# Test driven kernel development

Knut Omang, Open Source Summit Europe, 2018

## Agenda

- Test driven development (TDD)
- … in the context of the Linux kernel
- Unit testing in the kernel
  - KTF (Kernel Test Framework)

*“One test result is worth a 1,000 expert opinions.”*

*(from Test Driven Development for Embedded C, by James W. Grenning )*

# Test driven development (TDD)

- Write a (unit/”basic”) test first
- Run it and see it fail on the bug/missing feature
- Implement feature/fix bug/issue
- Run test again and get that good feeling …
- Commit test to continuous integration regression testing (CI)
  - Now nobody else will be able to break your fix without detection!
  - Get fewer embarrassments..
  - Sleep better at night…

## TDD + CI = true

- No point in adding tests if nobody runs them
- If only you run them, you get to fix all the bugs ;-)
- If your tests do not prevent merging of buggy commits by others, you get to fix the bugs the tests detect in other people's regressions too ;-)

- About scaling!
  - scaling out the test writing effort
  - get the full benefit of the tests
  - save human resources in coping with regressions, review broken code…

## The usual arguments against TDD...

- “Writing good tests take a long time...”
  - optimistic planning, unrealistic deadlines (for quality deliverance)
- “I’m a developer, not a tester..”
  - Good developers test their code
- “Writing test code is boring..”
  - Debugging incomprehensible complex issues under time pressure is worse...
- “We already have a lot of applications - no need for ‘synthetic’ tests..”
  - Complex applications much harder to debug than a simple focused failing test
- “Testing this is very difficult”
  - Some of the most challenging problems are debugging problems - cowardly shying away from a challenge? ;-)
  - Divide and conquer, improving tooling, likely to understand the problem better!
  - The alternative, does it terminate at all?
- “I have tested the code”
  - “I did these simple 16 manual steps, very easy to remember...”
  - “but wait: I don’t remember what I did, and now I can’t reproduce...”
  - “Very easy to test, you just run this simple program with these 25 parameters..”
  - “but wait: You need these two configuration files and a few setup scripts”

## My humble experience

- Code that isn't tested does not work…
- Resolution cost increases exponentially with distance from development
- Programming is 10% writing the target code, 90% work on testing
  - Easier, less frustrating, embarrasing, stressful to do it up-front
- Working test driven is more fun too..
  - That warm, fuzzy feeling of a "green" test suite run..
- Faults from full stack applications are usually harder to debug
  - result: More time in the debugger and less time coding, uncertainty about fw.progress!
- Lot to learn from writing the test code..
  - Willingness experience correlated, but young developers have the most to gain!

## Reality…

- Have to create output with perceived value within time limits..
- Putting out fire (due to lack of testing in the first place…)

Means:

- Cease opportunities to improve tests
- Do it right on significant new developments
  - New algorithms, interfaces, particularly complex code pieces
- When painful bugs surfaces, make sure they have a test

## Introducing TDD+CI for legacy code
## - not for the faint hearted…

- Potential is great but be prepared for an uphill battle!
  - general resistance against writing tests
  - test dev doesn’t automatically give credit, on the contrary…
  - short term ongoing development needs may complicate
  - component under test may not initially lend itself well to automated tests (baseline, APIs)
- And suppose you pull it off?
  - “This code hasn’t hit a single bug for a long time, so it must be easy…
    - why did we spend all that time developing tests for it??
  - Or: “We have all these issues in this other module (which by the way has no tests)
    so have to take some of your resources, sorry!”

*A project where all useful tests have been written is a dead project!*

## Properties of good unit test classes of tests

- Easy to run
  - normally/ideally just one way to run it - anyone can!
  - runs (relatively) quickly - short development cycle
- Easy(-ier) to debug
  - exercises one (or at least few ← pragmatism)  feature(s)
- Output of passed tests nice, compact, and easy to read/check (green)
  - and also gives some positive reinforcement (developers also get fuelled by “neat”)
- Output of failed tests focused and detailed/easy to pinpoint (yellow/red)
  - short, lend itself to automated reports etc..

