

Automated testing framework for The NetBSD Operating System: Brainstorming

Julio M. Merino Vidal <jmmv@NetBSD.org>

December 15, 2006

Abstract

This document contains an overview of random requirements and ideas that should be made part of an automatic testing framework for The NetBSD Operating System. Therefore keep in mind that anything exposed here can change (to better hopefully) during the effective development of the project.

This project is being presented as a possible candidate for my undergraduate thesis for the *Enginyeria en Informàtica* at the *Facultat d'Informàtica de Barcelona*, on which I will be working next semester.

Note that many examples in the text talk about tmpfs's regression test suite. This is because I wrote tmpfs (as part of Google's Summer of Code 2005 program) along its test suite and found it very annoying that there was no consistent framework to write nor run the tests. Hence this project proposal.

1 Project overview

1.1 The problem

An operating system is an extremely complex piece of software. It requires automatic testing to ensure its features work as expected and that they do not break over time.

At the moment, NetBSD includes some (few) regression tests in the `src/regress`¹ source tree. It is not possible to run all of them automatically and therefore few people uses them. As a result, some of these tests are already broken in some platforms supported by NetBSD and they can even panic the kernel!

1.2 Project goals

The following list shows all the ideal goals of the project. Some of them may be very hard to achieve and therefore might not be done during the time of a single semester. It should be negotiated which goals to prioritize.

- Design and implement an automated testing system for The NetBSD Operating System. This should include all possible test types such as regression tests, unit tests, etc.
- Consolidate the test results into a single and easy to read report.
- Support testing of both the kernel and user-land parts of the system.
- Minimize code duplication.
- Support multiple platforms (ideally all those supported by NetBSD).
- Ability to test the whole operating system at will after installation, without the need of the source tree.

¹<http://cvsweb.NetBSD.org/bsdweb.cgi/src/regress/>

- Allow some kind of benchmarking.
- Migrate existing tests to the new framework to demonstrate that it works and that it is flexible enough.
- Write new tests for common system functionality, to some extent. Full coverage is not the objective of this project; this could be an extremely time-consuming task and could require full knowledge of all areas of the system, something that no person alone has.

1.3 Why NetBSD?

The following list outlines multiple reasons to develop this system under NetBSD and not, e.g., a Linux distribution:

- NetBSD is a complete system: the test framework can be used to check both the kernel and user-space applications as a whole. This could be impossible under a GNU/Linux system. There you already have individual test suites for several components (`coreutils`, `textutils`, etc.) but you cannot reliably have a single test suite that checks everything including Linux (the kernel) itself.
- NetBSD focuses on correct and clean design. It could be great to also demonstrate that this produces correct code, and one way to do so is to have a large test suite that can be run by the system users at will.
- Being able to demonstrate that NetBSD behaves correctly in multiple areas can raise interest in this operating system in new researchers, system administrators needing to configure solid machines, developers and even end users.
- There is clearly a need for such a test system in NetBSD. As there is no way to run the existing tests easily they eventually break and the problems are not discovered until much later. This results in some test cases being able to, e.g., trivially crash the system.
- There is a lot of interest in this project by several NetBSD developers which means that it will be a great contribution to a big open source project. In fact, I presented this proposal to the Google Summer of Code 2006 program and it was one of the final sponsored candidates. Unfortunately, I could not work on it because I presented another project which was also selected, and I had more interest to work on that one.
- I am already familiar with NetBSD and it is my favorite free operating system.

1.4 Deliverables

This project will deliver the items listed below. As mentioned in the goals section, these may require a lot of work so some of them should be negotiated depending on the available time for the overall project.

- The C monitor: A library for test programs written in C that implements an homogeneous command-line interface across programs and provides common functionality required during tests (logging, test cases, detection of required features, etc.). See section 3.2.
- The POSIX shell monitor: Same as the C monitor but for test programs written in the POSIX shell language.
- The `regress` utility: A program that provides an interface to automate the execution of tests, collect the results, format them, etc. It is a higher-level interface than that offered by the test programs themselves. See section 2.5.
- The test daemon: It is still unknown if this will be needed. But it may be useful to have a daemon run on slave machines that handles the remote execution of tests. See section 2.3.

