Hako Technical Specification

Authors: Niall Ryan (21454746), Cathal O'Grady (21442084)

Supervisor: Prof. Stephen Blott

2025-05-02

Contents

1 Motivation

Modern programming education often prioritises ease of entry at the cost of system-level understanding. While abstraction helps to reduce the initial learning curve, it can leave learners with a fragmented mental model of how software interacts with the underlying machine. As a result, progressing beyond beginner-level programming often requires a difficult transition – one that demands unlearning oversimplifications and grappling with the complexity of real world systems.

We aim to address this gap with Hako, a platform-agnostic, unix-inspired educational environment designed to foster systems literacy from the outset. Hako is a client-side web application that simulates a light-weight Unix-like environment, complete with a simplified shell, terminal, file system, and graphical interface. Rather than shielding learners from complexity, it exposes them to a coherent, minimal system where every part is programmable and accessible. This approach emphasises code as a means of manipulating and understanding the system itself, rather than treating it as an isolated task.

Our design is constrained on purpose: tools are intentionally minimal, APIs are exposed but trimmed, and the system encourages exploration and modification. By simplifying without obscuring, we allow users – particularly younger learners – to develop a mental model of computing rooted in real systems behaviours. Hako is not just an IDE; it is an educational operating environment where the act of programming becomes inseperable from the system being programmed.

By lowering the barrier to understanding rather than hiding it, Hako aspires to teach programming as an act of systematic engagement – encouraging users to see code not just as syntax, but as a tool for mastering the systems they inhabit.

2 Research

Research for this project involved a mix of reading about potential technologies that we could use as well as prototyping things in them to see if they are viable for our particular use case. This meant that initially there was a lot of discussion amongst ourselves and sharing of findings and prototypes.

At first we didn't really have much of an idea of what we could use to solve the project we were proposing so we had a vague set of exploratory topics that we looked into. Some of the key problems that we focused our research into early on were:

- Window management/OS Desktop environment experience on the Web
- Persistent storage on the Web
- Execution environment on the Web

A lot of theses problems were further constrained by our **commitment to being fully client-side**, so as to never touch any data of the users.

The research into theses problems is dicussed in the following text. Note that these problems were only our initial research before we had started actually commiting work and implementing the features of the project. Due to the nature of the project and our expertise, we were constantly learning, researching and adapting as new requirements presented themselves.

2.1 Window Management

Initially a lot of concern with providing a desktop like experience on the Web was focused on the performance and responsiveness. This meant that we did research into various different solutions to rendering on the Web, like using canvas API, WebGL and WebGPU. We explored different options, including immediate mode ui libraries like imgui and egui. These libraries were great and provided a very responsive interface, however they were not very flexible so we were some what constrained in how we could design the look and feel of Hako – which we did not want to sacrifice on.

Additionally we explored what it would take to build our own UI library in OpenGL ES with C compiled down to WebAssembly using Emscripten to render on a canvas, prototyping basic window drawing and text rendering. It did not take long to realise that this would not be feasible or practical as it would take our focus away from the more important aspects of the project.

We also prototyped a solution using plain html, css and javascript. These were sufficient to get a prototype window mangement system implemented, however we were concerned in a similar manner to the previous exploration that it would be a lot of implementation burder on us, especially anticipating a very interactive user interface.

We eventually decided to use a Web framework and since we did not have a lot of UI experience, Svelte was very attractive as it mostly felt like writing regular html and css, just with convenient state management features and signals.

2.2 Persistent Storage

One of the challenges Hako's specification initially provided was to persist information between sessions.

We explored options such as:

- **Filesystem Access API** – Sandboxed, user consented, easily manage hierarchical filesystem on user disk (Only fully supported by chromium)

- **IndexedDB** – Files are stored in the browser's disk environment, no user consent required, more complex (Full support by all browsers - Safari support uncertain)
- **Emscripten Based Filesystem in Web Assembly** – maximum flexibility along with performance (Just proxying indexedDB via Emscripten)

We didn't want to restrict ourselves to only being compatible with Chromium-based browsers, so we decided against the Filesystem Access API.

Furthermore, interacting with IndexedDB directly is quite verbose, and not easy – but was an option available to us, and a better choice than the Filesystem Access API.

Eventually, through experimentation, we decided that the Emscripten based filesystem in Web Assembly provided us the best approach to implementing a filesystem.

2.3 Execution environment

Initial focus on the execution environment was more generally how we could execute arbitrary code entirely on the client, without any server providing the execution platform. We saw that there was some options for emulating linux in browser, particularly v86 which translates x86 assembly into WebAssembly, while this was a great project, the technology was daunting and we were not sure whether we would be able to get it work for our use case effectively.

Along with simply trying to compile different software to WebAssembly, notably Lua, we also looked into the various web-based libc solutions, like WASI, WASIX, Wasmer and Emscripten. The WASI standard was too limited, WASIX was under-documented and didn't seem like a well supported project and Wasmer was under-supported and more focused on providing one-off ports to various applications, rather than providing a target platform. This made us settle with Emscripten, because it was the most mature and flexible of the bunch and we were able to get an initial prototype Lua REPL running in the browser with relative ease when compared with the other solutions.

In fact, while we did have Lua in mind as a language for the platform, it was the ease in which it was able to be compiled for the Web which made it an even more attractive choice along with its projected simplicity of embedding inside of larger projects that we had heard stories about online (e.g. Roblox, Neovim and Pandoc).

3 Software Process

Hako's software process followed a mix of methods and processes, primarily **Lean** and **Agile**.

3.1 Project Management

For **workflow** and **project** management, we utilised **Kanban**.

Kanban was appropriate in this case, as the only stakeholders in this project are the developers themselves, so it made sense to follow the lean mentality, eliminating waste and focusing solely on delivering value for the fourth year project.

Kanban was envisioned via Jira, the issue and project tracking software.

A sample view into our backlog

We created three "Epics":

- **Design** – Issues and tasks specifically for designing the project, typically explorative
- **Develop** – Implementing core features, functionality and bug-fixes for Hako's MVP
- **Scope Creep** – Implementing extended features, "nice-to-haves", or items not stated in the functional specification

An example "Epic" can be seen below.

Both developers would occasionally meet for backlog refinement and grooming, but backlog items would also naturally form as they were encountered – as is intended with Lean software develop-

ment.

There were three colums for our Kanban approach:

- **TODO** – Items that are waiting to be actioned.
- **IN PROGRESS** – Items currently being actioned.
- **DONE** – The item has been completed.

Backlog items would additionally be defined with:

- **DoD** (Definition of Done) – clear exit criteria to prevent ticket creep
- **Priority** – How urgent an item is, to prioritise actioning important items
- **Work Size Estimate** – How big a piece of work is estimated to be
- **Epic** – Either 'Design', 'Develop' or 'Scope Creep' (prioritised in that order)

Developers would then pick up a ticket when finished with their current one.

> Story points and priority for this are offscreen, but were '8' and 'Medium' respectively

Below, you can see a cumulative issue flow graph, showing we were **working from September 2024 to May 2025**.
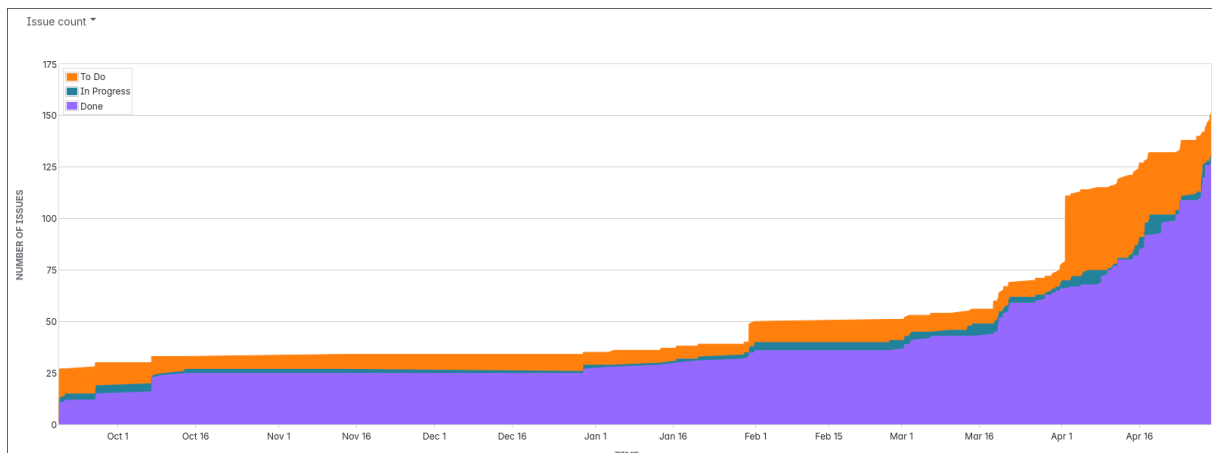
3.2 Engineering Practices

Hako's engineering practices bore heavy influence from Extreme Programming (XP), amongst other common Agile practices, with us taking upon:

- **Continuous Integration** – A full suite of tests running on new code ensuring code quality.
- **Continuous Deployment** – Once all tests pass, new code is **automatically** deployed.
- **Staging** – Having separate development and production branches (`dev` and `main` respectively) with deployment only on the former.
- **Pair Programming** – Both developers working together to solve difficult problems.
- **Rapid Prototyping** – Prototyping to mitigate risk and better understand technology / problems.
- **Collective Ownership** – Shared responsibility of all code, often patching eachother's assigned components.
- **Refactoring** – Reworking "smelly" code to improve its maintainability and extensibility.
- **Code Reviews** – Reviewing eachother's code in a Merge Request before merging to the development branch.

3.2.1 CI/CD

Hako used extensive CI/CD, with a Gitlab pipeline.
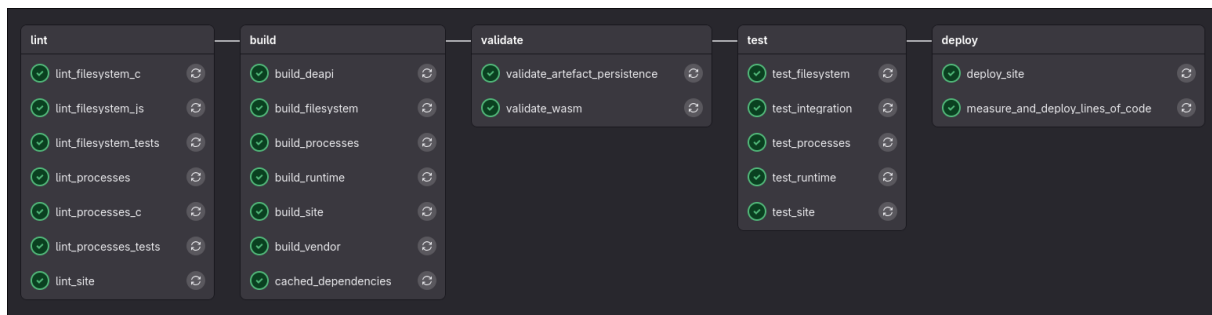
Our pipeline stages included:

- **Lint** – Ensuring code quality, static code analysis and early bug detection.
- **Build** – Building the entire project, aggressively caching 3rd party dependencies and vendors.
- **Validate** – Ensuring artefacts exist for testing, and WASM files are valid.
- **Test** – All testing, including unit, integration and e2e tests.
- **Deploy** – Only on `main`, for deploying to our own self-hosted server.

Build **aggresively** caches vendor dependencies as they are very large. This is handled specifically in the `build_vendor` job. `cached_dependencies` specifically caches 3rd party build dependencies, such as Just (similar to Make), the task manager we used - as we encountered rate limiting. (see Building for more details)

The Deploy stage will only run on the `main` branch, once the Test stage succesfully passes.

3.2.2 Staging

Staging was performed **throughout the entire development** of Hako.

Every Kanban item would have a single dedicated branch, which would be branched off of the dev branch.

Whenever the developers had decided they'd reached a satisfiable increment of features in Hako, they would then merge `dev` into `main`, triggering an auto-deploy if all tests passed.

3.2.3 Pair Programming

Pair programming was used often, typically for especially difficult problems that benefited from the oversight and combined skills of both developers.

Another often usecase of pair programming was when complicated regressions were introduced, or more difficult issues such as race conditions, heisenbuges, and more.

3.2.4 Rapid Prototyping

Hako is comprised of technologies and tools that the developers have absolutely no experience with, such as:

- **Web Assembly** – A low-level byte code instruction format for the web.
- **Emscripten** – A WebAssembly compiler and web-based libc implementation.
- **IndexedDB** – Persistent, site-local storage in the web.
- **Web Workers** – Threads of which the browser owns.

This is by no means an exhaustive list. There was an substantial amount of research performed (See Research section for more details), and problems that were come across (See Problems section for more details).

Due to this, we performed rapid prototyping often to mitigate risk; see what was possible, the constraints of the technologies we were dealing with, and whether our designs would work.

