Kernel Testing Guide

There are a number of different tools for testing the Linux kernel, so knowing when to use each of them can be a challenge. This document provides a rough overview of their differences, and how they fit together.

Writing and Running Tests

The bulk of kernel tests are written using either the kselftest or KUnit frameworks. These both provide infrastructure to help make running tests and groups of tests easier, as well as providing helpers to aid in writing new tests.

If you're looking to verify the behaviour of the Kernel â€” particularly specific parts of the kernel â€” then you'll want to use KUnit or kselftest.

The Difference Between KUnit and kselftest

KUnit (Documentation/dev-tools/kunit/index.rst) is an entirely in-kernel system for "white box" testing: because test code is part of the kernel, it can access internal structures and functions which aren't exposed to userspace.

KUnit tests therefore are best written against small, self-contained parts of the kernel, which can be tested in isolation. This aligns well with the concept of 'unit' testing.

For example, a KUnit test might test an individual kernel function (or even a single codepath through a function, such as an error handling case), rather than a feature as a whole.

This also makes KUnit tests very fast to build and run, allowing them to be run frequently as part of the development process.

There is a KUnit test style guide which may give further pointers in Documentation/dev-tools/kunit/style.rst

kselftest (Documentation/dev-tools/kselftest.rst), on the other hand, is largely implemented in userspace, and tests are normal userspace scripts or programs.

This makes it easier to write more complicated tests, or tests which need to manipulate the overall system state more (e.g., spawning processes, etc.). However, it's not possible to call kernel functions directly from kselftest. This means that only kernel functionality which is exposed to userspace somehow (e.g. by a syscall, device, filesystem, etc.) can be tested with kselftest. To work around this, some tests include a companion kernel module which exposes more information or functionality. If a test runs mostly or entirely within the kernel, however, KUnit may be the more appropriate tool.

kselftest is therefore suited well to tests of whole features, as these will expose an interface to userspace, which can be tested, but not implementation details. This aligns well with 'system' or 'end-to-end' testing.

For example, all new system calls should be accompanied by kselftest tests.

Code Coverage Tools

The Linux Kernel supports two different code coverage measurement tools. These can be used to verify that a test is executing particular functions or lines of code. This is useful for determining how much of the kernel is being tested, and for finding corner-cases which are not covered by the appropriate test.

Documentation/dev-tools/gcov.rst is GCC's coverage testing tool, which can be used with the kernel to get global or per-module coverage. Unlike KCOV, it does not record per-task coverage. Coverage data can be read from debugfs, and interpreted using the usual gcov tooling.

Documentation/dev-tools/kcov.rst is a feature which can be built in to the kernel to allow capturing coverage on a per-task level. It's therefore useful for fuzzing and other situations where information about code executed during, for example, a single syscall is useful.

Dynamic Analysis Tools

The kernel also supports a number of dynamic analysis tools, which attempt to detect classes of issues when they occur in a running kernel. These typically each look for a different class of bugs, such as invalid memory accesses, concurrency issues such as data races, or other undefined behaviour like integer overflows.

Some of these tools are listed below:

- kmemleak detects possible memory leaks. See Documentation/dev-tools/kmemleak.rst
- KASAN detects invalid memory accesses such as out-of-bounds and use-after-free errors. See Documentation/dev-tools/kasan.rst
- UBSAN detects behaviour that is undefined by the C standard, like integer overflows. See Documentation/dev-tools/ubsan.rst
- KCSAN detects data races. See Documentation/dev-tools/kcsan.rst
- KFENCE is a low-overhead detector of memory issues, which is much faster than KASAN and can be used in production. See Documentation/dev-tools/kfence.rst
- lockdep is a locking correctness validator. See Documentation/locking/lockdep-design.rst

- There are several other pieces of debug instrumentation in the kernel, many of which can be found in lib/Kconfig.debug

These tools tend to test the kernel as a whole, and do not "pass" like kselftest or KUnit tests. They can be combined with KUnit or kselftest by running tests on a kernel with these tools enabled: you can then be sure that none of these errors are occurring during the test.

Some of these tools integrate with KUnit or kselftest and will automatically fail tests if an issue is detected.