KUnit Architecture

The KUnit architecture can be divided into two parts:

- Kernel testing library
- kunit_tool (Command line test harness)

In-Kernel Testing Framework

The kernel testing library supports KUnit tests written in C using KUnit. KUnit tests are kernel code. KUnit does several things:

- Organizes tests
- Reports test results
- Provides test utilities

Test Cases

The fundamental unit in KUnit is the test case. The KUnit test cases are grouped into KUnit suites. A KUnit test case is a function with type signature `void (*)(struct kunit *test)`. These test case functions are wrapped in a struct called struct kunit_case.

Each KUnit test case gets a `struct kunit` context object passed to it that tracks a running test. The KUnit assertion macros and other KUnit utilities use the `struct kunit` context object. As an exception, there are two fields:

- `->priv`: The setup functions can use it to store arbitrary test user data.
- `->param_value`: It contains the parameter value which can be retrieved in the parameterized tests.

Test Suites

A KUnit suite includes a collection of test cases. The KUnit suites are represented by the `struct kunit_suite`. For example:

```
static struct kunit_case example_test_cases[] = {
        KUNIT_CASE(example_test_foo),
        KUNIT_CASE(example_test_bar),
        KUNIT_CASE(example_test_baz),
        {}
};

static struct kunit_suite example_test_suite = {
        .name = "example",
        .init = example_test_init,
        .exit = example_test_exit,
        .test_cases = example_test_cases,
};
kunit_test_suite(example_test_suite);
```

In the above example, the test suite `example_test_suite`, runs the test cases `example_test_foo`, `example_test_bar`, and `example_test_baz`. Before running the test, the `example_test_init` is called and after running the test, `example_test_exit` is called. The `kunit_test_suite(example_test_suite)` registers the test suite with the KUnit test framework.

Executor

The KUnit executor can list and run built-in KUnit tests on boot. The Test suites are stored in a linker section called `.kunit_test_suites`. For code, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/asm-generic/vmlinux.lds.h?h=v5.15#n945. The linker section consists of an array of pointers to `struct kunit_suite`, and is populated by the `kunit_test_suites()` macro. To run all tests compiled into the kernel, the KUnit executor iterates over the linker section array.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\[linux-master][Documentation][dev-tools][kunit]architecture.rst, line 87)

Unknown directive type "kernel-figure".

```
.. kernel-figure:: kunit_suitememorydiagram.svg
        :alt:   KUnit Suite Memory

        KUnit Suite Memory Diagram
```

On the kernel boot, the KUnit executor uses the start and end addresses of this section to iterate over and run all tests. For code, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/lib/kunit/executor.c

When built as a module, the `kunit_test_suites()` macro defines a `module_init()` function, which runs all the tests in the compilation unit instead of utilizing the executor.

In KUnit tests, some error classes do not affect other tests or parts of the kernel, each KUnit case executes in a separate thread context. For code, see: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/lib/kunit/try-catch.c?h=v5.15#n58

Assertion Macros

KUnit tests verify state using expectations/assertions. All expectations/assertions are formatted as:
`KUNIT_{EXPECT|ASSERT}_<op>[_MSG](kunit, property[, message])`

- `{EXPECT|ASSERT}` determines whether the check is an assertion or an expectation.
    - For an expectation, if the check fails, marks the test as failed and logs the failure.
    - An assertion, on failure, causes the test case to terminate immediately.
        - Assertions call function: `void __noreturn kunit_abort(struct kunit *).`
        - `kunit_abort` calls function: `void __noreturn kunit_try_catch_throw(struct kunit_try_catch *try_catch).`
        - `kunit_try_catch_throw` calls function: `void complete_and_exit(struct completion *, long) __noreturn;` and terminates the special thread context.
- `<op>` denotes a check with options: `TRUE` (supplied property has the boolean value "true"), `EQ` (two supplied properties are equal), `NOT_ERR_OR_NULL` (supplied pointer is not null and does not contain an "err" value).
- `[_MSG]` prints a custom message on failure.

Test Result Reporting

KUnit prints test results in KTAP format. KTAP is based on TAP14, see: https://github.com/isaacs/testanything.github.io/blob/tap14/tap-version-14-specification.md. KTAP (yet to be standardized format) works with KUnit and Kselftest. The KUnit executor prints KTAP results to dmesg, and debugfs (if configured).

Parameterized Tests

Each KUnit parameterized test is associated with a collection of parameters. The test is invoked multiple times, once for each parameter value and the parameter is stored in the `param_value` field. The test case includes a KUNIT_CASE_PARAM() macro that accepts a generator function. The generator function is passed the previous parameter and returns the next parameter. It also provides a macro to generate common-case generators based on arrays.

kunit_tool (Command Line Test Harness)

kunit_tool is a Python script `(tools/testing/kunit/kunit.py)` that can be used to configure, build, exec, parse and run (runs other commands in order) test results. You can either run KUnit tests using kunit_tool or can include KUnit in kernel and parse manually.

- `configure` command generates the kernel `.config` from a `.kunitconfig` file (and any architecture-specific options). For some architectures, additional config options are specified in the `qemu_config` Python script (For example: `tools/testing/kunit/qemu_configs/powerpc.py`). It parses both the existing `.config` and the `.kunitconfig` files and ensures that `.config` is a superset of `.kunitconfig`. If this is not the case, it will combine the two and run `make olddefconfig` to regenerate the `.config` file. It then verifies that `.config` is now a superset. This checks if all Kconfig dependencies are correctly specified in `.kunitconfig`. `kunit_config.py` includes the parsing Kconfigs code. The code which runs `make olddefconfig` is a part of `kunit_kernel.py`. You can invoke this command via: `./tools/testing/kunit/kunit.py config` and generate a `.config` file.
- `build` runs `make` on the kernel tree with required options (depends on the architecture and some options, for example: build_dir) and reports any errors. To build a KUnit kernel from the current `.config`, you can use the `build` argument: `./tools/testing/kunit/kunit.py build`.
- `exec` command executes kernel results either directly (using User-mode Linux configuration), or via an emulator such as QEMU. It reads results from the log via standard output (stdout), and passes them to `parse` to be parsed. If you already have built a 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`.
- `parse` extracts the KTAP output from a kernel log, parses the test results, and prints a summary. For failed tests, any diagnostic output will be included.