KUnit: Function Redirection and More

Brendan Higgins <brendanhiggins@google.com>
David Gow <davidgow@google.com>

(aka) LPC2022: Dublin Down On KUnit

Brendan Higgins <brendanhiggins@google.com>
David Gow <davidgow@google.com>

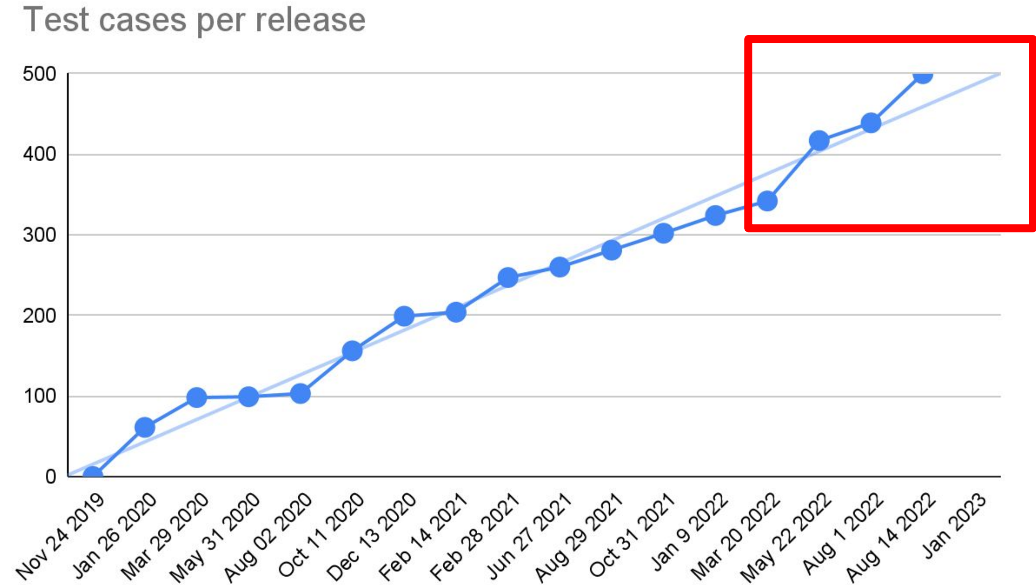
State of the KUnit

What's happened in the last year?

- Function redirection proposals
  - More on this later!
- New tooling features:
  - --kconfig_add and similar flags
  - + run_isolated
- More compatibility with modules and __init sections
  - Use `kunit_test_init_section_suite`
  - Tests built as modules no-longer conflict with module_init/_shutdown functions (Thanks Jeremy Kerr)
- KASAN support for UML
  - (Thanks Patricia Alfonso and Vincent Whitchurch)
- For the full list, see https://kunit.dev/release_notes.html

Test case growth

- Definitely seeing numbers picking up
- 500 test cases in 6.0-rc2
- Super linear growth for last 3 releases!



A quick list of fun new tests

- DRM subsystem tests / AMDGPU tests
- ChromeOS Embedded Controller Tests
- stackinit and overflow checking tests
- binfmt_elf has a test
- hlist
- and more!