- The `mkimage` utility: A program that creates a disk image, installs NetBSD into it, configures several details and finally executes a test inside it. See section 4.1.
- Conversion of existing tests: A conversion of all the existing tests inside `src/regress` to the new framework.
- New tests: Several new tests for other system functionality that is still uncovered. These will be provided mostly to illustrate all the different possibilities of the new framework but, if time permits, several tests for common functionality will also be implemented.

2 Testing infrastructure

The basic idea behind the new testing framework is to allow users of the system (administrators, researchers, end users, etc.) to run a full set of tests after installation *without having to deal with the system's source tree*. Therefore tests will be built and installed under a new hierarchy in the file system, something that will happen during a regular build of the whole operating system. In the examples below I will assume that such hierarchy is placed in `/usr/tests` and is distributed alongside formal system releases in a `tests.tgz` set².

2.1 Hierarchical organization

Tests are organized hierarchically forming a tree:

Test case The most basic testing unit. This includes, e.g. testing for equality (value returned by a function against the expected result) or testing for correct execution (whether a program exited with success or not).

Test program A test program is a collection of related test cases and can be seen, as a whole, as a test case for some specific functionality. It is standalone which means that it can be directly executed by the user from the command line.

Test suite A collection of related test programs (represented as a directory in the file system). Test suites can be nested (simply using subdirectories) and the most generic one tests the whole operating system.

This concept separation can already be found in the existing regression tests and is very common in other testing frameworks out there. For example: there is a test suite for tmpfs `regress/sys/fs/tmpfs` composed of multiple test programs (`t_mkdir`, `t_create`, etc.). Each test program contains multiple basic test cases. So, at the high level, you see `t_mkdir` is a test case for the `mkdir(2)` operation on a tmpfs file system, but on a lower level, `t_mkdir` is composed of several other test cases that stress-test that same operation in multiple ways.

2.2 Makefiles

The current test infrastructure relies on `make(1)` to build and run tests. This requires the availability of a source tree, all the development tools and, most importantly, keeps a lot of testing logic in the build scripts themselves.

In the new framework, the build infrastructure should be as simple as possible and must not contain any testing logic (such as logging); otherwise the test programs could not be executed standalone. In other words, `make(1)` must be used for what it was designed for: tracking dependencies, building the test programs and any required helpers.

Somewhat related to this, the tests are currently stored in a tree that is separate from the regular sources. Given the way that tests could be executed in the new framework, there is no need to keep this separation any more so the tests could be moved to where they make most sense: as close to the sources as possible.

²See <ftp://ftp.netbsd.org/pub/NetBSD/NetBSD-3.1/i386/binary/sets/> for a list of the sets that form NetBSD/i386 3.1.

2.3 Master/slave setup

Given the cross-platform nature of NetBSD some tests depend on a specific platform or on some specific hardware. There has to be a way to deploy a test suite or test program to a remote machine (slave), run it and gather the results. This is easy to achieve if we have the `tests.tgz` set mentioned above because it can be cross-built using NetBSD's `build.sh` script and later installed on the remote machines.

Generalizing the above requirement, it should be possible to have a master machine coordinating multiple slaves and receiving their results, collapsing them all in a single log. The master may not even be a NetBSD box.

It is still unknown if the slaves will require some kind of testing daemon that waits for testing petitions and serves them. Using SSH alone is probably enough.

2.4 Test report

A test program can report any of the following states after execution:

Passed The test was executed and finished successfully.

Failed The test was executed and failed for some specific reason which must be reported.

Skipped The test was not executed because some preconditions were not met (missing tools, kernel drivers, etc.), all of which must be logged. This is not an actual failure but the user must be aware of it.

Expected failure The test was executed and detected an invalid situation. However this failure was expected and therefore the test returns success. This is useful to test for functionality *known to be broken* in the system and to automatically detect when it gets fixed.