3.2.5 Collective Ownership

As pair programming and code inspection/review were common place, both developers became comfortable in the entire codebase, often patching eachothers code and assigned components.

3.2.6 Refactoring

Due to the unfamiliarity of the technologies involved, and regular rapid prototyping, technical debt was often introduced into the codebase.

Refactoring, when beneficial, was actioned. Due to the lean nature of the development of Hako, refactoring was only performed when crucial – to avoid slowing down of development.

3.2.7 Code Reviews

To ensure code quality and the collective ownership of the codebase, **all merges performed by a developer were assessed by the other**.

An example "Merge Request" is listed below.

More discussion is involved in this ticket below which can't be captured in a single screenshot

These would quite often involve change requests, inquiries into implementations and approaches – and increased the quality of our work in an efficient and valuable manner.

4 Design

4.1 Module Overview

Below is a diagram of the main dependencies between the modules of the system.
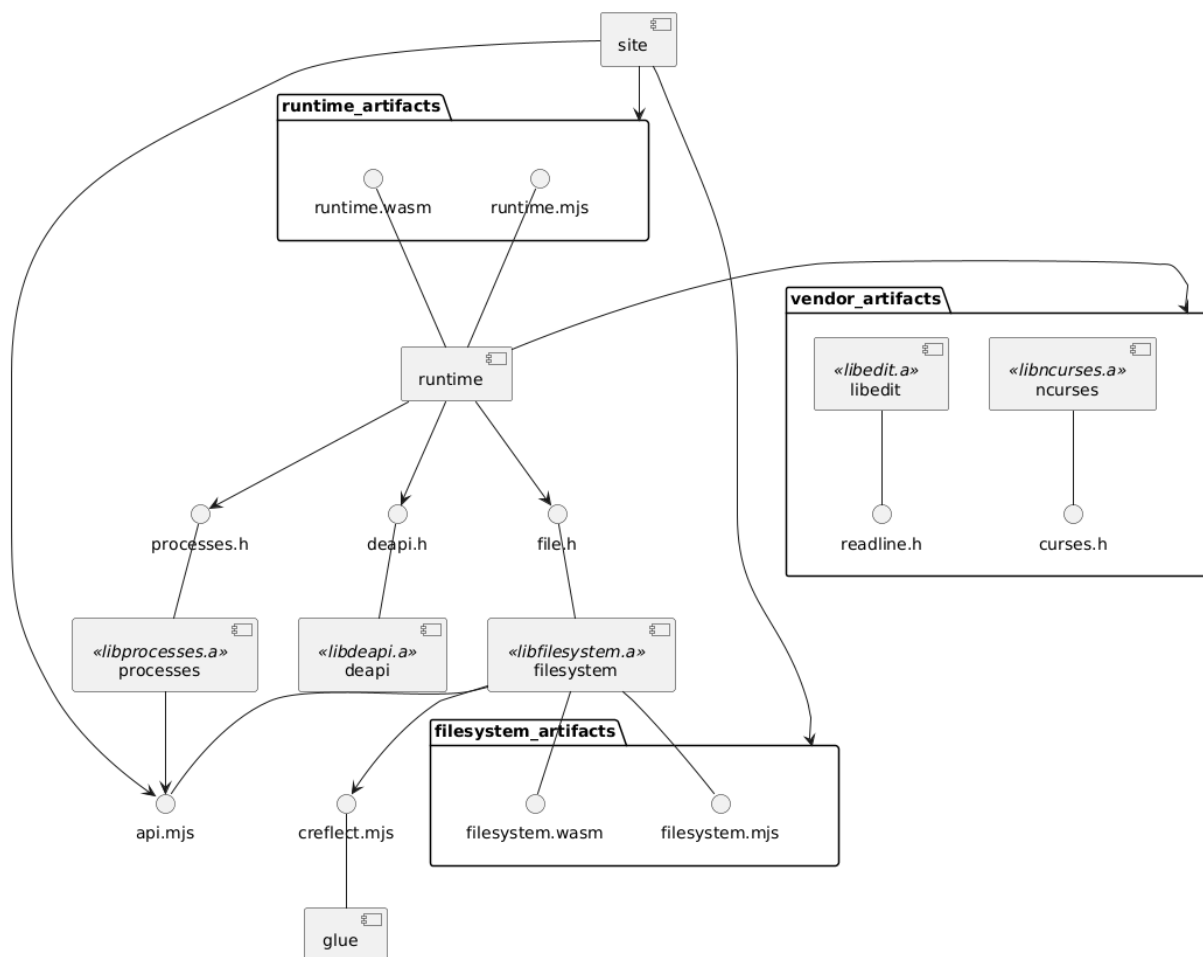
The system has the following core modules:

- **filesystem** – Filesystem C library + JS shims
- **processes** – Process management module
- **runtime** – Lua based wasm runtime for unix like environment
- **glue** – Runtime C type reflection for JS
- **deapi** – Desktop Environment API
- **vendor** – Third party libraries we custom build for wasm environment
- **site** – The website frontend component

The filesystem, deapi, processes and vendor artifacts are all built into archive files (.a) which are stat-

ically linked into our runtime. Some components, also produce artifacts that are used by the site directly. In particular, the filesystem produces a WebAssembly module which is instanciated by the website. This is because the website accesses the filesystem through a JS API (`src`/`filesystem`/ `src`/`api.mjs`), whereas the runtime accesses the filesystem C API directly. The runtime also produces a WebAssembly module which is used by the process module JS code to instanciate workers for new runtime instances (whenever you open a terminal).

4.2 Platform API

Since Hako is a platform for learning programming, there is a platform API which you use from within the website to program it. This provides the following modules:

- **file** – Provides file manipulation functionality, creating and deleting files and directories, etc
- **errors** – Provides error introspection as well as helpers to handle errors
- **process** – Provides process specific functionality, including spawning subprocesses, pipeing processes and I/O
- **terminal** – Provides terminal specific functionality, like clearing, readline-like support and querying the size of a terminal
- **window** – Provides functionality for manipulating the windows of the website, creating new windows, moving them and resizing them

There are multiple sources you can read to see all of the individual subroutines, like the "User Manual", the generated docs on the website, the `src`/`runtime`/`doc`/`api.lua` file or running the `help` command in the terminal of the system.

4.3 User Interface

The user interface was indirectly inspired by all of the operating systems that the authors of Hako have used. However especial inspiration was from Chrome OS, which has a very intuitive design language and has been found to be very easy for young people to pick up when compared other major operating systems like Window and MacOS.

The overall design of Hako's user interface is simple, you have a window area and task bar. The window area is where you can operate on your windows, and the task bar is used to manage applications.

Inspiration was also taken from Linux, in particular the Gnome desktop environment. The icons used in the system are taken from the adwaita icon theme, and the file manager and text editor icons are taken from the Gnome applications, gnome-terminal, nautilus and gnome-text-editor respectively.

The color theme for website is generated via the tool, Material Theme Builder. This was very useful as it kept the color language consistent, and allows for easy substitution of different themes in the future.

To make Hako look the same on all devices, it ships with its own default fonts. In particular Fira Code for monospace font (in terminal and text editor) and Fira Sans for everything else.

4.4 Architecture Diagram

The following diagrams show the various components of the system, building upon other components, and their respective languages.

In red are third party components we brought in. In white, are components we wrote entirely ourselves from scratch.

4.4.1 Site (Main Thread) Abstraction Diagram
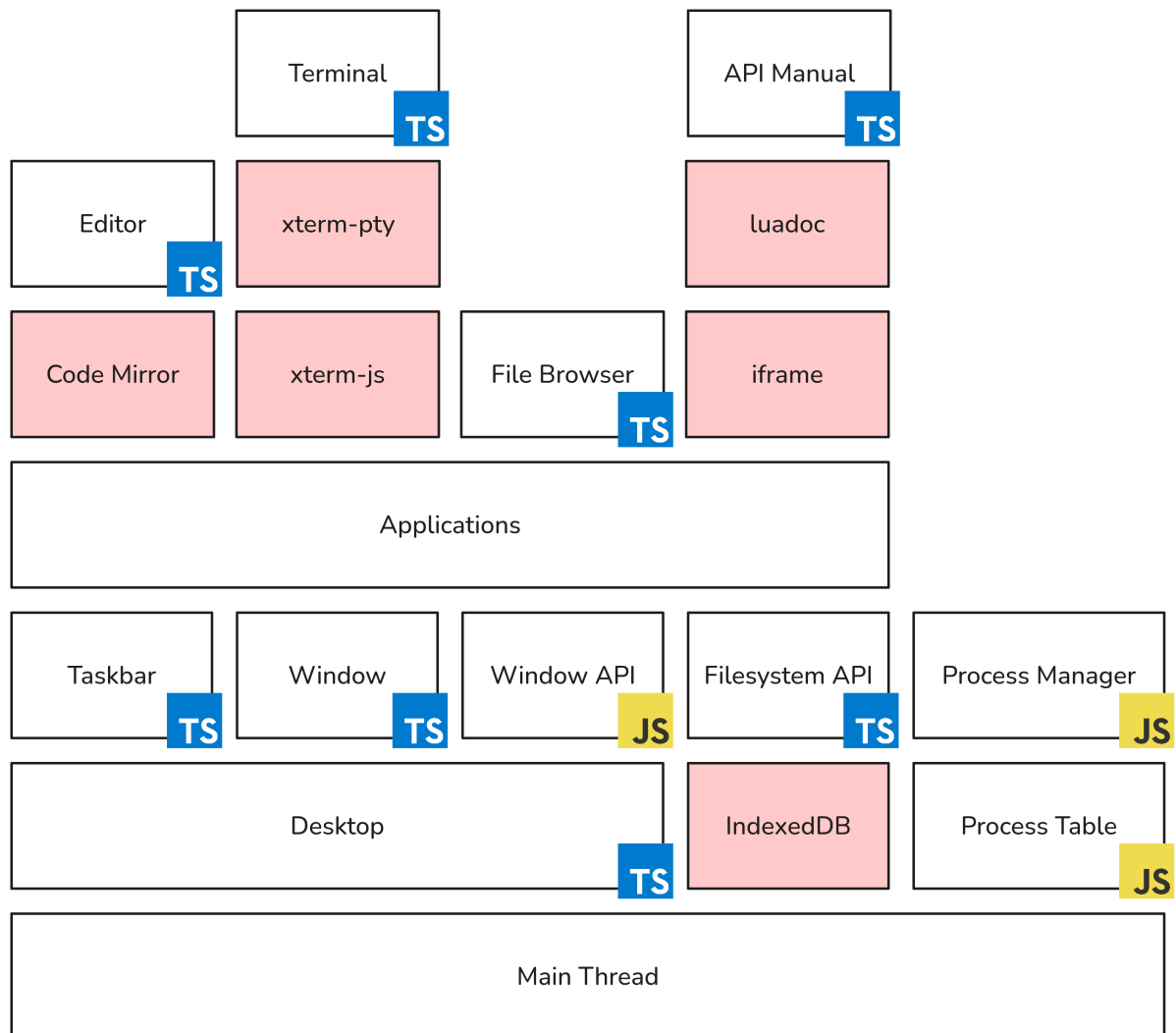
4.4.2 Process (Web Worker) Abstraction Diagram
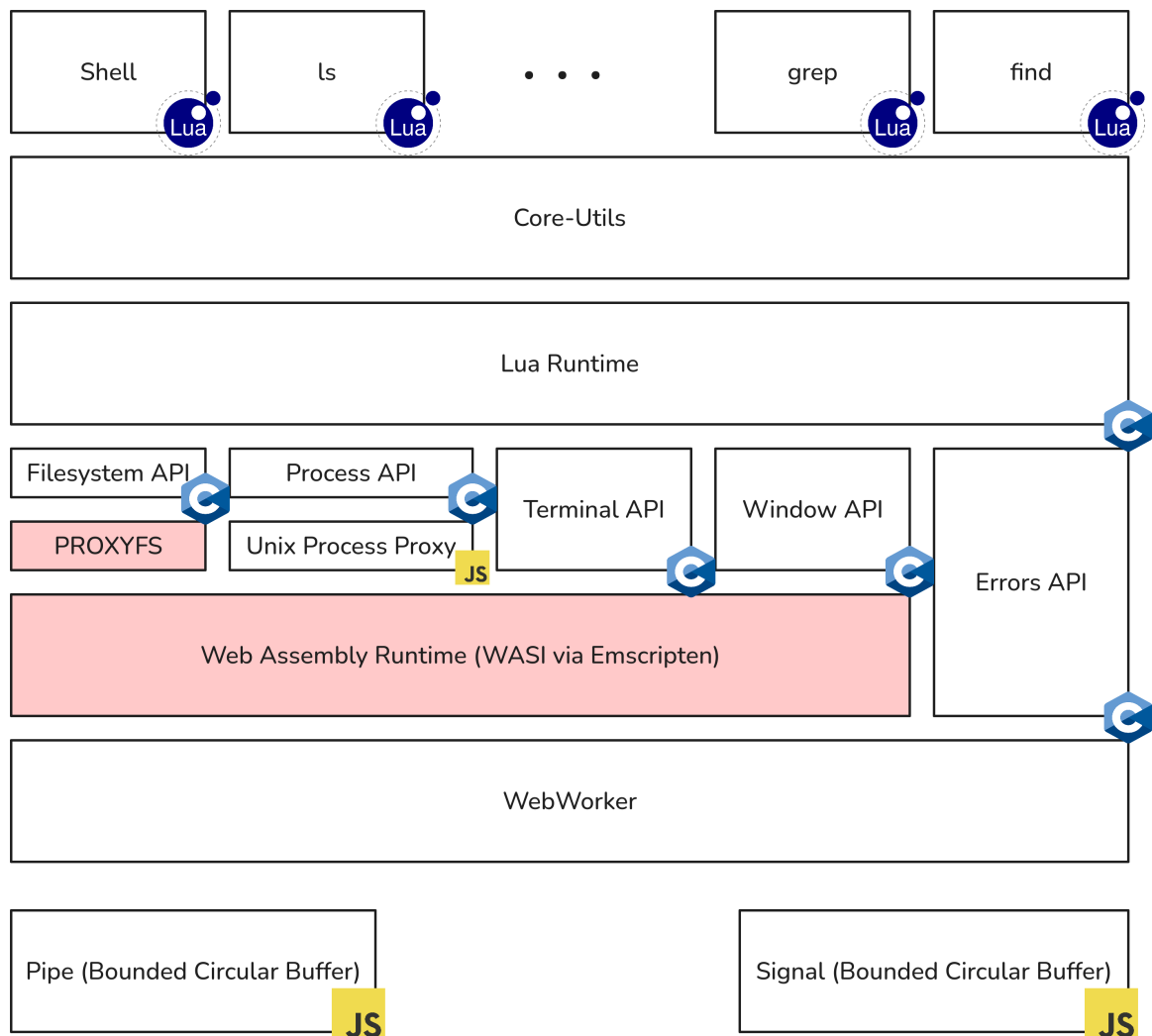
5 Implementation

