RT.js

Practical Real-Time Scheduling for Web Applications

Christian Dietrich, Stefan Naumann, Robin Thrift, Daniel Lohmann
Leibniz University Hanover
GitHub. The State of the Octoverse, 2019
OVERALL SUMMARY OF PROGRAMMING LANGUAGES

Which of the following programming languages, if any, do you use to build IoT solutions?

Eclipse IoT Working Group. IoT Developer Survey Results, 2018
IoT - Internet of Things

- on browser as frontend-language
- on Node.js as backend-language
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion

Button Press Handler
Background Job 2
Background Job 1
Renderer

Next Frame

Deadline miss!

Stefan Naumann
RT.js
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion

Button Press Handler
Background Job 2
Background Job 1
Renderer

Next Frame
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion

Button Press Handler

Background Job 2

Background Job 1

Renderer

Next Frame

Deadline miss!
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion

Button Press Handler
Background Job 2
Background Job 1
Renderer

Next Frame

Deadline miss!

Stefan Naumann
RT.js
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion

Button Press Handler

Background Job 2

Background Job 1

Renderer

Job Queue

BtnPrs

Executor

Next Frame

Deadline miss!
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
Scheduling in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion

Button Press Handler
Background Job 2
Background Job 1
Renderer

Next Frame
Deadline miss!
Problem 1: Long-running tasks block the executor

Long-running computations result in a reduced page-rendering frequency.
No Prioritization in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
- no prioritization possible
No Prioritization in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
- no prioritization possible
No Prioritization in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
- no prioritization possible
No Prioritization in the JavaScript Executor

- single-threaded
- first-come-first-served
- run-to-completion
- no priorization possible

Button Press Handler
Background Job 2
Background Job 1
Renderer

next period
Problems in the JavaScript Execution Model

Problem 1: Long-running tasks block the executor
Long-running computations result in a reduced page-rendering frequency.

Problem 2: No prioritization of tasks
An application cannot schedule its jobs according to their relative importance.
How would JavaScript solve it?

- asynchronous programming style discourages long running tasks
  - harder to read and maintain → Callback Hell
  - not suitable in all cases

```javascript
function cb () {
    console.log("Timer fired");
}
document.setTimeout ( cb, 4000 );
```

---

3 K. Gallaba, A. Mesbah, and I. Beschastnikh. *Don't call us, we'll call you: Characterizing callbacks in javascript.*
In *ESEM'15*, pages 1–10, Oct 2015
Proposing RT.js

- .js Code
- Transpiler
- .js Code
- +
- Runtime Library (incl. Scheduler)
- Upload

.js Code
with Premption Points
Proposing RT.js

.js Code ➔ Transpiler ➔ .js Code with Premption Points ➔ Runtime Library (incl. Scheduler) ➔ Upload

Stefan Naumann
Generators as Preemptable Jobs

- Generator: Function, returns (yield) and keeps state
- Compare: Iterator
Generators as Preemptable Jobs

- Generator: Function, returns \( \text{yield} \) and keeps state
- Compare: Iterator
Generators as Preemptable Jobs

- Generator: Function, returns \texttt{(yield)} and keeps state
- Compare: Iterator

\begin{center}
\begin{tikzpicture}
  \node[draw] (T1) {$*\text{foo()}$};
  \node[draw, below of=T1] (yield) {\texttt{yield}};
  \path[->] (T1) edge node {\texttt{yield}} (yield);
\end{tikzpicture}
\end{center}
Generators as Preemptable Jobs

- Generator: Function, returns (yield) and keeps state
  - Compare: Iterator
- **yield** → Preemption Point

```javascript
function T1 () {
  var i = 0;
  var result = 0;
  while ( i < 100000000 ) {
    // heavy computation
  }
  return result;
}
```

```javascript
function *T1 () {
  var i = 0;
  var result = 0;
  yield
  while ( i < 100000000 ) {
    // heavy computation
    yield
  }
  return result;
}
```
Generators as Preemptable Jobs

- Generator: Function, returns \( \texttt{yield} \) and keeps state
  - Compare: Iterator
- \texttt{yield} → Preemption Point

```javascript
function T1 () {
  var i = 0;
  var result = 0;
  while ( i < 100000000 ) {
    // heavy computation
  }
  return result;
}
```

```javascript
function *T1 () {
  var i = 0;
  var result = 0;
  yield
  while ( i < 100000000 ) {
    // heavy computation
    yield
  }
  return result;
}
```

- **Automatically** add Preemption Points
- to decorated functions
  - in \texttt{for}, \texttt{while}, \texttt{do-while}-loops
  - before function calls

Stefan Naumann
RT.js
Proposing RT.js

.js Code

Transpiler

Runtime Library (incl. Scheduler)