The test system must generate a user-friendly report after execution that lists, for each test, its result (as described above). Due to the master/slave design, the report has to consolidate the results of multiple test machines so that the developer can take a quick overview of the current status.

To generate user-friendly tests, the system has to support the conversion of logs to several formats such as plain text or XML. XML allows the later beautification using XSLT stylesheets. These allow presenting the report as, e.g. an HTML document with pretty tables formatted with CSS. Whether the internal system status should also be stored in XML is still undecided but should not matter at all in the general design.

The release engineering team currently serves a consolidated report for the status of automated builds of the whole system³. The testing report could follow the same spirit.

2.5 Test execution

The current testing infrastructure requires all Makefiles to specify all available test programs so that the `regress` target can run them all. This restricts the execution of tests to those users with a source tree available and with the development tools (the `comp.tgz` set) installed.

It is probably a good idea to allow the execution of tests easily without having a source tree available. In that case the tests have to be redistributed as part of the system as in the `tests.tgz` set already mentioned. End users could then install and use it to verify if the system is stable and functional on a new installation. Even more, the installer might even offer to do so for new setups!

If this is done, `make(1)` should not be involved in the test execution itself to avoid a dependency on the development tools. Then there should be some other way to automatically run a set of tests. This can either be in the form of a simple shell script or tool that iterates over the test directories. `pkgsrc` currently has a `pkg_regress` program that does just that. This tool could also be the one that implements the master/slave functionality described below.

Assuming we have the tool mentioned in the previous paragraph and that it is called `regress`, consider the following fictitious examples:

³<http://releng.netbsd.org/cgi-bin/builds.cgi>

```
/usr/tests# regress -f "privileges = root"
recursively runs all tests that require root privileges
```

```
/usr/tests# regress -l foo.log
recursively runs all tests and leaves their results in foo.log
```

```
/usr/tests/sys# regress
recursively runs all tests (kernel only due to the
directory we are in)
```

```
/usr/tests/sys/arch/mac68k# regress -r my.mac68k.machine
recursively runs all mac68-specific kernel tests on a
remote machine
```

```
/usr/tests/bin/cp# regress t_iflag
runs cp(1)'s "t_iflag" test
```

2.6 Configuration

The test system shall be configurable. Some tests are so generic that require to be specialized by the end user to make sense on some situations. On the other hand, there are details the framework cannot guess so they must be provided by the user; for example, the address of the remote machines to run machine-specific tests or the emulator to use to execute critical tests.

Given that tests are organized hierarchically in a tree, a configuration file that followed this same structure could be enough. At each node you could specify property/value pairs that affect that specific node and all its children. And a node could be either a test suite or a specific test program. For example:

```
node sys {
  node fs {
    # Run all file system tests within QEMU and
    # give them a 100-second long delay until a
    # panic is assumed.
    emulator = qemu
    emulator.timeout = 100s

    node tmpfs {
      # Size of the tmpfs mounted by its test
      # suite; makes no sense on other file
      # systems.
      size = 100M
    }
  }
}
```

This should be implemented at the framework level which then passes to each test program its specific configuration. Of course, each test program has to be configurable on its own to meet the requirement of standalone programs but the interface can be different. E.g. the above could be expressed, for a given tmpfs test, as:

```
/usr/tests/sys/fs/tmpfs# ./t_mkdir -s emulator=qemu \
-s emulator.timeout=100s -s size=100M
```

3 Test programs

Test programs must be self-sufficient: they should be executable on their own and behave as if they were executed from within the regress target (again, no testing functionality should be bundled in the Makefiles themselves). Of course the necessary requirements must be available. For example: a test program written in shell scripting may require the use of some C code to access an specific system functionality; this could be provided as a helper tool if not available as part of any of the standard system utilities. (See tmpfs's `h_tools.c` used by the multiple `t_*` shell scripts.)

Given the variety of stuff to check, test programs can be written either in C or shell scripting. In some situations writing tests in shell is a lot easier than in C, but in other cases it is impossible to test a feature from the shell alone. Of course some more-specific tests could be written in other languages (e.g. awk), but their "front-end" (monitor) could still rely on C or sh.

