Tips For Running KUnit Tests

Using `kunit.py run` ("kunit tool")

Running from any directory

It can be handy to create a bash function like:

```
function run_kunit() {
  ( cd "$(git rev-parse --show-toplevel)" && ./tools/testing/kunit/kunit.py run $@ )
}
```

> **Note**
>
> Early versions of `kunit.py` (before 5.6) didn't work unless run from the kernel root, hence the use of a subshell and `cd`.

Running a subset of tests

`kunit.py run` accepts an optional glob argument to filter tests. The format is `"<suite_glob>[.test_glob]"`.

Say that we wanted to run the sysctl tests, we could do so via:

```
$ echo -e 'CONFIG_KUNIT=y\nCONFIG_KUNIT_ALL_TESTS=y' > .kunit/.kunitconfig
$ ./tools/testing/kunit/kunit.py run 'sysctl*'
```

We can filter down to just the "write" tests via:

```
$ echo -e 'CONFIG_KUNIT=y\nCONFIG_KUNIT_ALL_TESTS=y' > .kunit/.kunitconfig
$ ./tools/testing/kunit/kunit.py run 'sysctl*.*write*'
```

We're paying the cost of building more tests than we need this way, but it's easier than fiddling with `.kunitconfig` files or commenting out `kunit_suite`'s.

However, if we wanted to define a set of tests in a less ad hoc way, the next tip is useful.

Defining a set of tests

`kunit.py run` (along with `build`, and `config`) supports a `--kunitconfig` flag. So if you have a set of tests that you want to run on a regular basis (especially if they have other dependencies), you can create a specific `.kunitconfig` for them.

E.g. kunit has one for its tests:

```
$ ./tools/testing/kunit/kunit.py run --kunitconfig=lib/kunit/.kunitconfig
```

Alternatively, if you're following the convention of naming your file `.kunitconfig`, you can just pass in the dir, e.g.

```
$ ./tools/testing/kunit/kunit.py run --kunitconfig=lib/kunit
```

> **Note**
>
> This is a relatively new feature (5.12+) so we don't have any conventions yet about on what files should be checked in versus just kept around locally. It's up to you and your maintainer to decide if a config is useful enough to submit (and therefore have to maintain).

> **Note**
>
> Having `.kunitconfig` fragments in a parent and child directory is iffy. There's discussion about adding an "import" statement in these files to make it possible to have a top-level config run tests from all child directories. But that would mean `.kunitconfig` files are no longer just simple .config fragments.
>
> One alternative would be to have kunit tool recursively combine configs automagically, but tests could theoretically depend on incompatible options, so handling that would be tricky.

Setting kernel commandline parameters

You can use `--kernel_args` to pass arbitrary kernel arguments, e.g.

```
$ ./tools/testing/kunit/kunit.py run --kernel_args=param=42 --kernel_args=param2=false
```

Generating code coverage reports under UML

Note

TODO(brendanhiggins@google.com): There are various issues with UML and versions of gcc 7 and up. You're likely to run into missing `.gcda` files or compile errors.

This is different from the "normal" way of getting coverage information that is documented in Documentation/dev-tools/gcov.rst.

Instead of enabling `CONFIG_GCOV_KERNEL=y`, we can set these options:

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools][kunit]running_tips.rst`, line 113)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        CONFIG_DEBUG_KERNEL=y
        CONFIG_DEBUG_INFO=y
        CONFIG_GCOV=y
```

Putting it together into a copy-pastable sequence of commands:

```
# Append coverage options to the current config
$ echo -e "CONFIG_DEBUG_KERNEL=y\nCONFIG_DEBUG_INFO=y\nCONFIG_GCOV=y" >> .kunit/.kunitconfig
$ ./tools/testing/kunit/kunit.py run
# Extract the coverage information from the build dir (.kunit/)
$ lcov -t "my_kunit_tests" -o coverage.info -c -d .kunit/

# From here on, it's the same process as with CONFIG_GCOV_KERNEL=y
# E.g. can generate an HTML report in a tmp dir like so:
$ genhtml -o /tmp/coverage_html coverage.info
```

If your installed version of gcc doesn't work, you can tweak the steps:

```
$ ./tools/testing/kunit/kunit.py run --make_options=CC=/usr/bin/gcc-6
$ lcov -t "my_kunit_tests" -o coverage.info -c -d .kunit/ --gcov-tool=/usr/bin/gcov-6
```

Running tests manually

Running tests without using `kunit.py run` is also an important use case. Currently it's your only option if you want to test on architectures other than UML.

As running the tests under UML is fairly straightforward (configure and compile the kernel, run the `./linux` binary), this section will focus on testing non-UML architectures.

Running built-in tests

When setting tests to `=y`, the tests will run as part of boot and print results to dmesg in TAP format. So you just need to add your tests to your `.config`, build and boot your kernel as normal.

So if we compiled our kernel with:

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools][kunit]running_tips.rst`, line 164)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        CONFIG_KUNIT=y
        CONFIG_KUNIT_EXAMPLE_TEST=y
```

Then we'd see output like this in dmesg signaling the test ran and passed:

**System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools]` [kunit]running_tips.rst, line 171)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        TAP version 14
        1..1
            # Subtest: example
            1..1
            # example_simple_test: initializing
            ok 1 - example_simple_test
        ok 1 - example
```

Running tests as modules

Depending on the tests, you can build them as loadable modules.

For example, we'd change the config options from before to

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools][kunit]running_tips.rst, line 188)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        CONFIG_KUNIT=y
        CONFIG_KUNIT_EXAMPLE_TEST=m
```

Then after booting into our kernel, we can run the test via

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools][kunit]running_tips.rst, line 195)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        $ modprobe kunit-example-test
```

This will then cause it to print TAP output to stdout.

Note

The `modprobe` will *not* have a non-zero exit code if any test failed (as of 5.13). But `kunit.py parse` would, see below.

Note

You can set `CONFIG_KUNIT=m` as well, however, some features will not work and thus some tests might break. Ideally tests would specify they depend on `KUNIT=y` in their `Kconfig`'s, but this is an edge case most test authors won't think about. As of 5.13, the only difference is that `current->kunit_test` will not exist.

Pretty-printing results

You can use `kunit.py parse` to parse dmesg for test output and print out results in the same familiar format that `kunit.py run` does.

```
$ ./tools/testing/kunit/kunit.py parse /var/log/dmesg
```

Retrieving per suite results

Regardless of how you're running your tests, you can enable `CONFIG_KUNIT_DEBUGFS` to expose per-suite TAP-formatted results:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools]

[kunit]running_tips.rst, line 230)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        CONFIG_KUNIT=y
        CONFIG_KUNIT_EXAMPLE_TEST=m
        CONFIG_KUNIT_DEBUGFS=y
```

The results for each suite will be exposed under `/sys/kernel/debug/kunit/<suite>/results`. So using our example config:

```
$ modprobe kunit-example-test > /dev/null
$ cat /sys/kernel/debug/kunit/example/results
... <TAP output> ...

# After removing the module, the corresponding files will go away
$ modprobe -r kunit-example-test
$ cat /sys/kernel/debug/kunit/example/results
/sys/kernel/debug/kunit/example/results: No such file or directory
```

Generating code coverage reports

See Documentation/dev-tools/gcov.rst for details on how to do this.

The only vaguely KUnit-specific advice here is that you probably want to build your tests as modules. That way you can isolate the coverage from tests from other code executed during boot, e.g.

```
# Reset coverage counters before running the test.
$ echo 0 > /sys/kernel/debug/gcov/reset
$ modprobe kunit-example-test
```