Tips For Writing KUnit Tests

Exiting early on failed expectations

`KUNIT_EXPECT_EQ` and friends will mark the test as failed and continue execution. In some cases, it's unsafe to continue and you 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 would use `kzalloc`, you should prefer `kunit_kzalloc` instead. KUnit will ensure the memory is freed once the test completes.

This is particularly useful since it lets you use the `KUNIT_ASSERT_EQ` macros to exit early from a test without having to worry about remembering to call `kfree`.

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 you don't want to expose functions or variables just for testing, one option is to conditionally `#include` the test file at the end of your .c file, e.g.

```
/* In my_file.c */

static int do_interesting_thing();

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

Injecting test-only code

Similarly to the above, it can be useful to add test-specific logic.

```
/* 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, see below.

Accessing the current test

In some cases, you need to call test-only code from outside the test file, e.g. like in the example above or if you're providing a fake implementation of an ops struct. There is a `kunit_test` field in `task_struct`, so you can access it via `current->kunit_test`.

Here's a slightly in-depth example of how one could 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 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);
}
```

Note: here we're able to get away with using `test->priv`, but if you wanted something more flexible you could use a named `kunit_resource`, see Documentation/dev-tools/kunit/api/test.rst.

Failing the current test

But sometimes, you might just want to fail the current test. In that case, we have `kunit_fail_current_test(fmt, args...)` which is defined in `<kunit/test-bug.h>` and doesn't require pulling in `<kunit/test.h>`.

E.g. say we had an option to enable some extra debug checks on some data structure:

```
#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
```

Customizing error messages

Each of the `KUNIT_EXPECT` and `KUNIT_ASSERT` macros have a `_MSG` variant. These take a format string and arguments to provide additional context to the automatically generated error messages.

```
char some_str[41];
generate_sha1_hex_string(some_str);

/* Before. Not easy to tell why the test failed. */
KUNIT_EXPECT_EQ(test, strlen(some_str), 40);

/* After. Now we see the offending string. */
KUNIT_EXPECT_EQ_MSG(test, strlen(some_str), 40, "some_str='%s'", some_str);
```

Alternatively, one can take full control over the error message by using `KUNIT_FAIL()`, e.g.

```
/* Before */
KUNIT_EXPECT_EQ(test, some_setup_function(), 0);

/* After: full control over the failure message. */
if (some_setup_function())
        KUNIT_FAIL(test, "Failed to setup thing for testing");
```

Next Steps

- Optional: see the Documentation/dev-tools/kunit/usage.rst page for a more in-depth explanation of KUnit.