



allotropia

LOWA or how we managed to run LibreOffice in your browser

Stories from the journey to port a massive c++
application to WASM/emscripten



allotropia

FrOSCon, 2022-08-20

Who's talking?

- Thorsten Behrens – thorsten.behrens@allotropia.de



allotropia

Why?

LibreOffice
Free software
for all

0 1 2 3 4 5 6 7 8 9

What?



- LOWA - LibreOffice WebAssembly
- native port of LibreOffice, running client-side in the browser
 - C++, cross-compiled via emscripten
 - Using WASM / W3C since 2019
- project is funded by NLnet / Horizon 2020, and allotropia software GmbH



A screenshot of a web browser window displaying the LibreOffice Dev Writer interface. The browser's address bar shows "localhost:6931/qt_soffice.html". The LibreOffice interface includes a menu bar (File, Insert, Format, Styles, Table, Form, Tools, Window, Help), a toolbar with various icons, and a text area. The main content area displays the text "LibreOffice in the Browser ?" in a large, bold, black font. Below the text is a row of five colorful icons representing different file types: a document, a spreadsheet, a presentation, a chart, and a database. At the bottom of the page, the text "Do more - easily, quickly" is displayed in a bold, black font. The footer of the browser window shows "314 words, 1.863 characters", "Default Page Style", and "English (USA)".



allotropia

Challenges

LibreOffice
Free software
for all

0 1 2 3 4 5 6 7 8 9

Challenges



allotropia

- maturity of platform (emscripten/WASM/browsers) unclear
 - we tried it in 2015 – not even got c++ exceptions to work
- size of the problem
 - >10M LoC
 - >100 3rd party libraries
 - notorious to break tool chains (“..breaking your tools since 1990..”)
 - size of the resulting WASM binary – browsers capsize hard
 - download size (also affecting cache-ability)
- Multi-threading & GUI event handling peculiarities
 - Multi-threaded document loading and UNO IPC are disabled
- Heap size tops out at 4GB, because of WASM32 32bit pointers
- Development / Debugging environment still very “fresh”

Challenges



- Size of the resulting WASM binary

Currently: packed = 35M, optimised = 150M, debug = 200M + ~1G
separate DWARF info

- Size of the filesystem image

~ 100M with all LO fonts, can be stored locally and split if needed

→ less downloads on updates

→ can use webfonts

- embeddability & programmability

– you want to use this from your JS framework

– massively powerful API (“UNO API”), how to bind that

Current LOWA status



- <https://wiki.documentfoundation.org/Development/WASM>
 - boils down to: use master, read static/README.wasm.md
- Branch with extra features:
<https://git.libreoffice.org/core/+refs/heads/feature/wasm>
- Master can now build and run:
 - A WASM soffice.(html|wasm)*
 - A WASM vcldemo.(html|wasm) – works mostly



allotropia

The journey

Timeline



- Project kickoff:
 - October 2020
- Build env & configury & emscripten setup
 - December 2020 – cross-building of a subset works
 - Docker builders for CI available
- Get the first LibreOffice-rendered pixel on the screen
 - October 2021 – after a death march of one year...
- Get Writer practically working
 - February 2022 – also merged ~all of the changes into master
 - first fully working demo presented at FOSDEM
- Ongoing:
 - get Calc working, get headless PDF conversion & JS framework connectors running

Core ideas



- LibreOffice is an autotools & GNU make project
 - stick to that, avoid other parallel build systems
 - and its already pretty portable, cross-compilation is supported out of the box
- LibreOffice has its own GUI abstraction
 - with plugins for Gtk, Qt/KF5, Win32 and OSX
 - with Qt5 supporting WASM natively, we went with that

Core ideas



- LibreOffice is basically c++ (by and large c++17)
 - we went with emscripten as platform compiler (pinned to 2.0.31 currently)
- We don't want to use any experimental WASM features
 - no threading
 - no dynamic linking (sadly require a re-tooling of the build system)
 - no native WASM exceptions
- We wanted to focus on Writer initially (and save size by not building/shipping the rest)

Major problems



- emscripten & browser tools
 - several moving targets
 - random setups (`emsdk activate / install` not repeatable)
 - In 2020: no source-level debugging, `SharedArrayBuffer` limitations, unstable WASM impls
- LibreOffice gbuild make system w/o support for static linking
 - GNU make with a ton of `$(eval.. & $(call ..` self-made, functional build system
 - 88 commits, 4kLOC change to add that
- LibreOffice gbuild make system with dependency loops
 - UNO component system for dependency inversion
 - once we link statically, we get loops

Major problems



allotropia

- LibreOffice UNO components
 - no static dependencies, but factory & runtime resolution
 - switched to static dependency per toplevel application
- LibreOffice needs a ton of secondary files (config, fonts, gui descriptions)
 - building a virtual embedded filesystem image
- Linker problems:
 - Link time grew quadratically with symbol amount
 - at some stage took >1h and >64GB to link
 - debug build now links in ~30s; not great but manageable
 - optimized build still needs huge amounts of memory and time, but saves 25% binary size with -O2
 - always separate debug data, downloadable on demand (DWARF)

Diversion: static build



allotropia

- turns out developing for WASM was super-hard
 - long link times, huge linking memory usage
 - impossible to get decent turn-around times
- turns out debugging WASM was not efficient
 - Basically reading disassembled WASM code
- So lets use known tools on the native side
 - made a native, static, single-library build work
 - automagically include all build components
 - based on the Android + iOS static build ideas
- And get *that* to work first



allotropia

Achievements unlocked

..aka demo..

