
Bild: Thorsten Hübner

# Der Nächste, bitte!

## Wie die Event-Loop asynchronen Code im Browser ermöglicht

**In der Regel läuft der Code von Webseiten in nur einem Thread, modernen Multicore-Prozessoren zum Trotz. Echte Parallelität gibt es so nicht, aber ein schlaues Programmiermodell erlaubt dennoch „asynchronen" Code, der nicht linear abgearbeitet wird.**

Von Lars Hupel

Diese Webseite verlangsamt Ihren Browser – so oder so ähnlich vermelden es Chrome, Firefox & Co., wenn ein Skript zu viel Rechenzeit benötigt und dadurch der ganze Tab (früher der ganze Browser) ins Stocken gerät. Der Grund hierfür ist schnell erklärt: Noch genau wie damals, als JavaScript im Netscape Navigator das Licht der Welt erblickte, teilt sich der JavaScript-Code die Ausführungszeit in einem Tab mit dem Rendering und der Ereignisbehandlung. Daraus folgt unter anderem, dass keine Klicks oder Tastatureingaben verarbeitet werden, solange eine JavaScript-Funktion rödelt.

c't Hardcore

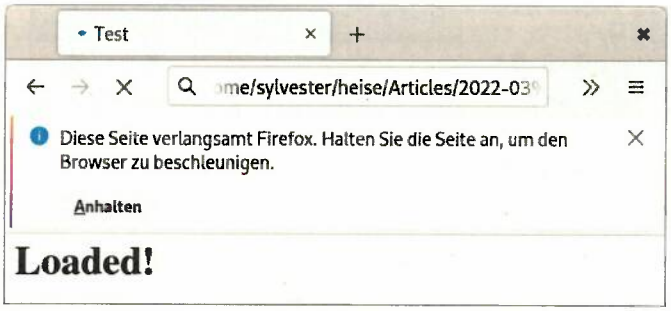
In Zeiten, in denen selbst billige Smartphones vier oder mehr Kerne haben, mag das überraschen. Warum verteilen sich JavaScript- und Browser-Code nicht schon längst auf diverse Threads? Und wenn sie es nicht tun, warum frieren Browser nicht viel häufiger ein? Moderne Websites lassen allerhand Skripte laufen, bei guter Programmierung lahmt der Browser trotzdem nicht – und bei schlechter Programmierung helfen auch mehr Kerne nicht.

Denn mehr Kerne bieten nur dann mehr Performance, wenn die Rechenleistung auch abgerufen werden kann. Die allermeisten Webseiten sind nicht CPU-intensiv, sondern I/O-intensiv. Das heißt, die Recheneinheiten langweilen sich, während die Seite auf Antwort vom Server, Interaktionen des Nutzers oder Daten von der Webcam wartet.

Im Alltag sieht man das sehr deutlich. Der Löwenanteil der Zeit bis zur fertig angezeigten Webseite vergeht, während alle

nötigen Ressourcen abgerufen werden, also HTML, Grafiken, Stylesheets et cetera. Von seltenen Ausnahmen abgesehen, ist die Ausführung von JavaScript nicht der limitierende Faktor. Deswegen teilen Browser jeder Seite auch nur einen Thread zu, was so auch die HTML- und JavaScript-Spezifikationen vorschreiben. Für die seltenen Ausnahmen gibt es „Worker“ (siehe Kasten auf S. 142).

**Wenn eine Meldung auftaucht, dass die Seite den Browser ausbremst, dann ist in aller Regel nicht mehr Leistung nötig, sondern bessere Programmierung.**




## Thread mit Schleife

Damit eine Website immer schön zackig auf Nutzereingaben reagiert und nicht „hängt“, braucht man also nicht mehr oder schnellere CPUs. Stattdessen braucht man eine Möglichkeit, auch dann auf Nutzereingaben zu reagieren, wenn die Website gerade auf etwas anderes wartet, etwa einen Dateidownload oder Bilddaten von der Webcam.

Genau für solche Fälle bieten moderne Betriebssysteme eine Lösung an: ereignisbasierte Ein- und Ausgabe (Event-based I/O). Bei klassischem I/O fordert ein Programm den Inhalt einer Datei an und wartet dann aktiv darauf, dass der Inhalt verfügbar wird. Dann verarbeitet es ihn und erst danach geht es weiter, eventuell mit der Anforderung für die nächste Datei. Man sagt, der Code arbeitet „synchron“. Bei Event-based I/O überwacht ein Programm stattdessen eine Reihe von Ereignisquellen und reagiert, wenn ein Ereignis auftritt. Die Programmausführung hängt davon ab, in welcher Reihenfolge Ereignisse auftreten und nicht in welcher Reihenfolge der Code notiert ist. Man spricht von „asynchroner“ Programmierung. Warteschlangen sorgen dafür, dass Ereignisse nach und nach abgearbeitet werden können, auch wenn sie sehr schnell aufeinander folgen.

