Unit testing implementation in the Linux kernel: A participant-observation in AMD's display driver

Introduction

One of the goals of automated tests is to help ensure software quality and robustness, especially when many developers around the globe are involved and contributing, such as in a project as the **Linux kernel**. This work presents a perspective on one specific type of testing within the Linux kernel: **unit testing**. We explored **KUnit** and focused on the **AMD's display driver** - the Linux kernel's largest driver in lines of code and a subsystem where **unit tests had not been implemented yet**.

Unit Testing and KUnit

Unit testing is a form of software testing where **small units of code are tested**.

KUnit is a **unit testing framework** in the Linux kernel with a **unified structure** that allows different subsystems to use it. Only a Linux kernel repository with version 5.5 and up and its dependencies are needed to run it.

AMD's Display Driver

This driver can be divided into two pieces: Display Core (DC) and Display Manager (DM). We wrote unit tests for the DC component, in particular, the **Display Mode Library (DML)**, which deals with floating-point arithmetic.

Using KUnit to Write Tests

We relied on **equivalence partitioning** and **boundary-value analysis** techniques for devising test cases and also analyzed **past regressions** in the code to cover them. Figure 1 shows the tests for `abs_i64()` based on these methodologies and using KUnit's API.

```c
/**
* abs_i64_test - KUnit test for abs_i64
* @test: represents a running instance of a test.
*/
static void abs_i64_test(struct kunit *test)
{
  KUNIT_EXPECT_EQ(test, 0ULL, abs_i64(0LL));
  KUNIT_EXPECT_EQ(test, 1ULL, abs_i64(-1LL));

  /* Argument type limits */
  KUNIT_EXPECT_EQ(test, (uint64_t)MAX_I64, abs_i64(MAX_I64));
  KUNIT_EXPECT_EQ(test, (uint64_t)MAX_I64 + 1, abs_i64(MIN_I64));
}
```

Figure 1. Test cases written for `abs_i64()` using KUnit

```c
static uint64_t abs_i64(int64_t arg)
{
  if (arg >= 0)
    return (uint64_t)(arg);
  else
    return (uint64_t)(-arg);
}
```

Figure 2. `abs_i64()` definition

Test Coverage

By combining **KUnit and Gcov**, we generate test coverage reports. Figures 3, 4, 5, and 6 show part of the test coverage we achieved.

LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level - drivers/gpu/drm/amd/display/dc					
Test:	coverage.info		Lines:	37	691	5.4 %
Date:	2022-12-09 01:23:36		Functions:	1	59	1.7 %

Filename	Line Coverage		Functions	
dc_dmub_srv.c	10.3 %	37 / 358	3.8 %	1 / 26
dc_edid_parser.c	0.0 %	0 / 18	0.0 %	0 / 3
dc_helper.c	0.0 %	0 / 295	0.0 %	0 / 27
dc_link.h	0.0 %	0 / 14	0.0 %	0 / 2
dm_services.h	0.0 %	0 / 6	0.0 %	0 / 1

Generated by: LCOV version 1.14

Figure 3. Report generated with *Gcov* showing the test coverage from DMUB

Directory	Line Coverage		Functions	
drivers/gpu/drm/amd/display/dc/dml	5.3 %	83 / 1567	4.0 %	6 / 149
drivers/gpu/drm/amd/display/dc/dml/calcs	1.7 %	74 / 4278	15.9 %	7 / 44
drivers/gpu/drm/amd/display/dc/dml/dcn20	2.9 %	219 / 7554	9.3 %	7 / 75

Figure 4. Report generated with *Gcov* showing the test coverage from DML

Filename	Line Coverage		Functions	
conversion.c	0.0 %	0 / 34	0.0 %	0 / 3
conversion.h	0.0 %	0 / 1	-	0 / 0
dc_common.c	0.0 %	0 / 26	0.0 %	0 / 5
fixpt31_32.c	48.7 %	74 / 152	27.8 %	5 / 18
vector.c	0.0 %	0 / 109	0.0 %	0 / 16

Figure 5. Report generated with *Gcov* showing the test coverage from fixed_3132

Filename	Line Coverage		Functions	
bw_fixed.c	94.0 %	63 / 67	80.0 %	4 / 5
custom_float.c	0.0 %	0 / 69	0.0 %	0 / 3
dce_calcs.c	0.0 %	0 / 2155	0.0 %	0 / 6
dcn_calc_auto.c	0.0 %	0 / 1126	0.0 %	0 / 4
dcn_calc_math.c	21.6 %	11 / 51	23.1 %	3 / 13
dcn_calcs.c	0.0 %	0 / 810	0.0 %	0 / 13

Figure 6. Report generated with *Gcov* showing the test coverage from bw_fixed

Lessons Learned

In low-level systems, unit testing presents challenges as we deal with **code closer to the metal and potentially hardware-dependent**.

Testing static functions, although not encouraged by some software engineering practitioners, does have its advantages, especially when testing the public functions that use them is infeasible.

Running unit tests without the specific hardware is easier than we first thought it would be because the code we tested was mostly self-contained.

Device mocking was not necessary, a concern we had as we wanted the tests to be run without the specific hardware.

Magali Lemes do Sacramento <magalilemes@usp.br>
Institute of Mathematics and Statistics, University of São Paulo
Advisors: Paulo Meirelles (IME-USP) and Rodrigo Siqueira (AMD)