**Browser-side PDF processing with Go and WebAssembly**

[#go](https://dev.to/t/go)[#webassembly](https://dev.to/t/webassembly)[#javascript](https://dev.to/t/javascript)[#webdev](https://dev.to/t/webdev)

**TLDR:** This [page](https://wcchoi.github.io/go-wasm-pdfcpu/) demonstrates how to use WebAssembly (compiled from a tool called [pdfcpu](https://github.com/hhrutter/pdfcpu) with Go v1.12) to extract the first page of a PDF file, done completely in client-side. Main idea is adding filesystem emulation support in Browser environment. To skip the prose below and go straight to the code, see the [Github repo](https://github.com/wcchoi/go-wasm-pdfcpu).

WebAssembly (wasm) is gaining a lot of buzz these days. Many languages are starting to experiment with/adopt this new technology and add wasm compilation support. Rust (wasm-pack, wasm-bindgen), Go, C/C++ (emscripten), Java (TeaVM), .NET (blazor) are a few examples.

In this article I am trying to document my attempt to port a command line application written in Go to wasm and using it in browser context. I've played with wasm using Emscripten for C/C++ codes before, but not with Go.

The wasm support for Go has landed in v1.11 and at the time of this writing the latest release version of Go is v1.12. It is said that the support is still maturing, with some limitations such as only the main function being exported right now (see [this issue](https://github.com/golang/go/issues/25612)), or not working on Android due to Out of Memory problem ([issue](https://github.com/golang/go/issues/27462)). Things may improve and change in future release, so please keep that in mind when reading the steps below.

Also, I am not a Go programmer (I've written <100 LoC). In fact we are not even going to change a single line of Go code in this article (but there will be lots of JavaScript/Node.js involved, be warned!). If you are looking for how to call to JavaScript from Go in wasm, there are many other outstanding resources online for that, but this is not one. Instead it is more about writing js to make the Go wasm run with file I/O support in the browser.

The target application is [pdfcpu](https://github.com/hhrutter/pdfcpu), a super useful utility for processing PDF files like extracting pages, optimizing the file size, etc. There are already tons of online sites for doing similar processing on PDF files without the need for users to download extra software on their computer, but most of them require uploading the file to a third-party server, which in some case - depending on where the server is located relative to you - the network transfer time (upload + download) is longer than the actual processing time. Also some documents are confidential and uploading to an external server may not be a good idea. If the pdf processing is completely done in the browser using wasm, these would become non-issues. Plus it can be made to work completely offline - that is if you cache the page's assets in the browser using things like Service Worker.

With that said, let's get started.

The very first step is to install the Go version v1.11+ (v1.12 is used for this article) and Node.js (I'm using version 12.0.0), which could be easily done by referring to the official documentations - [Go](https://golang.org/doc/install), [Node.js](https://nodejs.org/en/download/).

The next step is to try to build a native binary of pdfcpu, which is again not difficult, thanks to Go Modules support for this project. Referring to the [Github Repo](https://github.com/hhrutter/pdfcpu#installation) (Note: I am using the commit 9d476ddd92a for this article):

git clone https://github.com/hhrutter/pdfcpu

cd pdfcpu/cmd/pdfcpu

go build -o pdfcpu

You will see a binary executable pdfcpu in the folder, running ./pdfcpu version outputs pdfcpu version 0.1.23

Next let's try to build the wasm version (referred to as wasm module), in the same directory run:

GOOS=js GOARCH=wasm go build -o pdfcpu.wasm

We'll see the compiled wasm module output file pdfcpu.wasm, but how do we know if it will do anything?

From the [Go documentation](https://github.com/golang/go/wiki/WebAssembly), it's possible to execute the wasm file using Node.js. It requires running a js file called wasm\_exec.js located in misc/wasm directory of your Go installation (eg: /usr/local/go/misc/wasm, NOTE the js file must match the same version of Go used to compile the wasm file, so you can't just grab the latest wasm\_exec.js from the golang Github repo and expect it to work), so let's confirm that:

cp /usr/local/go/misc/wasm/wasm\_exec.js ./

node wasm\_exec.js pdfcpu.wasm version

Output:

pdfcpu version 0.1.23

So indeed the wasm file contains the code of pdfcpu.