Event-based I/O erlaubt einem Programm zum Beispiel, den Inhalt mehrerer Dateien gleichzeitig anzufordern. Danach kümmert es sich um andere Dinge oder macht einfach gar nichts (und belegt außer Arbeitsspeicher keine Ressourcen). Sobald eine der Dateien verfügbar ist, fügt das Betriebssystem ein Ereignis an eine Warteschlange an, das darüber informiert. Das Programm kann dann auf das Ereignis reagieren, zum Beispiel, indem es den jetzt verfügbaren Dateiinhalt verarbeitet.

Auf diese Weise „hängt“ das Programm nicht in Erwartung des Dateizugriffs. Wenn der Nutzer zwischendurch klickt oder tippt, kann es auf diese Ereignisse reagieren, obwohl die Datei immer noch nicht zur Verfügung steht, was die Reaktionsfreudigkeit einer Anwendung deutlich erhöht. Durch die Entkopplung von „Datei anfordern“ und „Datei verfügbar“ kann der I/O-Scheduler des Betriebssystems außerdem die Anfragen möglichst effizient umsortieren oder andere Optimierungen vornehmen.

Event-based I/O kommt auch bei Webseiten zum Einsatz. Intern bekommt jeder Browser-Tab eine Ereignisschleife („Event Loop“) zugewiesen. Die Schleife wartet auf Nachrichten vom Betriebssystem, etwa Mausbewegungen und -klicks, Timer-Ereignisse, Netzwerkpakete et cetera. Sobald ein Ereignis eintritt, ruft die Event-Loop die zugehörige Routine zur Behandlung auf. Erst wenn die Behandlung fertig ist, ist die Event-Loop wieder an der Reihe. Falls inzwischen ein weiteres Ereignis aufgetreten ist, ruft sie gleich die nächste Routine zu dessen Behandlung auf. Andernfalls wartet sie eben, bis das nächste Ereignis auftritt.

## Synchron und asynchron

Die Event-Loop ist aber kein magisches Mittel, das automatisch dafür sorgt, dass der Browser immer schön flott reagiert. Nach wie vor steht nur ein Thread zur Verfügung und wenn beispielsweise die Behandlung eines Ereignisses 30 Sekunden dauert, dann werden in dieser Zeit keine anderen Ereignisse verarbeitet und der Tab scheint zu hängen.

JavaScript bietet die nötigen Mittel, um Ressourcen optimal auszunutzen, ohne den Browser-Tab zu blockieren. Man unterscheidet grob zwischen zwei Arten von Schnittstellen, nämlich die synchronen und die asynchronen. Den Unterschied kann man sich sehr einfach verdeutlichen:

```
alert("Hello world");
document.addEventListener("click",
  () => alert("Hello world"));
```

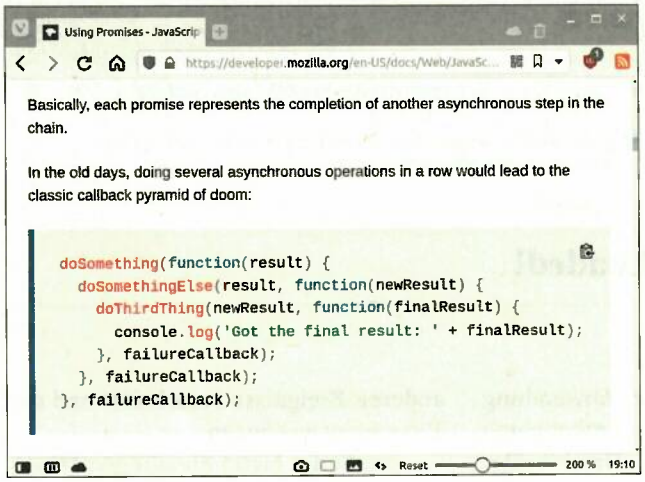
Die erste Zeile produziert ein Dialogfenster mit „Hello world“. Diese Aktion wird sofort ausgeführt. Das bedeutet vereinfacht gesagt, dass der Browser die Ausführung des JavaScript-Codes so lange anhält, wie der Dialog angezeigt wird. Erst wenn der Nutzer den Dialog schließt, wird die nächste Anweisung ausgeführt. Man spricht deswegen von einem synchronen Aufruf.