5.1 Building

The build for this project is particularly unique, due to the mix of technology being atypical. Since the project is a website all of the C code needs to be compiled into WebAssembly. The project uses Meson as the main build system for C code. Configuring Meson to compile code to WebAssembly is done by using a cross-compilation file, which describes the platform and compiler toolchain (i.e. CPU architecture (WASM), endianess, c compiler (Emcc), etc), you will find this file at `src/emscripten.ini`. As mentioned in Module Overview, the project is split up into modules which provide different

archive files. Each of these modules have separate `meson.build` files – these just describe how to build each module to meson.

Given that the modules are compiled separately, a higher-level task runner, Just, is used to build the entire site. This allows us to build all of the module artifacts separately and copy them into the site's **static**/ folder – which is not part of the source code, and is used for the particular case of build artifacts in the project. The file describing the higher level orchastrated build for Just is at the root of the source repository named `justfile`.

There is an exception to using Meson for building the C components of the project. When building our "vendor" dependencies, that is dependencies of which the source code was not written by the authors of this project, the Emscripten compiler toolchain is used directly due to the fact that the vendor dependencies like Libedit and NCurses have their own build systems and are not directly able to be built with meson (unlike Lua, whos build can be easily described by a meson wrap file).

For building and developing on the project, the following dependencies are needed:

- Emscripten
- Just
- GNUPG
- Nodejs
- Meson
- Ninja
- Ldoc

To simplify things, the project has a `flake.nix` file which provides a dev shell with all of the dependencies provided. This way you only need to install Nix, and run the following command in the root of the repository to be able to run our build:

```
1  nix develop
```

Then you can build the project by running:

```
1  just
```

If you are trying to deploy the website yourself, you might find the `Containerfile` useful. This can be used to build a container image, using a tool like Docker. Building the image can be done as follows, from the root of the repository (using Docker):

```
1  docker build -f Containerfile -t hakob .
```

Then you can run the container like so:

```
1  docker run -it -p 8000:80 --rm --name hakob localhost/hakob
```

This will host the website on port 8000.

5.2 Website (module `site`)

The website is written in Svelte-Kit with Typescript. The website's core purpose is to provide the UI components of the system. This includes the Window Manager, File Manager, Text Editor, Terminal and Desktop.

5.2.1 Window Manager

Since window management has to maintain a lot of state without re-rendering, windows are created as imperative components and managed manually. Manual management of window lifetimes (opening, closing, hiding and showing) is done through the API in `src/site/src/lib/windows.svelte.js`. This file stores the array of windows, keeps track of the order of the windows (their z-indices) and also provides an abstraction for creating instances of specific application windows (e.g. Terminal).

The actual logic for the interactive feature of the windows resides in `src/site/src/components/Window.svelte` however. This includes crutial actions like moving, resizing and exposing the lifecycle functionality of `windows.svelte.js` to the user.

Moving is implemented using custom drag and drop logic, this was done to optimise the performance for the specific use case of dragging objects to arbitrary regions on the screen. You can see below for reference of how dragging is handled. Updating the window position is put in a `requestAnimationFrame`, so that we syncronize with the next frame. Updating of the position is also throttled such that if we have a currently running update, we just increment the change in position, to be updated in some future frame. The callback throttling code was gotten from a blog by Nolan Lawson, which is a great resource for further reading on this topic.

```
1   let deltaX: number = 0;
2   let deltaY: number = 0;
3
4   function onDragWindow(ev: PointerEvent) {
5     deltaX += ev.screenX - (lastX ?? ev.screenX);
6     deltaY += ev.screenY - (lastY ?? ev.screenY);
7     throttleWrite(() => {
8       ctx.position.x += deltaX;
9       ctx.position.y += deltaY;
10      deltaX = 0;
11      deltaY = 0;
12      updatePosition();
13    })
14    lastX = ev.screenX;
```

```
15        lastY = ev.screenY;
16      }
17
18      function onHoldDecorations() {
19        if (!maximized) {
20          document.addEventListener("pointermove", onDragWindow);
21          document.addEventListener("pointerup", onReleaseDecorations);
22          overlay.toggleGrab();
23          onMove?.();
24        }
25      }
26
27      function onReleaseDecorations() {
28        document.removeEventListener("pointermove", onDragWindow);
29        document.removeEventListener("pointerup", onReleaseDecorations);
30        overlay.toggleGrab();
31        onStop?.();
32        lastX = undefined;
33        lastY = undefined;
34      }
```

Resizing is implemented by having the parent div of the window capture pointer events that are directly raised on it. Then a direct child has a margin, such that only a gap of this size defines the area in which pointer events are handled. You can see below the main logic for handling a pointer event that is determined as resizing. We calculate the delta position, update the size based on the section of the window the cursor is in and also addtionally update the window position for the cases when you are resizing from the top and left sides of the window.

```
1      function onDragResize(ev: MouseEvent) {
2        const moveX = ev.screenX - (lastX ?? ev.screenX);
3        const moveY = ev.screenY - (lastY ?? ev.screenY);
4
5        const [ dw, dh ] = lib.getResizeFromSect(globalSect, moveX, moveY);
6        updateInnerSize(dw, dh);
7
8        let dy = 0;
9        let dx = 0;
10
11       switch (globalSect) {
12         case lib.TOP_LEFT_CORNER:
13           dy = moveY;
14           dx = moveX;
15         case lib.TOP:
16         case lib.TOP_RIGHT_CORNER:
17           dy = moveY;
18           break;
19         case lib.LEFT:
20         case lib.BOTTOM_LEFT_CORNER:
21           dx = moveX;
```

```
22          break;
23      }
24
25      ctx.position.y = Math.max(-eventBorder, Math.min(ctx.position.y +
           dy, maxY));
26      ctx.position.x = Math.max(-eventBorder, Math.min(ctx.position.x +
           dx, maxX));
27      requestAnimationFrame(() => {
28        updatePosition();
29      })
30
31      lastX = ev.screenX;
32      lastY = ev.screenY;
33    }
```

5.2.2 File Manager

The file manager exposes a subset of the functionality of our JS filesystem API into an interactive user interface. The operations of the file manager are as follows:

- Create/Delete Files/Directories
- Rename Files/Directories
- Navigate filesystem (via clicking on folders or using the breadcrumbs)
- Moving Files/Directories

The code for the file manager can be found at `src`/`site`/`src`/`components`/`FileManager`.`svelte`.

Due to the fact that in JS, the filesystem is not mounted through a proxy, you cannot use functionality that changes the directory, as it would change the directory globally amongst all file manager instances. This is why you will note a `FSView` object being used in the code, it just simulates changing directories.

You will also note that a broadcast channel named "inotify" is used. This is the channel that the runtime posts messages on whenever file operations happen. This allows the file manager to receive live updates of newly created files from a runtime context and update the UI.

All file system manipulation operations are provided through a context menu. If you right click on a file or directory you get respective operations on them, whereas clicking elsewhere gives you a menu allowing you to create a file or directory.

For file moving specifically in contrast to the window manager the builtin browser drag and drop API is used, as its limitations were in line with the constraints of the file manager and it allows the maintenance burden to be offset onto the browser. The downside however is that despite it being a standard

browser API, it is often considered very difficult to use due to some of its strange behaviour. You can get a taste of this by the fact that for handling the drag enter and drag leave events you have to keep a counter as the browser will raise drag leave events if the dragged item moves between children of the drag target, you can see this in the code snippet below.

```
1    // This is needed due to how broken the DND html 5 api works.
2    // If you drag over a child element it raises a drag leave event for
3    // the parent. We just keep a reference counter to make sure we really
4    // did leave the parent element.
5    let counter = 0;
6
7    function onDragEnter<T extends EventTarget>(index: number):
         DragEventHandler<T> {
8      return (_ev) => {
9        const filepath = fileRefs[index].dataset.filepath;
10       const filetype = fileRefs[index].dataset.filetype;
11       const pathParts = filepath.split("/").filter(s => s.length !== 0)
            ;
12
13       if (filetype === "directory") {
14         counter++;
15         setDropZoneStyles(index);
16         if (timer !== undefined) clearTimeout(timer);
17         timer = setTimeout(() => {
18           fsView.changeDirAbs(pathParts);
19           // treat this as a sort of drag leave, reset counter and
                remove styles
20           counter = 0;
21           removeDropZoneStyles(index);
22         }, enterDirDelay);
23       }
24     }
25   }
26
27   function onDragLeave<T extends EventTarget>(index: number):
         DragEventHandler<T> {
28     return (_ev) => {
29       const filetype = fileRefs[index].dataset.filetype;
30
31       if (filetype === "directory") {
32         if (counter !== 0) counter--; // make sure we never get
                negative counter
33         if (counter === 0) {
34           removeDropZoneStyles(index);
35           if (timer !== undefined) clearTimeout(timer);
36           timer = undefined;
37         }
38       }
39     }
```

```
40        }
```

Addtionally the file manager has a special mode of operation as a file dialog. This is specifically used by the text editor, when opened directly, to allow the user to use the file manager and select a file to be opened. This mode is implemeneted by constructing a javascript promise on the editor which shares its resolve and reject functions with a new instance of the file manager. These functions are used to indicate a file selected by the user, or that the dialog was cancelled. Note also that this greys the window area and disables input, only bringing the file manager to the foreground (see Desktop for more details on the overlay).

5.2.3 Text Editor

For the text editor, CodeMirror is used, due to how modular it is. You can find the code that integrates it at `src`/`site`/`components`/`Editor`.`svelte`. The integration is mostly enabling the features of CodeMirror that we want: editing keybinds, history, search and syntax highlighting. The autosave functionality is implemented by debouncing "change"-like events raised by the editor and saving to the systems filesystem.

5.2.4 Terminal

The terminal is implemented using the following libraries: Xterm.js, Xterm-PTY. Xterm.js provides a primitive terminal interface, whereas Xterm-pty provides a PTY layer for Emscripten to use. A PTY allows us to have more advanced terminal features, as it handles things like line buffering, the mode of operation of the terminal as well as providing Termios for Emscripten programs - both of which are typically provided by the operating system, however being in the web means we have to provide these ourselves. Effectively we need this to support Libedit via Ncurses for interactive shell features (see Runtime for more details).

Note also that the pty provided by Xterm-PTY is also used as a handle to give terminal access to the runtime. This allows sub processes to share the same output as the process that created them. Essentially the slave side of the PTY is passed at creation of a process, allowing it to read input from the terminal, or produce output on the terminal. If you do not pass a PTY slave, the process must be setup to read standard input and write standard output on a pipe.

One unfortunate downside to using Xterm-pty, apart from the fact that the library is not very heavily tested (see Problems Solved for more details), is that it forces us to compile our runtime with specific Emscripten features like proxying the main thread into a worker via specifically the `PROXY_TO_PTHREAD` flag, i.e. you cant use other mechanisms of running the module in a worker.