## Unit test roles

- Test new code and new APIs
  - container class impl/usage
  - complex data structures, intricate use cases
- Tests becomes invariants for how the code is supposed to work
  - Trap if someone breaks it - now they got the work instead of you!
  - Tests as documentation of semantics
- Learn someone else’s code - how does this work?
  - Code your own assumptions – verify!
- Put up guards around assumptions made about other code
  - If your code relies on some property, make sure to capture if the property changes!

# Reviewing code…

- Reading other's code the hardest
  - Hardship inversely proportional to the quality of the code..
- A test suite == executable review?
  - Trying to understand someone else's change
  - Need to understand the original code
  - Need to understand the change
  - Convince oneself that there's no flaws:
  - Hypotheses: What if…, what if not.. --> tests?

## Testing the Linux kernel

- Higher stakes in kernel space
- Immense complexity
  - considering the cartesian product of all contexts and configs
- Test from user mode where possible
- But impossible to provoke all scenarios without kernel integration!
  - Testing kernel level APIs (external *and* internal)
  - Provoking error scenarios

## Testing the Linux kernel

Based on my limited oversight:

- “User” detected bugs..
- Added complexity of configuration options (ktest)
- Testing the basic operation (kselftests)
- Compile time: checkpatch, sparse, smatch, coccinelle,...
- Runtime: KASAN, lockdep, ...
- Random testing - Syzkaller
- Unit tests for some specific subsystems
- Running real use case workloads

## Good tools more than half the work..

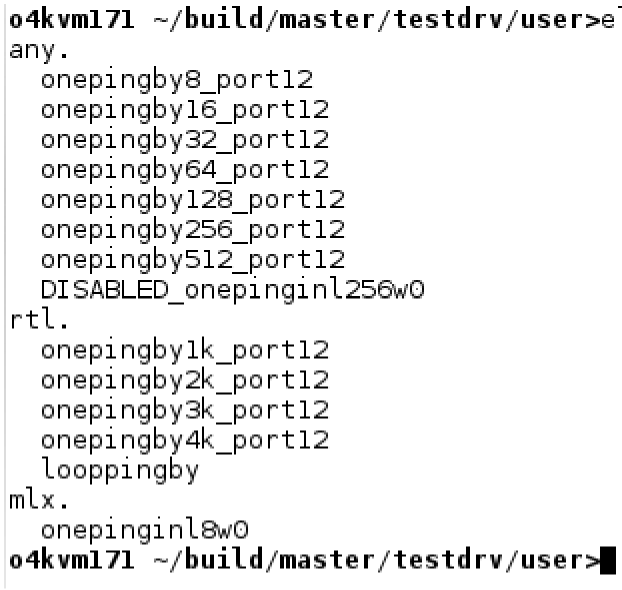
or

- and better quality results too!

