Synchronous Interface for OPFS

This document is public

Author: rstz@chromium.org
LGTM: fivedots@chromium.org
June 2021

The augmented OPFS provides a synchronous interface for file operations such as reading and writing. The synchronous interface is only available in dedicated workers and we do not plan on exposing a blocking API on the main thread. It is functionally equivalent to the asynchronous interface provided. The documentation discourages use of this synchronous interface unless it is used in conjunction with WebAssembly, and it is expected to be phased out within the next five years.

The decision is made with three goals in mind.

1. The augmented OPFS should provide a performant file system backend to unlock a great user experience for (legacy) applications compiled to WebAssembly.
2. The augmented OPFS should provide great developer experience when interacting with WebAssembly.
3. The web platform should stop providing a synchronous storage API if WebAssembly's support for asynchronous WebAPIs has sufficiently improved.

Next, we analyse each of these goals in more detail.

WebAssembly performance analysis

Developers such as Adobe and WebTorrent's maintainers have expressed interest and shown use cases for performant storage on the Web. It is worthwhile to try to unlock applications needing every bit of performance on the web.

In benchmarks using the Storage Foundation API, one can observe that asynchronous storage APIs perform poorly for some usage patterns, while being competitive for others. Compared to a synchronous interface, the asynchronous interface has a 30% - 50% decrease in performance and ca. 20% increase in binary size. We believe that this is indicative of user-observable slowdowns in real-world applications.

Our first benchmark measures the performance of an Emscripten / WebAssembly port of SQLite. SQLite itself only performs synchronous I/O.[1] The benchmark compares two configurations. The *sync* configuration uses Emscripten's standard file system over Storage Foundation's synchronous interface. The *async* configuration uses an experimental, Asyncify-based Emscripten file system over Storage Foundation's asynchronous interface. The

[1] No experiments were performed using the unmaintained module for async SQLite.

workload is provided by the SQLite development team that describes it as a “typical” workload of a database.

We observed that the overall slowdown of the asynchronous configuration is between 20% and 50%[2]. Part of this can be attributed to Asyncify’s overhead, yet individual, small read/write operations are up to 10 times slower than they are under the synchronous interface. The overhead seems to be a constant per call. Binary size increased by 20%. Detailed results can be found here. Our benchmarks are available to download.

We find that these results are in line with the observations of the Emscripten team about Asyncify’s performance. Providing an async file system requires us Asyncify to instrument many functions which hurts performance and binary size. Outside of Asyncify, the async overhead seems to be a small constant per call. In an additional experiment, SQLite was compiled to flush to disk after every write operation. Unsurprisingly, this makes write operations up to 1000 times slower. The async overhead of write operations shrinks to 9%. This strengthens our belief that the overhead per async operation is a very small constant and only becomes significant when the OS returns the operation’s result almost immediately.

The observations made in the SQLite benchmarks are in line with microbenchmarks we performed. In particular, we were able to observe a 10x slowdown when reading or writing very small chunks (< 1KB) of a large file (such as a database), but no slowdown when reading or writing larger chunks (>100KB). We will make our benchmarks available soon.

Developer experience with asynchronous APIs

A synchronous interface can provide a superior developer experience in many cases and is therefore preferred for non-storage Web APIs. Even for the case of storage APIs, synchronous interfaces have benefits when porting (legacy) C++ code to the web through WebAssembly. Most C++ code uses synchronous I/O for storage, refactoring that code to use asynchronous I/O can be expensive. Hence, even if WebAssembly had great built-in support for asynchronicity, many applications would not be able to leverage it easily. Right now, developers mostly use Asyncify which introduces significant complexity to building, optimizing and debugging a Webassembly module.

This is one of the reasons Emscripten, the major WebAssembly toolchain, implements file system support mainly through a synchronous in-memory file system. On node.js, Emscripten uses the synchronous node.js file system instead. A synchronous API for OPFS would enable a performant and persistent Emscripten file system on the web platform.