5.2.5 Documentation

The documentation application of the website is just an iframe which holds generated documentation site using Ldoc. The actual input to this generator is in the runtime module under `src`/`runtime`/ `docs`/`api`.`lua`. This Lua file just describes each of the platform API functions as well as their signatures and types. While the docs are generated using Ldoc, the format used in the `api`.`lua` file is a superset which supports more rich features like describing custom types, these are not represented in the generated docs unfortunately. However the extra type information does make the experience of developing Lua files using the API better with a LSP. Note however that no LSP client or server are shipped with Hako itself.

5.2.6 Desktop

When refering to the "Desktop", it is in reference to not only the high level component which lays out the top level components of the website (window area, taskbar), but also the miscellaneous interactive features of the website, such as dialog menus, context menus and the input overlay.

The window area, is just a container for the absolute positioned windows, it is just a place in which the imperative window components are mounted as well as the bounds to which they are maximised in. You can find the code for this in `src`/`site`/`src`/`components`/`Desktop`.`svelte`. Because of this dedicated area, maximisation is implemented by essentially changing the window from `postion`: `absolute` to `position`: `relative`.

The taskbar is a component which allows the user to open, close, show and hide applications. It is implemented in `src`/`site`/`src`/`components`/`TaskBar`.`svelte`. More specifically, in contrast to the buttons on the window component itself, which apply to that specific window, the taskbar provides more global functionality, like "show all windows", "hide all windows" and "close all windows" for each application type. Additionally it allows the user to cycle the focus of the different open instances of a given application, or swap focus between two different applications. The menu for said operations uses a context menu like in the file manager.

There are two main dialogs in the system; text dialog and alert dialog. Both dialogs use a generic popup menu implemented in `src`/`site`/`src`/`components`/`Popup`.`svelte`, and are respectively implemented at `src`/`site`/`src`/`components`/`InputText`.`svelte` and `src`/`site` /`src`/`components`/`Alert`.`svelte`. The text dialog is used by the file manager to create or rename files and directories. The alert dialog is also used by the file manager to indicate any errors raised by filesystem operations and the text editor when opened directly – prompting the user whether they want to select a file.

The "input overlay" is container which spans across the entire screen at the highest z-index. Its code can be found at `src`/`site`/`src`/`components`/`Overlay`.`svelte`. This is used by the website

to disable input events. For example, the overlay is used at initialisation to disable all website input when showing the loading screen. It is also used by the window manager to disable all local input, i.e. any input not on the root document element, so that we can prevent things like accidental selection while moving windows – not perfect but does prevent a lot of cases.

5.3 Runtime (module `runtime`)

The Hako platform API is provided as a Lua API. In fact, this API is exactly what is used to implement the shell and the core utilities in Hako. This was done to ensure that the platform API was sufficient in producing programs of reasonable complexity for learning. Lua was used for its ease of embedding in other code, making it a perfect fit for Hako as one can provide functionality from various different sources, and expose it in a transparent way to the user.

The runtime module is a C program that exposes all of the APIs of Hako to the end user. The Lua interpreter is statically linked into the runtime module and is used to execute Lua code from within. The intialisation of the runtime is broken into the following steps (code can be found at `src/runtime/src/main.c`):

1. Desktop Environment API (`deapi`) is initialised – this is needed to setup up an Emscripten code proxy queue to be able to run main thread code from the worker.
2. Setup Lua state and expose selected parts of the standard library.
3. Export platform API functions implemented in C to be called from Lua.
4. Set the `argv` global for the passed commandline arguments to the given runtime instance (process).
5. Emscripten proxy file system setup.
6. Environment Setup (e.g. current working directory)
7. Expose the inspect function – really useful for creating human readable string reprentations of Lua types, should really be part of the Lua standard library.
8. Set environment variables and configure libedit.
9. Execute the Lua code provided to the current runtime instance.

Hako only exposes a selection of modules of the Lua standard library: `string`, `table`, `math` and `debug`. Additionally Hako explicitly disables globals which would access I/O or the filesystem directly, as these are provided by the virtual operating system of Hako itself by other means (i.e. `file` and `process` API's). You can see in the below snippet how the runtime disables these selected globals.

```
1  static const char *excluded_globals[] = {"print", "dofile", "load", "
      loadfile"};
2  void exclude_globals(lua_State *L) {
```

```
3    for (size_t i = 0; i < sizeof(excluded_globals) / sizeof(const char
          *); i++) {
4      lua_pushnil(L);
5      lua_setglobal(L, excluded_globals[i]);
6    }
7  }
```

Additionally the runtime exposes two specific functions from the Lua standard library `os` module, `os`.`time` and `os.date`, however they are put under a custom namespace, `fmt`.

The C functions that are exposed to Lua are shims which provide our C API functions in an interface/-convention that Lua can understand. For example you can see in the following snippets how one of these shims is implemented:

```
1  int lfile__read_dir(lua_State *L) {
2    char **entries = NULL;
3    char *fpath = NULL;
4
5    lua_settop(L, 2);
6    const char *path = luaL_checkstring(L, 1);
7
8    fpath = fake_path(path);
9    if (fpath == NULL) {
10     lua_pushnil(L);
11     lua_pushnumber(L, E_DOESNTEXIST);
12     goto cleanup;
13   }
14
15   Error err = 0;
16   lua_newtable(L);
17
18   entries = file__read_dir(fpath, &err);
19   if (err != 0) {
20     lua_pushnil(L);
21     lua_pushnumber(L, err);
22     goto cleanup;
23   }
24
25   int idx = 0;
26   while (*(entries + idx) != NULL) {
27     char *entry = *(entries + idx);
28     lua_pushstring(L, entry);
29     free(entry);
30     lua_pushnumber(L, idx + 1);
31     lua_insert(L, -2);
32     lua_settable(L, -3);
33     idx++;
34   }
35
36   lua_pushnil(L);
```

```
37  cleanup:
38    free(entries);
39    free(fpath);
40    return 2;
41  }
```

You can see how the function internally calls our C filesystem API function `file__read_dir`, but also has to collect all of the entries of it's result into a Lua table (datastructure used in Lua for arrays as well as hash maps). You will also notice a call to `fake_path`, this normalises the path and ensures that it is called under the false root setup for persistence at `/persistent` (See False Root for more information). This just ensures that Lua code cannot escape this path. You can find the code for `fake_path` in `src`/`runtime`/`src`/`shared`.`c`, it is essentially a stack based path normalization implementation, which at the end prefixes the absolute path with `/persistent`.

Note also that for persistence inside of the runtime, each instance sets up a PROXYFS which forwards requests to the actual persistent filesystem which runs in a different emscripten module in a separate worker (see Sharing the Persistent Filesystem in the problems section for more details).

5.4 Process System (module `processes`)

Hako's process system is built upon:

- Web Workers – Threads the browser spawns and owns
- SharedArrayBuffers – Shared memory for inter-process communication which also allows for thread synchronisation
- Message Passing – Asynchronous, event-based communication between threads

There are five primary components in Hako's process system:

- **Process** – A Web Worker created by Emscripten (imitating pthreads) and intercepted by us to add additional features.
- **Signal** – Shared memory (SharedArrayBuffer) bounded circular array buffer between the main thread and process Web Worker to allow for synchronisation and syscall result returning.
- **Pipe** – Shared memory (SharedArrayBuffer) bounded circular array buffer between multiple processes for piping input and output.
- **Process Manager** – A process orchestrator that resides on the main thread and manages the lifetime of processes, and serves functions of the Process API.
- **Process Table** – A flat array of process information indexed by process identifiers (PIDs), handles allocation and de-allocation of processes.

A **highly** abstracted diagram of how these components interact.
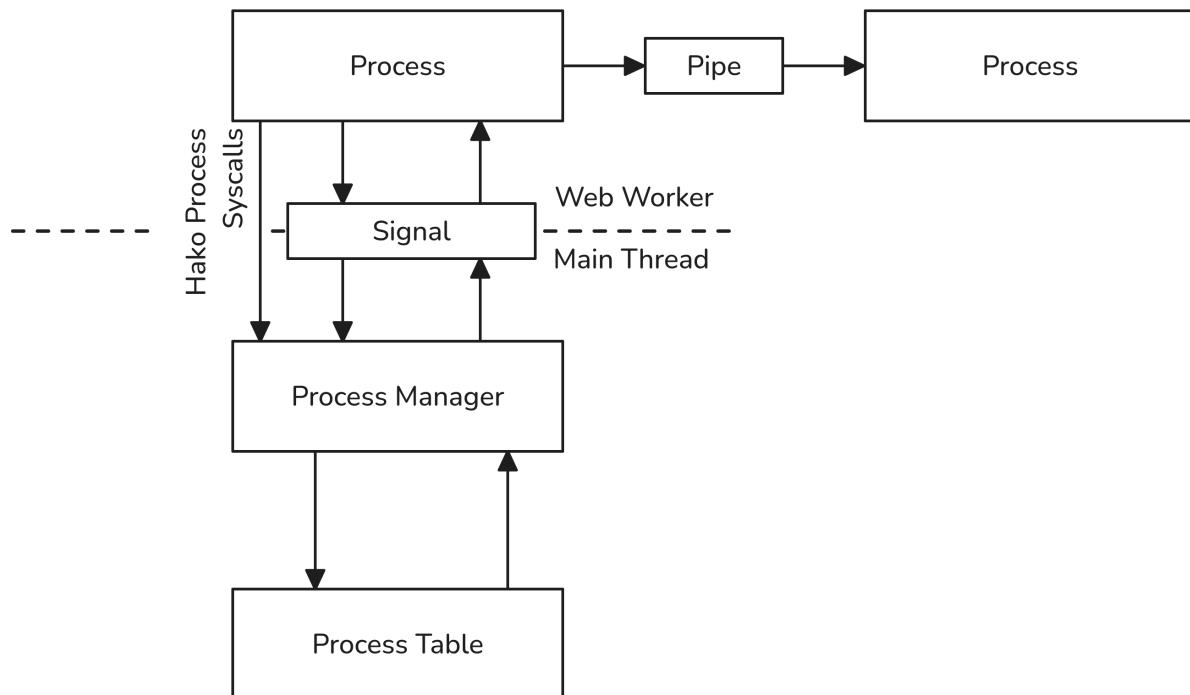
5.4.1 Process

Processes are at a very high level, Web Workers.

A process's I/O capabilities are:

1. Redirecting to the filesystem for input and output.
2. Piping to other processes for input and output.
3. Reading from and writing to the slave side of a PTY.

and prioritised in that manner.

When processes are created, they receive all information required in an initial message sent from the Process Manager on the main thread.

Information sent from the main thread:

- **pid** – The process's id.
- **cwd** – The process's initial current working directory.
- **stdin** – SharedArrayBuffer for the Stdin Pipe.
- **stdout** – SharedArrayBuffer for the Stdout Pipe.
- **stderr** – SharedArrayBuffer for the Stderr Pipe.
- **redirectStdin** – A path to redirect output to (`""` by default).

- **redirectStdout** – A path to redirect input to (`""` by default).
- **IsInATTY** – Whether the process's input is a TTY (`false` by default).
- **IsOutATTY** – Whether the process's output is a TTY (`false` by default).
- **IsErrATTY** – Whether the process's error is a TTY (`false` by default).
- **StreamDescriptor** – An enum for STDIN and STDOUT.
- **luaCode** – The Lua code to be ran by the runtime.
- **signal** – SharedArrayBuffer for the Signal.

And this is all sent in the Web Worker, within the `proc` object.

```
 1   self.proc = {
 2     pid: data.pid,
 3     cwd: data.cwd,
 4     args: data.args,
 5     stdin: new Pipe(0, data.stdin),
 6     stdout: new Pipe(0, data.stdout),
 7     stderr: new Pipe(0, data.stderr),
 8     redirectStdin: data.redirectStdin,
 9     redirectStdout: data.redirectStdout,
10     isInATTY: false,
11     isOutATTY: false,
12     isErrATTY: false,
13     StreamDescriptor,
14     luaCode: data.luaCode,
15     signal: new Signal(data.signal),
16     ...
```

The `proc` object contains these attributes, but it **also contains the Process JS API**.

For example, the `wait(pid)` process method, for blocking on another process.

```
 1   self.proc = {
 2
 3     ...
 4
 5     wait: (pid) => {
 6       // Tell the manager we'd like to wait on a process
 7       self.postMessage({
 8         op: ProcessOperations.WAIT_ON_PID,
 9         requestor: self.proc.pid,
10         sendBackBuffer: self.proc.signal.getBuffer(),
11         waiting_for: pid
12       });
13       changeState(ProcessStates.SLEEPING);
14       self.proc.signal.sleep();
15       let exitCode = self.proc.signal.read();
16       changeState(ProcessStates.RUNNING);
17       return exitCode;
18     },
```

Notice, the `proc` object is attached to the Web Worker's global scope `self`.

The reason why this is done is because the process and related systems such as the process manager **resides entirely in javascript**, but the runtime **resides entirely in C**.

So in order for the process to interact with the Process API, it needs to be able to access it – hence why all of this is transported and sent to the Web Worker on initialisation.

We then build the C Process API into the runtime, atop this thread-local state, via `EM_JS` macros provided by Emscripten for bridging C to JS.

