JavaScript-only Parallel Programming of Embedded Systems

Péter Gál

Abstract: The number of Internet-connected small embedded devices are increasing rapidly. Traditionally, such devices are programmed in some kind of compiled language, e.g., in C or C++. However, the number of embedded developers is small in contrast to JavaScript developers. Fortunately, there are multiple projects, which are providing JavaScript engines for such small devices and allow JavaScript developers to also program embedded devices. Nowadays, the embedded devices are not necessarily single-core systems anymore but can have multiple cores. For multi-core CPUs, the way to execute tasks or processes in parallel is a must. However, the JavaScript language itself does not have any thread, task, or parallelism concepts.

In this work, a way to run JavaScript scripts in parallel on embedded systems is presented. The proposed implementation adopts an already existing W3C standard as starting point to provide a familiar API for Web developers, and it is extended with properties required for embedded devices. By using the Worker concept, the embedded JavaScript programmer can leverage multi-core systems and also execute various tasks in separate contexts which do not block the main JavaScript execution.

Keywords: Embedded systems, JavaScript, Parallel programming, Workers

Introduction

The popularity of JavaScript is ever growing since it is was first used on the Web. Stack Overflow reports that at least 60% percent of developers are using JavaScript [7]. The most popular language on GitHub is also JavaScript [6]. In contrast, there are far less embedded developers reported by these surveys. However, the increasing number of Internet-connected devices in the Internet of Things movement results in a requirement for more embedded programmers. By enabling the use of JavaScript in embedded software or software components, JavaScript developers can contribute to the development of IoT devices.

Another interesting point in embedded systems is that there are more and more multi-core CPUs present in the embedded hardware segment. Using various parallel programming techniques, the embedded applications could benefit from the multi-core CPUs. Usually the concept of tasks or threads is available in embedded operating systems. FreeRTOS uses the task concept where a method can be started as a task and the OS schedules the execution of all tasks [2]. The ARM mbed OS uses the threading concept to enable parallel execution [3]. Currently, in the JavaScript language specification, there is no task or thread concept which would allow parallel execution.

In this paper, the current state of JavaScript parallel execution is investigated and a solution is presented for embedded devices.

Background

Running JavaScript on embedded devices is not an impossible task. There are already multiple engines designed for small devices, e.g., Espruino [1], Duktape [8], and JerryScript [5]. Each of these engines are created to have small memory footprint (both ROM and RAM) and to be as ECMAScript 5.1 compatible as possible.

The ECMAScript 2015 Language Specification introduces the Promise API for JavaScript [4]. The Promise objects represent a series of asynchronously invoked JavaScript functions, each of which is enqueued as job which needs to be executed by the JavaScript engine's event loop. As these functions are executed in a serial fashion by the event loop, two Promise functions cannot actually run in parallel. This means that this approach can not achieve true parallelism. The Promise object is available in modern browsers and also in the Node.js framework.

In Node.js, there is a possibility to execute JavaScript code in a child process. This is achieved by exposing the exec/frok mechanism of the OS to the JavaScript runtime. Child processes can communicate with their parent process via IPC channels via message passing. Using the child processes on non-embedded computers, parallelism can be achieved as each process runs separately. However, on

embedded devices it is highly possible that the OS does not support the creation of child processes. FreeRTOS and Mbed OS are such operating systems.

Fortunately the World Wide Web Consortium (W3C) have created a standard to allow executing JavaScript code on a separate thread. This is the Web Workers specification [9], which defines an API that allows web application developers to spawn background workers running JavaScript in parallel to their main page. Workers have no direct access to the main page contents but have means of communication via message passing. Values and objects sent in the messages are copied and not shared, therefore there are no synchronization issues while accessing data from multiple threads.

This Web Worker approach is a good base concept for the parallel execution of JavaScript code on embedded devices as it can be mapped to the task or thread concept of embedded OSs. For example, each new Worker can be expressed as the creation of a task and running the required code in that new context. Moreover, the Web Worker API is a technique well-known to JavaScript developers familiar with web-related programming, thus it enables the transfer of their knowledge to the embedded domain. This solution is presented in more detail in the following section.

Workers for Embedded

The Web Worker standard describes the `Worker` JavaScript objects which can be used to initialize and execute additional JavaScript contexts. To instantiate a Web Worker, only the script source is required as its first argument. At the time of the creation of the new context the Worker will automatically start. This can be observed in the line 1 of Listing 1. To send data to a newly created Web Worker, the worker object's `postMessage` method can be used, as depicted on line 5. The method serializes its parameter data and sends it to the worker's context. As the data is serialized, each JavaScript context will have a copy of the value. Only the `postMessage` method can be used to send data to the Web Workers. Additionally, the `postMessage` method in the Web Worker's context can send data to the parent context. Messages sent via the `postMessage` can be handled by implementing the `onmessage` method, both on the main script's worker object and in the new worker's context. The message handler method receives an `event` argument which describes message received. In this `event` object, the `data` property stores the content sent by the `postMessage` method. The data received this way is deserialized and can be treated as a normal JavaScript object or value. An example on how a message can be received from a worker is presented on lines 2 and 3 in Listing 1. It is important to note that the concrete serialization and deserialization mechanism is not mentioned in the standard but is left as an implementation-defined process instead. A Web Worker will continue to wait for messages till the parent context invokes the `terminate` method on the worker object. This call is omitted from the examples.