The test framework has to provide libraries to encapsulate most of the common stuff done by test programs, and these have to be equally available from any of the supported languages. These libraries include:

- Logging of the test program result.
- Automatic set up and clean up of test directories.
- Functions/macros to test for expected results (similar to an assertion but with proper logging).
- Common code to check for required features to actually run the test (privileges, missing tools, etc.).

The shell "library" may be made to rely on the C one, but if it is kept simple enough that may not be needed.

An example shell-based test program could be:

```
# example.sh.test

test_header() {
    ... described below ...
}

test_skip() {
    ... described below ...
}

test_body() {
    mkdir a || test_die
    test -d a || test_die
}
```

And similarly for a C file. As you can see the above is not executable on its own. It requires to be "linked" with the test monitor to work, as described later.

There might even be a need for `test_init()` and `test_finish()` functions that encapsulate all the test configuration (e.g. mounting a file system and unmounting it later on). This could allow running the body multiple times without executing the expensive (de)configuration operations. Could be useful for the benchmarking idea exposed below.

3.1 Verbosity

A test program should have the ability to report its progress to the user at run time. This must be configurable so that the user can choose which level of information wants to see:

Silent The test program generates no output.

```
# ./t_mkdir -v 0
#
```

Results only The test program results (as a whole) are printed in a compact form.

```
# ./t_mkdir -v 1
t_mkdir: OK.
#
```

Full The test program prints a detailed output of its activity (e.g. a message before each test case that specifies what is going to happen). This will be useful when debugging test failures and when executing tests that take very long.

```
# ./t_mkdir -v 2
t_mkdir: Running tests
    Directories can be created...
    Link count is updated after directory creation...
    Create many directories...
    Nested directories can be created...
t_mkdir: All tests were successful
#
```

Regardless of the verbosity configuration, all tests must log their results to the appropriate logging file and/or report success through the program's exit code. Otherwise silent output could be useless.

3.2 The monitor

A test program can abruptly fail leaving garbage all around. Test programs should be run within a monitor that catches any unexpected condition and tries to do the necessary cleanup.

This monitor can be built inside the test program itself (catching signals, doing some sanity checks, etc.) so it does not break the goal of making test programs be standalone. In the case of a C test, the monitor implements the main function and later delegates its tasks to the test_body. In a shell-based test, the monitor needs either be bundled inside the resulting source file or has to be sourced but the key idea is the same.

It is important to note that the monitor shall also provide a common front-end (command line options, verbosity behavior, etc.) to all tests as well as all the “library” facilities mentioned above.

3.3 Declarative headers

Test programs have to be described in some way so that the user knows what is going on. These descriptions also allow the user to filter some tests based on specific properties. For example:

- Test program purpose: textual description.
- Privileges required: root, regular user, N/A.
- Useful for benchmarking: yes, no.
- Already fixed bugs (PRs) that this test checks for.

These headers are useful, among other things, to quickly determine which tests to run. For example, if a regular user is executing a test suite, the test monitor can quickly decide if the test has to be skipped or not or request root privileges on demand using `sudo(8)`. This also allows to quickly run all tests applicable under a given condition.

The header is bundled into the source file by defining it in a special function. For example:

```
# begin example.sh.test

test_header() {
    set comment "Ensures that cp copies files"
    set benchmarking "Yes"
    set privileges "root"
}

test_body() {
    ...
}
```

The monitor could then allow controlled access to the header properties or handle some "skip" reasons automatically to minimize code duplication among tests:

```
$ ./example query privileges
root
$ ./example -v 1
example: SKIPPED - You are not root
$
```

3.4 Requirements

Some tests can only be run under certain conditions. For example some of them may require root privileges, others may require the presence of a given kernel subsystem, etc.

It is possible to abstract some of these restrictions and define them in a declarative way as described in the test headers above. This shall be done to avoid, as much as possible, code duplication across tests and to keep all logic centralized. This includes, but is not limited to:

- Required privileges.
- Required platform.
- Presence of an specific binary.
- Presence of a specific file system in the kernel.

