Kernel Test Framework Documentation

Release 1

Knut Omang

Sep 16, 2017

Contents

CHAPTER 1

1. Background and motivation

Kernel Test Framework (KTF) implements a unit test framework for the Linux kernel. There's a wide selection of unit test frameworks available for normal user land code testing, but I have so far not seen any similar frameworks that can be used with kernel code, to test details of both exported and non-exported kernel APIs. The hope is that providing an easy to use and convenient way to write simple unit tests for kernel internals, that this can promote a more test driven approach to kernel development, where appropriate.

Test driven development

Unit testing is an important component of Test driven development (TDD). The idea of test driven development is that when you have some code to write, whether it is a bug to fix, or a new feature or enhancement, that you start by writing one or more tests for it, have those tests fail, and then do the actual development.

Typically a test driven development cycle would have several rounds of development and test using the new unit tests, and once the (new) tests pass, you would also run all or a suitable subset (limited by execution time) of the old tests to verify that you have not broken any old functionality by the new code.

At this stage it is important that the tests that are run can be run quickly to allow them to be actively used in the development cycle. When time comes for submission of the code, a full, extensive set of both the tests the developer thinks can touch the affected code *and* all the other tests should be run, and a longer time to run tests can be afforded.

KTF tries to support this by using the module system of the kernel to support modularized test suites, where a user only need to insmod the test subsets that he/she wants to use right then. Different test suites may touch and require different kernel APIs and have lots of different module and device requirements. To enable as much reuse of the functionality of code developed within KTF it is important that any piece of test code has as few dependencies as possible.

Good use cases for KTF

Unit testing is at it's most valuable when the code to test is relatively error prone, but still might be difficult to test in a systematic and reproducable way from a normal application level. It can be difficult to trigger corner cases from a high abstraction layer, the code paths we want to exercise might only be used occasionally, or we want to exercise that

error/exception scenarios are handled gracefully. Particularly good use cases are simple APIs with a relativ complex implementation - such as container implementations eg. scatterlists, rbtrees, list, ... are obvious candidates

When *not* to use KTF

Writing kernel code has some challenges compared to user land code. KTF is intended for the cases where it is not easy to get coverage by writing simple tests from user land.

Why *you* would want to write and run KTF tests

Besides the normal write test, write code, run test cycle of development and the obvious benefits of delivering better quality code with fewer embarrassments, there's a few other upsides from developing unit test for a particular area of the kernel:

- A test becomes an invariant for how the code is supposed to work. If someone breaks it, they should detect it and either document the changes that caused the breakage by fixing the test or realize that their fix is broken before you even get to spend time on it.
- Kernel documentation while quite good in some places, does not always cover the full picture, or you might not find that sentence you needed while looking for it. If you want to better understand how a particular kernel module actually works, a good way is to write a test that codes your assumptions. If it passes, all is well, if not, then you have gained some understanding of the kernel.
- Sometimes you may find yourself relying on some specific feature or property of the kernel. If you encode a test that guards the assumption syou have made, you will capture if someone changes it, or if your code is ported to an older kernel which does not support it.

CHAPTER 2

2. Implementation

KTF consists of a kernel part and a user part. The role of the user part is to query the kernel for available tests, and provide mechanisms for executing a selected set or all the available tests, and report the results. The core ktf kernel module simply provides some APIs to write assertions and run tests and to communicate about tests and results with user mode. A simple generic Netlink protocol is used for the communication.

User mode implementation

Since reporting is something existing unit test frameworks for user space code already does well, the implementation of KTF simply leverages that. The current version supports an integration with gtest (Googletest), which provides a lot of these features in a flexible way, but in principle alternative implementations could use the reporting of any other user level unit test framework. The use of gtest also allows this documentation to be shorter, as many of the features in gtest are automatically available for KTF as well. More information about Googletest features can be found here: https://github.com/google/googletest

Kernel mode implementation

The kernel side of KTF implements a simple API for tracking test modules, writing tests, support functions and and a set of assertion macros, some tailored for typical kernel usage, such as `ASSERT_OK_ADDR_GOTO()` as a kernel specific macro to check for a valid address with a label to jump to if the assertion fails. After all as we are still in the kernel, tests would always need to clean up for themselves even though in the context of ktf.