These Web Worker mechanisms can be mapped to traditional embedded programming techniques. The creation of worker is the task creation. Communication via the `postMessage` and `onmessage` methods is similar to message queue operations in embedded devices. This is the main reason why the Web Worker concept is ideal for the embedded JavaScript world. Additionally it is important to mention that the Worker context is a separate one which does not block the main context.

```
1 var work = new Worker('script.js');
2 work.onmessage = function(event) {
3  // event.data
4 }
5 work.postMessage(10);
```

Listing 1: Example Web Worker initialization

Proof-of-Concept Implementation

A proof-of-concept implementation was created using the JerryScript [5] JavaScript engine and the FreeRTOS system [2]. For each new Worker created in JavaScript, an underlying FreeRTOS task is created. At the construction state, the script which should be executed will be associated with the worker object. As embedded systems do not necessarily have access to file systems, the filename of the given JavaScript filename will be resolved internally using a filename-source code mapping generated during the compilation of the embedded software.

After the new Worker object is created in the current JavaScript context, it is ready to be started. In the implementation, the Worker can be started with the `start` method. Upon starting the Worker the underlying task will first create a new JavaScript context to separate all methods and variables from the main JavaScript context. After this, the Worker's JavaScript code will be executed and the task enters an event loop state waiting for messages from the main JS context and from the worker context.

The underlying communication between the tasks in FreeRTOS is created via Queues. If the parent JavaScript context sends a message using the `postMessage` method to the child worker, underlying implementation sends a message to the task's own Queue. On any new message to the Queue, the worker's event loop wakes up and processes the entry on the Queue by sending the message to the `onmessage` event handler method. After handling the message, the event loop enters into a sleeping state waiting for furher messages. The message from the child worker to the parent one is done in the similar fashion.

An example main JavaScript and a worker script is shown in Listings 2 and 3. In the example, the `job.js` file is responsible for calculating the sum of the first `N` integer numbers. The `N` is provided for the script from the parent JavaScript context as an event message, as the line 2 illustrates. An input value, that is the `N` number, for the calculation can be observed on line 7 in Listing 2. After the sum is computed, the result is sent back from the worker on line 7 of Listing 3.

```
1 var work = new Worker('job.js', 120);
2 work.onmessage = function(event)
3 {
4  // event.data
5 }
6 work.start();
7 work.postMessage(10);
8 work.postMessage(30);
```

Listing 2: Main JavaScript

```
1 onmessage = function(event) {
2  var sum = 0, N = event.data;
3  for (var i = 0; i < N; i++)
4  {
5   sum += i;
6  }
7  postMessage(sum);
8 }
```

Listing 3: An example job.js JavaScript

Embedded Extensions for Workers

The API of the Web Workers provide minimalistic fine tuning options for a developer. However, more fine tuning is present for task executions in embedded systems. One of the more important things is the specification of the task stack size. In the embedded Worker approach, this information can be added via an additional parameter to the constructor call. This can be observed on line 1 in Listing 2 where the `Worker` function is called with the second parameter representing the allowed stack size for the new Worker task.

Another extension added to the original API is the introduction of the `start` method for worker objects. Using this method the start of a created Worker can be explicitly controlled. The start-up can be delayed if needed. This can be useful if the developer creates all required Worker objects first, then connects the message handlers, and finally starts up the required tasks.

Summary and Future Work

This paper presents a way to execute multiple JavaScripts in parallel on small embedded devices by reusing and extending an API which is already familiar to JavaScript developers: (Web) Workers. The Worker-based solution allows communication via message passing between runtime contexts. Using this approach, JavaScript developers can exploit the benefits of multi-core embedded systems. By running JavaScript on multi-core embedded system rapid prototyping is also encouraged.

There are still other Web Worker-based aspects to explore. For example, the Web Worker W3C standard describes Shared Workers which enables multiple connections between workers. Also, as the communication channels are always copying data between the contexts, the memory usage can increase rapidly, which is usually something to be avoided on embedded systems that typically have very limited memory on-board. By improving the way how messages are passed, e.g., by using shared memory, the implementation could reduce the memory impact and even potentially provide application speed-up.

Acknowledgment

This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled “Internet of Living Things”.

References

[1] Espruino. https://www.espruino.com/.

[2] Freertos. http://www.freertos.org.

[3] ARM. Mbed os. https://www.mbed.com/en/platform/mbed-os/.

[4] Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015.

[5] JS Foundation. Jerryscript. http://jerryscript.net/.

[6] GitHub. The state of the octoverse 2017. https://octoverse.github.com/, 2017.

[7] Stack Overflow. Developer survey results. https://insights.stackoverflow.com/survey/2017, 2017.

[8] Sami Vaarala. Duktape. http://duktape.org/.

[9] W3C. Web workers. W3c candidate recommendation, W3C, May 2012.