New Features

New features

- --kconfig_add
  - e.g. run with KASAN:
    ./tools/testing/kunit/kunit.py run --kconfig_add CONFIG_KASAN=y
  - (KASAN now supports UML, too!)
  - or, run a Rust kernel with its doctests enabled, built with clang, under qemu on x86_64:
    ./tools/testing/kunit/kunit.py run --arch x86_64 --kconfig_add CONFIG_RUST=y --make_options LLVM=1
- Or chain multiple --kunitconfig options:
  - ```
    ./tools/testing/kunit/kunit.py run \
        --kunitconfig fs/ext4/ \
        --kunitconfig fs/fat/
    ```
- suite_init
- hermetic testing
- kunit.enable
    ```

KTAP is happening

- KTAP is a standard for test output.
- It's upstream now.
  - Documentation/dev-tools/ktap.rst
- And version 2 is happening, thanks Frank!
- Speak now, or have your feature delayed 'til v3

New Maintainer!

- Not really
- David has been a de facto maintainer for a while
- I finally got around to documenting it in MAINTAINERS

Rust Support

- Support for doctests in Rust
  - Rust doctests are automatically converted to KUnit tests.
  - **`./tools/testing/kunit/kunit.py run --arch x86_64 --kconfig_add CONFIG_RUST=y --make_options LLVM=1`**
- UML support in Rust
  - Hitting a few bugs in rustc, but we have workarounds.
- Some interesting discussion around handling stack unwinding after assertions:
  - https://github.com/Rust-for-Linux/linux/issues/759
  - Also affects kernel BUGs
- Difference between binding and using KUnit calls directly, versus writing Rust tests and having the converted automatically.

Resource System

Understanding and Splitting the Resource System

- KUnit supports creating 'resources', which are attached to a test.
- Automatically cleaned up on test exit.
  - Both success and failure.
- Reference counted.
- Able to be 'looked up'
  - Either with a lookup function,
  - or by name (a 'named resource')
- Always represented as a struct kunit_resource*.

Problems

- Inconsistent allocation story.
  - Does KUnit allocate the struct kunit_resource, or does the user?
  - Who then frees it?
  - How do you safely free a resource early?
- Refcounts not always being used correctly.
- Type confusion when looking up resources.
  - Using the 'free' function as a key.
- Use for simple "I just want to free this" cases complicated.

Solution: Split and Simplify

- Simplify the existing API:
  - Focus on "I want to look up this resource" use-case
  - Reference-counting mandatory.
- Add a new 'kunit_defer' API.
  - Based on go's 'defer' statement.
  - Pass a function point and a void * context.
  - Execute on test shutdown, in opposite order.
  - Can 'cancel' or 'trigger' such functions early
  - Easy to wrap allocation / free functions.
- Make sure cleanup happens before the thread is destroyed.
  - (Wherever possible.)
  - Optionally halt / reboot when something goes badly wrong.

Function Redirection

Mocking: Subsystem and Hardware testing

- Unit testing requires isolating the "unit" being tested.
  - In KUnit, this means a "standalone" function.
  - Code which access hardware or otherwise affects / is affected by global state has problems.
- The traditional solution: "mock" versions of hardware or subsystems.
- Automating this is hard:
  - See kunit.dev/mocking.html
- Ultimately, some code (typically a function call) needs to be redirected from calling the "real" function to "test" code.
- Can either:
  - refactor code to support this (passing through test flags, or function pointers), or
  - intercept the call at the callee, and have it behave differently under test
- Both have their place, here are our implementations of the latter:
  - https://lore.kernel.org/lkml/20220910212804.670622-1-davidgow@google.com/

Static stubbing

- Just add this magic incantation to any function which might need replacing:
  - KUNIT_STATIC_STUB_REDIRECT(<function name>, <arguments>…);
- Enable redirection with:
  - kunit_activate_static_stub(test, <real fn>,<replacement fn>);
- No dependencies, works on all architectures.
- Compiles down to nothing if KUnit is not enabled
  - But some small performance cost if it isn't, even on functions not actively being redirected.
- Implementation:
  - https://lore.kernel.org/lkml/20220910212804.670622-2-davidgow@google.com/

static stub example

```c
/* This is a function we'll replace with static stubs. */
static int add_one(int i)
{
	/* This will trigger the stub if active. */
	KUNIT_STATIC_STUB_REDIRECT(add_one, i);
	return i + 1;
}
/* This is used as a replacement for the above function. */
static int subtract_one(int i)
{
	/* We don't need to trigger the stub from the replacement. */
	return i - 1;
}