Next let's run it in the browser (PS: the browser I used for testing is Chrome), referring to the same documatation page, we need to prepare an index.html file like this:

<html>

<head>

<meta charset="utf-8">

<script src="wasm\_exec.js"></script>

<script>

if (!WebAssembly.instantiateStreaming) { // polyfill

WebAssembly.instantiateStreaming = async (resp, importObject) => {

const source = await (await resp).arrayBuffer()

return await WebAssembly.instantiate(source, importObject)

}

}

const go = new Go();

WebAssembly.instantiateStreaming(fetch("pdfcpu.wasm"), go.importObject).then((result) => {

go.run(result.instance);

});

</script>

</head>

<body></body>

</html>

Let's start a static file server to test the page, but one thing to keep in mind is that the .wasm file should have MIME Type application/wasm for WebAssembly.instantiateStreaming to work, otherwise you will get error like this in the console when you visit index.html:

Uncaught (in promise) TypeError: Failed to execute 'compile' on 'WebAssembly': Incorrect response MIME type. Expected 'application/wasm'.

I use this Node.js script from <https://gist.github.com/aolde/8104861> and add wasm MIME Type as follow:

....

mimeTypes = {

"html": "text/html",

"jpeg": "image/jpeg",

"jpg": "image/jpeg",

"png": "image/png",

"js": "text/javascript",

"css": "text/css",

"wasm": "application/wasm",

};

.........

Run with node static\_server.js & and visit localhost:8080 on Chrome, then open the DevTools Console, we'll see:

pdfcpu is a tool for PDF manipulation written in Go.

Usage:

pdfcpu command [arguments]

The commands are:

attachments list, add, remove, extract embedded file attachments

changeopw change owner password

changeupw change user password

decrypt remove password protection

encrypt set password protection

extract extract images, fonts, content, pages, metadata

grid rearrange pages orimages for enhanced browsing experience

import import/convert images

merge concatenate 2 or more PDFs

nup rearrange pages or images for reduced number of pages

optimize optimize PDF by getting rid of redundant page resources

pages insert, remove selected pages

paper print list of supported paper sizes

permissions list, add user access permissions

rotate rotate pages

split split multi-page PDF into several PDFs according to split span

stamp add text, image or PDF stamp to selected pages

trim create trimmed version with selected pages

validate validate PDF against PDF 32000-1:2008 (PDF 1.7)

version print version

watermark add text, image or PDF watermark to selected pages

Completion supported for all commands.

One letter Unix style abbreviations supported for flags.

Use "pdfcpu help [command]" for more information about a command.

Cool, this is the stdout of running ./pdfcpu with no arguments

What if we want to specify the command line argument? We can do this by:

// in index.html

...

const go = new Go();

go.argv = ['pdfcpu.wasm', 'version']; // <- Add this line

...

Output in Chrome Console:

pdfcpu version 0.1.23

Now let's try to get pdfcpu to really do work on some PDF file instead of just printing usage/version to STDOUT, I will be using the PDF specification file obtained from <https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/pdf_reference_archives/PDFReference.pdf> as the test input file.

Before working on wasm side, let's see how the pdfcpu native binary executable works on the test file:

1. Validate PDF file
2. $ ./pdfcpu validate PDFReference.pdf
3. validating(mode=relaxed) PDFReference.pdf ...
4. validation ok
5. Extract the first page
6. $ ./pdfcpu trim -pages 1 PDFReference.pdf first\_page.pdf
7. pageSelection: 1
8. trimming PDFReference.pdf ...
9. writing first\_page.pdf ...
10. # first\_page.pdf is a 26KB pdf file

We can do the same thing with wasm using Node.js (but it takes much longer time - about 10X slower - compared to native binary)

$ node wasm\_exec.js pdfcpu.wasm validate PDFReference.pdf

validating(mode=relaxed) PDFReference.pdf ...

validation ok

$ node wasm\_exec.js pdfcpu.wasm trim -pages 1 PDFReference.pdf first\_page.pdf

pageSelection: 1

trimming PDFReference.pdf ...

writing first\_page.pdf ...

