KUnit - Unit Testing for the Linux Kernel

Brendan Higgins <brendanhiggins@google.com>
Who am I?

- Brendan Higgins <brendanhiggins@google.com>
- I work at Google
- Previously I worked on
  - server bringup at Google
  - OpenBMC
Please interrupt me!

- This talk is for you!
- I hate slide decks, so mine aren’t usually very good.
Why Unit Testing, Why KUnit?
Why Unit Testing, Why KUnit?

- I will let it speak for itself…
- (Demo time)
==sysctl-test==

- [FAILED] sysctl_test_null_tbl_data
- [PASSED] sysctl_test_api_dointvec_table_maxlen_unset
- [PASSED] sysctl_test_api_dointvec_table_len_is_zero
- [PASSED] sysctl_test_api_dointvec_table_read_not_positive
- [PASSED] sysctl_test_api_dointvec_table_read_happy_single_pos
- [PASSED] sysctl_test_api_dointvec_table_read_happy_single_neg
- [PASSED] sysctl_test_api_dointvec_table_write_happy_single_pos
- [PASSED] sysctl_test_api_dointvec_table_write_happy_single_neg
- [FAILED] sysctl_test_api_dointvec_write_single_less_int
  # sysctl_test_api_dointvec_write_single_less_int
  Expected -22 == proc_dointvec(&table, 1, user_buffer, -22 == -22)
  proc_dointvec(&table, 1, user_buffer, 
  not ok 9 - sysctl_test_api_dointvec_write_single
- [FAILED] sysctl_test_api_dointvec_write_single_greater_int
  # sysctl_test_api_dointvec_write_single_greater_int
  Expected -22 == proc_dointvec(&table, 1, user_buffer, -22 == -22)
  proc_dointvec(&table, 1, user_buffer, 
  not ok 10 - sysctl_test_api_dointvec_write_single

- [PASSED] kunit-resource-test
- [PASSED] kunit_resource_test_init
- [PASSED] kunit_resource_test_alloc
- [PASSED] kunit_resource_test_destroy
- [PASSED] kunit_resource_test_cleanup
- [PASSED] kunit_resource_test_proper_free_ordering
- [PASSED] kunit-try-catch-test
- [PASSED] kunit_test_try_catch
  kunit_test_try_catch

kunit_test Guy-01
How is this different?

- Well, it’s fast!
- It doesn’t depend on userland
- It has no external dependencies
- Writing tests is no different from writing normal kernel code
How is this different?

- Unit testing!
What’s unit testing?

- “Unit testing” is not synonymous with “testing”
- Testing is typically divided into 3 domains:
  - unit testing
  - integration testing
  - end-to-end/system testing
Types of Testing

- An end-to-end test:
  - usually tests the entire system from the perspective of the code under test
  - For example, someone might write an end-to-end test for the kernel by installing a production configuration of the kernel on production hardware with a production userspace and then trying to exercise some behavior that depends on interactions between the hardware, the kernel, and userspace
  - As close to production environment as possible
Types of Testing

- A unit test:
  - is supposed to test a single unit of code in isolation, hence the name
  - should be the finest granularity of testing and as such should allow all possible code paths to be tested in the code under test
  - this is only possible if the code under test is very small and does not have any external dependencies outside of the test’s control like hardware
  - should be very fast
  - Should be able to be run without any special set up
  - Executes in less than 100 ms
  - Multiple threads are discouraged
Types of Testing

- An integration test:
  - tests the interaction between a minimal set of components, usually just two or three
  - For example, someone might write an integration test to test the interaction between a driver and a piece of hardware
  - Or to test the interaction between the userspace libraries the kernel provides and the kernel itself
  - Nevertheless, one of these tests would probably not test the entire kernel along with hardware interactions and interactions with the userspace
  - Minimal external dependencies
  - Runs on the order of seconds/minutes
## Types of Testing