Improved WASM dev experience



allotropia

- since late 2021:
- now on Emscripten 2.0.23
 - `./emsdk install 2.0.23`
 - `./emsdk activate --embedded 2.0.23`
 - massively better build & link times
- Chrome debugging support
 - DWARF debug info included into binary
 - Much less linking time then generating huge source-maps
 - Debugging optimized code
 - <https://developer.chrome.com/blog/wasm-debugging-2020/>

Chrome Debug Setup



- DWARF debugging howto
- Install C/C++ DevTools Extension
- enable experimental WebAssembly debugging:

A screenshot of the Chrome DevTools Settings page, specifically the Experiments section. The left sidebar shows a list of settings categories: Settings, Preferences, Workspace, Experiments (highlighted), Ignore List, Devices, Throttling, Locations, and Shortcuts. The main content area is titled "Experiments" and contains a search filter box. Below the filter is a red warning message: "WARNING: These experiments could be unstable or unreliable". A list of experimental features follows, each with a checkbox and a help icon. The checkbox for "WebAssembly Debugging: Enable DWARF support" is checked and highlighted with a green oval. Other visible options include "Allow extensions to load custom stylesheets", "Capture node creation stacks", "Automatically pretty print in the Sources Panel", "Protocol Monitor", "Show CSP Violations view", "Record coverage while performance tracing", "Show option to expose internals in heap snapshots", "Source order viewer", "Timeline: event initiators", "Timeline: WebGL-based flamechart", "Console: Resolve variable names in expressions using sou", and "Emulation: Support dual screen mode".

Settings

Experiments

Filter

WARNING: These experiments could be unstable or unreliable

- Allow extensions to load custom stylesheets
- Capture node creation stacks
- Automatically pretty print in the Sources Panel
- Protocol Monitor ?
- Show CSP Violations view ?
- Record coverage while performance tracing
- Show option to expose internals in heap snapshots
- Source order viewer ?
- Timeline: event initiators
- Timeline: WebGL-based flamechart
- WebAssembly Debugging: Enable DWARF support ?
- Console: Resolve variable names in expressions using sou
- Emulation: Support dual screen mode ?

Chrome Debug Setup



- If necessary, tweak path mappings:
- Extensions → Details → Extension Options → Path substitutions
- And then serve your full source tree via `emrun` :
- `emrun --serve_root=<path>/<to>/core / instdir/program/qt_soffice.html`

A screenshot of the Chrome DevTools extension settings page for "C/C++ DevTools Support (DWARF)". The extension is currently enabled, as indicated by a blue toggle switch on the right. Below the extension name and description, there are two buttons: "Details" and "Remove". The description states: "DevTools Plugin for debugging C/C++ WebAssembly applications (using DWARF debug information). BETA version, use at your own risk."

A screenshot of the "Path substitutions" settings in Chrome DevTools. It shows two rows of input fields. The first row has "/build/tdf/libo-wasm/" in the left field and "http://localhost:6931/" in the right field, with a minus sign icon to the right. The second row has "/old/path" in the left field and "/new/path" in the right field, also with a minus sign icon. Below these fields is a button labeled "Add path substitution".



allotropia

Remaining problems

Problems still to tackle



allotropia

- No nested main loops / no blocking of the browser
 - You can run the main loop in a web worker, but then need a separate frontend.
 - Convert dialogs to async + no more Reschedule() calls.
 - `SAL_USE_SYSTEM_LOOP=1` make debugrun
 - Easy start: `grep Application::CreateMessageDialog`
 - Like commit [972aa39fb976e30ce73065b1eba69f4c78c17855](#)
 - Easy hack to reference: [tdf#146919](#)

Problems still to tackle



allotropia

- Fixed the WASM Qt backend:
 - use qtbase branch 5.15.2+wasm from [allotropia Github](#)
 - much more bugs than expected for Qt
 - upgrade to Qt6 / WASM (**not perfect yet either**)
- Alternatives:
 - port Gtk to WASM
 - but you need some kind of compositor for multiple windows..
 - use the same frontend as COOL... somehow
 - implement some “WASM-native” VCL plugin / WebGL

More problems to tackle



allotropia

- “Upload and Download” of local files
- Use browser APIs where / if possible
 - from spell-checking to ICU
- Implement persistent storage for the FS image and user files / data
- Use / Download translations + dictionaries (and keep in local storage)
- Implement a real UNO bridge, probably using WAT
- p2p document editing (as originally planned...)
- Port to a WASI
- Moonshots:
 - Switch to WASM modules AKA “dynloading”
 - Replace gbuild with Meson

Project plan



- switch focus onto JavaScript side:
 - GUI & embedding
 - sample code for VueJS
 - access to full UNO API (webIDL or embind) – at least cmd & callbacks & state checks need to wok
 - use browser APIs wherever possible
- usable HTML widget for rich text editing by Q4
- headless conversion (PDF & more)
- get Calc into shape

What to expect (and our vision for LOWA)



- **not** a replacement for desktop/mobile LibreOffice, or Collabora Online
- instead serving unmet needs:
 - your platform is the browser? here's your everything-works text widget!
 - require privacy-by-default, or end2end encryption? here's your no-data-ever-goes-to-any-server solution!
 - want planetary-scale for your product, but lack GAFAM's number of data centers? here's something that scales like a static website!
- want to play yourself? have a look, demo setup here:

<https://lab.allotropia.de/wasm>



allotropia

Questions & Answers

