Kselftests augmented:

Running kernelspace tests from userspace

Ricardo Cañuelo Navarro
rcn@igalia.com

Linux Plumbers Conference 2025

Main idea

Provide a core kernel feature and interface to extend the reach and capabilities of userspace-triggered tests.

Why?

- Partially bridge the gap between kernelspace and userspace tests.
    - There are other efforts with a similar goal. *
- Remove some of the restrictions of userspace tests.
- Enhance and add flexibility to kselftests.
- Encourage the standardization of test code in the kernel.

** LWN: Toward the unification of kselftests and KUnit (https://lwn.net/Articles/1029077)*

Basic interface example: ktest

- New kernel config option: **CONFIG_KTEST**.
    - Not enabled = no overhead.
- A ktest defines a function to test, a name for it and the size of the parameter data it needs.
- The ktest framework provides the basic functionalities:
    - Registering kernel functions as tests.
    - Listing the current registered tests.
    - Running them.

Basic interface example: ktest

sysfs-based userspace API: **`/sys/kernel/debug/ktest`**

```
# ls -l /sys/kernel/debug/ktest/
total 0
-r-------- 1 root root 0 Nov 27 15:15 ktests
-r-------- 1 root root 0 Nov 27 15:15 last_result
-r-------- 1 root root 0 Nov 27 15:15 last_test
--w------- 1 root root 0 Nov 28 10:48 run
```

Basic interface example: ktest

Listing available ktests returns their names and required argument sizes:

```
# cat /sys/kernel/debug/ktest/ktests
:mytest1,0
:mytest2,4
:mytest3,0
```

Basic interface example: ktest

Running a test:

```
# echo ':mytest1' > /sys/kernel/debug/ktest/run
[81260.677342] :mytest1 PASS
```

Checking that it finished and its result:

```
# cat /sys/kernel/debug/ktest/last_test
:mytest1
# cat /sys/kernel/debug/ktest/last_result
0
```

Defining, registering and running tests

Definition:

```
int mytest1(void *params)
{
     /* [Do test stuff ...] */

     return 0;
}

struct mytest2_params {
     int num;
} __packed;

int mytest2(void *params)
{
     struct mytest2_params *p = params;
     return p->num;
}
```

Defining, registering and running tests

Registration:

```
ktest_register(":mytest1", mytest1, 0);
ktest_register(":mytest2", mytest2, sizeof(struct mytest2_params));
```

Defining, registering and running tests

Running them as part of a kselftest:

```
TEST(mytest1)
{
	int ret;
	int psize;
	int result;

	if (!ktests_enabled()) {
		SKIP(return, "Ktests aren't enabled in the kernel");
	}
	ret = get_ktest(":mytest1", &psize);
	EXPECT_EQ(ret, 0);
	EXPECT_EQ(psize, 0);
	EXPECT_EQ(run_ktest(":mytest1", NULL, &result), 0);
	EXPECT_EQ(result, 0);
}
```

Helper functions (`get_ktest()`, `run_ktest()`, `ktest_done()`) would be part of the kselftests framework.

Defining, registering and running tests

Running them as part of a kselftest:

```
struct mytest2_params {
	int num;
} __packed;

TEST(mytest2)
{
	int ret;
	int psize;
	int result;
	struct mytest2_params p = {.num = 10};

	if (!ktests_enabled()) {
		SKIP(return, "Ktests aren't enabled in the kernel");
	}
	ret = get_ktest(":mytest2", &psize);
	EXPECT_EQ(ret, 0);
	EXPECT_EQ(psize, sizeof(p));
	EXPECT_EQ(run_ktest(":mytest2", &p, &result), 0);
	EXPECT_EQ(result, 10);
}
```

Defining, registering and running tests

Running them as part of a kselftest:

```
# ./ktest_test
TAP version 13
1..2
# Starting 2 tests from 1 test cases.
#  RUN           global.mytest1 ...
#            OK  global.mytest1
ok 1 global.mytest1
#  RUN           global.mytest2 ...
#            OK  global.mytest2
ok 2 global.mytest2
# PASSED: 2 / 2 tests passed.
# Totals: pass:2 fail:0 xfail:0 xpass:0 skip:0 error:0
```

Back to “why?”

Remove some of the restrictions of userspace tests:

Unpredictable conditions

When developing a subsystem, a driver or a syscall:

1) We can provide a function that acts as a test fixture that sets up certain conditions in the runtime environment.
2) Register the test fixture function as a ktest.
3) In our kselftest, we start by running the ktest to set up a particular starting point.

Back to “why?”

Remove some of the restrictions of userspace tests:

Enhance and add flexibility to kselftests

- No longer limited to interacting with the provided userspace interfaces.
- As always, there’s a danger of it becoming exploited and misused.

Back to “why?”

Encourage the adoption and standardization of test code in the kernel:

- Standard way to add and keep test code as part of drivers and core code in the kernel tree.
- Good practices:
  - Makes the test code more explicit and visible for developers.
  - Stronger obligation to keep kernel code and test code in sync.
- No need to keep ad-hoc out-of-tree test modules in the kselftests dirs.

Feedback and ideas welcome

- Do you find this makes sense?
- Anyone finds this useful?
- Implementation ideas?
- Pitfalls and drawbacks?

igalia

Join us!

https://www.igalia.com/jobs