Hako Testing Documentation

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

Supervisor: Prof. Stephen Blott

2025-05-02

Contents

1 Testing strategy

Hako was a very ambitious project, and ended up being a very large system, composed of almost **18,000 lines of code**.

Not only was Hako an ambitious project, but it was incredibly experimental and completely unfamiliar ground for both developers – with both students even being unsure of the capabilities of current web technology to fulfill Hako's specification.

In the end, Hako was delivered succesfully to what it was originally defined to be. In order to deliver such a difficult project under time constraint, we took upon Lean development – focusing solely on delivering value.

This heavily influenced our testing approach.

For example, early on in Hako's development, we did have a greater emphasises on testing. An example of this was having substancial tests already written for the filsystem.

But after multiple large changes to the filesystem, and even a full re-write, we discovered that preemptive testing was massively slowing us down and really unhelpful.

Our project required constant rapid prototyping and often rework. Writing tests under false or unsure assumptions created large amounts of work that provided no value.

Due to the fact that it was a complex and ambitious project, with a tight time constraint, involving constant prototyping and rework – we utilised:

- **test-as-you-go** – Writing tests after functionality has stabilised rather than upfront.
- **regression-focused testing** – Prioritise writing tests for maintaining behaviour during ongoing changes.

These methods suited us really well, and helped us ensure stability and validity of our system without sacrificing efficiency of development and allowed us to keep delivering as much value as possible.

1.1 Testing Types

1.1.1 User Testing

At the start of Hako's development, user testing was believed to be a valuable assessment of verifying Hako's end-user experience. Especially as our product was intended to be easy to use and educational.

However, **our primary demographic are children**, which is stated in all of our formal documentation.

After researching into the steps involved of performing user-testing on children, we realised this was a very difficult and delicate topic to approach – as it should be. But with the need to get Garda vetted, potential safe guarding courses, parental signatures, witnesses and more – we decided it didn't align with our Lean approach – and we were actually advised against it by our supervisor.

1.2 Static Analysis

1.2.1 Linting

Static analysis was an easy value-add for our testing approach as it's low-effort and compatible with a highly dynamic codebase such as ours.

We perform linting on our C, Javascript and Typescript code.

This is all performed in the `lint` stage in our pipeline, and has 7 jobs:

- `lint_filesystem_c`
- `lint_filesystem_js`
- `lint_filesystem_tests`
- `lint_processes`
- `lint_processes_c`
- `lint_processes_tests`
- `lint_site`

The various tools we use to perform linting are:

- Eslint – For Javascript
- TypeScript Type Checking – For TypeScript
- Clang-tidy – For C
- Cppcheck – For C

1.2.2 WebAssembly Validation

We validate our compiled WebAssembly modules to ensure they're well-formed, safe and that they conform to the WebAssembly specification.

This performs some basic operations such as:

- Checking types are correct and consistent.
- Checking stack operations are valid.
- Checking for out-of-bounds memory.

- Checking caller and callee constraints.
- Checking control flow is valid.
- and more…

This is to ensure our built WebAssembly artefacts are correct, and to reduce the likelihood that we're running unsafe code on the client.

1.2.3 Compiler Enforced Static Checks

The Meson build for compiling Hako's C modules (such as runtime) contain the flags:

- `-Wall` – Complain about all common warnings
- `-Wextra` – Enable even additional warnings, stricter checks

We then set the global Meson flag:

- `werror=true` – **fail on any warnings in the build**

This enforced strict developer discipline preventing us from ignoring warnings.

1.2.4 Code Review

Part of our software process includes Merge Requests, as discussed in the technical specification.

Before merging new Kanban items into our development branch, a code review would have to be conducted by the other member.

Explicit approval and merging was solely performed by the other member.

Often comments would be initiated by the reviewer to be resolved by the MR author.

This can be seen in our Gitlab Repository `Code > Merge Requests`, where we performed around 100 merge requests.