However, this is not scalable because, given the nature of tests, each of them might need to be skipped on a very specific condition. The test program needs to have a way to verify some conditions are met on its own and abort with an skipped result. So we could have:

```
# example.sh.test

test_skip() {
    if test -f /bin/cp; then
        true
    else
        log_skip "/bin/cp is not available"
        false
    fi
}
```

4 Other ideas

4.1 Kernel testing

Some kernel features cannot be tested once the system is up (e.g. Multiboot booting). There has to be a common framework (provided by the “libraries” or as a helper tool) to configure a virtual machine (QEMU, Xen, ...), populate it with appropriate contents, launch it and gather the results.

On a further thought, it could be useful to implement the creation of system images (to be later used within an emulator or virtualization system) in a standalone tool (let’s say `mkimage(8)`). It may come handy in more situations that testing alone. Writing such a tool is non-trivial because it needs to be portable; remember that the master system may not be running NetBSD!

This can also be useful to run critical tests that may crash the computer or tests that need to mess with the system’s configuration (e.g. enable a NFS server, export some specific directories and then run some commands). The former is difficult to run on a remote machine (what to do if it crashes?) but the latter is possible.

4.2 Benchmarking

Some tests are useful to benchmark the system. For example, the test that verifies that a given file system can be mounted correctly is not useful for a benchmark, but the test that does multiple `mkdirs` within it is.

The framework has to provide a way to properly mark such tests as useful for a benchmarking run. When such functionality is enabled, the framework can run such tests multiple times to gather somewhat stable statistics and later log them along the test results. (This is where the `test_init()` and `test_finish()` functions mentioned earlier may come handy because the benchmark could run the body multiple times bypassing the expensive setup operation.)

Benchmark tests should be easily comparable to a previous run and the system should spot which test have improved (or become worse) by a reasonable amount.

As far as I know, Monotone⁴ is using a similar approach to benchmark changes to the core program.

4.3 Naming scheme

It may be good to have some naming conventions on tests and helper tools to quickly differentiate them. Given that they will be executable programs, extensions should not be used. So we could have, e.g. a `t_` prefix or `_test` suffix for test programs and similarly for helpers (`h_` or `_helper`). `tmpfs` currently does this based on ideas taken from Monotone and Boost⁵’s test suites.

This is important if the tests end up living alongside the real program sources. Otherwise this is not required because the tests could be living in an independent tree separate from the sources.

5 Project extensions

In case the project got too short for the expected timeframe (4-5 months working full-time on it), here is a list of possible uninvestigated extensions that could be added:

- It would be nice to be able to compile the kernel’s file system code in user space as if it were a regular library or program⁶. This way, the testing framework could stress-test the file system code without worrying about system crashes nor data corruption: everything could be running inside user space.
- Add the ability to do test coverage scans for a particular system area. That is, allow the framework to report the user how much (and which) of the existing code has been run during a test; this is useful to demonstrate if the test suite covers all possible execution flows or not.

In order to do this, one’d need binaries built with profiling support and the framework would have to provide an automated way to analyze the execution results, generating an appropriate report at the end.

⁴<http://www.monotone.ca/>

⁵<http://www.boost.org/>

⁶<http://www.NetBSD.org/contrib/projects.html#usermode-ffs>.

- As already mentioned in the introduction, add new tests for several parts of the system. For example, it'd be a good idea to scan the list of open bug reports in <http://www.NetBSD.org/Gnats/> and add tests to reproduce the bugs described in them. I'd also work on fixing some of them.

This item alone is enough to cover any “free” time that may be left during the project development because there are a lot of open bugs and there are a lot of different system utilities/areas to be tested.

Furthermore, this could include an analysis of how to test some specific parts of the system because, oftentimes, they will require complex tests to be stressed. Just consider how one could test the system scheduler or the behavior of some concrete hardware devices; testing these may require changes to the code itself to be testing-friendly.