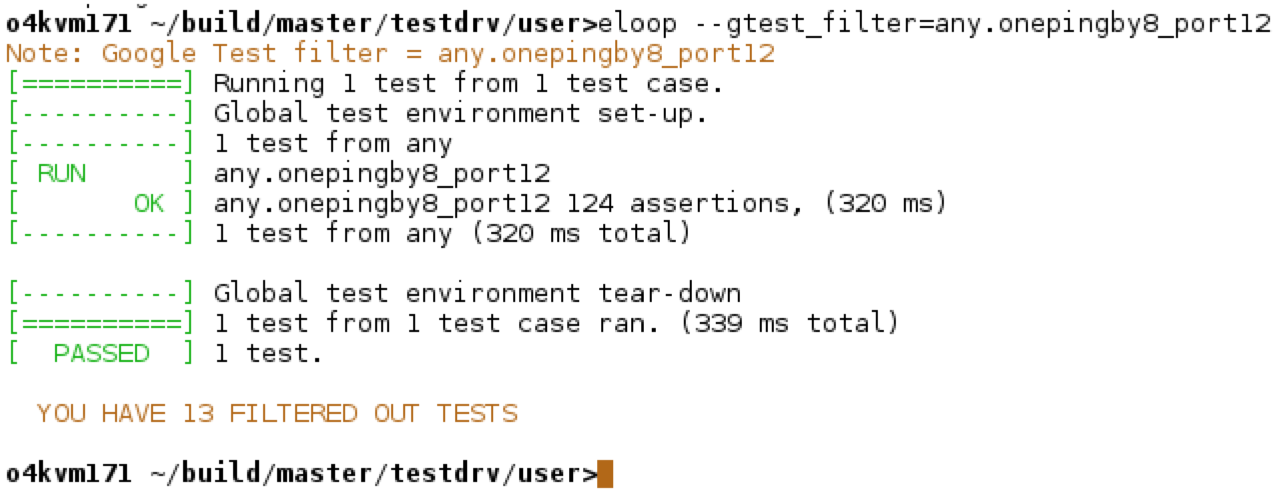
## KTF - Kernel Test Framework

- Once a test driven developer you never want to go back ;-)
- Source: https://github.com/oracle/ktf
  - Sphinx formatted docs: http://heim.ifi.uio.no/~knuto/ktf/
- A toolbox for writing modularized unit test suites in kernel code
- Simple way of running selected/all kernel tests from user land
- Error injection (by use of kprobes)
- Simple debugfs inspection
- Hybrid testing

# Leveraging existing work: gtest (GoogleTest)



```
o4kvm171 ~/build/master/testdrv/user>eloop --gtest_list_tests
any.
  onepingby8_port12
  onepingby16_port12
  onepingby32_port12
  onepingby64_port12
  onepingby128_port12
  onepingby256_port12
  onepingby512_port12
  DISABLED_onepinginl256w0
rtl.
  onepingby1k_port12
  onepingby2k_port12
  onepingby3k_port12
  onepingby4k_port12
  looppingby
mlx.
  onepinginl8w0
o4kvm171 ~/build/master/testdrv/user>
```



```
o4kvm171 ~/build/master/testdrv/user>eloop --gtest_filter=any.onepingby8_port12
Note: Google Test filter = any.onepingby8_port12
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from any
[ RUN      ] any.onepingby8_port12
[       OK ] any.onepingby8_port12 124 assertions, (320 ms)
[----------] 1 test from any (320 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (339 ms total)
[  PASSED  ] 1 test.

  YOU HAVE 13 FILTERED OUT TESTS

o4kvm171 ~/build/master/testdrv/user>
```

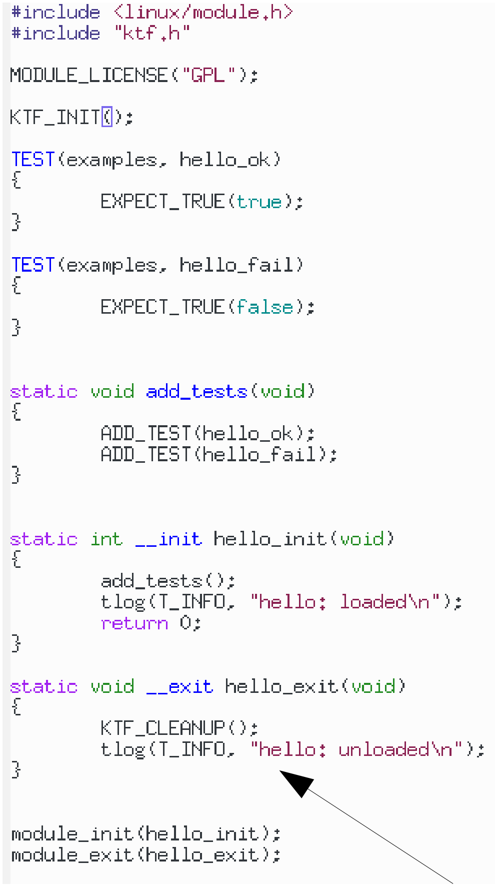
## Leveraging existing work: gtest (GoogleTest)

- C++ based unit test framework
- Reuse system for selecting/running/reporting the tests
- Kernel API made similar to gtest (but with C limitations, kernel req):
  - TEST(), TEST_F()
  - EXPECT_INT_EQ(A,B)
  - ASSERT_ADDR_NE(A, B)
  - ASSERT_OK_ADDR_GOTO(A, B, label)

