Writing Tests

Test Cases

The fundamental unit in KUnit is the test case. A test case is a function with the signature `void (*)(struct kunit *test)`. It calls the function under test and then sets *expectations* for what should happen. For example:

```
void example_test_success(struct kunit *test)
{
}

void example_test_failure(struct kunit *test)
{
        KUNIT_FAIL(test, "This test never passes.");
}
```

In the above example, `example_test_success` always passes because it does nothing; no expectations are set, and therefore all expectations pass. On the other hand `example_test_failure` always fails because it calls `KUNIT_FAIL`, which is a special expectation that logs a message and causes the test case to fail.

Expectations

An *expectation* specifies that we expect a piece of code to do something in a test. An expectation is called like a function. A test is made by setting expectations about the behavior of a piece of code under test. When one or more expectations fail, the test case fails and information about the failure is logged. For example:

```
void add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, add(1, 1));
}
```

In the above example, `add_test_basic` makes a number of assertions about the behavior of a function called `add`. The first parameter is always of type `struct kunit *`, which contains information about the current test context. The second parameter, in this case, is what the value is expected to be. The last value is what the value actually is. If `add` passes all of these expectations, the test case, `add_test_basic` will pass; if any one of these expectations fails, the test case will fail.

A test case *fails* when any expectation is violated; however, the test will continue to run, and try other expectations until the test case ends or is otherwise terminated. This is as opposed to *assertions* which are discussed later.

To learn about more KUnit expectations, see Documentation/dev-tools/kunit/api/test.rst.

> **Note**
>
> A single test case should be short, easy to understand, and focused on a single behavior.

For example, if we want to rigorously test the `add` function above, create additional tests cases which would test each property that an `add` function should have as shown below:

```
void add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, add(1, 1));
}

void add_test_negative(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 0, add(-1, 1));
}

void add_test_max(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, INT_MAX, add(0, INT_MAX));
        KUNIT_EXPECT_EQ(test, -1, add(INT_MAX, INT_MIN));
}

void add_test_overflow(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, INT_MIN, add(INT_MAX, 1));
}
```

Assertions

An assertion is like an expectation, except that the assertion immediately terminates the test case if the condition is not satisfied. For

example:

```
static void test_sort(struct kunit *test)
{
        int *a, i, r = 1;
        a = kunit_kmalloc_array(test, TEST_LEN, sizeof(*a), GFP_KERNEL);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, a);
        for (i = 0; i < TEST_LEN; i++) {
                r = (r * 725861) % 6599;
                a[i] = r;
        }
        sort(a, TEST_LEN, sizeof(*a), cmpint, NULL);
        for (i = 0; i < TEST_LEN-1; i++)
                KUNIT_EXPECT_LE(test, a[i], a[i + 1]);
}
```

In this example, the method under test should return pointer to a value. If the pointer returns null or an errno, we want to stop the test since the following expectation could crash the test case. *ASSERT_NOT_ERR_OR_NULL(...)* allows us to bail out of the test case if the appropriate conditions are not satisfied to complete the test.

Test Suites

We need many test cases covering all the unit's behaviors. It is common to have many similar tests. In order to reduce duplication in these closely related tests, most unit testing frameworks (including KUnit) provide the concept of a *test suite*. A test suite is a collection of test cases for a unit of code with a setup function that gets invoked before every test case and then a tear down function that gets invoked after every test case completes. For example:

```
static struct kunit_case example_test_cases[] = {
        KUNIT_CASE(example_test_foo),
        KUNIT_CASE(example_test_bar),
        KUNIT_CASE(example_test_baz),
        {}
};

static struct kunit_suite example_test_suite = {
        .name = "example",
        .init = example_test_init,
        .exit = example_test_exit,
        .test_cases = example_test_cases,
};
kunit_test_suite(example_test_suite);
```

In the above example, the test suite `example_test_suite` would run the test cases `example_test_foo`, `example_test_bar`, and `example_test_baz`. Each would have `example_test_init` called immediately before it and `example_test_exit` called immediately after it. `kunit_test_suite(example_test_suite)` registers the test suite with the KUnit test framework.