How could we ask pdfcpu.wasm to operate on the test pdf file in the browser? In the above examples, pdfcpu (whether the native binary or the wasm module run by Node.js) was given the path of the test pdf file as a command line argument, and it will read the bytes of file from the filesystem. But in the browser, there's no filesystem access.

Let's dig deeper into the wasm\_exec.js file to see what's happening when Node.js is running the wasm module, I find the following code snippet that is of interest:

....

// Map web browser API and Node.js API to a single common API (preferring web standards over Node.js API).

const isNodeJS = global.process && global.process.title === "node";

if (isNodeJS) {

global.require = require;

global.fs = require("fs");

// ..... other

} else {

let outputBuf = "";

global.fs = {

constants: { O\_WRONLY: -1, O\_RDWR: -1, O\_CREAT: -1, O\_TRUNC: -1, O\_APPEND: -1, O\_EXCL: -1 }, // unused

writeSync(fd, buf) {

outputBuf += decoder.decode(buf);

const nl = outputBuf.lastIndexOf("\n");

if (nl != -1) {

console.log(outputBuf.substr(0, nl));

outputBuf = outputBuf.substr(nl + 1);

}

return buf.length;

},

write(fd, buf, offset, length, position, callback) {

if (offset !== 0 || length !== buf.length || position !== null) {

throw new Error("not implemented");

}

const n = this.writeSync(fd, buf);

callback(null, n);

},

open(path, flags, mode, callback) {

const err = new Error("not implemented");

err.code = "ENOSYS";

callback(err);

},

read(fd, buffer, offset, length, position, callback) {

const err = new Error("not implemented");

err.code = "ENOSYS";

callback(err);

},

fsync(fd, callback) {

callback(null);

},

};

}

....... the rest

So we can see that if wasm\_exec.js is run by Node.js, it can read from the filesystem because it is using the fs module from Node.js, but if it is running in browser context (the else branch), a stub for fs is used and many needed functions are not yet implemented.

Let's try to fix that! There's a project called [BrowserFS](https://github.com/jvilk/BrowserFS) that emulates the Node.js filesystem API for the browser, we'll use it in place of the fs stub in wasm\_exec.js

In index.html, add the script tag to BrowserFS CDN js file to the head tag and initialize it, we also try to write the test pdf file to the InMemory FS (as /test.pdf in the FS) and try to run the validate command on /test.pdf:

<head>

<script src="https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js"></script>

<script src="wasm\_exec.js"></script>

<script>

// Configures BrowserFS to use the InMemory file system.

BrowserFS.configure({

fs: "InMemory"

}, function(e) {

if (e) {

// An error happened!

throw e;

}

// Otherwise, BrowserFS is ready-to-use!

var fs = BrowserFS.BFSRequire('fs');

var Buffer = BrowserFS.BFSRequire('buffer').Buffer;

// Write the test pdf file to the InMemory FS

fetch('/PDFReference.pdf').then(function(res) { return res.arrayBuffer() }).then(function(buffer) {

fs.writeFile('/test.pdf', Buffer.from(buffer), function(err) {

// check it is there

fs.readFile('/test.pdf', function(err, contents) {

console.log(contents);

done();

});

});

});

function done() {

const go = new Go();

go.argv = ['pdfcpu.wasm', 'validate', '/test.pdf'];

WebAssembly.instantiateStreaming(fetch("pdfcpu.wasm"), go.importObject).then((result) => {

go.run(result.instance);

});

}

});

</script>

</head>

Also need to change wasm\_exec.js to use BrowserFS in browser context:

...

// Map web browser API and Node.js API to a single common API (preferring web standards over Node.js API).

const isNodeJS = global.process && global.process.title === "node";

if (isNodeJS) {

global.require = require;

global.fs = require("fs");

// ..... other

} else {

var myfs = global.BrowserFS.BFSRequire('fs');

global.Buffer = global.BrowserFS.BFSRequire('buffer').Buffer;

global.fs = myfs;

// ... Delete or comment out the original global.fs = {....}

// let outputBuf = "";

}

...