Die zweite Zeile hingegen registriert eine Routine, die beim Klick irgendwo auf der Seite ausgeführt wird. Die Abarbeitung erfolgt asynchron, denn der Code wartet an dieser Stelle nicht, bis das Ereignis eintritt. Daher wird der übergebene Handler (im Beispiel `() => alert("Hello world")`) oft als „Callback“-Funktion bezeichnet. Sie kommt nur zum Zug, wenn das Ereignis eintritt (im Beispiel ein Mausklick) und gerade kein anderer JavaScript-Code ausgeführt wird. Ohne Klick-Ereignis wird der Handler gar nicht ausgeführt und wenn noch anderer Code läuft, kann der Handler erst dann auf das Klick-Ereignis reagieren, wenn die Ausführung des Codes beendet ist. Daher rührt auch die eingangs erwähnte Fehlermeldung: Falls der Code einer Website sehr lange läuft, kommt die Event-Verarbeitung praktisch gar nicht mehr an die Reihe und die Website reagiert auf nichts.

Langwierige Aktionen, etwa das Laden von Daten, sollte man also mög-

### c't kompakt

- Echte Parallelität ist beim Scripten von Webseiten unüblich und nur selten sinnvoll.
- Das Programmiermodell der Event-Loop erlaubt dennoch nicht-linearen Code.
- Moderne JavaScript-Versionen bieten verschiedene Möglichkeiten, die Event-Loop zu nutzen.




**„In the old days ..." – für Mozilla liegt die Callback-Hölle offenbar bereits in dunkler Vergangenheit.**

lichst nicht synchron programmieren, sondern über asynchrone Events abwickeln:

```
const req = new XMLHttpRequest();
req.open("GET", "/more_data.txt");
req.addEventListener("load",
  () => alert(req.responseText));
req.send();
```

Dieser Code erzeugt zuerst eine HTTP-Anfrage (`XMLHttpRequest`, auch „XHR" genannt) und initialisiert sie mit `open()`. Anschließend registriert der Code einen Callback-Handler, der die Antwort verarbeitet. Danach schickt `send()` den Request ab und das Programm läuft direkt weiter. Es könnte zum Beispiel weitere Requests aufsetzen und abschicken. Erst wenn die Datei more_data.txt geladen wurde (und gerade kein Code mehr ausgeführt wird), kommt der Handler des Load-Events zum Zug. Im Beispiel gibt er den Dateiinhalt per `alert()` aus.

## Callback-Hölle

Baut man die Webanwendung nach diesen Prinzipien, sammeln sich flugs zahlreiche Callbacks an. Wenn die Callbacks nicht völlig unabhängig voneinander sind, muss man sie außerdem ineinander verschachteln. Das macht den Code schnell unübersichtlich und es fällt schwer, den gesamten Stapel im Blick zu behalten. Welcher Teil des Codes läuft synchron? Welche Teile werden durch welche Events ausgelöst? Für dieses Problem hat sich der Begriff „Callback-Hölle" eingebürgert.

Als einfaches Beispiel soll wieder eine Datei geladen werden. Deren URL steht aber in einer anderen Datei, die man daher zuerst anfragen muss:

```
const r1 = new XMLHttpRequest();
const r2 = new XMLHttpRequest();
r1.open("GET", "/url.txt");
r1.addEventListener("load", () => {
  const url = r1.responseText;
  r2.open("GET", url);
  r2.addEventListener("load", () => {
    const text = r2.responseText;
    console.log(text);
  });
  r2.send();
});
r1.send();
```

Übersichtlich ist das schon in diesem simplen Beispiel nicht. Mehr Callbacks oder komplexere Abhängigkeiten machen es noch komplizierter. Die nötigen Verschachtelungen führen Schritt für Schritt zu immer weiteren Einrückungen im Code, sodass man auch von der „callback pyramid of doom" spricht.

Mit dem JavaScript-Sprachstandard ES2015 (auch ES6 genannt) wurden 2015 sogenannte „Promises", also Versprechen, eingeführt, um die Callback-Hölle abzuschaffen. Ein solches Versprechen ist ein Objekt, das sich in einem von drei Zuständen befindet: schwebend, erfüllt (mit einem Wert) oder verworfen (mit einem Fehler). Legt man ein neues Promise an, schwebt es zwar zunächst, steht aber sofort zur Verfügung, ohne den Programmablauf aufzuhalten. Es ist sozusagen eine leere Wert-Hülse. Später, wenn sein innerer Wert verfügbar wird (oder definitiv nicht mehr verfügbar wird), geht es in den erfüllten (oder verworfenen) Zustand über.