> **Note**
>
> A test case will only run if it is associated with a test suite.

`kunit_test_suite(...)` is a macro which tells the linker to put the specified test suite in a special linker section so that it can be run by KUnit either after `late_init`, or when the test module is loaded (if the test was built as a module).

For more information, see Documentation/dev-tools/kunit/api/test.rst.

Writing Tests For Other Architectures

It is better to write tests that run on UML to tests that only run under a particular architecture. It is better to write tests that run under QEMU or another easy to obtain (and monetarily free) software environment to a specific piece of hardware.

Nevertheless, there are still valid reasons to write a test that is architecture or hardware specific. For example, we might want to test code that really belongs in `arch/some-arch/*`. Even so, try to write the test so that it does not depend on physical hardware. Some of our test cases may not need hardware, only few tests actually require the hardware to test it. When hardware is not available, instead of disabling tests, we can skip them.

Now that we have narrowed down exactly what bits are hardware specific, the actual procedure for writing and running the tests is same as writing normal KUnit tests.

> **Important**
>
> We may have to reset hardware state. If this is not possible, we may only be able to run one test case per invocation.

Common Patterns

Isolating Behavior

Unit testing limits the amount of code under test to a single unit. It controls what code gets run when the unit under test calls a function. Where a function is exposed as part of an API such that the definition of that function can be changed without affecting the rest of the code base. In the kernel, this comes from two constructs: classes, which are structs that contain function pointers provided by the implementer, and architecture-specific functions, which have definitions selected at compile time.

Classes

Classes are not a construct that is built into the C programming language; however, it is an easily derived concept. Accordingly, in most cases, every project that does not use a standardized object oriented library (like GNOME's GObject) has their own slightly different way of doing object oriented programming; the Linux kernel is no exception.

The central concept in kernel object oriented programming is the class. In the kernel, a *class* is a struct that contains function pointers. This creates a contract between *implementers* and *users* since it forces them to use the same function signature without having to call the function directly. To be a class, the function pointers must specify that a pointer to the class, known as a *class handle*, be one of the parameters. Thus the member functions (also known as *methods*) have access to member variables (also known as *fields*) allowing the same implementation to have multiple *instances*.

A class can be *overridden* by *child classes* by embedding the *parent class* in the child class. Then when the child class *method* is called, the child implementation knows that the pointer passed to it is of a parent contained within the child. Thus, the child can compute the pointer to itself because the pointer to the parent is always a fixed offset from the pointer to the child. This offset is the offset of the parent contained in the child struct. For example:

```
struct shape {
        int (*area)(struct shape *this);
};

struct rectangle {
        struct shape parent;
        int length;
        int width;
};

int rectangle_area(struct shape *this)
{
        struct rectangle *self = container_of(this, struct rectangle, parent);

        return self->length * self->width;
};

void rectangle_new(struct rectangle *self, int length, int width)
{
        self->parent.area = rectangle_area;
        self->length = length;
        self->width = width;
}
```

In this example, computing the pointer to the child from the pointer to the parent is done by `container_of`.

Faking Classes

In order to unit test a piece of code that calls a method in a class, the behavior of the method must be controllable, otherwise the test ceases to be a unit test and becomes an integration test.

A fake class implements a piece of code that is different than what runs in a production instance, but behaves identical from the standpoint of the callers. This is done to replace a dependency that is hard to deal with, or is slow. For example, implementing a fake EEPROM that stores the "contents" in an internal buffer. Assume we have a class that represents an EEPROM:

```
struct eeprom {
        ssize_t (*read)(struct eeprom *this, size_t offset, char *buffer, size_t count);
        ssize_t (*write)(struct eeprom *this, size_t offset, const char *buffer, size_t count);
};
```