If we run it, we can see the console.log call in fs.readFile sucessfully report the content bytes of the test file, but we get another cryptic exception:

Uncaught (in promise) TypeError: Reflect.get called on non-object

at Object.get (<anonymous>)

at syscall/js.valueGet (wasm\_exec.js:304)

at syscall\_js.valueGet (:8080/wasm-function[1649]:3)

at syscall\_js.Value.Get (:8080/wasm-function[1632]:123)

at syscall.init.ializers (:8080/wasm-function[1698]:649)

at syscall.init (:8080/wasm-function[1699]:354)

at os.init (:8080/wasm-function[1817]:299)

at fmt.init (:8080/wasm-function[1884]:328)

at flag.init (:8080/wasm-function[1952]:241)

at main.init (:8080/wasm-function[4325]:247)

Seems that the Go Runtime that got compiled to wasm will try to call to the JS land when accessing global.fs object, but something get wrong. From the exception stack trace there's not much useful information for debugging.

Comparing the original stub fs in wasm\_exec.js and that of BrowserFS, I noticed that the constants property is undefined for BrowserFS's fs, adding that back (using the one from Node.js fs.constants, keeping only the ones staring with O\_), the error goes away:

...

global.fs = myfs;

global.fs.constants = {

O\_RDONLY: 0,

O\_WRONLY: 1,

O\_RDWR: 2,

O\_CREAT: 64,

O\_EXCL: 128,

O\_NOCTTY: 256,

O\_TRUNC: 512,

O\_APPEND: 1024,

O\_DIRECTORY: 65536,

O\_NOATIME: 262144,

O\_NOFOLLOW: 131072,

O\_SYNC: 1052672,

O\_DIRECT: 16384,

O\_NONBLOCK: 2048,

};

But we get another error:

exit code: 1

It seems to indicate that something went wrong and the program exit with exit code 1, similar to what would happen in shell.

We can still work on something. One reason there is so little log is that the original stub of global.fs in wasm\_exec.js contatins console.log calls that I think is responsible for logging the STDOUT/STDERR of the wasm module, but BrowserFS implementation does not support that, so we check for the fd passed to fs.write/fs.writeSync, if fd is 1 or 2 (corresponding to STDOUT/STDERR), we use the original stub function, otherwise we call to BrowserFS

// ... Add to wasm\_exec.js, below the global.fs.constants = {...} mentioned above

let outputBuf = "";

global.fs.writeSyncOriginal = global.fs.writeSync;

global.fs.writeSync = function(fd, buf) {

if (fd === 1 || fd === 2) {

outputBuf += decoder.decode(buf);

const nl = outputBuf.lastIndexOf("\n");

if (nl != -1) {

console.log(outputBuf.substr(0, nl));

outputBuf = outputBuf.substr(nl + 1);

}

return buf.length;

} else {

return global.fs.writeSyncOriginal(...arguments);

}

};

global.fs.writeOriginal = global.fs.write;

global.fs.write = function(fd, buf, offset, length, position, callback) {

if (fd === 1 || fd === 2) {

if (offset !== 0 || length !== buf.length || position !== null) {

throw new Error("not implemented");

}

const n = this.writeSync(fd, buf);

callback(null, n, buf);

} else {

return global.fs.writeOriginal(...arguments);

}

};

After adding that, we now get:

validating(mode=relaxed) /test.pdf ...

wasm\_exec.js:89 can't open "/test.pdf": open /test.pdf: Invalid argument

wasm\_exec.js:135 exit code: 1

exit @ wasm\_exec.js:135

runtime.wasmExit @ wasm\_exec.js:269

runtime.wasmExit @ wasm-020eb99a-871:3

runtime.exit @ wasm-020eb99a-860:2

syscall.Exit @ wasm-020eb99a-579:26

os.Exit @ wasm-020eb99a-1802:65

main.process @ wasm-020eb99a-4283:215

main.main @ wasm-020eb99a-4281:591

runtime.main @ wasm-020eb99a-466:673

...

We have some progress now, the STDOUT/STDERR is working again, and we saw an "Invalid argument" error.

I got stuck on this part for a while, but later find a way out.