This allows us to access thread-local JS state, calling it from Web Assembly (C compiled with Emscripten)

```
1  // proc__wait(int pid, Error *err)
2  EM_JS(int, proc__wait, (int pid, Error *err), {
3    let exitCode = self.proc.wait(pid);
4    setValue(err, 0, 'i32');
5    return exitCode;
6  })
```

> Above is the corresponding C Process API, matching the JS Process API `wait(pid)` call above

This combination of technology and methods allow us to access the JS Process API, and build a runtime C Process API on top of it. **Giving the runtime access to control and interact with its own and other process instances**.

5.4.1.1 Intercepting Emscripten's pThread

Processes were originally implemented completely from scratch by us via Web Workers, but as the runtime evolved, we realised that we would be constrained by Emscripten.

The runtime runs in a pthread (for reasons stated in Terminal) – specifically Emscripten's interpretation of what a `pthread` would look like in the web, its own implementation of the pthread library.

This poses two main challenges:

- Emscripten is entirely in charge of the web worker, and even hides it.
- Emscripten never yields to the event loop, so message passing is impossible.

This was **very problematic for us**, as we needed to attach custom data to the Web Worker's global scope to allow the runtime to have access to the Process API. Unfortunately, Emscripten doesn't provide this.

All we needed to do was send a single message to the Web Worker to provide it all the state it needed, but this required:

1. Finding where Emscripten hides the Web Workers for its pthreads.
2. Being able to send the Emscripten pthread Web Worker a message.
3. Defining in the Emscripten pthread Web Worker how to handle this message.

We managed to do this by **embedding custom code into Emscripten's generated WebAssembly initialisation code**, intercepting it, and attaching our own behaviour.

We found through inspection of the Emscripten pthread being initialised how to intercept it, though admittedly it is somewhat gruesome.

Emscripten's WebAssembly initialisation code runs on both the main thread and the web worker it creates and owns, so we can inject code on both to try and intercept it.

We intercept on **both the main thread and the Emscripten Web Worker thread**.

```
1  // If we're in an emscripten PTHREAD (webworker), intercept the thread
2  if (ENVIRONMENT_IS_PTHREAD) {
3    interceptThreadCreation();
4
5  // Otherwise intercept the main thread
6  } else {
7    const isNode = typeof window == "undefined";
8    ProcessManager = isNode ? globalThis.ProcessManager : window.
         ProcessManager;
9
10   interceptMainThread();
11 }
```

A diagram to better visualise the interception:
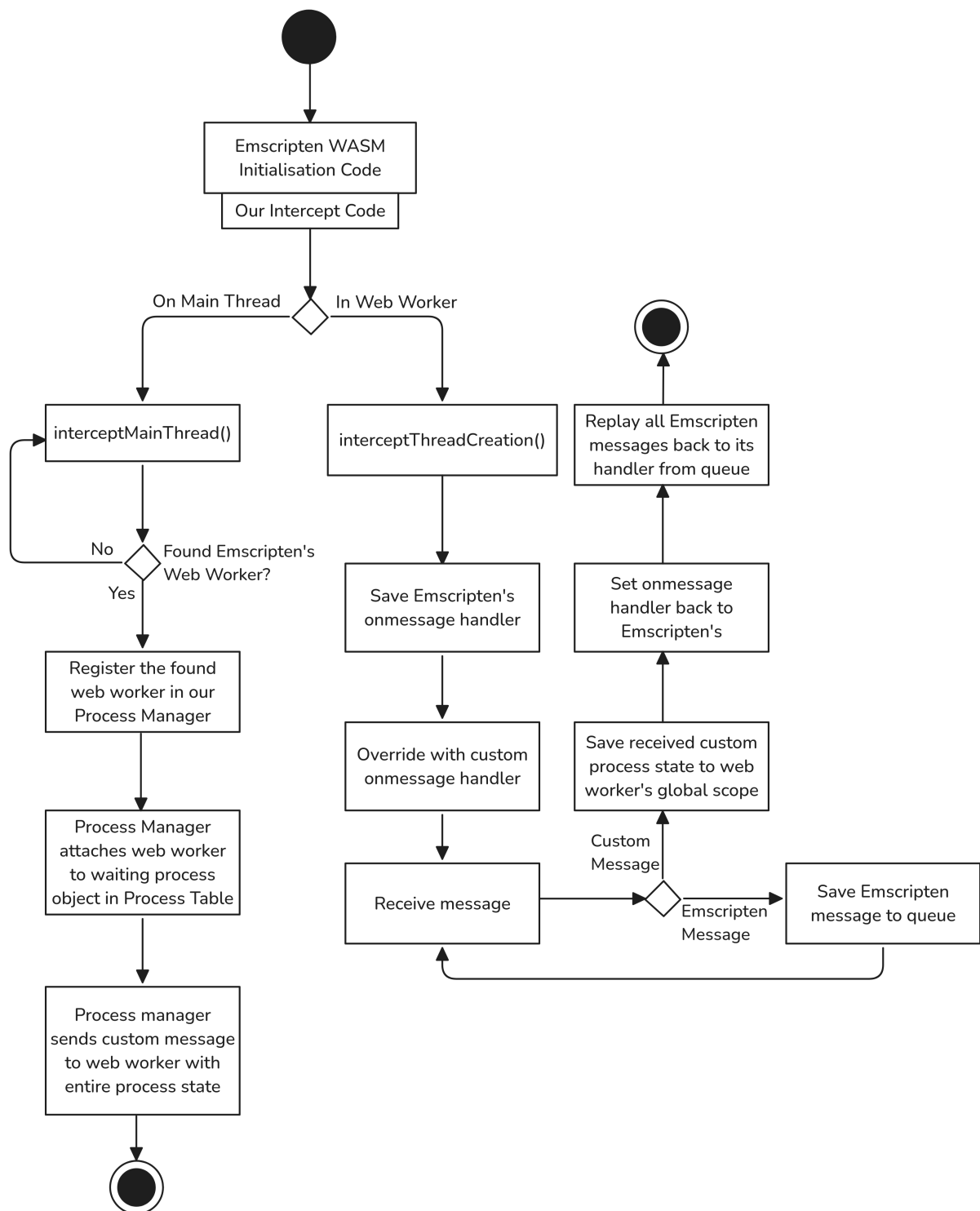
The steps to intercepting Emscripten's initialisation in the Main Thread:

1. Execute a polling `interceptMainThread()` function at the end of the Emscripten initialisa-

tion code.

2. Find the Web Worker Emscripten creates in a global `PThread` object attached to the main thread's window.

3. Once detected, we register the Web Worker in our own Process Manager.

4. The Process Manager then associates this Web Worker with an entry in the process table waiting to be initialised with a Web Worker.

5. The Process Manager then sends a custom message to the captured Web Worker, with all custom process state included.

```
1   // Function intercepts main thread to send messages to main thread
2   function interceptMainThread() {
3     let running = PThread.runningWorkers[0];
4     if (running === undefined) {
5       setTimeout(interceptMainThread, 100);
6       return;
7     }
8     // Register the worker to a process
9     ProcessManager.registerWorker(running);
10  }
```

The steps to intercepting Emcripten's initialisation in the Web Worker:

1. Execute `interceptThreadCreation()`.

2. Create a queue to temporarily store intercepted Emscripten messages (from the main thread to its Web Worker).

3. Store emscripten's custom `onmessage` handler.

4. Override the Web Worker's `onmessage` with our own message handler.

5. Direct all messages to the temporary queue to store them.

6. Wait for a custom message from the process manager with our custom process data.

7. Attach received custom process data to the Web Worker's global scope.

8. Go back to the original Web Worker's message handler.

9. Handle the saved Emscripten messages.

```
1   // Function intercepts web worker thread to temporarily override custom
2   function interceptThreadCreation() {
3     let queue = [];
4     let defaultHandleMessage = self.onmessage;
5
6     function goBack() {
7       // Set onmessage back
8       self.onmessage = defaultHandleMessage;
9       for (let msg of queue) { // Run any captured events
10        defaultHandleMessage(msg);
11      }
12    }
```

```
13
14    self.onmessage = async (e) => {
15      switch (e.data.cmd) {
16      case "custom-init":
17        await initWorkerForProcess(e.data);
18        goBack();
19        break;
20      default:
21        queue.push(e);
22      }
23    }
24  }
```

As a summary; we intercept Emscripten's initialisation on both the main thread, and its Web Worker, in order to send a single message containing our custom process state, to enable the Process API in the runtime.

5.4.2 Inter-Process Communication

Hako's processes communicate with two parties:

- **Other Processes (Web Workers)** – via Pipes
- **Process Manager (Main Thread)** – via Signals

Both Pipes and Signals are essentially the same implementation.

They are both circularly bounded buffers on shared memory (SharedArrayBuffers) with read and write functionality.

A diagram showing the interaction of two processes piped together via a Pipe is shown below.
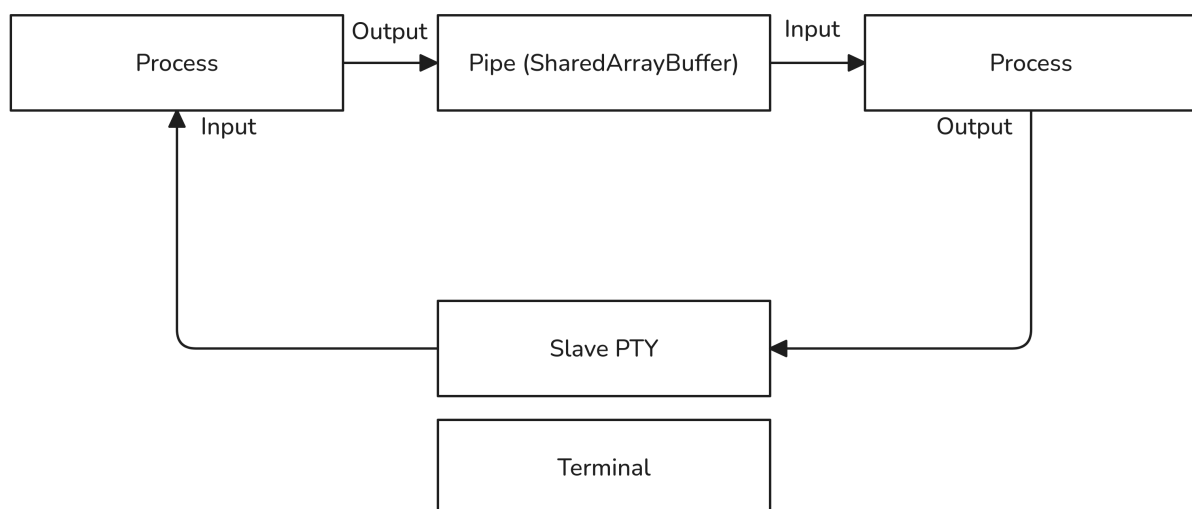
Here is a snippet of `Pipe`'s write function, which writes onto the SharedArrayBuffer, synchronising via javascript Atomics.

```
 1   /**
 2    * Writes data to buffer.
 3    * Returns -1 if closed, 0 otherwise
 4    */
 5   write(data) {
 6     if (this.#closed) return -1;
 7
 8     const encoded = this.encoder.encode(data);
 9     for (let i = 0; i < encoded.length; i++) {
10       while (true) {
11         const wr = Atomics.load(this.control, 1);
12         const rd = Atomics.load(this.control, 0);
13         // Buffer is full if the next write index equals the read
                pointer
14         if (((wr + 1) % this.data.length) === rd) {
15           // Buffer full; wait on the read pointer
16           Atomics.wait(this.control, 0, rd);
17           continue;
18         }
19         // Write the byte at the current write pointer
20         this.data[wr] = encoded[i];
21         const newWr = (wr + 1) % this.data.length;
22         Atomics.store(this.control, 1, newWr);
23         // Notify the reader that new data is available
24         Atomics.notify(this.control, 1, 1);
25         break;
26       }
27     }
28     return 0;
29   }
```

> Signal's implementation of `write` differs very slightly, as there is no concept of EOF

Each process has a pipe for:

- Stdin
- Stdout
- Stderr (Although not used)

So processes can easily be piped together via different streams.

Each process has one signal which it shares with the main thread, the primary difference between pipes and signals being that **signals allow sleeping and notifying of the Web Worker thread.**

As stated in the Intercepting Emscripten section, Emscripten's Web Workers refuse to yield to the event loop, so we cannot use traditional message passing mechanisms to pass data back and forth from the

Process Manager on the main thread, and the Web Worker itself.

So we are forced to use shared memory via SharedArrayBuffers, and utilise the synchronisations allowed on these - such as `Atomics.wait` and `Atomics.notify` as seen above.

The typical sequence of events of a Web Worker sending a request to the Process Manager on the main thread, and receiving a result back is:

1. The Web Worker posts a message to the main thread with data specifying the type of request and information to go along with the request, etc.
2. The Web Worker then sleeps on its `signal` via `signal.sleep()`.
3. The Process Manager in the main thread receives the message, and actions it.
4. Once finished processing the request, the Process Manager writes to the requesting process's `signal` via `signal.write(response)`.
5. The Process Manager then calls `signal.wake()`, which wakes the requesting process.
6. Finally, the requesting process reads the result from the signal.