And we want to test code that buffers writes to the EEPROM:

```
struct eeprom_buffer {
        ssize_t (*write)(struct eeprom_buffer *this, const char *buffer, size_t count);
        int flush(struct eeprom_buffer *this);
        size_t flush_count; /* Flushes when buffer exceeds flush_count. */
};

struct eeprom_buffer *new_eeprom_buffer(struct eeprom *eeprom);
void destroy_eeprom_buffer(struct eeprom *eeprom);
```

We can test this code by *faking out* the underlying EEPROM:

```
struct fake_eeprom {
        struct eeprom parent;
        char contents[FAKE_EEPROM_CONTENTS_SIZE];
};

ssize_t fake_eeprom_read(struct eeprom *parent, size_t offset, char *buffer, size_t count)
{
        struct fake_eeprom *this = container_of(parent, struct fake_eeprom, parent);

        count = min(count, FAKE_EEPROM_CONTENTS_SIZE - offset);
        memcpy(buffer, this->contents + offset, count);

        return count;
}

ssize_t fake_eeprom_write(struct eeprom *parent, size_t offset, const char *buffer, size_t count)
{
        struct fake_eeprom *this = container_of(parent, struct fake_eeprom, parent);

        count = min(count, FAKE_EEPROM_CONTENTS_SIZE - offset);
        memcpy(this->contents + offset, buffer, count);

        return count;
}

void fake_eeprom_init(struct fake_eeprom *this)
{
        this->parent.read = fake_eeprom_read;
        this->parent.write = fake_eeprom_write;
        memset(this->contents, 0, FAKE_EEPROM_CONTENTS_SIZE);
}
```

We can now use it to test `struct eeprom_buffer`:

```
struct eeprom_buffer_test {
        struct fake_eeprom *fake_eeprom;
        struct eeprom_buffer *eeprom_buffer;
};

static void eeprom_buffer_test_does_not_write_until_flush(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;
        struct eeprom_buffer *eeprom_buffer = ctx->eeprom_buffer;
        struct fake_eeprom *fake_eeprom = ctx->fake_eeprom;
        char buffer[] = {0xff};

        eeprom_buffer->flush_count = SIZE_MAX;

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0);

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0);

        eeprom_buffer->flush(eeprom_buffer);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0xff);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0xff);
}

static void eeprom_buffer_test_flushes_after_flush_count_met(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;
        struct eeprom_buffer *eeprom_buffer = ctx->eeprom_buffer;
        struct fake_eeprom *fake_eeprom = ctx->fake_eeprom;
        char buffer[] = {0xff};

        eeprom_buffer->flush_count = 2;

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0);

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0xff);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0xff);
}

static void eeprom_buffer_test_flushes_increments_of_flush_count(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;
        struct eeprom_buffer *eeprom_buffer = ctx->eeprom_buffer;
        struct fake_eeprom *fake_eeprom = ctx->fake_eeprom;
```

```
        char buffer[] = {0xff, 0xff};

        eeprom_buffer->flush_count = 2;

        eeprom_buffer->write(eeprom_buffer, buffer, 1);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0);

        eeprom_buffer->write(eeprom_buffer, buffer, 2);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[0], 0xff);
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[1], 0xff);
        /* Should have only flushed the first two bytes. */
        KUNIT_EXPECT_EQ(test, fake_eeprom->contents[2], 0);
}

static int eeprom_buffer_test_init(struct kunit *test)
{
        struct eeprom_buffer_test *ctx;

        ctx = kunit_kzalloc(test, sizeof(*ctx), GFP_KERNEL);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ctx);

        ctx->fake_eeprom = kunit_kzalloc(test, sizeof(*ctx->fake_eeprom), GFP_KERNEL);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ctx->fake_eeprom);
        fake_eeprom_init(ctx->fake_eeprom);

        ctx->eeprom_buffer = new_eeprom_buffer(&ctx->fake_eeprom->parent);
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, ctx->eeprom_buffer);

        test->priv = ctx;

        return 0;
}

static void eeprom_buffer_test_exit(struct kunit *test)
{
        struct eeprom_buffer_test *ctx = test->priv;

        destroy_eeprom_buffer(ctx->eeprom_buffer);
}
```

Testing Against Multiple Inputs

Testing just a few inputs is not enough to ensure that the code works correctly, for example: testing a hash function.

