kunit_tool How-To

What is kunit_tool?

kunit_tool is a script (`tools/testing/kunit/kunit.py`) that aids in building the Linux kernel as UML (User Mode Linux), running KUnit tests, parsing the test results and displaying them in a user friendly manner.

kunit_tool addresses the problem of being able to run tests without needing a virtual machine or actual hardware with User Mode Linux. User Mode Linux is a Linux architecture, like ARM or x86; however, unlike other architectures it compiles the kernel as a standalone Linux executable that can be run like any other program directly inside of a host operating system. To be clear, it does not require any virtualization support: it is just a regular program.

What is a .kunitconfig?

It's just a defconfig that kunit_tool looks for in the build directory (`.kunit` by default). kunit_tool uses it to generate a .config as you might expect. In addition, it verifies that the generated .config contains the CONFIG options in the .kunitconfig; the reason it does this is so that it is easy to be sure that a CONFIG that enables a test actually ends up in the .config.

It's also possible to pass a separate .kunitconfig fragment to kunit_tool, which is useful if you have several different groups of tests you wish to run independently, or if you want to use pre-defined test configs for certain subsystems.

Getting Started with kunit_tool

If a kunitconfig is present at the root directory, all you have to do is:

```
./tools/testing/kunit/kunit.py run
```

However, you most likely want to use it with the following options:

```
./tools/testing/kunit/kunit.py run --timeout=30 --jobs=`nproc --all`
```

- `--timeout` sets a maximum amount of time to allow tests to run.
- `--jobs` sets the number of threads to use to build the kernel.

Note

This command will work even without a .kunitconfig file: if no .kunitconfig is present, a default one will be used instead.

If you wish to use a different .kunitconfig file (such as one provided for testing a particular subsystem), you can pass it as an option.

```
./tools/testing/kunit/kunit.py run --kunitconfig=fs/ext4/.kunitconfig
```

For a list of all the flags supported by kunit_tool, you can run:

```
./tools/testing/kunit/kunit.py run --help
```

Configuring, Building, and Running Tests

It's also possible to run just parts of the KUnit build process independently, which is useful if you want to make manual changes to part of the process.

A .config can be generated from a .kunitconfig by using the `config` argument when running kunit_tool:

```
./tools/testing/kunit/kunit.py config
```

Similarly, if you just want to build a KUnit kernel from the current .config, you can use the `build` argument:

```
./tools/testing/kunit/kunit.py build
```

And, if you already have a built UML kernel with built-in KUnit tests, you can run the kernel and display the test results with the `exec` argument:

```
./tools/testing/kunit/kunit.py exec
```

The `run` command which is discussed above is equivalent to running all three of these in sequence.

All of these commands accept a number of optional command-line arguments. The `--help` flag will give a complete list of these, or keep reading this page for a guide to some of the more useful ones.

Parsing Test Results

KUnit tests output their results in TAP (Test Anything Protocol) format. kunit_tool will, when running tests, parse this output and print a summary which is much more pleasant to read. If you wish to look at the raw test results in TAP format, you can pass the `--raw_output` argument.

```
./tools/testing/kunit/kunit.py run --raw_output
```

The raw output from test runs may contain other, non-KUnit kernel log lines. You can see just KUnit output with `--raw_output=kunit`:

```
./tools/testing/kunit/kunit.py run --raw_output=kunit
```

If you have KUnit results in their raw TAP format, you can parse them and print the human-readable summary with the `parse` command for kunit_tool. This accepts a filename for an argument, or will read from standard input.

```
# Reading from a file
./tools/testing/kunit/kunit.py parse /var/log/dmesg
# Reading from stdin
dmesg | ./tools/testing/kunit/kunit.py parse
```

This is very useful if you wish to run tests in a configuration not supported by kunit_tool (such as on real hardware, or an unsupported architecture).

Filtering Tests

It's possible to run only a subset of the tests built into a kernel by passing a filter to the `exec` or `run` commands. For example, if you only wanted to run KUnit resource tests, you could use:

```
./tools/testing/kunit/kunit.py run 'kunit-resource*'
```

This uses the standard glob format for wildcards.

Running Tests on QEMU

kunit_tool supports running tests on QEMU as well as via UML (as mentioned elsewhere). The default way of running tests on QEMU requires two flags:

`--arch`
: Selects a collection of configs (Kconfig as well as QEMU configs options, etc) that allow KUnit tests to be run on the specified architecture in a minimal way; this is usually not much slower than using UML. The architecture argument is the same as the name of the option passed to the `ARCH` variable used by Kbuild. Not all architectures are currently supported by this flag, but can be handled by the `--qemu_config` discussed later. If `um` is passed (or this this flag is ignored) the tests will run via UML. Non-UML architectures, e.g. i386, x86_64, arm, um, etc. Non-UML run on QEMU.

`--cross_compile`
: Specifies the use of a toolchain by Kbuild. The argument passed here is the same passed to the `CROSS_COMPILE` variable used by Kbuild. As a reminder this will be the prefix for the toolchain binaries such as gcc for example `sparc64-linux-gnu-` if you have the sparc toolchain installed on your system, or `$HOME/toolchains/microblaze/gcc-9.2.0-nolibc/microblaze-linux/bin/microblaze-linux-` if you have downloaded the microblaze toolchain from the 0-day website to a directory in your home directory called `toolchains`.

In many cases it is likely that you may want to run an architecture which is not supported by the `--arch` flag, or you may want to just run KUnit tests on QEMU using a non-default configuration. For this use case, you can write your own QemuConfig. These QemuConfigs are written in Python. They must have an import line `from ..qemu_config import QemuArchParams` at the top of the file and the file must contain a variable called `QEMU_ARCH` that has an instance of `QemuArchParams` assigned to it. An example can be seen in `tools/testing/kunit/qemu_configs/x86_64.py`.

Once you have a QemuConfig you can pass it into kunit_tool using the `--qemu_config` flag; when used this flag replaces the `--arch` flag. If we were to do this with the `x86_64.py` example from above, the invocation would look something like this:

```
./tools/testing/kunit/kunit.py run \
        --timeout=60 \
        --jobs=12 \
        --qemu_config=./tools/testing/kunit/qemu_configs/x86_64.py
```

Other Useful Options

kunit_tool has a number of other command-line arguments which can be useful when adapting it to fit your environment or needs.

Some of the more useful ones are:

`--help`
: Lists all of the available options. Note that different commands (`config`, `build`, `run`, etc) will have different supported options. Place `--help` before the command to list common options, and after the command for options specific to that command.

`--build_dir`
: Specifies the build directory that kunit_tool will use. This is where the .kunitconfig file is located, as well as where the .config and compiled kernel will be placed. Defaults to `.kunit`.

`--make_options`
: Specifies additional options to pass to `make` when compiling a kernel (with the `build` or `run` commands). For example, to enable compiler warnings, you can pass `--make_options W=1`.

`--alltests`
: Builds a UML kernel with all config options enabled using `make allyesconfig`. This allows you to run as many tests as is possible, but is very slow and prone to breakage as new options are added or modified. In most cases, enabling all tests which have satisfied dependencies by adding `CONFIG_KUNIT_ALL_TESTS=1` to your .kunitconfig is preferable.

There are several other options (and new ones are often added), so do check `--help` if you're looking for something not mentioned here.