An example of a calling `process.create` from Lua, which requires synchronisation and information passing from both the Web Worker and the Process Manager on the main thread.

You can see that a process creation object `obj` is passed down into the main thread from the Web Worker, and a `pid` is returned back.
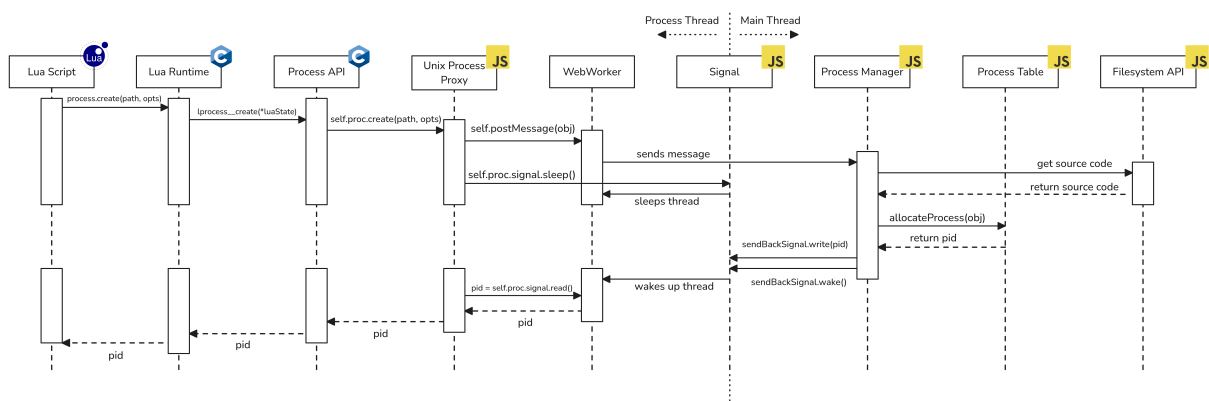
5.4.3 Process Manager

The Process Manager is the orchestrator of all Hako processes and resides on the main thread.

When Emscripten creates Web Workers, they're intercepted and registered with our Process Manager so we can have full control over their lifetime. As described above in Intercepting Emscripten.

The Process Manager owns:

- The Process Table.

- A queue of new processes awaiting Emscripten Web Workers.
- A map set of processes waiting on others.

Some of the Process Manager's methods:

- `createProcess(obj)` – For creating a new process.
- `registerWorker(worker)` – Registering an intercepted Emscripten Web Worker to one of our new processes.
- `getProcess(pid)` – Returns all process information of that pid.
- `listProcesses()` – Lists all processes in the process table.
- `killProcess(pid)` – Kills and cleans up a process of pid.
- `exitProcess(exitCode)` – Called by a process for exiting with an exit code.
- `pipe(outPid, inPid)` – Pipes the `outPid`'s stdout with the `inPid`'s stdin.

The Process Manager also accepts and handles requests from processes via message passing.

Requests that the Process Manager accepts are:

- `ProcessOperations.CHANGE_STATE` – Change a process's current state in the process table (e.g. 'Starting', 'Running', etc).
- `ProcessOperations.WAIT_ON_PID` – Allows a process to sleep on another process until it's finished execution, receiving its exit code.
- `ProcessOperations.CREATE_PROCESS` – Allows a process to create another with options, receiving the new `pid` in response.
- `ProcessOperations.KILL_PROCESS` – Allows a process to kill another via referencing its `pid`.
- `ProcessOperations.LIST_PROCESSES` – Provides a serialised JSON view of the process table.
- `ProcessOperations.PIPE_PROCESSES` – Allows the piping of two processes.
- `ProcessOperations.START_PROCESS` – Allows a process to be started after creation.
- `ProcessOperations.EXIT_PROCESS` – Allows a process to exit with an exit code.

Processes sleep on their `Signal` buffer immediately after sending a message to the main thread. The Process Manager writes the result into the requesting process's `Signal` buffer, and wakes the process again after writing the result.

5.4.4 Process Table

The process table is an array, indexed by the process identifier, containing all information required by our process implementation.

It is instantiated by the Process Manager, with the default max number of processes being set to 128.

It has the responsibility of:

- Allocating space in the process table for new processes.
- Creating a process's `Pipes` and `Signal`.
- Creating the process data structure and filling it with information.
- Starting the Emscripten runtime module.
- Attaching a Web Worker to a process in the table.
- Freeing / Cleaning up processes in the table.
- Serialising itself into JSON.

Here is an example method of the Process Table allocating a process:

```javascript
 1  async allocateProcess(processData) {
 2    // Find the next available PID
 3    while (this.nextPID < this.maxPIDs && this.processTable[this.
          nextPID] !== null) {
 4      this.nextPID++;
 5    }
 6
 7    // Table is full
 8    if (this.nextPID >= this.maxPIDs) {
 9      throw new CustomError(CustomError.symbols.PROC_TABLE_FULL);
10    }
11
12    // ============= Initialise Process Entry =============
13
14    // Create MessageChannels for inter-process communication
15    const stdinPipe = new Pipe(this.pipeSize);
16    const stdoutPipe = new Pipe(this.pipeSize);
17    const stderrPipe = new Pipe(this.pipeSize);
18    const signal = new Signal();
19
20    const process = {
21      args: processData.args,
22      stdin: stdinPipe,
23      stdout: stdoutPipe,
24      stderr: stderrPipe,
25      luaCode: processData.luaCode,
26      signal: signal,
27      time: Date.now() / 1000,
28      state: ProcessStates.STARTING,
29      pty: processData.slave,
30      pipeStdin: processData.pipeStdin,
31      pipeStdout: processData.pipeStdout,
32      redirectStdin: processData.redirectStdin,
33      redirectStdout: processData.redirectStdout,
34      cwd: processData.cwd,
```

```
35          fakePath: processData.fakePath,
36          start: processData.start
37        }
38
39        // Place the new process object in the table
40        this.processTable[this.nextPID] = process;
41
42        let newProcessPID = this.nextPID++;
43
44        // Attach Module to process
45        // Return the allocated PID and increment it
46        return {
47          pid: newProcessPID,
48        };
49      }
```

The reason why we're dividing the result of `Date.now()` by 1000 is because we are restricted by WASM32, which can only represent 32-bit integers. `Date.now()` returns an epoch nano second timestamp, which cannot be represented.

5.5 Filesystem (module `filesystem`)

Hako imitates a regular filesystem, similar to `fat32` or `ext4` in behaviour as we wanted to imitate a filesystem that users would be familiar with.

Furthermore, it is **persistent** and **completely client-side**, as our target demographic are children. We do not want to store any information – and a **core constraint of this project is to be completely client-side**.

The technology we utilised for this was IndexedDB, a low-level API for client-side storage of significant amounts of structured data, including files and blobs.

Furthermore, Emscripten has support for IndexedDB too for imitating a persistent filesystem.

The filesystem is **written entirely in C** and compiled to WebAssembly via Emscripten.

For example, below is our file/directory remove (i.e. unlink) method.

```
1  // Unlinks a node
2  void file__remove(const char *restrict path, Error *restrict err) {
3    // Permission checks
4    struct stat st;
5    bool file_exists = (stat(path, &st) == 0);
6
7    if (!file_exists) {
8      *err = translate_errors(EEXIST);
9      return;
10   }
```

```
11
12    // If it's a system file, fail - user can't remove system files
13    if (is_system_file(&st)) {
14      *err = translate_errors(EROFS);
15      return;
16    }
17
18    int error = unlink(path);
19    if (error < 0) {
20      *err = translate_errors(errno);
21      return;
22    }
23    *err = 0;
24    return;
25  }
```

This allows the runtime to easily access it – however we did also have to **expose the filesystem to Javascript** for the filesystem initialisation and bootstrap process, as well as for the file manager application.

Additionally, Emscripten does a lot of the heavy lifting for us with its own web libc implementation - translating standard filesystem libc calls to its javascript-based Virtual File System (VFS).

5.5.1 Exposing to Javascript

As the filesystem is written in C and compiled to WebAssembly, we need to create a stub API in Javascript to interact with it.

This involves some pretty interesting behaviour in order to interact with WebAssembly, such as **allocating and de-allocating heap and stack memory in Javascript**.

The following is the primary stub, allowing us to call WebAssembly functions.

```
1    function callWithErrno(fnName, returnType, argTypes = [], args = [])
       {
2      let sp = Module.stackSave();
3
4      let errorPointer = Module.stackAlloc(4);
5
6      let returnVal = Module.ccall(
7        fnName,
8        returnType,
9        [...argTypes, "number"],
10       [...args, errorPointer]
11     );
12
13     let errno = Module.getValue(errorPointer, 'i32');
14
```

```
15        Module.stackRestore(sp);
16
17        return { returnVal, errno };
18    }
```

All of our filesystem functions have an error out parameter, so we have to **allocate on the WebAssembly stack** (specifically four bytes for an integer) to pass into the WebAssembly call.

An example of us using this stub is for our remove (unlink) call below, where we also pass a path. Note this would be analogous to the previously listed C implementation of remove, which you can see at Intercepting Emscripten.

```
1    Filesystem.remove = (path) => {
2      let errorStr = null;
3
4      let { errno } = callWithErrno(
5        "file__remove",
6        null,
7        ["string"],
8        [path],
9      )
10
11     if (errno > 0) {
12       errorStr = errnoToString(errno);
13     }
14
15     return { error: errorStr }
16   }
```

> Notice the Filesystem object – this is attached to the window object to be easily accessible from the main thread.

5.5.2 Permissions

Hako is a **single-user system**, so there is no need for multi-user permissions.

Hako's permission system only uses the 'user-bits' of traditional Linux filesystems (0700) along with an extra 'system-bit' flag (0010) for signifying it's **a protected system file**.

Permissions work as expected, but restrictions exist on system files. Which you can only read and execute. Any other operations such as unlinking or renaming aren't permitted.

5.5.3 Initialisation and Bootstrapping

One of the first operations the +page.svelte component performs when it's mounted is to load the filesystem.

Initialising the Filesystem involves:

1. Instantiating the filesystem WebAssembly module.
2. Waiting until it's instantiated.
3. Initialising the filesystem JS API.
4. Calling the `initialiseFS()` JS API method to begin the bootstrap process.

Bootstrapping the Filesystem involves:

1. Detecting if the `"/persistent"` directory mountpoint exists.
2. Creating it if it doesn't exist.
3. Mounting IndexedDB to `"/persistent"` with `autoPersist` enabled.
4. Checking if the `"/bin"` path directory exists within this mount.
5. Creating it if not.
6. Moving system files (core-utils and shell) into the `"/bin"` directory.
7. Raising the 'system-bit' on all system files and directories for integrity.
8. Performing a sync on the filesystem, ensuring all previous user data is loaded.

You can actually see this being performed when you open Hako, as a 'loading' popup will appear during this – although it's quite speedy, so it's easy to miss.

An example of some of the bootstrapping code is available below:

```
1    // Mount the root
2    try {
3      M.FS.mount(M.IDBFS, {autoPersist : true}, persistentRoot);
4    } catch (err) {
5      console.error("[JS] Failed to mount filesystem:", err);
6    }
7
8    // Util function for performing an action if a path doesn't exist
9    function ifNotExists(path, doit) {
10     if (!M.FS.analyzePath(path, false).exists) {
11       return doit(path);
12     }
13   }
14
15   function moveFilesIn() {
16     console.log("Moving fresh system files in...");
17
18     // Initialise system files
19     let systemFilePath = "/persistent/bin";
20
21     // WARNING: IDBFS requires write access – however users will not
          be
22     //          able modify regardless due to the PROTECTED_BIT being
23     //          raised signifying it's a system file (0o010)
```

```
24        ifNotExists(systemFilePath, (p) => M.FS.mkdir(p, 0o710));
25
26        // Move lua files into correct place in IDBFS
27        for (const luaFile of M.FS.readdir("/luaSource")) {
28          if (luaFile == "." || luaFile == ".." || luaFile == "luaSource"
              ) continue;
29
30          let sourcePath = `/luaSource/${luaFile}`;
31          let systemPath = `${systemFilePath}/${luaFile}`;
32
33          // Move files into systemFilePath
34          let data = M.FS.readFile(sourcePath);
35          ifNotExists(systemPath, (p) => {
36            M.FS.writeFile(p, data);
37            // Set correct permissions on file
38            M.FS.chmod(p, 0o710);
39            console.log(`ADDED: ${p}`);
40          });
41        }
42      // Set correct permissions so parent directory cannot be modified
            either
43      // INFO: I don't believe we're currently using dir permission
          bits, but future proofing regardless
44      M.FS.chmod(systemFilePath, 0o710);
45    }
```

> This is just a snippet of the code too, there is more involved

This routine is restorative too, as it only moves in the files that are missing - so after the first bootstrapping of the filesystem, it has very little to do.

