Enhancing FreeBSD Test Suite Parallelism with Kyua's Jail Feature

BY IGOR OSTAPENKO

Testing is a rather broad concept today. Regardless of approaches, academic views, or specific situations, it is difficult to resist the simple desire not to force the end user to test on our behalf. Even a simple error can significantly harm the business in multiple ways: reputation, time-to-market, conversion rate, expansion, etc. The industry has evolved from relying on complex manual checks before release to embracing automation whenever and wherever possible. Automated testing offers its benefits: shorter cycles, more frequent feedback, additional confidence in the result, less fear of changes, and so on. Even if automated testing is a significant part of the development process, it is just one of many practices improving software delivery.

It is difficult to resist the simple desire not to force the end user to test on our behalf.

Organizing Tests with Kyua

The FreeBSD Test Suite is typically located in `/usr/tests`, with its infrastructure built on Kyua, a testing framework created by Julio Merino. Kyua offers an expressive test suite definition language, a safe runtime engine, and powerful report generation. Tests can be written using anything or without relying on a specific library. The unit and integration tests found in the suite often utilize libraries like `atf-c(3)`, `atf-c++(3)`, `atf-sh(3)`, or `pytest`.

Kyua operates with the following hierarchy of key concepts: test suite > test program > test case.

A test suite groups several binaries (test programs) into one set with a single name. A Lua script is used to describe a test suite, which is typically saved in a special kyuafile(5). Let's consider the following existing file:

```
# cat -n /usr/tests/sys/kern/Kyuafile
       1  -- Automatically generated by bsd.test.mk.
       2
       3  syntax(2)
       4
       5  test_suite("FreeBSD")
       6
       7   atf_test_program{name="basic_signal", }
```

```
[skipped]
       34  atf_test_program{name="sonewconn_overflow", required_programs="python",
required_user="root", is_exclusive="true"}
       35  atf_test_program{name="subr_physmem_test", }
       36  plain_test_program{name="subr_unit_test", }
[skipped]
       45  atf_test_program{name="unix_seqpacket_test", timeout="15"}
       46  atf_test_program{name="unix_stream", }
       47  atf_test_program{name="waitpid_nohang", }
       48  include("acct/Kyuafile")
       49  include("execve/Kyuafile")
       50  include("pipe/Kyuafile")
```

Line #3 specifies the required version of the syntax used. Line #5 sets the name for the test suite. Usually, all **`/usr/tests/**/Kyuafile`** descriptions are collected into a single test suite named FreeBSD. If a binary is based on ATF libraries, it's registered using **`atf_test_program`**, so Kyua can leverage ATF capabilities and specifics for such a test program. If it's not based on a library supported by Kyua and simply communicates results via exit code, **`plain_test_program`** construct is used instead. There is also **`tap_test_program`** for test programs that communicate results using the good old Test Anything Protocol.

Each Kyuafile describes test binaries only within its directory. However, **`/usr/tests`** are structured the way that each test directory explicitly includes its subdirectories, as illustrated in lines #48, #49, and #50. Consequently, running tests in this directory will execute all tests from the **`sys/kern`** sub-tree, including those in **`sys/kern/acct`**, **`sys/kern/execve`**, and **`sys/kern/pipe`**:

```
# kyua test -k /usr/tests/sys/kern/Kyuafile
```

Line #1 indicates that Kyuafile is not created manually in the FreeBSD Test Suite. Instead, as with most components of the FreeBSD build system, the process is handled through a Makefile, which builds the test programs and generates the corresponding Kyuafile. To understand the process in detail, the generated **`/usr/tests/sys/kern/Kyuafile`** can be compared with its source in /**`usr/src/tests/sys/kern/Makefile`** — it's a straightforward approach.

A test program that is not registered in the Kyuafile will not be recognized by Kyua and therefore will not be executed.

