Kernel Testing with KUnit: Bridging the Gap

David Gow <davidgow@google.com>

Who am I?

- Hello!
- Using Linux full time since 2006
- Developer on KUnit for the last 3 years

What are we discussing?

- What, why, and how of testing?
- KUnit and kselftest: what they are and when to use each.
- What's changing and improved
    - KTAP, QEMU support, Documentation
- Where to from here?

What aren't we discussing?

- Step-by-step how to write tests.

Testing, kselftest, and KUnit

Why (and How) Should You Test the Kernel?

- Because you want it to work.
- Security and Reliability bugs in the kernel are *bad*.
- Regressions are bad.
- Tests were written as ad-hoc test scripts and modules.

- KUnit and kselftest are standardise these tests.
- They're both in-tree: the tests are included (and kept in-sync with) the kernel.
- Being run automatically on a number of CI systems.

What test framework should I use?

- kselftest: scripts that run from userspace
  - Any kernel data / code needs to be exposed somehow
  - Can easily set up state from userspace, run programs, etc
- KUnit: the test code is part of the kernel
  - Can access internal kernel functions/data
  - More structured, smaller tests
  - "A single C function"
  - Difficult to write integration tests, particularly those which touch userspace.
  - Must be written in C (or maybe Rust)

Other Testing Tools

- Dynamic Analysis tools:
  - Sanitizers: KASAN, KCSAN, UBSAN, KFENCE, etc.
  - Leak checkers: kmemleak
  - Validators: lockdep
  - Don't run "tests", but identify unsafe behaviour
  - Can be run alongside KUnit/kselftest — integrations exist.
- Code coverage
  - gcov
  - kcov
- See the 'kernel testing guide' for more info:
  - https://www.kernel.org/doc/html/latest/dev-tools/testing-overview.html

The Challenges Faced in 2021

Integration:

- kselftest and KUnit serve similar purposes, but there are reasons to use one over the other in some circumstances.
- The same people need to use both.
- Porting tests from one to the other.
- The same systems (CI, tooling) want to aggregate results from both.

Feature Gaps:

- KUnit comes with a bunch of built-in tooling, but it was very KUnit-specific
- It only really worked under UML (User-Mode Linux).

The KTAP format

A Test Result Format

- A structured, machine-readable format for test results
  - Tools can pretty-print and summarise output
  - CI systems can collate and correlate output from different runs.
  - Still human-readable.
- TAP: the Test Anything Protocol
  - https://testanything.org/tap-version-13-specification.html
  - Simple
  - A bit too simple: no nested tests, etc
  - Every test extended it in a slightly different, incompatible way.
- We need an updated format.
  - TAP14: Draft update to the spec.
  - Abandoned.
  - Some licensing weirdness.
- New one: KTAP — Kernel TAP
  - A standardisation of what kselftest and KUnit are doing
  - Still parsable by most existing tooling
  - Some unnecessary stuff removed (embedded yaml)

Results (KTAP format)

```
KTAP version 1
1..1
    KTAP version 1
    1..36
    ok 1 - list_test_list_init
    ok 2 - list_test_list_add
    ok 3 - list_test_list_add_tail
    ok 4 - list_test_list_del
    ok 5 - list_test_list_replace
    ok 6 - list_test_list_replace_init
    ok 7 - list_test_list_swap
    […]
    ok 35 - list_test_list_for_each_entry
    ok 36 - list_test_list_for_each_entry_reverse
ok 1 - list-kunit-test
```

Results (KTAP format)

```
KTAP version 1
1..1
    KTAP version 1
    1..4
    # example_simple_test: initializing
    ok 1 - example_simple_test
    # example_skip_test: initializing
    ok 2 - example_skip_test # SKIP this test should be skipped
    ok 3 - example_mark_skipped_test # SKIP this test should be skipped
    # example_all_expect_macros_test: initializing
    # Oh, no! An error!
    not ok 4 - example_all_expect_macros_test
# example: pass:1 fail:1 skip:2 total:4
# Totals: pass:1 fail:1 skip:2 total:4
not ok 1 - example
```

Parsing KTAP with KUnit

- KUnit includes a parser for KTAP output
- ./tools/testing/kunit/kunit.py parse
  - Accepts either a filename or stdin
- Prints a nice summary:

```
[16:55:12] ============================================================
[16:55:12] =================== example (4 subtests) ===================
[16:55:12] [PASSED] example_simple_test
[16:55:12] [SKIPPED] example_skip_test
[16:55:12] [SKIPPED] example_mark_skipped_test
[16:55:12] # example_all_expect_macros_test: initializing
[16:55:12] # Oh no! An error!
[16:55:12] [FAILED] example_all_expect_macros_test
[16:55:12] ===================== [FAILED] example =====================
[16:55:12] ============================================================
[16:55:12] Testing complete. Passed: 1, Failed: 1, Crashed: 0, Skipped: 2, Errors: 0
```