1.3 Adhoc Testing

1.3.1 Rapid Prototyping

Due to the difficult nature of the techology we utilised to create Hako, we were almost constantly prototyping to create various system components.

To avoid time-consuming re-work, we performed a lot of adhoc testing to verify behaviour and edge-cases. Lots of re-building and interacting with the development server, manually modifying state in the browser's console, clarifying poor assumptions, etc.

Once functionality was more stable, we would often write integration tests to maintain correct behaviour in our very dynamic codebase.

1.3.2 Dogfooding

Dogfooding is the act of utilising your own product or service to further test it in an adhoc environment.

Because Hako is a development environment – once we had our system built to a sufficient state, we actually **developed our shell and core-utils within Hako**. This was a concious decision as we knew it would force us to find a large amount of bugs that we otherwise would not have seen if we had not tested the bounds of our APIs by implementing a suite of software on the platform. Once their implementation was finished, they would then be transferred into our repository.

This was very useful in further testing of the system from the end-users perspective, and is possibly some of the most useful testing we performed overall.

Due to the amount of core-utils and the complexity of the shell, we actually spent a decent amount of time developing in Hako, and squashing many bugs, jarring experiences, etc – and improved the product overall.

1.3.3 Unit Testing

Unit tests were the most difficult to apply to our project, due to the aformentioned experimental technology, constant clarification of assumptions, incredibly dynamic codebase, and more.

From experience, we would often write plentiful tests, and then end up having to rewrite them all due to our approach changing or various incorrect assumptions arising – it was just a bad and not very useful utilisation of our time.

However, there was still room for them in extremely critical parts of Hako.

One such example is our `Pipe` interprocess communication mechanism, which is a thread-safe bounded circular buffer of shared memory. It needed unit-level testing as it was a critical piece of code, and had to be thread-safe. These are available under `src/processes/tests/pipes.test.js`.

An example testcase below creates a `Pipe` object with a buffer of 9 bytes. We then write a message longer than the buffer, reading until a newline – this is to ensure the synchronisation aspects of a full buffer, and that notifying sleeping threads works correctly for `Pipe.readLine()`.

```
1   it('readLine should stop reading at a newline message larger than
        buffer', (done) => {
2     const pipe = new Pipe(9);
3     const message = "Hello this is a test!\n";
4     const pipeBuffer = pipe.getBuffer();
5
6     const writer = new Worker('./tests/writerNoEOF.js', {
7       workerData: { buffer: pipeBuffer, message }
8     });
9
10    const reader = new Worker('./tests/readLineReader.js', {
11      workerData: { buffer: pipeBuffer }
12    });
13
14    reader.on('message', (msg) => {
15      expect(msg).to.equal(message);
16      done();
17    });
18    reader.on('error', done);
19    writer.on('error', done);
20  });
```

An example of our testing output for `Pipe` in the `test_processes` job:

```
> mocha tests/*.test.js
  Pipe Tests
    ✔ readAll should read until EOF (51ms)
    ✔ readExact should read until exactBytes consumed (46ms)
    ✔ readExact should read until EOF (53ms)
    ✔ readAll should read until EOF with wrapping buffer multiple times (46ms)
    ✔ read should read a maximum of maxBytes (47ms)
    ✔ read should stop reading at EOF (46ms)
    ✔ read should stop reading when buffer empty (47ms)
    ✔ readLine should stop reading at a newline (46ms)
    ✔ readLine should stop reading at a newline message larger than buffer (47ms)
    ✔ readLine should stop reading at EOF (47ms)
    ✔ EOF shouldn't remove existing data in pipe (152ms)
    ✔ read shouldn't consume EOF (150ms)
    ✔ readAll shouldn't consume EOF (151ms)
    ✔ readLine shouldn't consume EOF (150ms)
    ✔ readExact shouldn't consume EOF (152ms)
    ✔ should handle high throughput with multiple readers (172ms)
    ✔ isClosed should reflect whether pipe is closed or not
  17 passing (1s)
```

We perform quite a few tests on our `Pipe` implementation, testing various edgecases, and testing its concurrent safety.

