KUnit:
Past, Present, and Future

David Gow <davidgow@google.com>

What is KUnit?

What is KUnit?

- A unit testing framework for the Linux Kernel
  - Aimed at 'in-kernel' testing
  - Tooling to make writing and running tests easier.
- Designed for small, fast, self-contained tests
  - Think "a single kernel function"
- An effort to standardise such tests
  - Tests produce a common, machine parsable (K)TAP output format
- Can be run under any kernel architecture
  - Either built-in to the kernel to run at startup, or as a module
  - Can be run (with provided tooling) under User-Mode Linux (UML), as a normal x86-64 Linux binary
    - Allows for extremely fast testing!

Where is KUnit?

- KUnit is included in Linux 5.5+
  - KUnit and KUnit tests can be enabled with Kconfig entries.
  - e.g. CONFIG_KUNIT and CONFIG_KUNIT_ALL_TESTS
  - A python wrapper which configures, builds, runs, and parses results included
    - ./tools/testing/kunit/kunit.py run
  - The in-kernel test framework lives:
    - lib/kunit/
  - Tests sit alongside the code being tested
    - (typically in a file ending _kunit.c, building a <thing>_kunit.ko module)
- KUnit documentation:
  - Documentation/dev-tools/kunit/index.rst
  - KUnit's website: https://kunit.dev/

How are tests structured?

- A test is a single function:
  - Runs some setup
  - Runs the code under test
  - Asserts the resultant state matches expectations
- Test suites:
  - A collection of related tests.
  - Can have shared initialisation / teardown code.

Example: list_test

lib/list-test.c:

```
static void list_test_list_move_tail(struct kunit *test)
{
	struct list_head a, b;
	LIST_HEAD(list1);
	LIST_HEAD(list2);

	list_add_tail(&a, &list1);
	list_add_tail(&b, &list2);

	/* before: [list1] -> a, [list2] -> b */
	list_move_tail(&a, &list2);
	/* after: [list1] empty, [list2] -> b -> a */

	KUNIT_EXPECT_TRUE(test, list_empty(&list1));

	KUNIT_EXPECT_PTR_EQ(test, &b, list2.next);

	KUNIT_EXPECT_PTR_EQ(test, &a, b.next);
}
```

Example: suites

```
static struct kunit_case list_test_cases[] = {
        KUNIT_CASE(list_test_list_init),
        KUNIT_CASE(list_test_list_add),
        KUNIT_CASE(list_test_list_add_tail),
[…]
        KUNIT_CASE(list_test_list_for_each_prev_safe),
        KUNIT_CASE(list_test_list_for_each_entry),
        KUNIT_CASE(list_test_list_for_each_entry_reverse),
        {},
};

static struct kunit_suite list_test_module = {
        .name = "list-kunit-test",
        .test_cases = list_test_cases,
};

kunit_test_suites(&list_test_module);

MODULE_LICENSE("GPL v2");
```

Running tests (with kunit_tool)

- Create a '.kunitconfig' file in the build directory ([srcdir]/.kunit by default)
  - Include the config options you need for testing:
    - CONFIG_KUNIT=y
    - CONFIG_LIST_KUNIT_TEST=y
- Run './tools/testing/kunit/kunit.py run'
  - If you want the raw kernel output in TAP format, rather than the parsed summary, use the '--raw_output' option

Results (TAP format)

```
TAP version 14
1..1
    # Subtest: list-kunit-test
    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
```

```
TAP version 14
1..1
   # Subtest: list-kunit-test
   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
   # list_test_list_del_init: EXPECTATION FAILED at
lib/list-test.c:161
   Expected list_empty_careful(&a) to be true, but is false
   not ok 8 - list_test_list_del_init
   ok 9 - list_test_list_move
   ok 10 - list_test_list_move_tail
   ok 36 - list_test_list_for_each_entry_reverse
not ok 1 - list-kunit-test
```