QEMU

Architectures and Tooling

- KUnit works on all architectures supported by the kernel.
- Some of the KUnit tooling was UML-specific
- kunit_tool now has better support for other architectures
  - Can now cross-compile
  - KUnit comes with configs and QEMU scripts to run across many architectures
- Just add the --arch=[arch] option
- Also a --cross_compile option to pick a compiler manually

Architecture support

- In addition to UML, we support the following out of the box:
  - i386
  - x86_64
  - arm
  - arm64
  - alpha
  - powerpc
  - riscv
  - s390
  - sparc
- Don’t see your architecture? No problem:
  - Extra architectures can be defined in a python file.

Other New Features

Since Last Year's LCA

Visit: https://kunit.dev/release_notes.html

- Tests can now be SKIPped.
    - Just use kunit_skip() or kunit_mark_skipped()
- Test statistics:
    - Even if you're not using kunit_tool, counts of passed, failed, skipped, tests.
- UBSAN integration
- Drastically improved documentation
- A huge number of bug and usability fixes.

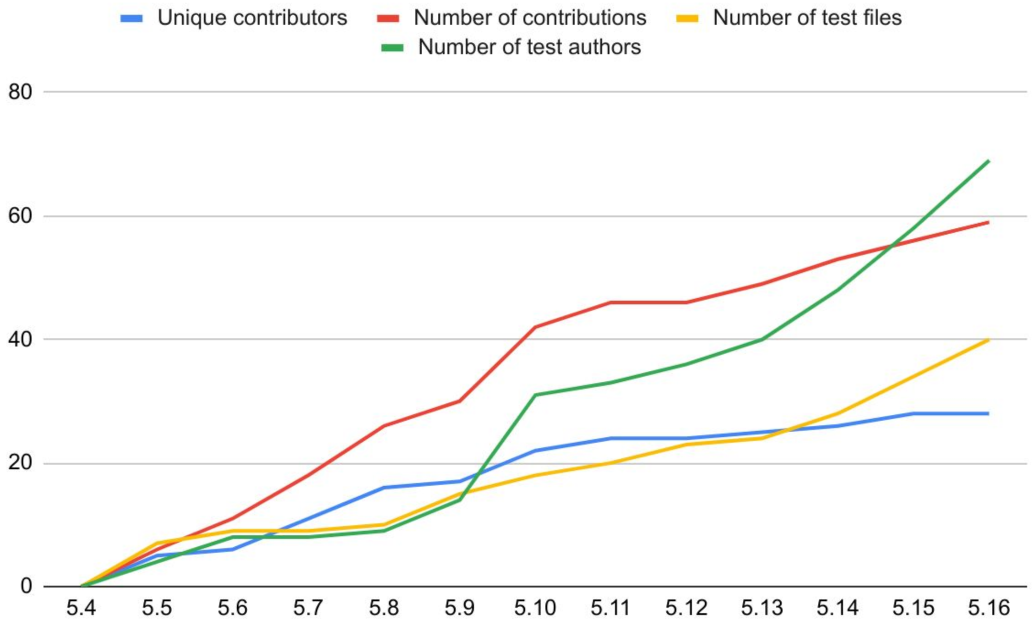
Since Last Year's LCA

Visit: https://kunit.dev/release_notes.html

- .kunitconfig fragments:
  - Each subsystem can now include a default .config for tests
- --kconfig_add:
  - Add an extra kconfig option to the current kernel
- Test filtering
  - Run only tests which match a glob
- Hermetic testing
  - --run_isolated option allows suites/tests to be run on separate kernel invocations

New Tests!

- In 5.11, we had 20 test suites (204 individual tests)
- In 5.16, we have 40 test suites (324 individual tests)
  - Despite the introduction of parameterised tests merging a number of existing tests
- New tests include:
  - timestamp conversions
  - KFENCE
  - ALSA SoC topology
  - ASPEED SDHCI phase tests
  - Thunderbolt / USB4
  - mptcp
  - s390 stack unwinding
  - command-line options parsing
  - DAMON (Data Access MONitor)
  - SLUB memory allocator
  - memset/memcpy/memmove
  - kprobes
  - Maths functions
  - Hashing!
- …and more!



The Future

What's coming soon?

- More KTAP standardisation fixes.
- Improved support for running KUnit tests as modules.
- More tests and test examples, particularly testing hardware.
- Reduced memory usage (even further!)
- Yet more bugfixes and documentation.

What do you want?

- Have you used KUnit or kselftest?
- Is anything blocking you from doing so?
- What tests should you run for a subsystem? How would you know?
- Would you want to get test results / know how a patch has been tested?
- How much refactoring of code to make it testable is too much?

Questions / Comments?