There is no explicit mention of test cases in the Kyuafile because test cases are defined at a lower level, within a test program, and it requires support from the library used. It's often more convenient to group several similar tests (i.e. test cases) within a single test program binary rather than creating multiple separate test programs. Plain test programs are expected to provide only one test case, typically named "main" after the **`main()`** function. In contrast, tests based on ATF libraries can report multiple test cases. In Kyua, what we generally call a test is referred to as a test case, which is treated as a unit of execution. Consequently, a test suite described in a Kyuafile might seem to reference only a single test program, but it could contain dozens or more test cases. The **`kyua list`** command lists test cases in the format **`<test program>:<test case>`**, which can also be used with other commands, for example, to run a specific test case individually:

```
# cd /usr/tests/sys/kern
# kyua test unix_dgram:basic
```

Each test case can have optional metadata properties in the form of key/value pairs, which modify Kyua's behavior for that specific test case. The Kyuafile example above shows that the same metadata can be applied to all test cases within a test program. The illustrated properties are:

- `timeout` allows changing the default value, which is 300 seconds.
- If the binaries specified in `required_programs` are not found either by their full path or within the `PATH`, the test case will be marked as skipped with the respective message.
- `required_user="root"` will skip the test if Kyua is not running with root privileges, while `required_user="unprivileged"` will ensure that the test is run without root access rights.
- `is_exclusive="true"` specifies that the test cannot be run concurrently with other tests.

Parallelism and Jails

Kyua can run test cases in parallel when configured to do so. By default, the `parallelism` setting is set to 1, which means tests are run sequentially. This can be adjusted in `kyua.conf(5)` or specified as an option:

```
# kyua -v parallelism=8 test
```

Test cases that require exclusive access to a shared resource should be marked with `is_exclusive="true"` so Kyua knows not to run them in parallel with others. Kyua operates in two phases. First, it runs all non-exclusive test cases, which can be executed in parallel if configured so. The second phase runs all exclusive test cases sequentially. To keep the test suite efficient, it's best to avoid adding new exclusive tests when possible and create non-exclusive versions instead. Otherwise, the test suite may become too time-consuming to execute.

However, some tests utilize the `jail(8)` feature to handle scenarios that are otherwise difficult to reproduce. For example, network module testing often involves creating temporary jails to verify module behavior by poking it from the host through `epair(4)`. There are several reasons why such tests must be marked as exclusive: they usually re-use the same jail naming for convenience (while each jail in the system must have a unique name), the same IP addresses officially allocated for demo purposes are used to configure interfaces on the host side leading to potential conflicts with a shared routing table, and other related issues. While these problems could be addressed by test cases themselves, doing so would significantly increase the complexity for test authors and maintainers, and some issues might be impossible to resolve without external intervention. This is where the latest Kyua version comes into play.

To keep the test suite efficient, it's best to avoid adding new exclusive tests.

Execution Environment Concept

In 15-CURRENT Kyua provides a new concept called "execution environment". It's going to be available from 14.2-RELEASE.

By default, tests continue to run as before by spawning child processes — this is referred to as the host execution environment. A test case can opt-in to use a different execution environment by specifying a new metadata property called **`execenv`**. The general sequence of steps applied to each test case has been extended to include the following:

1. Execution environment initialization
2. Test execution
3. (optional) Test cleanup
4. Execution environment cleanup

Currently, Kyua supports only one additional execution environment — the jail environment. While it can be configured for individual test cases, the following example shows how to apply the execenv metadata property to all test cases within a test program:

```
atf_test_program{name="test_program", execenv="jail"}
```

This configuration causes Kyua to provide each test case in the **`test_program`** with its own temporary jail in which to execute. If a test case declares a cleanup routine, it will be executed within the same jail. Kyua uses `jail(8)` for creating these jails, and test cases can pass additional parameters through a new metadata property called **`execenv_jail_params`**:

```
atf_test_program{name="test_program", execenv="jail", execenv_jail_params="vnet allow.raw_
sockets"}
```

As long as the names of sub-jails do not conflict among different parent jails, and each jail can have its own VNET stack, we can easily isolate tests — such as the network tests mentioned earlier — into separate jails and run them in parallel by removing the **`is_exclusive`** flag. It depends on environment and configuration, but there are reports that **`netpfil/pf`** test suite runs 4 or 5 times faster using the same environment — taking just a few minutes instead of half an hour.