Results (kunit_tool)

```
[22:49:46] Configuring KUnit Kernel …
[22:49:46] Building KUnit Kernel …
[22:49:52] Starting KUnit Kernel …
[22:49:57]
============================================================
[22:49:57] ======== [PASSED] list-kunit-test ========
[22:49:57] [PASSED] list_test_list_init
[22:49:57] [PASSED] list_test_list_add
[22:49:57] [PASSED] list_test_list_add_tail
[22:49:57] [PASSED] list_test_list_del
[22:49:57] [PASSED] list_test_list_replace
[22:49:57] [PASSED] list_test_list_replace_init
[22:49:57] [PASSED] list_test_list_swap
[22:49:57] [PASSED] list_test_list_del_init
[22:49:57] [PASSED] list_test_list_move
[…]
[22:49:57] [PASSED] list_test_list_for_each
[22:49:57] [PASSED] list_test_list_for_each_prev
[22:49:57] [PASSED] list_test_list_for_each_safe
[22:49:57] [PASSED] list_test_list_for_each_prev_safe
[22:49:57] [PASSED] list_test_list_for_each_entry
[22:49:57] [PASSED] list_test_list_for_each_entry_reverse
[22:49:57]
============================================================
[22:49:57] Testing complete. 36 tests run. 0 failed. 0
crashed.
[22:49:57] Elapsed time: 10.216s total, 0.001s
configuring, 6.069s building, 0.000s running
```

```
[22:41:59] Configuring KUnit Kernel …
[22:41:59] Building KUnit Kernel …
[22:42:03] Starting KUnit Kernel …
[22:42:07]
============================================================
[22:42:07] ======== [FAILED] list-kunit-test ========
[22:42:07] [PASSED] list_test_list_init
[22:42:07] [PASSED] list_test_list_add
[22:42:07] [PASSED] list_test_list_add_tail
[22:42:07] [PASSED] list_test_list_del
[22:42:07] [PASSED] list_test_list_replace
[22:42:07] [PASSED] list_test_list_replace_init
[22:42:07] [PASSED] list_test_list_swap
[22:42:07] [FAILED] list_test_list_del_init
[22:42:07]     # list_test_list_del_init: EXPECTATION FAILED at
lib/list-test.c:161
[22:42:07]     Expected list_empty_careful(&a) to be true, but is
false
[22:42:07]     not ok 8 - list_test_list_del_init
[…]
[22:42:07] [PASSED] list_test_list_move
[22:42:07] [PASSED] list_test_list_for_each
[22:42:07] [PASSED] list_test_list_for_each_prev
[22:42:07] [PASSED] list_test_list_for_each_safe
[22:42:07] [PASSED] list_test_list_for_each_prev_safe
[22:42:07] [PASSED] list_test_list_for_each_entry
[22:42:07] [PASSED] list_test_list_for_each_entry_reverse
[22:42:07]
============================================================
[22:42:07] Testing complete. 36 tests run. 1 failed. 0 crashed.
[22:42:07] Elapsed time: 7.732s total, 0.001s configuring, 3.550s
building, 0.000s running
```

Other neat tricks:

- KUnit can manage memory and resources
  - Cleaned up on test exit (failure or success)
  - Use, e.g., kunit_kzalloc()
- Parameterised and data driven tests
  - We'll look at a bit more later
- Other useful assertion / expectation variants:
  - KUNIT_EXPECT_STREQ(test, a, b): compares strings
  - KUNIT_EXPECT_*_MSG(test, a, b, fmt, …): provide a specific error message
- KUnit logging tools:
  - kunit_log() macro will output log both to dmesg and to the test log in debugfs

What's changed?

KUnit since 5.5

KUnit first accepted upstream in Linux 5.5

Since then:

- Module support & debugfs test output
- Named resources
- Improved TAP output / executor
- KASAN integration
- Parameterised test support
- Continuous Integration support
- Many misc. fixes
- Lots of tests.

Module Support

- KUnit tests can now be built into modules, and will run at module load time.
- Useful for integrating with existing test systems.
- Non-UML architectures.
- Tests which need to access user memory.

Named resources

- It's now possible to associate a named resource with a test, and have it automatically cleaned up when the test completes (whether it succeeds or fails)
- Also useful for storing test-specific metadata
  - Used by the KASAN integration to expect specific KASAN failures

Improved TAP output / executor

- The first version of KUnit ran tests as initcalls()
  - No centralised knowledge of what KUnit tests were built-in
  - TAP output couldn't count number of tests
- KUnit tests now run via an 'executor' which calls tests explicitly as part of the init process
  - TAP output now includes the test summary line

KASAN integration

- If KASAN is enabled:
  - Memory errors will cause tests to fail (if kasan_multishot enabled)
- KASAN's own tests largely ported to KUnit
  - KUnit supports 'expecting' an invalid memory access
  - Unlike previous tests, where output had to be compared manually to a 'known good' to get any results, most tests can now report their own success/failure.
  - Some tests yet to be ported:
    - Access to user memory
    - Stack traces under RCU, etc.