Phasing out the synchronous interface

We acknowledge that the Web Platform aims to offer asynchronous-only storage and network APIs. It is very reasonable to expect WebAssembly and the tooling surrounding it to improve

[2] The variance between individual runs was reasonably small as calculated in the detailed results.

support for asynchronous Web APIs over time. A proposal to improve support for asynchronicity is currently in early Phase 1. No consensus has yet been reached on its exact scope and there is no prototyping by a major browser vendor at the time of this writing. Therefore, a usable, shipped implementation will not be available before way into 2022.

Our goal is to deprecate the synchronous OPFS interface at this point. In order to be able to do so, we will

- Explicitly communicate to web developers that the synchronous interface is to be avoided outside of very specific WebAssembly use cases.
- Commit to providing a polyfill for the sync APIs using the async APIs and the Wasm support.
- Communicate a concrete deprecation plan as soon as possible

Alternatives considered

Offering an asynchronous interface only

The main argument against a synchronous interface is that synchronous APIs will be made obsolete through improvements in WebAssembly and native code. Our counterpoint is that these improvements will be done over a multi-year time span of uncertain length. A synchronous interface could unlock important use cases much faster and be deprecated later on.

In the WebAssembly development, it is indeed possible that Stack Switching, or some other proposal, will reach wide adoption by the end of 2022. We are closely monitoring the current state of the proposal and will react to changes that present themselves during our design discussions. It is possible that the community subgroup decides to go for a more lightweight approach that only enables use cases such as ours, as discussed recently. See the appendix for a detailed discussion of WebAssembly's efforts to improve support for asynchronous APIs.

Run the synchronous interface as a separate proposal

We could split the synchronous interface for OPFS into a separate proposal. This would help us streamline the discussions about the main augmented OPFS, and allow us to gather additional developer feedback specifically through an origin trial for the synchronous interface.

Splitting the synchronous interface from the main proposal also allows us to standardize the feature with a clear deprecation path from the get go, informing developers that the synchronous API will go away by the end of 2023. Long-term support after deprecation could be achieved through a polyfill bundled with Emscripten.

Landing the augmented OPFS without the synchronous interface would, however, likely break some central use cases of the API. The augmented OPFS would fail to bridge the feature gap between a native platform (even node.js) and the web platform, which hurts adoption of the API.

It would also force early partners that currently rely on Storage Foundation to perform a costly switch to the asynchronous OPFS.

Provide a synchronous interface for directory operations

We are open to extending OPFS to make directory operations (such as opening, deleting or listing files) available synchronously in workers if significant interest arises during our specification discussions.

Appendix

Asynchronous WebAssembly

At a high level, WebAssembly's design separates the stack and the heap (linear memory). A program cannot perform nontrivial modifications to the stack. It is possible to save the current stack to the heap, but this is an expensive operation. Additionally, WebAssembly only maintains a single stack that is shared with the Javascript runtime. This means that suspending a WebAssembly module (like Asyncify does) is expensive in terms of time and code size: there must be explicit code within the module that saves the current state by copying the stack to the heap.

When calling *synchronous* Web APIs (or any synchronous Javascript function), execution of the WebAssembly module pauses until the API call is completed. Since Javascript and WebAssembly share the same stack, there is no need to modify the stack: the WebAssembly module naturally resumes when the result is available.
When calling an *asynchronous* Web API, Javascript unwinds the stack and returns control to the event loop. The event loop may now call the WebAssembly module again, but it must make sure that it calls the correct function with the correct stack. Currently, there is no cheap way of doing this.

The Stack Switching proposal aims to introduce some "cheap" way of saving the stack and thereby suspending a WebAssembly module. There is a broad list of goals for the proposal and multiple ideas have been discussed publicly (e.g., Wasm/k, typed continuations, Await) in the subgroup meetings. At the time of this writing, it is unclear which one of the competing approaches will gain the most support. It is, however, reasonable to assume that the winning approach will adequately address the issue of calling asynchronous Web APIs from Wasm. The subgroup is very aware of the issues with asynchronous Web APIs and recently discussed solving this simpler problem first.