Aus der Callback-Hölle führen Promises, weil sie sich verketten lassen: Methoden wie `then()`, mit denen man auf Erfüllung (oder Verwerfung) eines Promises reagiert, geben selber wieder ein Promise zurück. Das schafft die Verschachtelungen und Einrückungen ab, obwohl der Code weiter aus asynchronen Callbacks besteht.

Neue Browser-Schnittstellen verwenden von Haus aus Promises. Zum Beispiel gibt es als Ablösung für XHR die Funktion `fetch()`:

```
fetch("/url.txt")
  .then(f1 => f1.text())
  .then(url => fetch(url))
  .then(f2 => f2.text())
  .then(text => console.log(text));
```

Dieser Schnipsel erfüllt die gleiche Funktion wie das XHR-Beispiel von oben, ist

## Parallelismus mit Workern

JavaScript-Code innerhalb einer Webseite wird zwar immer nur mit einem Thread ausgeführt, mit sogenannten Web Workern gelingt die Parallelisierung aber doch. `Worker` erlauben es einer Seite, eine JavaScript-Datei in einem eigenen Thread zu starten:

```
const worker = new Worker("/prog.js");
```

Hauptskript und Worker-Skript sind voneinander abgekoppelt. Es ist vom Hauptskript aus nicht möglich, im Worker definierte Funktionen aufzurufen oder auf seine Variablen zuzugreifen. Umgekehrt darf ein Worker auch nicht auf das Document Object Model (DOM) der Webseite zugreifen oder Funktionen oder Variablen des Hauptskripts verwenden.

Der einzige Kommunikationsweg ist über Nachrichten. Beide Skripte können per `postMessage()` Nachrichten verschicken, welche die Gegenseite per Event-Handler verarbeitet:

```
worker.addEventListener(
  "message",
  event => console.log(event.data)
);
```

Die Programmierung mit Web Workern haben wir bereits in [1] detailliert beschrieben.

aber fast so lesbar wie synchroner Code. Im Unterschied dazu läuft das Programm allerdings nach diesem Schnipsel direkt weiter, ohne darauf zu warten, dass die Dateien geladen werden. Die `then()`-Kette wird asynchron abgearbeitet, jeweils wenn eine Datei geladen wurde und gerade kein anderer Code läuft.

## Abwarten

Noch einfacher wird es mit einer Syntaxerweiterung für Promises, die das JavaScript-Sprachkomitee ausgetüftelt hat:

```
async function loadFile() {
  const f1 = await fetch("/url.txt");
  const url = await f1.text();
  const f2 = await fetch(url);
  const text = await f2.text();
  console.log(text);
}
```

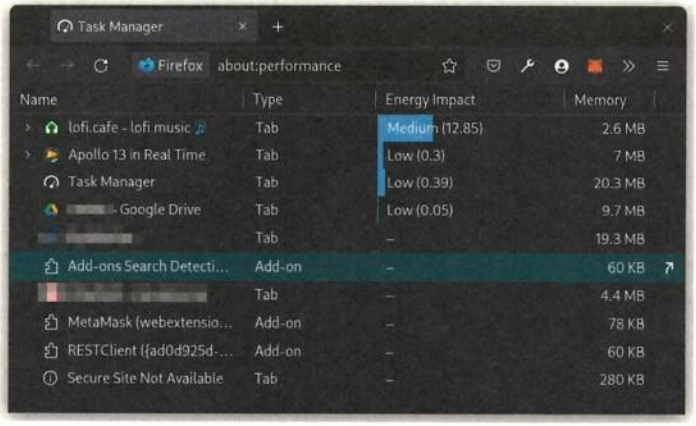
Dank der zwei Schlüsselwörter `async` und `await` kann man Funktionen, die intern auf Promises zurückgreifen, in scheinbar synchronem Stil schreiben. So entsteht oberflächlich der Eindruck, Aufrufe wie `await fetch()` würden blockieren und die Ausführung erst nach dem Aufruf fortgesetzt. In Wahrheit werden Callbacks verkettet und der Code asynchron abgearbeitet. Dieses Code-Stück und das obige mit der `then()`-Verkettung behandelt die Browser-Engine weitestgehend gleich.

Eine Einschränkung gibt es allerdings: `await`-Aufrufe müssen in eine `async`-Funktion verpackt werden. Dies hat technische Gründe; sehr neue Browserengines weichen die Einschränkung auf und erlauben `await` ohne explizites `async` in Modulen.