# Kernel Test Framework (ktf) implementation

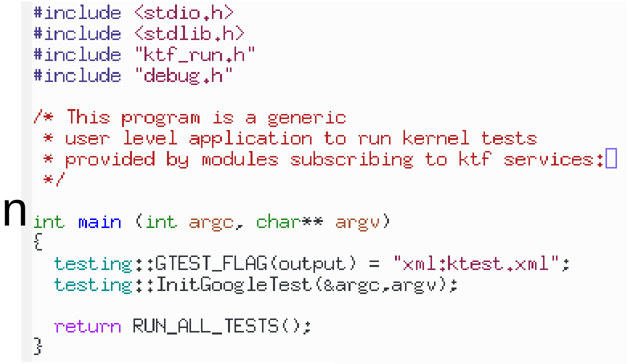
- Generic netlink protocol to query/run/report tests++
- Defines header with macros for creating tests and making assertions
- Defines a few necessary datatypes (ktf_handle, ktf_case, ktf_context)
- Some support utilities
- Kernel logic implemented by (minimal) ktf module
- Users implements test suites as individual modules dependent on ktf
- Aid to get started with new tests suites

# ktf - hello world test

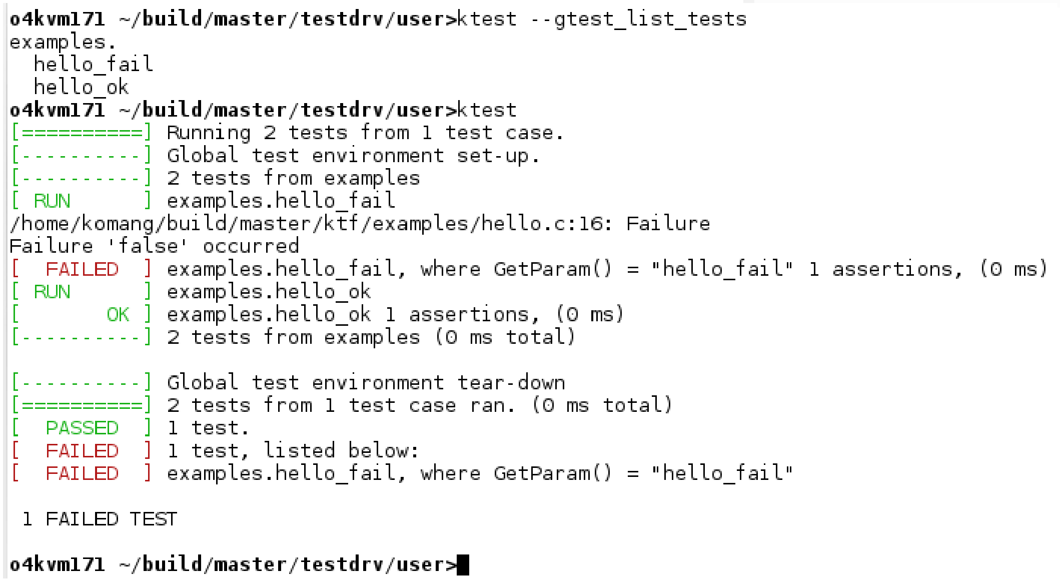


```
#include <linux/module.h>
#include "ktf.h"

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\n");
        return 0;
}

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


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

user mode part (generic):
g++ -lktf  ktfrun.cpp -o ktfrun



```
#include <stdio.h>
#include <stdlib.h>
#include "ktf_run.h"
#include "debug.h"

/* This program is a generic
 * user level application to run kernel tests
 * provided by modules subscribing to ktf services:
 */

int main (int argc, char** argv)
{
  testing::GTEST_FLAG(output) = "xml:ktest.xml";
  testing::InitGoogleTest(&argc,argv);

  return RUN_ALL_TESTS();
}
```



```
o4kvm171 ~/build/master/testdrv/user>ktest --gtest_list_tests
examples.
  hello_fail
  hello_ok
o4kvm171 ~/build/master/testdrv/user>ktest
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from examples
[ RUN      ] examples.hello_fail
/home/komang/build/master/ktf/examples/hello.c:16: Failure
Failure 'false' occurred
[  FAILED  ] examples.hello_fail, where GetParam() = "hello_fail" 1 assertions, (0 ms)
[ RUN      ] examples.hello_ok
[       OK ] examples.hello_ok 1 assertions, (0 ms)
[----------] 2 tests from examples (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] examples.hello_fail, where GetParam() = "hello_fail"

 1 FAILED TEST

o4kvm171 ~/build/master/testdrv/user>
```

echo 0xfff > /sys/module/ktf/parameters/debug_mask

## Questions/demo...