KTF supports two distinct classes of tests:

- Pure kernel mode tests
- Combined tests

Pure kernel mode tests are tests that are fully implemented in kernel space. This is the most straightforward mode and resembles ordinary user land unit testing in kernel mode. If you only have kernel mode tests, you will only ever need one user level program similar to user/ktest.cpp, so all test development takes place on the kernel side.

Combined tests are for testing and making assumptions about the user/kernel communication, for instance if a parameter supplied from user mode is interpreted the intended way when it arrives at it's kernel destination. For such tests you need to tell ktf (from user space) when the kernel part of the test is going to be executed. Apart from that it works mostly like a normal gtest user level test.

KTF as a separate git project

A lot of test infrastructure and utilities for the Linux kernel is implemented as part of the linux kernel git project. This has some obvious benefits, such as

- Always being included
- When APIs are changed, test code can be updated atomically with the rest of the kernel
- Possible higher visibility and easier access
- Easier integration with internal kernel interfaces useful for testing.

On the other hand keeping KTF as a separate project allows

- With some use of `KERNEL_VERSION` and `LINUX_VERSION_CODE`, up-to-date KTF code and tests can be allowed to work across kernel versions.
- This in turn allows newly developed tests to be tested on older kernels as well, possibly detecting more bugs, or instances of bugs not backported.
- User code can be integrated with the kernel code to be built with a single command
- A smaller kernel git module

I am open to suggestions from the community on whether inclusion in the kernel tree or a separate project (or both...) is the best approach.

CHAPTER 3

3. KTF kernel specific features

Running tests, examining results via debugfs

We provide debugfs interfaces for examining the results of the last test run and for running tests which do not require configuration specification. ktf testsets can be run via:

cat /sys/kernel/debug/ktf/run/<testset>

Individual tests can be run via

cat /sys/kernel/debug/ktf/run/<testset>-tests/<test>

Results can be displayed for the last run via

cat /sys/kernel/debug/ktf/results/<testset>

Individual tests can be run via

cat /sys/kernel/debug/ktf/results/<testset>-tests/<test>

These interfaces eliminate the use of the netlink socket API and provide a simple way to keep track of test failures. It can be useful to log into a machine and examine what tests were run without having console history available, and in particular:

cat /sys/kernel/debug/ktf/run/*

...is a useful way of running all KTF tests.

Reference to module internal symbols

When working with unit tests, the need to access non-public interfaces often arises. In general non-public interfaces is of course not intended to be used by outside programs, but a test framework is somewhat special here in that it is often necessary or desirable to unit test internal data structures or algorithms even if they are not exposed. The program under test may be a complicated beast by itself and merely exercising the public interfaces may not be flexible enough to stress the internal code. Even if it is possible to get the necessary “pressure” from the outside like that, it might be much more challenging or require a lot more work.

The usual method to gain access to internal interfaces is to be part of the internals. To some extent this is the way a lot of the kernel testing utilities operate. The obvious advantages of this is that the test code 'automatically' follows the module and it's changes. The disadvantage is that test code is tightly integrated with the code itself. One important goal with KTF is to make it possible to write detailed and sophisticated test code which does not affect the readability or complexity of the tested code.

KTF contains a small python program, `resolve`, which parses a list of symbol names on the form:

```
#module first_module
#header first_header.h
private_symbol1
private_symbol2
...
#header second_header.h
#module next_module
...
```

The output is a header file and a struct containing function pointers and some convenience macro definitions to make it possible to 'use' the internal functions just as one would if within the module. This logic is based on kallsyms, and would of course only work if that functionality is enabled in the kernel KTF compiles against.

Requesting callbacks when a certain function gets called/returns

Tap into function entry using KTF entry probes. Many tests need to move beyond kernel APIs and ensure that side effects (logging a message etc) occur. A good way to do this is to probe entry of relevant functions. In order to do so in KTF you need to:

- define an entry probe function with the same return value and arguments as the function to be probed
- within the body of the entry probe function, ensure return is wrapped with KTF_ENTRY_PROBE_RETURN(<return value>);
- entry probes need to registered for use and de-registered when done via KTF_[UN]REGISTER_ENTRY_PROBE(<kernel function name>).