.js Code

with Premption Points

Upload

Stefan Naumann

RT.js
Scheduling Layer in JavaScript

- Manages Job queue itself, prioritizing
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$
Priorities: $T_1 = T_2 < T_3$
Priorities: $T_1 = T_2 < T_3$
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$
Priorities: $T_1 = T_2 < T_3$

Rescheduling

Browser Loop

RT.js

Round

Slice

Budget

yield

RT.js Scheduler

Button Press!

activate(T3)
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$

RT.js Scheduler
Event Handling
Rendering

Rescheduling
yield

Budget
Slice
Round

Browser Loop

RT.js Scheduler

RT.js Scheduler

Rendering
Priorities: \( T_1 = T_2 < T_3 \)
Priorities: \( T_1 = T_2 < T_3 \)

activate(T2)

Browser Loop

Round

Slice

Budget

RT.js Scheduler

RT.js Scheduler

T1 T1 T1

T1

Button Press!
Priorities: $T_1 = T_2 < T_3$

activate(T2)

Button Press!
Priorities: $T_1 = T_2 < T_3$

activate(T2)

terminate()

Button Press!
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$

activate(T2)

Button Press!
Optimizations and Scheduling Example

Priorities: \( T_1 = T_2 < T_3 \)

![Diagram showing RT.js Scheduler, Browser Loop, Round, Slice, and Budget with tasks T1, T2, T3 being scheduled and activated at specific times.](image)

- RT.js Scheduler
- Browser Loop
- Round
- Slice
- Budget

activate(T2) → terminate()

activate(T3)
Priorities: $T_1 = T_2 < T_3$

activate(T2)

terminate()

Budget
Slice
Round

RT.js Scheduler
RT.js Scheduler
RT.js Scheduler

Browser Loop

$\text{Button Press!} \rightarrow \text{activate(T3)}$
Priorities: $T_1 = T_2 < T_3$

activate(T2)

terminate()

activate(T3)

Button Press!

activate(T3)
Optimizations and Scheduling Example

Priorities: $T_1 = T_2 < T_3$

activate(T2)

terminate()

activate(T3)

Button Press!
Synchronisation Guarantees

- single-threaded ✓
- non interleaving ✗
  - Synchronisation Primitives (Non Preemptive Critical Section)
Synchronisation Guarantees

- single-threaded ✔
- non interleaving ❌

- Synchronisation Primitives (Non Preemptive Critical Section)
Missed-Deadline Ratio of the Generated Task Sets

- 1,000 Generated task sets, with 15 tasks each
- Each jobs busy waits for its WCET
- Run on Node.js v8.10.0

- FP-RM - Fixed Priority (Rate Monotonic)
- EDF - Earliest Deadline First

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04
Scheduling Overhead

- Run on Node.js v8.10.0
- Scheduling overhead max +13.2% (bucket average)
- Over all task sets, the median overhead for
  - Fixed Priority-Rate Monotonic (FP-RM) is 4.4%
  - Earliest Deadline First (EDF) is 3.8%

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04
Makro Benchmark in the Browser

- A website, consisting of 4 components
  - Box Task, animates a box
  - Input Task, releases an AES-job
  - AES Task, encrypts input
  - A generated task set ($u=0.75$)

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04
Makro Benchmark

- Run on Firefox 67.0 and Chromium 73.0.3683.86

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04
Conclusion

- RT.js allows prioritization of jobs
- Automatically introducing Preemption points
- The JavaScript engine was not modified
- Applicable to existing code and platforms

github.com/luhsra/RT.js
Microbenchmark: Overhead of yields

Benchmarking the `yield`-statement on Node.js V8.10.0

```javascript
function *genForLoop () {
    let counter = 0;
    for ( let i = 0; i < COUNT; i++ ) {
        yield;
        counter += 1;
    }
    return counter;
}

function *genFor2Loop () {
    let y = yield *genForLoop();
    return y;
}
```

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04
Microbenchmark: Overhead of yields

- Benchmarking the `yield`-statement on Node.js V8.10.0
- Upper bound for budget variable in tight loop at around 300

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04
Microbenchmark: Overhead of yields

- Benchmarking the `yield`-statement on Node.js V8.10.0
- Upper bound for budget variable in tight loop at around 300

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Per Yield</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.98</td>
<td>0.02</td>
<td>–</td>
</tr>
<tr>
<td>Generator (1 Level)</td>
<td>18.80</td>
<td>0.01</td>
<td>17.82</td>
</tr>
<tr>
<td>Generator (2 Level)</td>
<td>49.10</td>
<td>0.21</td>
<td>48.13</td>
</tr>
<tr>
<td>Generator (3 Level)</td>
<td>71.45</td>
<td>0.19</td>
<td>70.47</td>
</tr>
</tbody>
</table>

Benchmark-PC: Intel Core i5-6400, 2.70 GHz, 32 GB Memory, Ubuntu 18.04