Parameterised Testing

- Run the same test code repeatedly with different inputs
  - KUnit will collate the results.
- A 'generator' function is used to allow number and value of inputs to be determined at runtime.
- Useful for 'data driven testing', allowing test data to be read from a table (e.g., standardised test vectors), or generated from code.

```
static void timestamp_expectation_to_desc(const struct timestamp_expectation *t,
                                          char *desc)
{
        strscpy(desc, t->test_case_name, KUNIT_PARAM_DESC_SIZE);
}

KUNIT_ARRAY_PARAM(ext4_inode, test_data, timestamp_expectation_to_desc)
```

Continuous Integration

- Goal: ensure KUnit tests are not being broken by new changes upstream
- Support for running "all tests"
  - The KUNIT_ALL_TESTS config option enables all tests with satisfied dependencies.
    - Useful if you have an existing config, and want to test it.
  - New kunit.py run --alltests option
    - Uses make allyesconfig under UML to run as many tests as possible
    - UML tends to break a bit: there's a list of broken configs which are disabled.
- KernelCI support
  - Working, but not yet fully enabled. Runs kunit.py --alltests.
- Linaro LKFT
  - Running KUnit tests on ARM and x86-64
  - (Including KASAN tests, which don't work under UML yet!)

Tooling updates

- kunit_tool now supports running subtasks individually
  - e.g. Building a kernel with …/kunit.py build
  - The parser can be run on arbitrary input with …/kunit.py parse
- JSON output for test parsing
- Test results can be output in the JSON format used by KernelCI
- kunit_tool should no-longer pollute the source directory
  - kunit_tool defaults to using .kunit as a build directory
  - kunitconfig files are now .kunitconfig in the build directory (Thanks Andy Shevchenko)
- New naming guidelines for tests, suites, modules, etc:
  - See Documentation/dev-tools/kunit/style.rst

New tests

- Power Management / QOS
- Multipath TCP: Crypto and Token
- KASAN
- KCSAN
- Bitfields
- Command-line parsing
- Thunderbolt / USB4
- IO Ports / Resources
- And more…

The Future

Mocking and Hardware Testing

- Testing drivers is hard: need to intercept reads/writes to hardware
- Ways to approach it:
  - Refactor code to allow a "fake" interface to be passed in
  - Forcibly intercept functions ("function mocking")
  - Provide ways of intercepting access to platform IOMEM and similar
- KUnit has experimented with providing features to support this:
  - "Class Mocking" — macros to generate ops structs, classes, OOP constructs
    - An RFC of this is available
  - "Function Mocking" — somewhat problematic interception of functions with weak linking and/or ftrace
  - "Platform Mocking" — implementing stub interfaces under UML, adding hooks, etc, to allow fake devices

Skippable test support

- The (K)TAP specification allows tests to be programmatically skipped.
- Plan is to allow individual testcases (or entire suites) to be skipped if prerequisites aren't met
  - Doesn't count either as failure or success.
  - A skipped test will not fail the entire suite.
- Prototype exists: hopefully this'll be added soon!

Bugfixes and tooling improvements

- Standardisation of output between KUnit and kselftest
  - Tim Bird's proposed KTAP output format
  - Reworking kunit_tool's parser to better support non-kunit TAP output
    - e.g. nested subtests, flexibility in where directives are placed
- Improved tooling / processes for testing individual subsystems
  - Support having separate kunitconfig files for individual subsystems.
  - kunit_tool can then accept the path to a subsystem's config, and run these
  - Work out a way for subsystem maintainers to request contributors run a specific set of tests before sending patches.
    - e.g. having a test script, a list of instructions in MAINTAINERS, etc.
- Tooling support for running tests against the current kernel
  - By loading modules and using debugfs to read results.

Questions / Comments?