See example h4.c in examples/ for a simple case where we probe printk() and ensure it is called.

Sometimes is is also useful to check that an intermediate function is returning an expected value. Return probes can be used to register/probe function return. In order to probe function return:

- define a return probe point; i.e KTF_RETURN_PROBE(<kernel function>)
- within the body of the return probe the return value can be retrieved via KTF_RETURN_VALUE(). Type will obviously depend on the function probed so should be cast if dereferencing is required.
- return probes need to be registered for use and unregistered when done via KTF_[UN]REGISTER_RETURN_PROBE(<kernel function name>).

See example h4.c in examples/ for a simple case where we verify return value of printk().

Coverage analytics

While other coverage tools exist, they generally involve gcc-level support which is required at compile-time. KTF offers kernel module coverage support via kprobes instead. Tests can enable/disable coverage on a per-module basis, and coverage data can be retrieved via:

```
# more /sys/kernel/debug/ktf/coverage
```

For a given module we show how many of its functions were called versus the total, e.g.:

```
# cat /sys/kernel/debug/ktf/coverage
MODULE                #FUNCTIONS     #CALLED
selftest              14             1
```

We see 1 out of 14 functions was called when coverage was enabled.

We can also see how many times each function was called:

```
MODULE          FUNCTION                    COUNT
selftest        myelem_free                 0
selftest        myelem_cmp                  0
selftest        ktf_return_printk           0
selftest        cov_counted                 1
selftest        dummy                       0
```

In addition, we can track memory allocated via kmem_cache_alloc()/kmalloc() originating from module functions we have enabled coverage for. This allows us to track memory associated with the module specifically to find leaks etc. If memory tracking is enabled, /sys/kernel/debug/ktf/coverage will show outstanding allocations - the stack at allocation time; the memory address and size.

Coverage can be enabled via the "ktfcov" utility. Syntax is as follows:

ktfcov [-d module] [-e module [-m]]

"-e" enables coverage for the specified module; "-d" disables coverage. "-m" in combination with "-e" enables memory tracking for the module under test.

CHAPTER 4

4. Building and installing KTF

To build KTF, clone the ktf project, then:

```
cd ktf
autoreconf
```

Create a build directory somewhere outside the source tree to allow the same KTF source tree to be used against multiple versions of the kernel. Assuming for simplicity that you want to build for the running kernel but you can build for any installed `kernel-*-devel`:

```
cd <a dedicated build tree for this kernel>
mkdir build
cd build
~/<path_to_source_root>/ktf/configure KVER=`uname -r`
make
```

Now you should have got a `kernel/ktf.ko` that works with your test kernel and modules for the `examples` and KTF `selftest` directories. You are ready to create your own test modules based on KTF!

CHAPTER 5

5. Example test code

Here is a minimal dummy example of a KTF unit test suite that defines two tests, `hello_ok` and `hello_fail`. The test is in the examples directory and is built with KTF:

```
#include <linux/module.h>
#include "ktf.h"

MODULE_LICENSE("GPL");

KTF_INIT();

TEST(examples, hello_ok)
{
        EXPECT_TRUE(true);
}

TEST(examples, hello_fail)
{
        EXPECT_TRUE(false);
}


static void add_tests(void)
{
        ADD_TEST(hello_ok);
        ADD_TEST(hello_fail);
}


static int __init hello_init(void)
{
        add_tests();
        tlog(T_INFO, "hello: loaded");
        return 0;
}
```

```
static void __exit hello_exit(void)
{
        KTF_CLEANUP();
        tlog(T_INFO, "hello: unloaded");
}


module_init(hello_init);
module_exit(hello_exit);
```

To run the test, cd to your KTF build tree and insmod the ktf module and the module that provides the test:

```
insmod kernel/ktf.ko
insmod examples/hello.ko
```

Now you should be able to run one or more of the tests by running the application `ktfrun` built in `user/ktfrun`. You should be able to run that application as an ordinary user:

```
ktfrun --gtest_list_tests
ktfrun --gtest_filter='*fail'
ktfrun --gtest_filter='*ok'
```

There are more examples in the examples directory. KTF also includes a `selftest` directory used to test/check the KTF implementation itself.