We can write a helper macro or function. The function is called for each input. For example, to test `sha1sum(1)`, we can write:

```
#define TEST_SHA1(in, want) \
        sha1sum(in, out); \
        KUNIT_EXPECT_STREQ_MSG(test, out, want, "sha1sum(%s)", in);

char out[40];
TEST_SHA1("hello world",  "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed");
TEST_SHA1("hello world!", "430ce34d020724ed75a196dfc2ad67c77772d169");
```

Note the use of the `_MSG` version of `KUNIT_EXPECT_STREQ` to print a more detailed error and make the assertions clearer within the helper macros.

The `_MSG` variants are useful when the same expectation is called multiple times (in a loop or helper function) and thus the line number is not enough to identify what failed, as shown below.

In complicated cases, we recommend using a *table-driven test* compared to the helper macro variation, for example:

```
int i;
char out[40];

struct sha1_test_case {
        const char *str;
        const char *sha1;
};

struct sha1_test_case cases[] = {
        {
                .str = "hello world",
                .sha1 = "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed",
        },
        {
                .str = "hello world!",
                .sha1 = "430ce34d020724ed75a196dfc2ad67c77772d169",
        },
};
for (i = 0; i < ARRAY_SIZE(cases); ++i) {
```

```
        sha1sum(cases[i].str, out);
        KUNIT_EXPECT_STREQ_MSG(test, out, cases[i].sha1,
                               "sha1sum(%s)", cases[i].str);
}
```

There is more boilerplate code involved, but it can:

- be more readable when there are multiple inputs/outputs (due to field names).
    - For example, see `fs/ext4/inode-test.c`.
- reduce duplication if test cases are shared across multiple tests.
    - For example: if we want to test `sha256sum`, we could add a `sha256` field and reuse `cases`.
- be converted to a "parameterized test".

Parameterized Testing

The table-driven testing pattern is common enough that KUnit has special support for it.

By reusing the same `cases` array from above, we can write the test as a "parameterized test" with the following.

```
// This is copy-pasted from above.
struct sha1_test_case {
        const char *str;
        const char *sha1;
};
struct sha1_test_case cases[] = {
        {
                .str = "hello world",
                .sha1 = "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed",
        },
        {
                .str = "hello world!",
                .sha1 = "430ce34d020724ed75a196dfc2ad67c77772d169",
        },
};

// Need a helper function to generate a name for each test case.
static void case_to_desc(const struct sha1_test_case *t, char *desc)
{
        strcpy(desc, t->str);
}
// Creates `sha1_gen_params()` to iterate over `cases`.
KUNIT_ARRAY_PARAM(sha1, cases, case_to_desc);

// Looks no different from a normal test.
static void sha1_test(struct kunit *test)
{
        // This function can just contain the body of the for-loop.
        // The former `cases[i]` is accessible under test->param_value.
        char out[40];
        struct sha1_test_case *test_param = (struct sha1_test_case *)(test->param_value);

        sha1sum(test_param->str, out);
        KUNIT_EXPECT_STREQ_MSG(test, out, test_param->sha1,
                               "sha1sum(%s)", test_param->str);
}

// Instead of KUNIT_CASE, we use KUNIT_CASE_PARAM and pass in the
// function declared by KUNIT_ARRAY_PARAM.
static struct kunit_case sha1_test_cases[] = {
        KUNIT_CASE_PARAM(sha1_test, sha1_gen_params),
        {}
};
```

Exiting Early on Failed Expectations

We can use `KUNIT_EXPECT_EQ` to mark the test as failed and continue execution. In some cases, it is unsafe to continue. We can use the `KUNIT_ASSERT` variant to exit on failure.

```
void example_test_user_alloc_function(struct kunit *test)
{
        void *object = alloc_some_object_for_me();

        /* Make sure we got a valid pointer back. */
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, object);
        do_something_with_object(object);
}
```

Allocating Memory

Where you might use `kzalloc`, you can instead use `kunit_kzalloc` as KUnit will then ensure that the memory is freed once the test completes.