/*
 * This test shows the use of static stubs.
 */
static void example_static_stub_test(struct kunit *test)
{
	/* By default, function is not stubbed. */
	KUNIT_EXPECT_EQ(test, add_one(1), 2);
	/* Replace add_one() with subtract_one(). */
	kunit_activate_static_stub(test, add_one, subtract_one);
	/* add_one() is now replaced. */
	KUNIT_EXPECT_EQ(test, add_one(1), 0);
	/* Return add_one() to normal. */
	kunit_deactivate_static_stub(test, add_one);
	KUNIT_EXPECT_EQ(test, add_one(1), 2);
}
```

ftrace Stubbing

- Like static stubbing, but using ftrace to redirect function calls.
  - No need for a function prologue macro, but functions can't be inline.
- Almost identical API to static stubbing:
  - kunit_activate_ftrace_stub(test, <real fn> <replacement>);
- No performance overhead at all for un-redirected functions.
- Requires ftrace and livepatch, which are only available on some architectures.
- Implementation:
  - https://lore.kernel.org/lkml/20220910212804.670622-3-davidgow@google.com/

ftrace stub example

```
/* This is a function we'll replace with an ftrace stub. */
static int KUNIT_STUBBABLE add_one(int i)
{
	return i + 1;
}
/* This is used as a replacement for the above function. */
static int subtract_one(int i)
{
	return i - 1;
}
static void example_ftrace_stub_test(struct kunit *test)
{
#if !IS_ENABLED(CONFIG_KUNIT_FTRACE_STUBS)
	kunit_skip(test, "KUNIT_FTRACE_STUBS not enabled");
#else
	/* By default, function is not stubbed. */
	KUNIT_EXPECT_EQ(test, add_one(1), 2);
	/* Replace add_one() with subtract_one(). */
	kunit_activate_ftrace_stub(test, add_one, subtract_one);
	/* add_one() is now replaced. */
	KUNIT_EXPECT_EQ(test, add_one(1), 0);
	/* Return add_one() to normal. */
	kunit_deactivate_ftrace_stub(test, add_one);
	KUNIT_EXPECT_EQ(test, add_one(1), 2);
#endif
}
```

Open Questions

- How useful is function redirection?
- How dangerous is it to replace a widely-used function at runtime?
  - Even if this change is scoped to a single test's kthread.
- Are maintainers okay with disabling inlining or adding these function redirect macros?
  - Even if they compile to nothing if KUnit is disabled.
  - Several people (Android, Red Hat) are building production kernels with KUNIT compiled in!
- Static stubbing, ftrace-based stubbing, both?, neither?, swappable implementations?
  - Can/Do we use static_call to optimise the static_stub implementation?
  - The Code Tagging feature might be interesting here.
  - What can we do to support more architectures for ftrace?

Questions / Comments?

Or visit kunit.dev/ and subscribe to
kunit-dev@googlegroups.com

Backup Slides

Hardware Mocking

- Built on top of function redirection stuff
- logic_iomem
- Current focus on platform drivers

Hardware Mocking - Register Description

```
static struct kunit_fake_register_map_entry
aspeed_i2c_fake_register_map[] = {
    KUNIT_FAKE_REG_32_NOP(ASPEED_I2C_FUN_CTRL_REG),
    KUNIT_FAKE_REG_32_NOP(ASPEED_I2C_AC_TIMING_REG1),
    KUNIT_FAKE_REG_32_NOP(ASPEED_I2C_AC_TIMING_REG2),
    KUNIT_FAKE_REG_32_VAR(ASPEED_I2C_INTR_CTRL_REG,
                          struct aspeed_i2c_fake_device,
                          interrupts_active),
    KUNIT_FAKE_REG_32_RW(ASPEED_I2C_INTR_STS_REG,
                         aspeed_i2c_fake_read_intr_sts_reg,
                         aspeed_i2c_fake_write_intr_sts_reg),
    KUNIT_FAKE_REG_32_RW(ASPEED_I2C_CMD_REG,
                         aspeed_i2c_fake_read_command_reg,
                         aspeed_i2c_fake_write_command_reg),
    KUNIT_FAKE_REG_32_NOP(ASPEED_I2C_DEV_ADDR_REG),
    KUNIT_FAKE_REG_32_RW(ASPEED_I2C_BYTE_BUF_REG,
                         aspeed_i2c_fake_read_byte_buf_reg,
                         aspeed_i2c_fake_write_byte_buf_reg),
    {},
};
```

Hardware Mocking - Register Description

- `KUNIT_FAKE_REG_32_NOP` - Does nothing. Legal to access, but does nothing.
  - Useful for initial prototyping. Figuring out what the hardware does.
- `KUNIT_FAKE_REG_32_VAR` - Stores a value that is easily retrievable, otherwise does nothing.
  - A lot of registers don’t do anything kernel visible right away.
- `KUNIT_FAKE_REG_32_RW` - Does whatever you want.
  - Anything more complicated than storing a value usually requires arbitrary logic.

Hardware Mocking - Register Description