Remember the wasm module on Node.js wasm run just fine? There must be some difference between the two implementation of fs (the Node.js one and BrowserFS), we can use that as a starting point for troubleshooting.

We can use Proxy in JavaScript to print the function arguments and the return value whenever a function in fs module get called, by adding these lines in wasm\_exec.js:

.....

var handler = {

get: function(target, property) {

if(property in target && target[property] instanceof Function) {

return function() {

console.log(property, 'called', arguments);

if (arguments[arguments.length - 1] instanceof Function) {

var origCB = arguments[arguments.length - 1];

var newCB = function() {

console.log('callback for', property, 'get called with args:', arguments);

return Reflect.apply(origCB, arguments.callee, arguments);

}

arguments[arguments.length - 1] = newCB;

}

return Reflect.apply(target[property], target, arguments);

}

} else {

return target[property]

}

}

}

// Map web browser API and Node.js API to a single common API (preferring web standards over Node.js API).

const isNodeJS = global.process && global.process.title === "node";

if (isNodeJS) {

global.require = require;

var myfs = require("fs");

global.fs = new Proxy(myfs, handler); // <- "install" the handler for proxy

// ... the rest

} eles {

var myfs = global.BrowserFS.BFSRequire('fs');

global.Buffer = global.BrowserFS.BFSRequire('buffer').Buffer;

// ..... the previous global.fs.constants = {...}, global.fs.write = function (...) {...}

global.fs = new Proxy(global.fs, handler); // <- "install" the handler for proxy;

}

Now run the Node.js again with stdbuf -o 0 node wasm\_exec.js pdfcpu.wasm validate PDFReference.pdf | tee trace.log

We will get lots of output detailing each call to fs module with the arguments and return value (to the callback), kind of like using strace:

.....

open called { '0': 'PDFReference.pdf', '1': 0, '2': 0, '3': [Function] }

callback for open get called with args: { '0': null, '1': 11 }

fstat called { '0': 11, '1': [Function] }

callback for fstat get called with args: { '0': null,

'1':

Stats {

dev: 1275115201,

mode: 33204,

nlink: 1,

uid: 1000,

gid: 1000,

rdev: 0,

blksize: 4096,

ino: 3889238,

size: 5158704,

blocks: 10080,

atimeMs: 1555990816488.329,

mtimeMs: 1555987073908.2253,

ctimeMs: 1555987073908.2253,

birthtimeMs: 1555987073908.2253,

atime: 2019-04-23T03:40:16.488Z,

mtime: 2019-04-23T02:37:53.908Z,

ctime: 2019-04-23T02:37:53.908Z,

birthtime: 2019-04-23T02:37:53.908Z } }

fstat called { '0': 11, '1': [Function] }

callback for fstat get called with args: { '0': null,

'1':

Stats {

dev: 1275115201,

mode: 33204,

nlink: 1,

uid: 1000,

gid: 1000,

rdev: 0,

blksize: 4096,

ino: 3889238,

size: 5158704,

blocks: 10080,

.....

Running on the browser, we get an error in some call:

open called Arguments(4) ["/test.pdf", 0, 0, ƒ, callee: ƒ, Symbol(Symbol.iterator): ƒ]

callback for open get called with args: Arguments [ApiError, callee: ƒ, Symbol(Symbol.iterator): ƒ]

0: ApiError

code: "EINVAL"

errno: 22

message: "Error: EINVAL: Invalid flag: 0"

path: undefined

stack: "Error

at new ApiError (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:5430:22)

at new FileFlag (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:5551:15)

at Function.getFileFlag (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:5565:42)

at FS.open (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:6103:69)

at Object.\_fsMock.<computed> [as open] (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:7006:28)

at Proxy.<anonymous> (http://localhost:8080/wasm\_exec.js:29:37)

at syscall/js.valueCall (http://localhost:8080/wasm\_exec.js:371:31)

at syscall\_js.valueCall (wasm-function[1653]:3)

at syscall\_js.Value.Call (wasm-function[1636]:482)

at syscall.fsCall (wasm-function[1691]:666)"

syscall: ""

\_\_proto\_\_: Error

callee: ƒ ()

length: 1

Symbol(Symbol.iterator): ƒ values()