This is useful because it lets us use the `KUNIT_ASSERT_EQ` macros to exit early from a test without having to worry about remembering to call `kfree`. For example:

```
void example_test_allocation(struct kunit *test)
{
        char *buffer = kunit_kzalloc(test, 16, GFP_KERNEL);
        /* Ensure allocation succeeded. */
        KUNIT_ASSERT_NOT_ERR_OR_NULL(test, buffer);

        KUNIT_ASSERT_STREQ(test, buffer, "");
}
```

Testing Static Functions

If we do not want to expose functions or variables for testing, one option is to conditionally `#include` the test file at the end of your .c file. For example:

```
/* In my_file.c */

static int do_interesting_thing();

#ifdef CONFIG_MY_KUNIT_TEST
#include "my_kunit_test.c"
#endif
```

Injecting Test-Only Code

Similar to as shown above, we can add test-specific logic. For example:

```
/* In my_file.h */

#ifdef CONFIG_MY_KUNIT_TEST
/* Defined in my_kunit_test.c */
void test_only_hook(void);
#else
void test_only_hook(void) { }
#endif
```

This test-only code can be made more useful by accessing the current `kunit_test` as shown in next section: *Accessing The Current Test*.

Accessing The Current Test

In some cases, we need to call test-only code from outside the test file. For example, see example in section *Injecting Test-Only Code* or if we are providing a fake implementation of an ops struct. Using `kunit_test` field in `task_struct`, we can access it via `current->kunit_test`.

The example below includes how to implement "mocking":

```
#include <linux/sched.h> /* for current */

struct test_data {
        int foo_result;
        int want_foo_called_with;
};

static int fake_foo(int arg)
{
        struct kunit *test = current->kunit_test;
        struct test_data *test_data = test->priv;

        KUNIT_EXPECT_EQ(test, test_data->want_foo_called_with, arg);
        return test_data->foo_result;
}

static void example_simple_test(struct kunit *test)
{
        /* Assume priv (private, a member used to pass test data from
         * the init function) is allocated in the suite's .init */
        struct test_data *test_data = test->priv;

        test_data->foo_result = 42;
        test_data->want_foo_called_with = 1;

        /* In a real test, we'd probably pass a pointer to fake_foo somewhere
         * like an ops struct, etc. instead of calling it directly. */
```

```
        KUNIT_EXPECT_EQ(test, fake_foo(1), 42);
}
```

In this example, we are using the `priv` member of `struct kunit` as a way of passing data to the test from the init function. In general `priv` is pointer that can be used for any user data. This is preferred over static variables, as it avoids concurrency issues.

Had we wanted something more flexible, we could have used a named `kunit_resource`. Each test can have multiple resources which have string names providing the same flexibility as a `priv` member, but also, for example, allowing helper functions to create resources without conflicting with each other. It is also possible to define a clean up function for each resource, making it easy to avoid resource leaks. For more information, see Documentation/dev-tools/kunit/api/test.rst.

Failing The Current Test

If we want to fail the current test, we can use `kunit_fail_current_test(fmt, args...)` which is defined in `<kunit/test-bug.h>` and does not require pulling in `<kunit/test.h>`. For example, we have an option to enable some extra debug checks on some data structures as shown below:

```
#include <kunit/test-bug.h>

#ifdef CONFIG_EXTRA_DEBUG_CHECKS
static void validate_my_data(struct data *data)
{
        if (is_valid(data))
                return;

        kunit_fail_current_test("data %p is invalid", data);

        /* Normal, non-KUnit, error reporting code here. */
}
#else
static void my_debug_function(void) { }
#endif
```