Implicit Parameters and Hierarchical Jails

Since a test case and its optional cleanup routine run in separate child processes, Kyua implicitly appends the **`persist`** parameter to keep the temporary jail alive, allowing both child processes to run within the same jail. Kyua ensures that the temporary jail is removed during the "Execution environment cleanup" step.

Spawning jails by network tests is a common practice. This raises the question of whether a test case, which is already running inside a jail, is permitted to create sub-jails. In principle, this is allowed as long as system limits are not exceeded. Each jail has a limit on the number of sub-jails that can be created. The following new read-only sysctl variables, introduced in 15-CURRENT, provide this information:

```
# sysctl security.jail.children
security.jail.children.cur: 0
security.jail.children.max: 999999
```

Apparently, the above looks to be the highest jail in the hierarchy, so called **`prison0`**, and almost a million of jails can be created according to the current and maximum values. When jail(8) is used to create a new jail, it applies the following default configuration:

```
# jail -c command=sysctl security.jail.children
security.jail.children.cur: 0
security.jail.children.max: 0
```

This indicates that no sub-jails are allowed. Obviously, test cases attempting to create new jails would fail under these conditions. To address this, Kyua assists by adding another implicit parameter that allows the maximum number of child jails, calculated as the parent jail's maximum limit minus one. Although it's possible to configure this from the test case side using the **execenv_jail_params** metadata property, it appears to be cumbersome and repetitive work.

The following formula clarifies how Kyua creates temporary jails and how this process can be modified using metadata properties:

```
jail -qc name=<name> children.max=<parent_max-1> <test case defined params> persist
```

The name of a temporary jail is derived from the test program path and test case name. For example, the test case **/usr/tests/sys/kern/unix_dgram:basic** will use a temporary jail named **kyua_usr_tests_sys_kern_unix_dgram_basic**.

kldload concerns

Since all jails, except for **prison0**, lack the privilege to load kernel modules, this creates inconveniences if a test case relies on the jail execution environment.

Kyua's original philosophy is to be usable by both developers and users. This means that a system administrator should be able to run the test suite after OS upgrade to ensure everything is functioning as expected. Clearly, such a host is not a test lab where developers can freely experiment, break things, or cause fire. Therefore, tests should be designed to avoid disrupting the normal operation of the host unless explicitly instructed. That's why the FreeBSD Test Suite has configuration variables like **allow_sysctl_side_effects** to follow this approach. Despite the fact that the suite is primarily treated as a developer tool, many existing tests adhere to this principle by checking if a required module is loaded, rather than loading it implicitly. A system administrator would not appreciate it if, for example, tests of a firewall, which is not used by a host, inadvertently affect its traffic or even make it inaccessible.

Therefore, the recommended strategy is to use **kldstat -q -m <module-name>** within the test case to check for the presence of required modules and skip the test if the module is not found. The configuration of the FreeBSD CI ensures that all necessary modules are loaded and required software packages are installed before running the test suite.

execenvs and WITHOUT_JAIL

A new engine configuration variable is provided — **execenvs**. By default, it is set to a list of all supported execution environments:

```
# kyua config
architecture = aarch64
execenvs = host jail
parallelism = 1
platform = arm64
unprivileged_user = tests
```

This variable can be manipulated through kyua.conf(5) or specified as an option for the kyua(1) CLI. For instance, the following command will execute only host-based tests and skip all others:

```
# kyua -v execenvs=host test
```

If the system is built without jail support, only the default host execution environment will be available. Consequently, any tests that require the jail execution environment will be skipped.