5.6 Core-Utils

There are a large amount of core-utils, and we do not want to blow up the document by discussing each one in detail, so we'll moreso discuss the general approach we took to implementing them.

The development of the core-utils and the shell were actually performed in the Hako virtual operating system. That is to say that we performed dogfooding, where we used our own self-hosted system for development of the utilities. This was greatly beneficial for finding bugs, jarring behaviour, and generally just experiencing the system as an end user, and improving it as a whole.

In choosing core utilities for Hako, we looked at various definitions and standards of core utils, and took a select subset of utilities to implement ourselves.

Standards / implementations we looked at:

- GNU Coreutils

- Opengroup Standard Utilities
- Busy Box

The specific set we decided to implement were:

- cat
- chmod
- clear
- cp
- echo
- find
- grep
- kill
- ls
- mkdir
- mv
- ps
- pwd
- rm
- rmdir
- touch

We would then look at the flags of each core-util, and decide on which ones we would and wouldn't support – although we did put a great deal of effort in supporting as many as we thought would be well suited.

For example, `ls`:

```
 1  $ ls --help
 2  Usage: ls [OPTION]... [FILE]...
 3  List information about the FILEs (the current directory by default)
 4  Sorts entries alphabetically by default.
 5
 6  Options:
 7   -1, --one       list one file per line
 8   -a, --all       do not ignore directories starting with .
 9   -A, --almost-all  do not list implied . and ..
10   -R, --recurse   follow sub-directories and list their contents too
11   -l, --long      long format
12   -i, --inode     list inode numbers
13   -s, --block     lists allocated blocks
14   -p,             append / to directory names
15   -c, --ctime     list ctime
16   -u, --atime     list atime
17   -h, --human     list human readable sizes (1K, 243M, 2G)
```

```
18   -S, --size        sort by size
19   -X, --ext         sort by extension
20   -t, --smtime      sort by mtime
21   -m, --sctime      sort by ctime
22   -e, --satime      sort by atime
23   -r, --reverse     reverse sort order
```

As you can see, there are many flags supported, but more 'niche' or unapplicable flags are emitted. Such as '−g' (group directories first), '−H' (follow symlinks).

Core-utils were also implemented by both members, and were implemented last – so there are differing styles in their implementation. But the external appearance of the utilities are consistent.

There are also a couple of additional custom non-standard core-utils:

- **help** – A large manual, somewhat akin to man.
- **lua** – Runs lua inline in the shell, useful for debugging.

All of the core-util code is available in src/filesystem/luaSource.

5.7 Shell

The shell was essentially the first core-util written in order to make Hako usable for the dogfooding process of creating the other core-utils within the system.

It's a simple shell that supports:

- **Pipelines** - Joining two or more commands together via the pipe operator (|).
- **Short Circuit Evaluation** - Conditional operation based on the outcome of previous commands via the OR (||) and AND (&&) operator.
- **I/O Redirection** - Redirecting a process's input or output via the < and > operators respectively.
- **Command Grouping** - Commands can be grouped in their execution (without subshelling) via { and } operators.

The grammar representative of the recursive decent parser for the shell is as follows:

```
 1  Lines := Line (';' Line)*
 2
 3  Line := Pipeline ( ('&&' | '||') Pipeline )* ('&')?
 4
 5  Pipeline := Command ( '|' Command )*
 6
 7  Command := SimpleCommand
 8            | GroupedCommand
 9
10  GroupedCommand := '{' Lines '}'
```

```
11
12   SimpleCommand := word ( word )* Redirects?
13
14   Redirects := RedirectIn RedirectOut
15              | RedirectOut RedirectIn
16              | RedirectIn
17              | RedirectOut
18
19   RedirectIn := ('<' | '<<') word
20   RedirectOut := ('>' | '>>') word
```

> Note, the background symbol & is supported in the parser for extensibility, but background processes were never implemented.

A line of input into the shell is processed as follows:

- **Lexical Analysis** – Input characters are turned into tokens (e.g. STRING, AND_OPERATOR, etc)
- **Syntactical Analysis** – A handwritten recursive descent parser which parses the terminals and non-terminals of the grammer aforementioned, producing an Abstract Syntax Tree
- **AST Tranversal** – Walking the AST tree to execute line(s) of input.

There is a substantial amount of code included in this, and it's quite dense, as lexing, parsing and AST traversal was all manually implemented, so we'll just discuss the general patterns applied.

For lexing, we would just iterate over the input characters, occasionally having a single lookahead for ambiguities, and then fill a flat array with such tokens.

For parsing, standard recursive descent was implemented, with functions:

- `parse_lines`
- `parse_line`
- `parse_pipeline`
- `parse_command`
- `parse_grouped_command`
- `parse_single_command`
- `parse_redirect`

This would create an Abstract Syntax Tree (AST).

Finally, the most complex part was walking the AST. For this, a **structured, context-aware top-down, pre-order AST traversal** was implemented.

More specifically, AST walking consists of:

- **Context-carrying** – Passing contextual state around the tree during traversal (e.g. collecting process IDs to lazily execute, etc).

- **Deferred Execution** – Commands are collected in context, and then executed lazily, only just as their evaluation is required (e.g. for short circuit evaluation).
- **Pipe Closure Inversion** – To support the complicated semantics of piping into and out of command groups

The complexities involved in the pipe closure inversion were due to:

- Shortcuts in our implementations (technical debt) – such as a lack of pipe table.
- Restrictions imposed upon us by our technologies – Emscripten's Web Workers do not yield to the event loop, so message passing cannot be used, and complicated data structures such as SharedArrayBuffers can only be sent this way, so we can not update a process's pipes after it has been created.

The methods that specifically handle the AST evaluation are:

- `exec_lines`
- `exec_line`
- `exec_pipeline`
- `exec_pids`

This is perhaps some of the most complex code in our project, and snippets are difficult to understand out of context, so we won't include any here – but the code is available in Hako within the `"/bin"` directory or at `src/filesystem/luaSource/shell.lua` in the repository.

6 Problems Solved

Note that this is by no means an exaustive list, and especially for this project. There were many problems that we faced due to the mere fact that we were unfamiliar with the particularly obscure technology that we used. Many of our problems are certainly forgotten at this point. The following list is the most memorable and interesting set of problems that we were faced with. Other problems either directly or indirectly discussed in the rest of the document are omitted from here for brevity.

6.1 Self Hosting – Bundler Issues

6.1.1 Problem

When trying to port the website to be deployable – deployed on our self-hosted webserver serving https://hakoapp.com (uptime subject to reliability of home network). In development, we could get by using the development web server provided by SvelteKit, however for the deployment the site

needed to be bundled so that it could be ran through a Caddy reverse proxy on the server. Bundling in SvelteKit is done through Vite, which uses Rollup under the hood. The problem is that it seems to be fragile. Often it optimises out imports that it shouldn't. This means that certain imports do not actually produce the contents of the module itself, but strangely it produces the name of the import itself.

6.1.2 Resolution

The solution was to actually switch to a very manual dynamic import, and tell vite to not transform it. This meant that we would have to change some imports from:

```
1        const { initialiseAPI, Filesystem } = await import("/api.mjs?url")
            as unknown as { initialiseAPI: Function, Filesystem: any };
```

To the following:

```
1        const { initialiseAPI, Filesystem } = await import(/* @vite-ignore
            */ new URL("/api.mjs", import.meta.url).href) as unknown as {
            initialiseAPI: Function, Filesystem: any };
```

6.2 Auto Deploy

6.2.1 Problem

We wanted to be able to have any merge from our development branch into main branch to automatically trigger a deployment when the CI passes.

6.2.2 Resolution

Since the website is fully client side and just a static bundle, it made sense to just have a simple webserver running on our server that would serve an API key protected set of endpoints for submitting new bundles to be deployed. The API key is stored as a gitlab secret that our pipeline can securely access to hit the website whenever a CI build and test is complete.

The webserver is implented in Go and the code can be found at `src/deploy-server`. It is run under Caddy on our own server.

6.3 Sharing the Persistent Filesystem

6.3.1 Problem

The filesystem in Hako is backed by a persistent store in IndexedDB, via IDBFS. IDBFS needs to be mounted onto the memory based VFS (MEMFS) provided by emscripten to enable persistence. The problem is that you cannot just mount IDBFS into each runtime instance separately, as this will cause all sorts of bizarre behaviour, indicative of a race condition/contention between different instances.

6.3.2 Solution

The solution is to have the JS API just directly access the filesystem Emscripten module, whereas each runtime accesses it through a PROXYFS which ensures that they are all writing to the same persistent filesystem, just with separate file descriptor tables and working directory state.

6.4 Integrating Text Editor With Custom Filesystem

6.4.1 Problem

As mentioned before the JS API accesses the persistent filesystem directly. This means that file descriptors that you get back from these API's calls are managed by that filesystem. The working directory state is also managed by this filesystem for the same reason. This is problematic as instances of the file manager cannot simply use `change_dir` whenever the user is clicking through folders on the file manager as it would change the current working directory globally.

6.4.2 Resoution

The solution was to have a virtual view of the filesystem to enable changing directory scoped to each file manager independently. This functionality was provided as a class, `FSView`, whose code resides at `src`/`site`/`src`/`lib`/`files`.`js`. This just manages the current directory as well as providing funtionality for resolving relative paths based on that state.

6.5 Filesystem initialization race condition

6.5.1 Problem

Filesystem intialization was not properly setup to block the caller trying to initialise it until it was setup and ready properly. This meant that there was a race condition between filesystem initialisation

and performing operations on the filesystem. This was particularly egregious as when we wrote the filesystem tests we thought we had configured Puppeteer to not persist between different test cases, i.e. files created in one test case would not be accessible in another test case. This assumption was wrong however, and what was actually happening is the filesystem was being intialized for each test case independantly and persistence was not happening between the test cases only because it wasn't initialised yet and it was actually writing to the in memory VFS. This bug was difficult to track down as it was only presenting itself as failed tests very irregularly.

6.5.2 Resolution

The fix was to make sure that filesystem initialization would only resolve a promise when it was actually setup and ready to be read from and written to. This also concequently broke all of our filesystem tests that were assuming persistence was disabled. So we additionally had to fix them to not assume this and in fact this allowed us to test that persistence was working properly in the test cases as well. Note also that the loading screen in Hako is blocked by this filesystem intialisation step.

6.6 Bugs in Xterm-PTY

6.6.1 Problem

Xterm-PTY is a library that we needed to integrate Emscripten and Xterm.js (see Terminal for more details). Unfortunately it is a very small project maintained at least from what we could see, by a single benevelant individual. That being said, a couple issues were found throughout the project, both of which were raised with the maintainer. The first issue was that when reading a buffered line of input, if the terminal was resized raising a `SIGWINCH` signal it would interrupt the reading of the current line (you can see the issue discussion here - leath-dub is the user name of Cathal O'Grady). The second issue was that EOF (End Of File) was not being sent properly when typeing Ctrl-D, this was because when `EAGAIN` was returned twice from reading from the TTY, it was not interpretted correctly as a EOF and would return `EAGAIN` to the caller instead of a read of 0 (as should be returned from read on EOF) – (you can see more discussion here).

6.6.2 Resolution

The first issue was thankfully solved by the maintainer of Xterm-PTY. The second issue was however a large blocker for us at the time, so we implemented a fix ourselves and made a PR to Xterm-PTY. You can see the PR here. The fix was to just detect when `EAGAIN` is raised twice and actually return a read of 0 to indicate EOF to the caller.

6.7 Exposing svelte UI to runtime

6.7.1 Problem

Unlike other APIs like the filesystem and Process Manager, the Desktop API, which allows Lua to access the windows of the website and perform operations on them (as well as open new windows), needed to interract with the Svelte application. This is so it could manipulate the state of the Window component running in svelte (i.e. change the x, y and width, height).

6.7.2 Resolution

The solution was to have every Window component set a context and table of functions providing the API to operate on said context. The really important thing however is to ensure that when opening a window, you need to call `flushSync()` to ensure that Svelte waits for `onMount` to be called and finished executing, which correctly sets up the context and table of functions on its respective window object. You can see in the `onMount` in `src`/`site`/`src`/`components`/`Window.svelte` how it sets a context and function table on its respective window object here:

```
1    let win = windows.getWindowByID(id);
2    win.state.ctx = ctx;
3    win.state.vtable = vtable;
```

All of this allows the desktop API to manipulate windows through the window API in `windows.svelte.js` file (technically it doesn't directly call the API).

6.8 Caching in Pipeline

6.8.1 Problem

Hako needs the latest version of the Just task runner, as it uses some of its newer features in its justfile. This meant that in the pipeline we needed to download the install script rather than installing it from the Ubuntu package repositories directly. The problem is that initially we were pulling the install script every time the pipeline ran, and we ended up getting rate limited by Just.

6.8.2 Resolution

The solution was to add caching of this install script into the pipeline. We also additionally cache things like building `ncurses` and `libedit`, as they are quite large and make the CI build take a long time. They are also not code that we change, so it makes sense to aggressively cache their build artefacts.