<table>
<thead>
<tr>
<th></th>
<th>Unit Test</th>
<th>Integration Test</th>
<th>End-to-End Test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td>Small</td>
<td>Medium</td>
<td>Large</td>
</tr>
<tr>
<td><strong>Coverage</strong></td>
<td>High</td>
<td>Medium</td>
<td>Small</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>Fast</td>
<td>Medium</td>
<td>Slow</td>
</tr>
<tr>
<td><strong>Confidence in Parts</strong></td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Confidence in System</strong></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>
KUnit is not a new idea

- KUnit is a typical example of an x-unit style testing framework
- Many other x-unit testing frameworks/libraries have been written before:
  - JUnit (for Java)
  - Python’s unittest module (for Python)
  - Googletest/Googlemock (for C++)
What’s x-unit?

- Usually has a concept of a test suite broken up into test cases
- Each test case tests one aspect of functionality
- Together, the test suite tests a logical unit: a class, or a group of related functions
- The test cases often share a test fixture:
  - setup
  - teardown
  - context
KUnit Example

```c
static void list_del_init_test(struct test *test)
{
    struct list_head a, b;
    LIST_HEAD(list);

    list_add_tail(&a, &list);
    list_add_tail(&b, &list);

    /* before: [list] -> a -> b */
    list_del_init(&a);

    /* after: [list] -> b, a initialised */
    KUNIT_EXPECT_EQ(test, list.next, &b);
    KUNIT_EXPECT_EQ(test, b.prev, &list);
    KUNIT_EXPECT_TRUE(test, list_empty_careful(&a));
}
```
More on x-unit

- [https://google.github.io/kunit-docs/third_party/kernel/docs/usage.html](https://google.github.io/kunit-docs/third_party/kernel/docs/usage.html)
- [https://martinfowler.com/bliki/Xunit.html](https://martinfowler.com/bliki/Xunit.html)
Where are we going?
Where is KUnit today

- Initial patchset is in linux-next
- We have:
  - Basic test suite and test case definition
  - Expectations and assertions
  - Test resource management API
  - Tests!
  - Command line script for parsing test results (and other stuff)
What are KUnit’s current challenges?

- The Linux kernel wasn’t written to be unit tested
- The code is pretty good for the most part
  - The Linux kernel is reasonably well structured: encapsulation, code sharing, modularity, etc
  - Actually pretty object oriented
- Dependencies are poorly defined
  - Kbuild cares about feature dependencies, not code dependencies
  - Some core dependencies are just always implicitly included
Where does KUnit fit into the kernel’s test paradigm?

- There are a lot of test frameworks/suites for the Linux kernel
- Most are end-to-end tests
- There are currently not many unit tests, all are ad hoc
Where does KUnit fit into the kernel’s test paradigm?

- A well tested code base has:
  - Lots of unit tests (~80%)
  - A moderate number of integration tests (~15%)
  - And some end-to-end tests (~5%)
Where does KUnit fit into the kernel’s test paradigm?

- Lot’s of unit tests (~80%)
  - This is where KUnit lives
- A moderate number of integration tests (~15%)
  - ???
- And some end-to-end tests (~5%)
  - We got this covered (kselftest, xfstest, etc)
What we are planning on doing in the future?

- Improving Documentation
- Mocking/Faking
  - C-style class mocking
  - Function mocking
  - Faking hardware
- Improving test result parsing
  - Make test parser stand alone
  - Improve output
- CI/CD
  - We have an open source CI/CD system - make it work with LKML
  - Provide coverage results for every patch
What are we planning on focussing on near term?

- Getting usage
- Improving documentation
- Improving usage
- Adding tests
- (Maybe) Work on matchers API
- (Maybe) Work on mocking
Let’s talk!

● What types of things should we be focusing on?
  ○ Near term?
  ○ Long term?

● What should we do about the dependency problem?
  ○ Should Kbuild know about code dependencies?

● What should be done about integration testing?
  ○ KUnit can currently address kernel - hardware boundary.
  ○ KUnit can be made to address user space - kernel boundary.
  ○ Is this what we want?
Connect with us!

- kunit-dev@googlegroups.com
- linux-kselgtest@vger.kernel.org
- #kunit on oftc.net