Obwohl durch `await` die Illusion eines linearen Programmablaufs entsteht, kann der Browser I/O-Operationen parallelisieren. Der folgende Code stößt mehrere HTTP-Requests an, die zur gleichen Zeit auf ihre Erledigung warten:

```
const responses = await Promise.all([
  fetch("/data1.json"),
  fetch("/data2.json"),
  fetch("/data3.json")
]);
```

Auch hier läuft nach wie vor nur ein Thread im Browser-Tab. (Die Kontrolle über das Netzwerk-I/O ist an das Betriebssystem delegiert.) Trotzdem steht `responses` als schwebendes Promise sofort zur Verfügung und die Programmausführung läuft weiter. Wenn alle drei Dateien geladen wurden, erfüllt sich `responses` und Code, der per `then()` oder `await` darauf gewartet hat, wird ausgeführt.




**Moderne Browser nutzen durchaus mehrere Prozesse – aber um Tabs voneinander zu isolieren, in der Regel nicht für Parallelität innerhalb eines Tabs.**

## Mikrotasking

Auch wenn der Code synchron aussieht und nirgendwo `addEventListener()` oder dergleichen steht: Operationen auf Promises behandelt ebenfalls die Event-Loop. Dafür erzeugt die Browserengine eine Art virtuelle Benachrichtigung, wenn ein Promise erfüllt oder verworfen wird. So bündelt sie alle Events an zentraler Stelle und arbeitet sie in definierter Reihenfolge ab.

Im Detail ist es etwas komplizierter, denn Browser unterscheiden zwischen Tasks und Microtasks, die in verschiedenen Warteschlangen auflaufen. Ein Task ist – vereinfacht gesprochen – ein Stück JavaScript-Code, welches sequenziell abgearbeitet werden muss. Beispielsweise führt ein Klick auf einen Button dazu, dass der Brower den zugehörigen `click`-Callback als Task plant. Wenn der Callback fertig ausgeführt wurde, dann ist der Task beendet und die Abarbeitung der Event-Loop ist wieder an der Reihe. Zwischen mehreren Tasks kann der Browser einen neuen Rendervorgang durchführen, muss er aber nicht. Tasks arbeiten Browser in der Reihenfolge ab, in der sie anfallen.

Microtasks sind dem Namen nach zu urteilen kleinere Tasks. Eine genaue Definition davon fällt schwer – die Browser weisen subtile Unterschiede auf. Grob gesagt: Wenn während der Ausführung eines Tasks ein Promise erfüllt wird, dann ruft der Browser die zugehörigen Callbacks unmittelbar nach Fertigstellung des Tasks auf. Das bedeutet auch, dass er eine Funktion mit mehreren `await`-Anweisungen möglicherweise en bloc abarbeitet, ohne dass zwischendurch die Event-Loop die Kontrolle bekommt. Eine Garantie gibt es dafür allerdings nicht. Events und Promises bleiben asynchroner Code, der zwar irgendwann ausgeführt wird, aber in der Regel erst dann, wenn kein anderer Code mehr läuft.

## Fazit

Der Kerngedanke der asynchronen Programmierung ist, dass man die vorhandenen Ressourcen optimal ausnutzen kann – obwohl man nur einen einzigen Thread zur Verfügung hat. Damit das klappt, darf man aber keinen Code schreiben, der selbst auf Ereignisse wartet. Das würde den Browser oder zumindest das eigene Browser-Tab blockieren. Stattdessen sollte man – explizit per Event oder Promise oder implizit über `await` – die Kontrolle an die Event-Loop zurückgeben und das erwartete Ereignis in einem Callback behandeln.

Seit der ersten Veröffentlichung von Chrome ist es en vogue, dass Browser pro Tab einen Prozess starten. Falls eine Webseite wegen schlechter Programmierung hängen sollte, reißt sie zumindest nicht auch alle anderen Tabs mit in den Abgrund. Der Nachteil: Viele Prozesse benötigen auch viel Hauptspeicher. Die Frage, welche Tabs sich einen Prozess teilen müssen, und ob und wann alte Tabs dauerhaft schlafen gelegt werden, muss jeder Browser-Hersteller für sich beantworten. Um das letzte Quäntchen Performance aus moderner Hardware herauszukitzeln, wird an diesen Stellen kontinuierlich optimiert. *(syt@ct.de)* c't

### Literatur

[1] Oliver Lau, Würze fürs Web, Verteiltes Rechnen mit JavaScript, c't 9/2012, S. 190