```
KUNIT_FAKE_REG_32_RW(ASPEED_I2C_CMD_REG,
                             aspeed_i2c_fake_read_command_reg,
                             aspeed_i2c_fake_write_command_reg),

static u32
aspeed_i2c_fake_read_command_reg(struct kunit_fake_device *fd,
                                 struct kunit_fake_register_map_entry *entry,
                                 unsigned long offset)
{
	struct aspeed_i2c_fake_device *i2c_fake = fd->priv;

	if (i2c_fake->sda_hung) {
		return ASPEED_I2CD_BUS_BUSY_STS | ASPEED_I2CD_SCL_LINE_STS;
	} else if (i2c_fake->scl_hung) {
		return ASPEED_I2CD_BUS_BUSY_STS | ASPEED_I2CD_SDA_LINE_STS;
	} else if (i2c_fake->busy) {
		i2c_fake->busy = false;
		return ASPEED_I2CD_BUS_BUSY_STS |
		        ASPEED_I2CD_SDA_LINE_STS |
		        ASPEED_I2CD_SCL_LINE_STS;
	}
	return 0;
}
```

Hardware Mocking - Register Description

```
static void aspeed_i2c_master_xfer_idle_bus(struct kunit *test)
{
        struct aspeed_i2c_driver_test_ctx *ctx = test->priv;
        struct aspeed_i2c_fake_device *i2c_fake = ctx->i2c_fake;
        struct i2c_client *client = ctx->client;
        u8 msg[] = {0xae, 0x00};
        int i;

        i2c_fake->busy = true;
        KUNIT_ASSERT_EQ(test,
                        ARRAY_SIZE(msg),
                        i2c_master_send(client, msg, ARRAY_SIZE(msg)));
        KUNIT_EXPECT_FALSE(test, i2c_fake->busy);
        KUNIT_ASSERT_EQ(test, i2c_fake->msgs_count, 1);
        KUNIT_EXPECT_EQ(test, client->addr, i2c_fake->msgs->addr);
        KUNIT_EXPECT_EQ(test, i2c_fake->msgs->len, ARRAY_SIZE(msg));
        for (i = 0; i < ARRAY_SIZE(msg); i++)
            KUNIT_EXPECT_EQ(test, i2c_fake->msgs->buf[i], msg[i]);
}
```

Hardware Mocking - Limitations

- Depends on mocking interrupt and reset
    - Necessary for this driver, mocking other functions might be necessary for other drivers
    - Would it be better to do something specific to interrupt like logic_iomem?
    - What functions are too important to have a static stub?
- logic_iomem was not intended to be used this way
- A lot of gross platform driver helpers
- Device API and OF API heavily abused

Hardware Mocking - What do you think?

- Are we on the right track?
- Does anyone care?
- Should we fake hardware descriptions?
- Anyone familiar with roadtest?
    - Would that be better?
- Are platform drivers the right place to start?

The Challenges Faced in 2021/2022

Feature Gaps:

- QEMU support
- Modules support
- Mocking support
- Gotchas in the Resource system
- (K)TAP standardisation
- Android (and others) compiling KUnit (and disabling it) on production kernels.