Running the process tests is done by running `just test-processes` (note make sure you have dependencies installed and that `npm install` is ran in the `src/processes` directory)

1.3.4 Regression Testing

A lot of our testing focus in this project was put into regression testing. In particular we have mostly integration tests, some of which were written directly following bugs that we had fixed in the system.

The project has very diverse sets of test cases spanning different frameworks, due to the fact that the project spans various technologies.

- Mocha (JS)
- Pupeteer (JS)
- Playwright (JS)
- Node built-in test library (JS)
- Unity via Meson (C)
- Custom Lua testing library (Lua)

In particular we have the following general integration tests:

- Filesystem JS API tests – testing all of the functions in the filesystem.
- Lua Filesystem API Shim tests – testing that the usage of Lua library itself with respect to our filesystem was correct.
- Selective Platform regression tests – testing APIs that had bugs in the end user platform that had been previously introduced.

You can see that especial focus was put into ensuring that the filesystem was correct, as seeing as most components are built atop the filesystem, it is very criticial that the filesystem operates correctly – not introducing corruption or data loss.

More detail about all of these is discussed below.

1.3.4.1 Filesystem JS API Tests

These test cases are run in the `test_filesystem` job of our GitLab pipeline. You can find the source code for the tests at `src/filesystem/tests/filesystem.test.js`. These particular test cases use Puppeteer to run our code in a sandboxed headless browser. They use Mocha for defining the test cases and assertions within them.

You can see below an example of the test case that tests the `Filesystem.truncate` function. Note that everything inside of the `page.evaluate` function is run in the headless browser environment provided by Puppeteer.

```
1   it("Truncate a file after writing", async () => {
2     const assertions = await page.evaluate(async () => {
3       let fd, error;
4       let assertions = [];
5
6       ({ fd, error } = window.Filesystem.open("/persistent/truncate.txt
            ", "rwc"));
7       assertions.push({ cond: error === null, msg: "error opening file"
             });
8       ({ error } = window.Filesystem.write(fd, "Hello, world!"));
9       assertions.push({ cond: error === null, msg: "error writing file"
            });
10      ({ error } = window.Filesystem.truncate(fd, 5));
11      assertions.push({ cond: error === null, msg: "error truncating
            file"});
12      ({ error } = window.Filesystem.goto(fd, 0));
13      assertions.push({ cond: error === null, msg: "error moving cursor
            "});
14      let data, size;
15      ({ error, data, size } = window.Filesystem.readAll(fd))
16      assertions.push({ cond: error === null, msg: "failed to read file
            "});
```

```
17        assertions.push({ cond: size === 5, msg: "truncate did not reduce
              file to expected size"});
18        assertions.push({ cond: data === "Hello", msg: "truncate did not
              leave the correct data in the file"});
19        ({ error } = window.Filesystem.close(fd));
20        assertions.push({ cond: error === null, msg: "error closing file"
              });
21
22        return assertions;
23      });
24
25      for (let assertion of assertions) {
26        assert.ok(assertion.cond, assertion.msg);
27      }
28    });
```

