GSoC 2022 Proposal | Rewrite PJDFSTest suite

Introduction

This project aims to rewrite the PJDFSTest suite. Today, the tests are written in a mix of shell script and C. This approach has provided some flexibility and usability, allowing to use syscalls within a shell environment. However, it also has disadvantages, the main ones being performance, code duplication and higher entry barrier for potential contributors. We want to improve the test suite, by switching to a unique language, Rust. Rust has numerous advantages, in particular, fearless multithreading, low-level handling and increased safety.

TL;DR Rewrite the tests in Rust, with a custom-built test runner for running them standalone, and rely on Kyua for running tests along with ATF support to get reports.

Mentor: asomers (Alan Somers <asomers@FreeBSD.org>)

whoami?

Name: Sayafdine Said
GitHub: musikid
Discord: musikid.#7043
Matrix: musikid64
Mail address: musikid@outlook.com
Languages: French, English
Timezone: Paris, France (UTC+2)

Currently a computer science student at Sorbonne University, I am proficient in Rust, Python, C/C++ and JavaScript. I run Arch Linux and FreeBSD, so I am pretty familiar with the Unix environment. I wrote fixes for some open source projects, but never been more involved. I also did a lot of yak shaving, fancy being the most complete project I produced. This GSoC is the perfect occasion to be involved in the open source community, particularly for the FreeBSD project, that I love!

Summary

We want to write a Rust binary, which:

- is self-contained (embed all the tests),
- can execute all the tests (with the test runner),
- can filter according to conditions,
- *is compatible with ATF*.

Details

Test collection

The project will not rely on the Rust testing framework, therefore, we will need to do the test collection ourselves. As an example, Criterion uses declarative macros:

- to collect the tests in a group (`criterion_group!`), which is turned into a function,
- and aggregate all these functions into the main one (`criterion_main!`).

We want to adopt a similar approach, with some nuances, however.

Since we want to be able to list the tests, collecting them in a function seems troublesome. Instead, we will collect them in a slice, along with/within a structure for easing listing them.

Test runner

The test runner will not need to have a lot of fancy features. It has to support:

- filtering/skipping tests according to conditions,
- printing the status,
- reading a configuration file.

Configuration file

The format needs further investigation, but we should at least support filtering by syscalls availability.

Macros

`pjdfs_group!` Make a group of tests.

`pjdfs_main!` Make the main function.

NOTE: The next macros are not top priorities, more of a convenience if I already implemented the first ones correctly.

`#[pjdfs_test(conditions)]` Declare a test.

The attribute might come with arguments to declare conditions. For example, some tests need to be run as root, or some syscalls are not available everywhere. So, we can imagine an interface analogous to `#[cfg]`. For example:

```rust
#[pjdfs_test(user = "root")]
fn my_test() {
}
```

In the experimental Python rewrite, Python decorators are used.

`#[case(args)]`

Declare parameterization, used with `pjdfs_test`.

We take inspiration from pytest parameterize and rstest case, and add a new parameter for the test.

`#[fixture(conditions)]` Declare a fixture.

We take inspiration from pytest and rstest, and add the fixture to the test's parameters.

The attribute might come with arguments to declare conditions (like `pjdfs_test`). Because we want to avoid duplication, when tests share conditions, we can declare a common fixture.

For example:

```rust
#[fixture(user = "root", feature = "posix_fallocate")]
fn posix_fallocate_and_root_user() -> Something {}

#[pjdfs_test]
fn posix_fallocate_1(posix_fallocate_and_root_user: Something) {}

#[pjdfs_test(my_own_condition = "yes")]
fn posix_fallocate_2(posix_fallocate_and_root_user: Something) {}
```

Layout

Currently, tests are organized by syscalls, and the rewrite should adopt a similar approach. Like explained in the previous section, we declare the tests by groups.

For example:

symlink/mod.rs

```rust
pjdfs_group!(posix_fallocate, posix_fallocate_1, posix_fallocate_2);
```

main.rs

```rust
pjdfs_main!(posix_fallocate);
```

Command-line arguments

The command-line interface should also be compatible with ATF. It has a really simple interface, consisting only of two running modes:

- `-l`, to list all the tests and their conditions.
- `[-r resfile] [-s srcdir] [-v var1=value1 [.. -v varN=valueN]] test_case`, to run a test case.

Otherwise, running the program without arguments should call the runner with all the tests meeting the current conditions.

For further configuration, we might also add CLI arguments in addition of the configuration file, however, it's not a priority compared to having a configuration file.

ATF support

To support ATF, we need to be able to output the tests in ATF metadata format. The format is fairly simple, consisting only of `key: value` pairs. For example:

```
ident: test_1
descr: this is a test
require.user: root
```

ATF supports specifying some conditions. However, except for privileges, I did not find any condition which intersects with the ones used in the project. Thus, the only required keys to support are `ident` and `descr`. We can also add `require.user`, and eventually `timeout` if we want to delegate it to `kyua`.

Multithreading

We need to provide test isolation to be able to support multithreading. Further investigations are needed.

Timeline

After implementing the test collection along with the fixtures, I will to write a single-threaded test runner. Then, as a secondary objective, I aim to add ATF support so we can rely on `kyua` test runner, to inherit from its high quality reporting. Finally, I will try to add support for multithreading to our test runner.

Though, adding multithreading is definitely not a priority, neither is ATF support.

NOTE: All the time intervals imply writing the documentation along with the code.

Community bonding (May 20)

- Get more familiar with the current codebase
- Review the missing syscalls/functionality
- Determine the test structure's shape
- Determine the configuration file fields
- Discuss on other details

1st week (June 13)

- Iterate on the project's design

2nd week - 4th week (June 20)

- Start implementing test collection

5th - 7th weeks (July 11)

- Implement test runner
- Start writing the tests

Phase 1 evaluation period (July 25)

8th - 9th weeks (August 1)

- Continue writing the tests

10th - 12th weeks (August 15)

- Continue writing the tests
- For unexpected delay
- *Implement test macros*
- *Add ATF support*
- *Add multithreading support*

13th week - End (September 4)

- Document extensively
- For unexpected delay

Relevant links

https://github.com/pjd/pjdfstest/issues/59

https://github.com/musikid/pytest-atf.git

https://www.freebsd.org/cgi/man.cgi?query=atf-test-program&sektion=1&apropos=0&manpath=FreeBSD+13.0-RELEASE+and+Ports

https://www.freebsd.org/cgi/man.cgi?query=atf-test-case&sektion=4&apropos=0&manpath=FreeBSD+13.0-RELEASE+and+Ports

https://nexte.st/book/how-it-works.html

https://github.com/jmmv/kyua/wiki/About