\_\_proto\_\_: Object

So the Go wasm runtime passes a value to BrowserFS that it does not accept (the fs.open function's 2nd parameter flags, in this case 0 is passed), digging through the source code, it seems that BrowserFS's open function can only accept string for argument flags ('r', 'w', 'w+', etc), so we can manually convert this in wasm\_exec.js:

(Ref: <https://nodejs.org/api/fs.html#fs_file_system_flags>)

global.fs.openOriginal = global.fs.open;

global.fs.open = function(path, flags, mode, callback) {

var myflags = 'r';

var O = global.fs.constants;

// Convert numeric flags to string flags

// FIXME: maybe wrong...

if (flags & O.O\_WRONLY) { // 'w'

myflags = 'w';

if (flags & O.O\_EXCL) {

myflags = 'wx';

}

} else if (flags & O.O\_RDWR) { // 'r+' or 'w+'

if (flags & O.O\_CREAT && flags & O.O\_TRUNC) { // w+

if (flags & O.O\_EXCL) {

myflags = 'wx+';

} else {

myflags = 'w+';

}

} else { // r+

myflags = 'r+';

}

} else if (flags & O.O\_APPEND) { // 'a'

throw "Not implmented"

}

// TODO: handle other cases

return global.fs.openOriginal(path, myflags, mode, callback);

};

Running that, we get some progress but end up with a new error:

Uncaught (in promise) TypeError: Cannot read property 'get' of undefined

at storeValue (wasm\_exec.js:245)

at syscall/js.valueCall (wasm\_exec.js:388)

at syscall\_js.valueCall (:8080/wasm-function[1653]:3)

at syscall\_js.Value.Call (:8080/wasm-function[1636]:482)

at syscall.fsCall (:8080/wasm-function[1691]:666)

at syscall.Close (:8080/wasm-function[1682]:399)

at internal\_poll.\_\_FD\_.destroy (:8080/wasm-function[1771]:215)

at internal\_poll.\_\_FD\_.decref (:8080/wasm-function[1768]:212)

at internal\_poll.\_\_FD\_.Close (:8080/wasm-function[1772]:282)

at os.\_\_file\_.close (:8080/wasm-function[1799]:224)

If we compare trace.log (Node.js) with the console output (BrowserFS), we can notice that the Stat object passed to the callback of fs.fstat is different, so again we manually "patch" that in wasm\_exec.js:

global.fs.fstatOriginal = global.fs.fstat;

global.fs.fstat = function(fd, callback) {

return global.fs.fstatOriginal(fd, function() {

var retStat = arguments[1];

delete retStat['fileData'];

retStat.atimeMs = retStat.atime.getTime();

retStat.mtimeMs = retStat.mtime.getTime();

retStat.ctimeMs = retStat.ctime.getTime();

retStat.birthtimeMs = retStat.birthtime.getTime();

return callback(arguments[0], retStat);

});

};

Continuing, there are lots of call to read now, and finally, the output

validation ok

Awesome, so our BrowserFS + patching approach works!

Next, let's try something that would write out some data - extracting the first page of PDF to first\_page.pdf (see go.argv below), in index.html:

function done() {

const go = new Go();

WebAssembly.instantiateStreaming(fetch("pdfcpu.wasm"), go.importObject).then((result) => {

go.argv = ['pdfcpu.wasm', 'trim', '-pages', '1', '/test.pdf', '/first\_page.pdf'];

var st = Date.now();

go.run(result.instance);

console.log('Time taken:', Date.now() - st);

fs.readFile('/first\_page.pdf', function(err, contents) {

console.log("after run main:", err, contents);

});

});

}

It gives another error:

callback for writeOriginal get called with args:

