compute power of single cores, like clock speed and TODO, is in decline [1]. On the other hand the number of cores in the average CPU has increased significantly since the Introduction of the first multi-core CPU and is steadily rising [1]. It follows that the importance for programming languages to be able to make us of these additional cores and software that does so also increases.

JavaScript is best-known as the scripting language for Web pages, although it is also supported by many non-browser environments like Node.js or Deno. It is lightweight, interpreted or just-in-time compiled, prototype-based and features first-class functions [2]. Because of the event-based nature of JS, concurrency is innate to the language. It's achieved through the use of Callbacks and more recently with promises and async/await. This form of concurrency can solve a lot of problems that would otherwise end in down time like waiting for I/O or network requests. While this enables running different tasks concurrent instead of strictly sequentualy, the execution is still limited to one thread. Other use cases such as ones that require a lot of CPU power, or ones that require a lot of memory, don't benefit from this approach. Some of these were solved individually by the runtime environments like browsers by introducing special APIs, but for a lot of the needs of software needs this does not suffice. These use cases are best handled by parallelism and the use of multiple threads.

In the past, the platforms that JavaScript ran on did not provide any thread support, so the language was thought of as single-threaded. While the ECMAScript specification has no concept of threads, it specifies realms, execution context and its memory management. The implementation of threads and the corresponding APIs is left to the runtime environments.

In the browser this concept has been in use for a long time e. g. for seperating execution contexts of different tabs. With the introduction of web, shared and service workers in the browser and worker threads in nodejs and deno, there are also APIs for developers to achieve real multithreading in all major environments. These differ in their indivdual options and have different use cases. While they have been around for a long time now, the usage of these possibilities is still quite low and not well understood by a lot of developers. This is also visible in the comparatively low number of available documentation and examples

compared to other features of the language.

Communication between Threads or JavaScript contexts is done via message channels and the postMessage API. While this API allows sending JSON Objects and some builtin data types, it also has strict limitations when it comes to other objects. It is not possible to send Function Objects or preserve prototype information.[2] This greatly impacts the way multi-threaded programs can be designed, especially for libraries (e. g. using polymorphy (OOP) and compositon (FP) across thread bounds becomes non trivial).

## 1.2 Research question and scope

The goal of this thesis is to explore the available options and the reasoning behind the restrictions regarding the communication between contexts in general and explain how the underlying mechanism work.

It starts with an introduction into the relevant concepts underlying processes, threads and interprocess communication on the operating system layer. After that a short overview of JavaScript and its core concepts is given, with a more in detail explanation on the more important ones. To understand the relation between the os layer and JavaScript code, the usual compiling process will als be looked at with the V8 JavaScript Engine serving as an example. With the basics covered, the current possibilities and restrictions in interprocess communication in JavaScript will be established.

Subsequently, Function Objects (and the corresponding subtypes) will be considered in particular. In addition prototypically experiment(?) which dependencies and restrictions arise here with the transmission over thread bounds. (e. g. serialize/deserialze necessary?).

# 2 Concurrency

## 2.1 Processes

## 2.2 Threads

## 2.3 Interprocess Communication

# 3 JavaScript

JavaScript (JS) is a general purpose programming language that is best-known as the scripting language for the Web and is defined by the ECMAScript Specification. More precisely by the ECMAScript Language Specification (ECMA-262) and the ECMAScript Internationalization API Specification (ECMA-402) [2].

## 3.1 Core concepts

JavaScripts core concepts include being prototype-based, supporting multiple paradigms such as object-oriented and functional programming and having higher order functions [2]. The following sections give a brief overview over the most important core concepts.

### 3.1.1 Inheritance and the prototype chain

Some info about Inheritance

### 3.1.2 Higher order functions

### 3.1.3 Closures

### 3.1.4 Strict mode

### 3.1.5 Event Loop

### 3.1.6 Memory management

## 3.2 Compilation

# 4 Current state of parallelism in JavaScript

ECMAScript itself has no notion of Threads. It only has the concept of Realms and execution contexts. Instead the Engines and envrionments that JavaScript is executed in provide methods to create new execution contexts (Threads) and interact with them.

## 4.1 Message channels

## 4.2 Transferables

## 4.3 Shared memory and Atomics

## 4.4 Differences in server and client side JavaScript

In the Browser these are different kind of Workers. Currently there are Web, Shared and Service workers. The Web Worker is akin to a ”normal” Thread. It lives only as long as its parent Context and can only communicate with it and contexts that are related to the parent context.
The Shared Worker allows communication with multiple ”Parent” contexts. It lives as long as atleast one parent context is still alive.
The Service Worker can persist even after the parent context is already closed. It can communicate with all other contexts that have the same origin.
nodejs and deno provide Worker Threads. Theses behave similar to threads known in other programming languages.

## 4.5 Open proposals

# 5 Restrictions in interprocess communication

## 5.1 Functions and methods

## 5.2 Class information

# 6 Discussion

# 7 Outlook

## References

[1] todo. *todo*. todo. 2020.

[2] MDN. *JavaScript*. en-US. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 06/27/2022).

## List of abbreviations

| | |
|---|---|
| $JS$ | JavaScript |

# List of Figures

# List of Tables

## Danksagung

Wenn Du hier auf Deutsch schreiben möchtest, wird durch den Befehl ’selectlanuage’ die deutsche Rechtschreibung bzw. Silbentrennung aufgerufen.

# Appendix