Getting Started Examples

The following example, based on atf-sh(3), illustrates how to configure the jail environment at the test case level. It also reminds the importance of root user privileges.

```
# cat /usr/src/tests/sys/kern/test_program.sh
atf_test_case "case1" "cleanup"
case1_head()
{
       atf_set descr 'Test that X does Y'
       atf_set require.user root
       atf_set execenv jail
       atf_set execenv.jail.params vnet allow.raw_sockets
}
case1_body()
{
       if ! kldstat -q -m tesseract; then
               atf_skip "This test requires tesseract"
       fi

       # test code...
}
case1_cleanup()
{
       # cleanup code...
}

atf_init_test_cases()
{
       atf_add_test_case "case1"
}
```

A single line addition to the Makefile is enough for this test program:

```
# grep test_program /usr/src/tests/sys/kern/Makefile
ATF_TESTS_SH+= test_program
```

The build system will prepend the `#!/usr/libexec/atf-sh` shebang line, install the script without the `.sh` extension at `/usr/tests/sys/kern/test_program`, and register it in the `Kyuafile` accordingly:

```
# grep test_program /usr/tests/sys/kern/Kyuafile
atf_test_program{name="test_program", }
```

Having multiple test cases within a single test program can lead to a Don't Repeat Yourself situation. To handle this, common metadata can be moved up to the test suite level in a Kyuafile, allowing it to apply to the entire test program rather than repeating it for each test case. However, individual test cases can still override these properties if necessary:

```
# cat /usr/src/tests/sys/kern/test_program2.sh
atf_test_case "case2"
case2_head()
{
       atf_set descr 'Test that A does B'
}
case2_body()
{...}

atf_test_case "case3"
case3_head()
{
       atf_set descr 'Test that Foo does Bar'
       atf_set execenv.jail.params vnet allow.raw_sockets
}
case3_body()
{...}

atf_init_test_cases()
{
       atf_add_test_case "case2"
       atf_add_test_case "case3"
}
```

Now the main configuration is provided on the test program level:

```
# grep test_program2 /usr/src/tests/sys/kern/Makefile
ATF_TESTS_SH+= test_program2
TEST_METADATA.test_program2+= execenv="jail",execenv_jail_params="vnet"
```

As a result, Kyua consolidates metadata defined at different levels into the following:

```
# kyua list -k /usr/tests/sys/kern/Kyuafile -v test_program2
test_program2:case2 (FreeBSD)
    description = Test that A does B
    execenv = jail
    execenv_jail_params = vnet
test_program2:case3 (FreeBSD)
    description = Test that Foo does Bar
    execenv = jail
    execenv_jail_params = vnet allow.raw_sockets
```

It's important to note the key difference in metadata property naming conventions between ATF and Kyua — dots (`execenv.jail.params`) versus underscores (`execenv_jail_params`). Additionally, names themselves may vary slightly, the `kyuafile(5)` and `atf-test-case(4)` manual pages can be compared for that.

To switch an existing test to the jail execution environment, the `is_exclusive="true"` metadata property should be negated or removed. Otherwise, the test will not benefit from parallel execution.

Further Reading

The entry point to the FreeBSD Test Suite is described in tests(7). For test authors, the following wiki page is a valuable starting point: https://wiki.freebsd.org/TestSuite/DeveloperHowTo.

The official Kyua wiki is an excellent resource for historical aspects, design rationale, and feature overviews. Detailed information on execution environments can be found in the kyua.conf(5) and kyuafile(5) manual pages.

Also, reviewing how existing jail-based tests are written and organized is crucial to avoid reinventing the wheel. The PF test suite located in `/usr/src/tests/sys/netpfil/pf` is a great source for understanding of the established practices.

While retroactively adding tests to the existing code can be an enormous effort, incorporating tests that address bug fixes is a worthwhile opportunity to enhance the FreeBSD Test Suite, and therefore the project as a whole.

IGOR OSTAPENKO is a FreeBSD contributor with a wide range of software development experience in various areas, whether it's systems for manipulating and testing navigation devices, enterprise solutions for optimizing business processes, reverse-engineering, or B2B/B2C startups.