TypeError: buffer$$1.copy is not a function

at SyncKeyValueFile.writeSync (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:8560:29)

at SyncKeyValueFile.write (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:8523:27)

at FS.write (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:6386:14)

at Object.\_fsMock.<computed> [as writeOriginal] (https://cdnjs.cloudflare.com/ajax/libs/BrowserFS/2.0.0/browserfs.js:7006:28)

at Proxy.<anonymous> (http://localhost:8080/wasm\_exec.js:29:37)

at Object.global.fs.write (http://localhost:8080/wasm\_exec.js:108:34)

at Proxy.<anonymous> (http://localhost:8080/wasm\_exec.js:29:37)

at syscall/js.valueCall (http://localhost:8080/wasm\_exec.js:406:31)

at syscall\_js.valueCall (wasm-function[1653]:3)

at syscall\_js.Value.Call (wasm-function[1636]:482)

We find that the buf passed to fs.write has no copy method. So we change that to:

global.fs.write = function(fd, buf, offset, length, position, callback) {

if (fd === 1 || fd === 2) {

if (offset !== 0 || length !== buf.length || position !== null) {

throw new Error("not implemented");

}

const n = this.writeSync(fd, buf);

callback(null, n, buf);

} else {

// buf:

arguments[1] = global.Buffer.from(arguments[1]);

return global.fs.writeOriginal(...arguments);

}

};

Finally we get the bytes of the first page in Console log! (You may take a look at the file oldindex.html in the repo for the code up to this point)

Now that it is working fine (at least for the two cases we tested, for other cases we can use the same method of comparing the BrowserFS implementation with that of Node.js output and patch global.fs.XXX in wasm\_exec.js), the remaining is to create a web UI that allows users to drag+drop/select the PDF file to process, let the wasm handle the task in a Web Worker to prevent blocking the main page UI, then report back the result or let the user download the processed pdf files in the browser.

You may take a look at a demo page [here](https://wcchoi.github.io/go-wasm-pdfcpu)

**Conclusion:**

We managed to do some hacks on wasm\_exec.js combining with BrowserFS to make a Go command line utility running in the browser successfully. As the wasm support for Go get more mature, there may be official support for filesystem emulation in browser (similar to that of Emscripten) in the future, or there will be support for exporting particular function in the wasm module that allows directly working on bytes instead of jumping through the hoops of file I/O.

If you want to see the final code, please go to the [Github Repo](https://github.com/wcchoi/go-wasm-pdfcpu).

You can also check out my other projects at <https://github.com/wcchoi>

**Issues:**

I claimed in the very beginning that wasm could be used in the client-sdie to replace some file processing in server, but this approach is not without its problems:

1. Large WebAssembly Module Size
   * If we are connecting to localhost, it is not an issue, but the pdfcpu.wasm is 8MiB in size, which is very large and defeat the stated advantage of less network transfer (upload + download) compared to uploading to external server.
   * It can be solved by gziping the wasm file, or better use brotli to compress, in my test, gzip -9 reduce the file size to 1.8MiB and brotli -9 to 1.5MiB, much smaller than uncompressed one
   * If it is still too large, we can manually modify the Go code to split the functions into individual command line tools (one tool for merge, another for split PDF, etc), then separately compile those into wasm and load only the wasm module for the specific task user requests
2. Slow Execution compared to native
   * When using the native binary, on one particular test (extracting the first page of a 5MiB PDF file), the processing time is only 1s, but using Node.js and wasm, it is 15s, 15X slower
   * On the Browser, it's about the same: 13-14s
   * So sometimes it may still be faster to simply upload to a powerful server for the processing, even taking the time needed for file upload/download into consideration
   * Also the client's machine may be resources contrained, and cannot process large files in the browser (the Tab will just crash if that happens)
   * But it is very likely that Browser's wasm runtime will be getting faster, and the Go compiler's wasm target backend generate better/faster code in the future
   * Right now I don't know of any profiling tool for wasm to see why it is slow, but using the Source Tab in Chrome's DevTools and clicking 'pause script execution' randomly, I notice that many times it stops at functions that (maybe?) related to Memory Allocation or Garbage Collection, perhaps when the GC support for wasm arrives in the future, things will be faster.

**Relevant Projects:**

There are many libraries that do PDF processing in browser already, some using Emscripten port of C/C++ libraries, other in pure js. If you have such need for your project, here are some examples:

* <https://github.com/DevelopingMagic/pdfassembler>
* <https://github.com/jrmuizel/qpdf.js>
* <https://github.com/manuels/unix-toolbox.js-poppler>