Running the tests is done by running `just test-filesystem` (note make sure you have dependencies installed and that `npm install` is ran in the `src/filesystem` directory) You can also see the output of running these tests below:

```
> mocha tests/*.test.js
  Filesystem tests
    ✔ Check whether Filesystem object exists on the window object
    ✔ Check whether we can open a file (create)
    ✔ Confirm opening an existing file with create fails
    ✔ Check whether closing a file works
    ✔ Check we can write to an open file
    ✔ Check we can read an open file
    ✔ Check whether opening with only a read flag disallows writes
    ✔ Check whether opening with only a write flag disallows reads
    ✔ Confirm opening a non-existent file without create flag fails
    ✔ Create a file, write to it, close it, re-open it, and read it using Read
    ✔ Create a file, write to it, close it, re-open it, and read it using ReadAll
    ✔ Confirm opening protected system files for writing is blocked
    ✔ Confirm opening protected system files for reading is allowed
    ✔ Get the stat of a file
    ✔ Get the fdstat of a file
    ✔ Permit (chmod) a file
    ✔ Confirm a read-only file can't be written to
    ✔ Confirm a write-only file can't be read
    ✔ Confirm shifting an open file's cursor works
    ✔ Confirm goto of an open file's cursor works
    ✔ Confirm removing a file deletes it
    ✔ Confirm remove fails on protected system files
    ✔ Move a file
    ✔ Ensure user can't move a protected file
    ✔ Ensure user can't overwrite a protected file with a move
    ✔ Make a directory
    ✔ Remove a directory
    ✔ Read a directory
    ✔ Change active directory
    ✔ Truncate a file after writing
  30 passing (7s)
```

1.3.4.2 Lua Filesystem API Shims

We wrap our C based filesystem API in Lua shims to expose them to the Lua virtual machine. We wanted to test our use of Lua specifically without depending on the Web browser. We wrote native tests for this use case specifically as they would run directly on hardware and not through a browser running WebAssembly which makes them much faster. This did mean we had to tweak our build to be able to build the runtime as a native library as well as guard Emscripten specific things behind

preprocessor macros.

Theses tests are run in the `test_runtime` step of the pipeline. You can find the code for these tests in `src/runtime/test/file-api.c`. They use the Unity testing library to run. Below is an example test case testing that the `file.permit` Lua API we expose correctly sets the permissions on a file. You can see that it runs a large C string storing Lua code and makes assertions about what is returned by running said code.

```c
void test_file_permit(void) {
  TEST_ASSERT_MESSAGE(L != NULL, "Lua is not initialized properly");
  int top = lua_gettop(L);

  snprintf(static_fmt_buf, STATIC_FMT_SIZE,
      "local fd, err = file.open('/tmp/%d-file-permit', 'c')\n"
      "if err ~= nil then\n"
      "  return err\n"
      "end\n"
      "err = file.permit('/tmp/%d-file-permit', 'rw')\n"
      "if err ~= nil then\n"
      "  return err\n"
      "end\n"
      "local st, err = file.fdstat(fd)\n"
      "if err ~= nil then\n"
        "return err\n"
      "end\n"
      "if st.perm ~= 'rw' then\n"
      " return ''\n"
      "end\n"
      "return 0", unique_test_id, unique_test_id);
  if (LUA_OK != luaL_dostring(L, static_fmt_buf)) {
    const char *err = lua_tostring(L, -1);
    fprintf(stderr, "lua code failed to run: %s\n", err);
    TEST_FAIL();
  }
  TEST_ASSERT_EQUAL_INT_MESSAGE(LUA_TNUMBER, lua_type(L, -1), "the
      permissions are wrong");
  TEST_ASSERT_EQUAL_INT_MESSAGE(0, lua_tonumber(L, -1), "api function
      returned an error code");

  lua_settop(L, top);
}
```