6.9 Runtime C Type Reflection in JS

6.9.1 Problem

When exposing a C function to Javascript from WebAssembly, any types that the C function takes or returns are not accessible from Javascript. This means that you need to keep track of the size of the types, and in the case of C structs the offsets of the fields on the struct. Initially we were just ensuring that the structs were `packed` – which means that the compiler would not add padding between fields trying to maintain a byte memory alignment (although wasm32 doesnt seem to have pointer alignment requirements). Then we would just manually reference the offsets to given fields from Javascript. This was very cumbersome and barely maintainable, especially if you had to add new fields to a struct.

6.9.2 Resolution

We solved this by exporting type information as globals on our C APIs. For example you can see below how we export the size and offset information of the `StatResult` struct (which is what is used to hold information returned by the `file__stat` function) – full code at `src/filesystem/src/main.h`:

```c
typedef struct __attribute__((packed)) {
  int size;
  int blocks;
  int blocksize;
  int ino;
  int perm; // permissions (Only user: 01 Read, 001 Write, 0001 Execute
      ) 20
          // bytes
  int type; // 0: file, 1: directory
  Time atime;
  Time mtime;
  Time ctime;
} StatResult;

#ifdef FILE_IMPL
const int sizeof_ReadResult = sizeof(ReadResult);
const int offsetof_ReadResult__data = offsetof(ReadResult, data);
const int offsetof_ReadResult__size = offsetof(ReadResult, size);
const int sizeof_Time = sizeof(Time);
const int offsetof_Time__sec = offsetof(Time, sec);
const int offsetof_Time__nsec = offsetof(Time, nsec);
const int sizeof_StatResult = sizeof(StatResult);
const int offsetof_StatResult__size = offsetof(StatResult, size);
const int offsetof_StatResult__blocks = offsetof(StatResult, blocks);
```

```
24  const int offsetof_StatResult__blocksize = offsetof(StatResult,
        blocksize);
25  const int offsetof_StatResult__ino = offsetof(StatResult, ino);
26  const int offsetof_StatResult__perm = offsetof(StatResult, perm);
27  const int offsetof_StatResult__type = offsetof(StatResult, type);
28  const int offsetof_StatResult__atime = offsetof(StatResult, atime);
29  const int offsetof_StatResult__mtime = offsetof(StatResult, mtime);
30  const int offsetof_StatResult__ctime = offsetof(StatResult, ctime);
31  #endif
```

You will also notice that the exported globals are guarded by `FILE_IMPL`, this is just so we can avoid duplicate symbol definitions by importing this header in multiple files (i.e. only the filesystem module itself defines the `FILE_IMPL` macro).

Using this information which Javascript can access via the Emscripten API to lookup the symbols, a helper module for runtime reflection was written allowing for transparent access to a C structure. The code for this is at `src/glue/creflect.mjs`.

You can see in the following snippet taken from `src/filesystem/api/api.mjs` how we use, in particular, the `StructView` class to access the C struct on the WebAssembly module to interpret the pointer to `StatResult` data:

```javascript
1   function extractStatFields(statResultPtr) {
2     const stats = new StructView(Module, "StatResult", statResultPtr);
3
4     const permNum = stats.perm;
5     const type = stats.type;
6     let typeString = type == 1 ? "directory" : "file";
7
8     let permString = "";
9     if ((permNum & 0o400) == 0o400) permString += "r";
10    if ((permNum & 0o200) == 0o200) permString += "w";
11    if ((permNum & 0o100) == 0o100) permString += "x";
12
13    // Edge case for system/protected files
14    if ((permNum & 0o710) == 0o710) permString = "rx";
15
16    const atime = new StructView(Module, "Time", stats.addressof("atime
        "));
17    const mtime = new StructView(Module, "Time", stats.addressof("mtime
        "));
18    const ctime = new StructView(Module, "Time", stats.addressof("ctime
        "));
19
20    return {
21      size: stats.size,
22      blocks: stats.blocks,
23      blocksize: stats.blocksize,
24      ino: stats.ino,
```

```
25        type: typeString,
26        perm: permString,
27        atime: { sec: atime.sec, nsec: atime.nsec },
28        mtime: { sec: mtime.sec, nsec: mtime.nsec },
29        ctime: { sec: ctime.sec, nsec: ctime.nsec },
30      }
31    }
```

6.10 Unreliable Gitlab Runners

6.10.1 Problem

The Gitlab Runners used for student projects were down or in a broken state at times (generally unreliable).

6.10.2 Resolution

We had two fixes to this issue, the first was to use gitlab-ci-local to run CI on our local machines. This was also useful in general to test your code before actually pushing to gitlab. Secondly we setup self-hosted Gitlab Runners on our laptops aswell as our server which allowed us to run the pipelines after disabling the instance runners that were broken.

6.11 Compiling Emscripten to Node

6.11.1 Problem

We wanted to test the Lua runtime outside of the website. We also wanted to avoid more tests running in Puppeteer (like filesystem tests) as they are very slow and increase CI time. This meant running the tests using Node, which Emscripten does support.

6.11.2 Resolution

We had to extend our build to also build our Emscripten modules to target Node and addtionally sprinkle `typeof window === "undefined"` checks in our existing code to determine whether our global APIs were to be read from the `window` object, in a browser context, or the `globalThis` object, in a Node context.

6.12 Debugging Emscripten

6.12.1 Problem

A continous source of difficulty in this project was debugging our emscripten modules. This was especially pertenant when trying to solve the aforementioned Xterm-PTY issues.

6.12.2 Resolution

The unfortunate reality is that resolving issues with Emscripten ended up with us pain stakingly reading the many thousand auto-generated Javascript lines of code to try to understand what was happening. This was a slow process but it did work at tracking down issues with emscripten.

6.13 Supporting Readline-Like Functionality in the Terminal

6.13.1 Problem

We wanted to support interactive shell features like history, line editing keybinds and tab completion.

6.13.2 Resolution

We solved this by statically building Ncurses and Libedit into our runtime and using them inside of the shell via our exposed API function: `terminal.prompt`. We just had to make sure to configure the builds for Ncurses and Libedit to work at targetting the WebAssembly environment.

6.14 Embedding Lua System Files

6.14.1 Problem

During Hako's filesystem's bootstrap, we needed to figure out how to include Lua source files in the filesystem's initialisation, so it can check for their existence, and write them to disk if missing.

6.14.2 Resolution

We used Emscripten's 'packaging files' feature to embed the Lua source files in Emscripten's virtual file system.

Specifically, Emscripten allows you to embed files into its binary, which automatically makes them available in its filesystem.

We added the `--embed-file luaSource` build flag to the filesystem component, which made them automatically readily available in the filesystem's root, under `/luaSource`.

We could then use this to write them to the persistent filesystem mount if missing.

6.15 Simulate False Root

6.15.1 Problem

Emscripten automatically instantiates its own in-memory virtual filesystem. We then mount IDBFS onto this, which is Emscripten's FS interface implementation using IndexedDB.

However, we want this implementation detail to be hidden from the user, with all Hako system calls believing the root to be our IDBFS mountpoint.

6.15.2 Solution

We perform string manipulation in the runtime on every path provided to Hako's platform API.

This involved path normalisation and prepending our mountpoint.

A simple implementation involving a custom stack was used – although somewhat made trickier by lots of string manipulation in the C language.

6.16 Filesystem Re-write

6.16.1 Problem

The first write of Hako's filesystem was ultimately unusable.

The filesystem's implementation was fragmented between mostly Javascript and a little C – heavily leveraging the browser's Filesystem API, instead of our own custom implementation.

This was a problem as the majority of the filesystem's usage would be by Hako's runtime, which required a C filesystem API. So we would have to translate Filesystem calls from Javascript, to C, to Lua, and all the way back again – highly inefficient.

As well as this, the original Hako filesystem API wasn't ideal, and was too complex in some areas, and missing features in other areas.

6.16.2 Solution

Both developers sat down and held a planning meeting, architecting out the design for the filesystem, and specifically defining Hako's filesystem API before diving into its implementation.

This involved a full re-write in C of the filesystem.

7 Results

7.1 Technical Achievements

7.1.1 Persistent Filesystem in the Web

Hako has a persistent filesystem in the Web. It is a traditional hierarchical single-user filesystem, which can store arbitrary information.

This was achieved by using Emscripten's IDBFS, which is Emscripten's virtual filesystem built atop IndexedDB, and writing our own custom filesystem calls and permission logic.

7.1.2 Client-Side Code Execution in the Web

We successfully embedded Lua on the Web, in which execution is entirely client side using WebAssembly. Addtionally we provide an all in one platform for writing and developing of code on the Web.

7.1.3 Platform API

We designed a custom "standard library" for the Hako platform which exposes the functionality of the platform through a programmable interface. This includes functionality like operating on files, processes and the windows and appications of the Desktop.

7.1.4 Self-Hosted Coreutils

We developed a suite of applications which are subsets of the standard utilities that you would find on a Unix system. Crucially these core utilities were developed in Hako itself, entirely using the Platform API we had built. Even the shell was developed from scratch using only our Platform API.

7.1.5 Process-Emulation on the Web

Hako has a Unix-inspired process system wrapped around Web Workers. There is process orchestration (via the Process Manager), and a flat process table. Hako's process system is flat, non-hierarchical, but processes can still synchronise and communicate with eachother.

Thread-safe inter-process communication mechanisms were also developed, via Pipes (communicating with other threads) and Signals (communicating with the process orchestrator).

7.1.6 Desktop Environment

We successfully provided a rich user experience which is akin to native operating systems like Windows, MacOS, Linux and ChromeOS. This environment has its own unique style along with a focus on simplicity and intentionality.

7.1.7 Auto-Deploy System and Self-Hosting

We wrote our own deploy server in Golang. It's a simple REST server protected via an API token. It accepts artefacts and hosts them for us on our own server.

7.2 What we Learnt

We used and combined a lot of new unconventional technology that aren't inherently main-stream or common.

Specifically, new technologies that we had to become familiar with were:

- WebAssembly.
- Emscripten.
- IndexedDB.
- Meson & Ninja.
- Typescript.
- Svelte.
- Vite.
- Puppeteer.
- Playwright.
- Unity (testing framework).
- Lua.
- Web Workers.

- JS libraries (Xterm.js, Xterm-pty, CodeMirror)

Additionally there were many concepts either one or both authors were unfamiliar with in building a virtual operating system:

- Implementing a floating window manager.
- Shared memory model on the Web.
- Synchronisation of Web Workers.
- Implementing custom inter-process communication of Web Workers.
- Synchronisation and race conditions between entire system components.
- Handwritten lexing and parsing.
- Debugging system with poor tooling for inspection
- Debugging third party dependencies, not assuming external code "just works".
- Complementing project work with Open Source contribution and discussion
- Writing reliable C code under atypical constraints
- Taming the build orchastration of independent components spanning different technologies.
- Handling continuous integration of a large build system, caching certain components.
- Compiling third party C projects using Emscripten (e.g. ncurses and libedit)
- Managing the growing complexity and accumulated technical debt of a larger project relative to previous work of the authors.
- Designing an easy and usable platform API guided by the development of our self-hosted utilities.

Due to the unfamiliar and unusual mix of technology being used, there was a lot of prototyping throughout, especially early on. One interesting thing we learned the hard way was that in a project like this, adding extensive tests too early risked a lot of rework and time spent fixing incorrect assumptions. We discuss this more in our testing document.

8 Testing

See seperate testing documentation for testing content.

9 Future Work

9.1 Stretch Goals

In our functional specification, we separated some parts of the system as being stretch goals, or "nice-to-haves". In particular we proposed a peer to peer task system in which one peer would act as the

mentor, submitting tasks which could be completed by the students/other peers. This component was not completed as part of the final delivery, however we do think there was value in this idea and would suggest it as good place for future work on the project. You can see further details of this proposed feature in our functional specification.

9.2 Emscripten

Emscripten has a large amount of technical debt and using it made us eager at times to stop what we were doing and rewrite the project ourselves. The main problem we have with Emscripten is that it handles too much that it doesn't need to inside of Javascript, for example the file system APIs are implemented in Javascript almost entirely and exposed to WebAssembly, rather than the other way round. Addtitionally emscriptens functionally is provided by pasting generated code into a large Javascript file, making certain combinations of features completely broken. This large Javascript file is also paired with a WebAssembly module which makes issues very difficult to root cause as there is always a question of where an issue originated in (i.e. is it in the Javascript code or is it in the WebAssembly code ?)

9.3 Export filesystem

One feature which we think has a great use case, is being able to export your filesystem so that you can share it with different devices our amongst other people. This would allow you to also effectively back up your entire filesystem with the click of a button.

9.4 Graphics library

Exposing a graphics library, which would allow for basic rendering onto some window/canvas, as one of the core APIs of the platform would be especially great considering the target audience. Being able to teach programming and systems with more visual feedback would be a primary concern of an API like this.