CHAPTER 6

6. KTF programming reference

KTF itself contains no tests but provides primitives and data structures to allow tests to be maintained and written in separate test modules that depend on the KTF APIs.

KTF API Overview

For reference, the following table lists a few terms and classes of abstractions provided by KTF:

Item	description
Test module	A kernel object file (.ko) with ktf tests in it
struct ktf_handle	At least 1 per test module. Use macros KTF_INIT() and KTF_CLEANUP() to set up and tear down handles.
struct ktf_context	0-n per test module - test module specific context for the test, such as eg. a device or another kernel object.
KTF_INIT()	Call this at the global level in the main file for each test module.
KTF_CLEANUP()	Call this in the __exit function to clean up
EXPECT_*	non-fatal assertions
ASSERT_*	"fatal" assertions that would cause return/goto
TEST(s, n) {...}	Define a simple test named 's.n' with implicit arguments 'ctx' and '_i' for context/iteration.
DECLARE_F(f) {...}	Declare a new test fixture named 'f' with additional data structure
SETUP_F(f, s) {...}	Define setup function for the fixture
TEARDOWN_F(f, t) {...}	Define teardown function for the fixture
INIT_F(f, s, t) {...}	Declare the setup and tear down functions for the fixture
TEST_F(s, f, n) {...}	Define a test named 's.n' operating in fixture f
KTF_ENTRY_PROBE (f, r, ...) {...}	Define function entry probe for function f with return value r. Arguments following r must match function arguments. Must be used at global level.
KTF_ENTRY_PROBE_RETURN(r)	Return from probed function with return value r. Must be called within KTF_ENTRY_PROBE().
KTF_REGISTER_ENTRY_PROBE(f)	Enable probe on entry to kernel function f.
KTF_UNREGISTER_ENTRY_PROBE 6.	Disable probe on entry to kernel function f.
KTF_RETURN_PROBE(f) {...}	Define function return probe for function f. Must be used at a global level.
KTF_RETURN_VALUE()	Retrieve return value in body of return probe.
KTF_REGISTER_RETURN_PROBE 6.	Enable probe for return of function f.
KTF_UNREGISTER_RETURN_PROBE 6.	Disable probe for return of function f.
ktf_cov_enable(m, flags)	Enable coverage analytics for module m. Flag must be either 0 or KTF_COV_OPT_MEM.
ktf_cov_disable(m)	Disable coverage analytics for module m.

The `KTF_INIT()` macro must be called at a global level as it just defines a variable `__test_handle` which is referred to which existence is assumed to continue until the call to KTF_CLEANUP(), typically done in the `__exit` function of the test module.

Assertions

Below is example documentation for some of the available assertion macros. For a full overview, see `kernel/ktf.h`

ASSERT_TRUE(C)

fail and return if **C** evaluates to false

Parameters

C Boolean expression to evaluate

ASSERT_FALSE(C)
fail and return if **C** evaluates to true

Parameters

C Boolean expression to evaluate

ASSERT_TRUE_GOTO(C, _lbl)
fail and jump to **_lbl** if **C** evaluates to false

Parameters

C Boolean expression to evaluate

_lbl Label to jump to in case of failure

ASSERT_FALSE_GOTO(C, _lbl)
fail and jump to **_lbl** if **C** evaluates to true

Parameters

C Boolean expression to evaluate

_lbl Label to jump to in case of failure

ASSERT_LONG_EQ(X, Y)
compare two longs, fail and return if **X** != **Y**

Parameters

X Expected value

Y Actual value

ASSERT_LONG_EQ_GOTO(X, Y, _lbl)
compare two longs, jump to **_lbl** if **X** != **Y**

Parameters

X Expected value

Y Actual value

_lbl Label to jump to in case of failure

EXPECT_TRUE(C)
fail if **C** evaluates to false but allow test to continue

Parameters

C Boolean expression to evaluate

Index