To run these tests you need to run `just test-runtime`. The output of them is below (removing any build output ran before the tests).

```
559  ---------------------------------- output -----------------------------------
560  ../../src/runtime/test/file-api.c:553:test_file_open:PASS
561  ../../src/runtime/test/file-api.c:554:test_file_close:PASS
562  ../../src/runtime/test/file-api.c:555:test_file_write_and_read:PASS
563  ../../src/runtime/test/file-api.c:556:test_file_read_all:PASS
564  ../../src/runtime/test/file-api.c:557:test_file_shift:PASS
565  ../../src/runtime/test/file-api.c:558:test_file_jump:PASS
566  ../../src/runtime/test/file-api.c:559:test_file_remove:PASS
567  ../../src/runtime/test/file-api.c:560:test_file_move:PASS
568  ../../src/runtime/test/file-api.c:561:test_file_make_dir:PASS
569  ../../src/runtime/test/file-api.c:562:test_file_remove_dir:PASS
570  ../../src/runtime/test/file-api.c:563:test_file_change_dir:PASS
571  ../../src/runtime/test/file-api.c:564:test_file_read_dir:PASS
572  ../../src/runtime/test/file-api.c:565:test_file_stat:PASS
573  ../../src/runtime/test/file-api.c:566:test_file_fdstat:PASS
574  ../../src/runtime/test/file-api.c:567:test_file_permit:PASS
575  ../../src/runtime/test/file-api.c:568:test_file_truncate:PASS
576  ----------------------
577  16 Tests 0 Failures 0 Ignored
578  OK
579  -------------------------------------------------------------------------------
580  Ok:                 1
581  Expected Fail:      0
582  Fail:               0
583  Unexpected Pass:    0
584  Skipped:            0
585  Timeout:            0
```

> NOTE: 16 tests are run under a higher level Meson test step “Test lua file API”.

1.3.4.3 Selective Platform Regression Tests

At one point in the project, apart from the Lua shim tests, we did not have a place to test other APIs. In a similar train of thought to Lua shim tests, we wanted to avoid more in-the-browser tests as they were slower to run. For this reason we decided to alter our build again to additionally have the runtime target node, which in this case would be faster than the browser based tests but slower than the native compiled tests. The upside however is that unlike the native tests we would be able to test things like processes as node has its own form of worker threads that Emscripten supports.

These tests are ran in the `test_integration` step of the pipeline. They actually use a small set of Lua functions we wrote ourselves for asserting conditions and checking errors inside of our on plat-

form. You can find the source code for the tests in the `src/test/` directory, which holds the runner Javascript file `integration.mjs` and Lua code `integration.lua` which is instantiated to run inside of our runtime by the aforementioned Javascript file.

You can see an example test case below that tests that you can pipe data between two processes. This was written after finding bugs in our pipe implentation while creating processes in the runtime.

```
test("Pipes", function ()
  local data = "xhR2KIyEUBQZLD7laHT7nouF0jY7byCSKhBcXHddit3bo8Tmq+kuhfvq7E9R3TRphWRzcjsVemh2"
  local writer_src = string.format([[
    output("%s", { newline = false })
    process.close_output()
  ]], data)

  local reader_src = [[
    local inp, err = input_all()
    if err ~= nil then
      output(err)
      error("reader failed")
    end
    assert(inp ~= nil)
    local fd, err = file.open("/return", "wc")
    if err ~= nil then
      output(err)
      error("reader failed")
    end
    file.write(fd, inp)
    file.close(fd)
  ]]

  ensure_file("/pipe-writer.lua", writer_src)
  ensure_file("/pipe-reader.lua", reader_src)

  local wtr = unwrap("process.create", "/pipe-writer.lua", { pipe_in = true, pipe_out = true })
  local rdr = unwrap("process.create", "/pipe-reader.lua", { pipe_in = true, pipe_out = false })

  unwrap("process.pipe", wtr, rdr)
  unwrap("process.start", rdr)
  unwrap("process.start", wtr)
  unwrap("process.wait", rdr)

  check(data == filedata("/return"), "Reader outputted unexpected text")
  unwrap("file.remove", "/return")
end)
```

You can run the tests using `just test-integration`. The output will look like so:

```
322  Finished bootstrap
323  Runtime emscripten module loaded
324  Attached worker to registeredProcess:
325  [JS] Sync completed succesfully!
326  Runtime emscripten module loaded
327  Attached worker to registeredProcess:
328  Runtime emscripten module loaded
329  Attached worker to registeredProcess:
330  [JS] Sync completed succesfully!
331  [JS] Sync completed succesfully!
332  TEST:0: Pipes :PASS
333  TEST:1: File does not exist :PASS
334  TEST:2: File needs open write :PASS
335  TEST:3: File read all :PASS
336  4 Tests 0 Failures
```

1.3.5 End-To-End Testing

For testing the UI we did look into using a unit testing library like Vitest which would work with well with Vite (used by SvelteKit). The problem is that for our project some of our dependencies, in particular Xterm.js, did not work properly with the DOM emulation libraries like Happy-dom and Jsdom. This led us to use a headless browser based solution similar to what we did for the Filesystem JS tests. In this case we used Playwright.

These test cases test that when user interacts with the UI itself the system responds correctly. You can find these tests under the `src/site/tests/` directory. An example test case that tests that you can create a file in the File Manager application is below. Note that it uses the UI programatically to

create a file via the context menu and checks using the Filesystem JS API that the file was successfully created.

```
test('Create file', async ({ page }) => {
  showConsole(page);
  await page.goto('/');

  await waitCustom(page, "loaded");
  await openFileManager(page);

  const fileManager = page.locator('#window-0');
  const boundingBox = await fileManager.boundingBox();

  if (boundingBox) {
    const centerX = boundingBox.x + boundingBox.width / 2;
    const centerY = boundingBox.y + boundingBox.height / 2;
    await page.mouse.click(centerX, centerY, { button: 'right' });
  }

  await page.click('div:text("New File")');
  await page.keyboard.insertText("MyFile");
  await page.keyboard.press("Enter");

  // Check that the file is visible in the file manager
  expect(page.locator('p:text("MyFile")')).toBeAttached();

  // Check that the file is created in the filesystem
  expect(await page.evaluate(async () => {
    let { entries } = window.Filesystem.read_dir("/persistent");
    return entries.includes("MyFile");
  })).toBe(true);
});
```

You can run these test cases in the `src/site/` directory using `npm run test`. The output will look like so:

```
$ playwright test

Running 16 tests using 2 workers

  ✓  1 tests/terminal.test.ts:10:1 › Terminal closes on Ctrl-D (1.2s)
  ✓  2 tests/file-manager.test.ts:10:1 › Create file (5.8s)
  ✓  3 tests/terminal.test.ts:24:1 › Terminal closes on Ctrl-C (985ms)
  ✓  4 tests/windows.test.ts:4:1 › Open terminal (441ms)
  ✓  5 tests/windows.test.ts:15:1 › Close terminal (674ms)
  ✓  6 tests/windows.test.ts:29:1 › Open file manager (431ms)
  ✓  7 tests/windows.test.ts:40:1 › Close file manager (631ms)
  ✓  8 tests/windows.test.ts:54:1 › Open editor (435ms)
  ✓  9 tests/windows.test.ts:65:1 › Open editor file dialog (511ms)
  ✓  10 tests/windows.test.ts:83:1 › Cancel editor (673ms)
  ✓  11 tests/file-manager.test.ts:40:1 › Delete file (614ms)
  ✓  12 tests/windows.test.ts:97:1 › Open manual (538ms)
  ✓  13 tests/file-manager.test.ts:73:1 › Create directory (589ms)
  ✓  14 tests/windows.test.ts:108:1 › Close manual (909ms)
  ✓  15 tests/file-manager.test.ts:103:1 › Delete directory (579ms)
  ✓  16 tests/windows.test.ts:122:1 › Open terminal creates runtime (471ms)

  16 passed (11.9s)
```

1.4 Continous Software Engineering

Finally, Continuous Software Engineering was utilised, performing the aforementioned stages when pushing to our origin for continuous integration and deployment:

- Static Analysis
- Unit Tests
- Integration Tests
- End-to-End Tests

Furthermore, if we're on the `main` branch, and all tests pass, a deploy is made to our self-hosted custom deploy server.

Our pipeline on the `main` branch can be seen below: