University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

# Unit testing implementation in the Linux kernel

## *A participant-observation in AMD's display driver*

Magali Lemes do Sacramento

Final Essay

mac 499 — Capstone Project

Supervisor: Paulo Meirelles
Co-supervisor: Rodrigo Siqueira

São Paulo
2022

# Resumo

Magali Lemes do Sacramento. **Implementação de testes de unidade no kernel Linux: *Uma observação participante do driver de display da AMD***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Testes de software são uma parte intrínseca do desenvolvimento de software. Testar código manualmente pode ser complicado e contraprodutivo, portanto, ferramentas automatizadas surgiram para realizar este trabalho mais facilmente. O kernel Linux não poderia ser diferente e, devido a seu tamanho e diferentes casos de uso, dispõe de ferramentas para testá-lo em uma variedade de cenários. Entre os diferentes tipos de testes de software no kernel, nós nos concentramos em testes de unidade, em que as unidades, as menores partes testáveis de código, são o alvo a ser testado. Este projeto apresenta uma perspectiva em relação à introdução de testes de unidade em um subsistema do kernel. Para atingir isso, seguimos uma abordagem de observação participante e exploramos o framework nativo de testes de unidade do Linux, o KUnit. Nosso objeto de estudo para a introdução dos testes foi o driver de display da AMD, o maior driver do Linux em linhas de código e também um subsistema onde testes de unidade ainda não haviam sido implementados. Como resultado, discutimos como elaboramos os casos de teste e as escolhas de projeto feitas ao longo do caminho, resumindo nossas lições e recomendações.

**Palavras-chave:** Testes de unidade. Linux. KUnit. Testes de software.

# Abstract

Magali Lemes do Sacramento. **Unit testing implementation in the Linux kernel: *A participant-observation in AMD's display driver***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Software testing is an intrinsic part of software development. Testing code manually can be cumbersome and counterproductive, so automated tools emerged to perform this job more easily. The Linux kernel could not be any different and, due to its size and different use cases, disposes of tools to test it against a variety of scenarios. Among the different types of software testing in the kernel, we focus on unit tests, where units, the smallest testable pieces of code, are the target to be tested. This project presents a perspective on introducing unit tests in a kernel subsystem. To achieve that, we followed a participant observation approach and explored Linux native unit testing framework, KUnit. Our object of study for introducing the tests was AMD display driver, Linux's largest driver in lines of code and also a subsystem where unit tests hadn't been implemented yet. As a result, we discuss how we devised test cases and the design choices made along the way, summarizing our lessons and recommendations.

**Keywords:** Unit tests. Linux. KUnit. Software testing.

# Contents

## Appendixes



## Annexes

# Chapter 1

# Introduction

Software testing is unquestionably a vital piece of software development. It allows catching regressions early on, avoiding the costs of finding bugs later, better ensures the overall software quality, and is one way to validate changes made to the code. The greater the variety of good tests in the software, the more robust it is.

Software testing becomes even more advantageous for projects with a massive codebase. In the case of the Linux kernel, a free software project with over 20,000,000 lines of code and lots of developers from around the globe contributing to it at a hectic pace, users could not be the only resource for testing, so having a set of different automated tools to test it is beneficial for the project.

Such a complex system naturally presents challenges when it comes to testing: how can the developer ensure that a proposed change works across the many different architectures, configurations, and hardware that Linux supports? Performing these manual checks can be at times inconvenient. Thus, developers can rely on the available testing tools in Linux or run their custom scripts. Still, when it comes to testing, the main challenge mentioned by some of the Linux kernel maintainers is the need for more hardware for it (Schmitt, 2022).

Among all the different types of software testing, we highlight unit tests, a way to test code by analyzing its internal logic, and, in most cases, hardware-independent. Its advantage lies in the fact that they are relatively faster to run compared to the other types of testing, providing quicker feedback when detecting possible errors. Due to its significance, the Linux kernel has a standardized framework for unit testing, named KUnit, used throughout this work to implement unit tests in the kernel.

KUnit was merged into the kernel mainline, the main tree maintained by Linus Torvalds, in 2019, making it a quite recent addition to the kernel. Its interface makes writing unit tests across different subsystems easy and uniform. The dependencies for running KUnit tests are the same as the Linux kernel, making the learning curve for using this tool relatively gentle. However, KUnit usage is still quite low in the kernel, with around 30% of maintainers stating that they never heard about it, according to a survey conducted by Schmitt, 2022.

In this work, we explored and understood how to implement unit tests within Linux

development context, focusing on unit tests using KUnit and its features, since we believe a greater adoption of this tool can be very favorable for the kernel. This work is closer to qualitative research, and the target of our observations is to write tests for the display driver developed by AMD (Advanced Micro Devices, Inc.). More specifically, we lean towards the ethnography framework since we directly dealt with and interacted with the community – namely, the Linux community – in which we were interested.

We deemed the AMD display driver subsystem a good candidate for introducing unit tests for two reasons. The first one is due to the mentorship we had: one of our mentors works at AMD, and the other two are external contributors to the same subsystem we wanted to explore; this resulted in us being granted an AMD Radeon™ RX 5700 XT 50th Anniversary Graphics video card, displayed in Annex A as Figure A.1 and now a property of the University of São Paulo, to explore and learn more about its internals. The second one is because the AMD display driver has an abundance of mathematical functions that we judged to be good targets for covering with unit tests.

In ethnographic research, the researcher gets immersed in the field they are studying, becoming its member and documenting what is learned along the way from this fresh point of view (Iacono *et al.*, 2009). It can be achieved through different means, such as participant observation and action research. In participant observation, the researcher learns the practices of the group being studied by engaging and interacting with it, also becoming a practitioner. In action research, the researcher is motivated by promoting change in the context of what is being studied. In contrast to participant observation, this approach makes a clear distinction between the roles of researcher and practitioner (Robey and Taylor, 2018).

Another methodology that comes close to describing how we conducted this work is the case study, where the researcher investigates specific events and the circumstances brought by the practitioners (Iacono *et al.*, 2009). Since the researcher is not necessarily directly involved in the experience being studied, we determine this as a distinction from the ethnographic approach we took. Our study mainly followed the participant-observation methodology by participating in two groups.

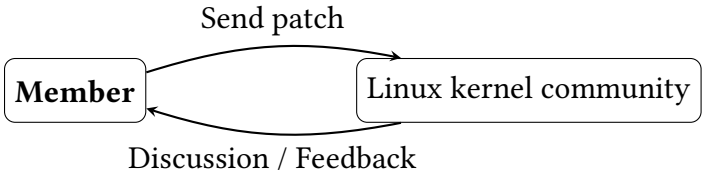



**Figure 1.1:** *Direct interaction with the Linux kernel community*

Making contributions to the Linux kernel is a challenging task. It requires looking for a change, finding the correct repository to work on and mailing lists to send the patch to, and setting up a development environment for dealing with all of this. As we intended to contribute to writing unit tests, we needed first to get used to this workflow and the community around it. Initially, we made general contributions to the AMD display driver subsystem and continued to do so throughout the project, interacting in a flow similar to

the one represented by Figure 1.1.

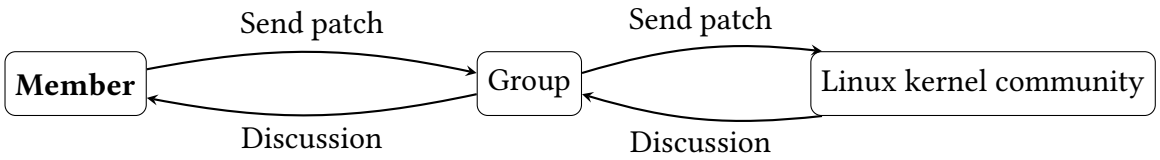



**Figure 1.2:** *Interaction with the Linux kernel community after internal discussions*

Specific to our goal of introducing unit tests to the AMD display driver, we were part of a group with contributors that took part in the Google Summer of Code program and whose projects[1,2,3] had the same target as this study. In this smaller group, we internally developed and discussed the infrastructure, design choices, and methodologies for the tests before sending them to the Linux community for more feedback. Figure 1.2 describes the flow of this interaction.

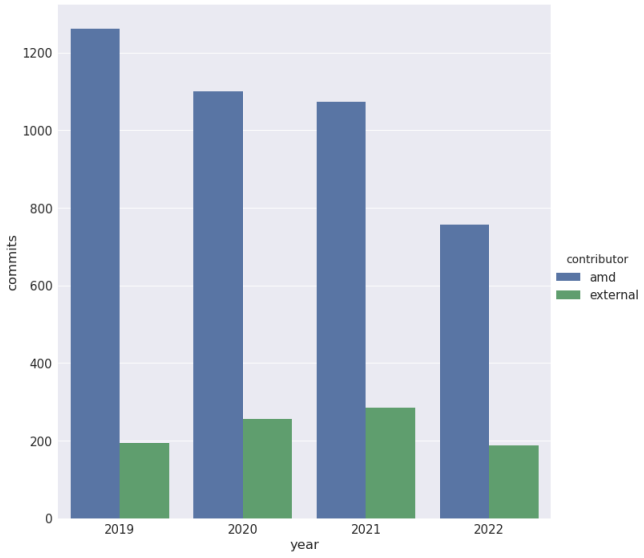



**Figure 1.3:** *Comparison between AMD and external contributors in the AMD Display Driver*

In summary, we can split the participant observation into two aspects. Firstly, we contributed and interacted with the AMD display driver and KUnit community, gathering their overall feedback about the tests we wrote and our choices. Moreover, we made other contributions not directly related to the scope of this work, but that served as a form of learning about Linux development workflow. The second aspect is from being part of a smaller community of external contributors to the AMD display driver, who were interested in introducing unit tests into the subsystem, discussing and gathering lessons throughout the process. Most of the contributions to the AMD display driver

[1] https://summerofcode.withgoogle.com/programs/2022/projects/fATmfPlL

[2] https://summerofcode.withgoogle.com/programs/2022/projects/6AoBcunH

[3] https://summerofcode.withgoogle.com/programs/2022/projects/JYeBJNnX

subsystem are made by developers employed by AMD, shown in Figure 1.3 obtained from Program A.4. Therefore, as external contributors, proposing new changes in this environment can present some challenges.

The remainder of this capstone project consists of four more chapters. Chapter 2 explains the context where this work lies, going over essential concepts related to the Linux kernel, software testing, in particular, unit testing, and how the AMD display driver, our target of study, is structured. Chapter 3 mainly describes the steps to write the unit tests and the design choices in adopting KUnit in a device driver. It also contains the lessons we learned while implementing the tests. Chapter 4 summarizes all the work and conclusions and presents probable future works based on what was accomplished. Chapter 5 covers the author's thoughts while working on the project. Moreover, to help the reader, throughout the manuscript, we adopted the following conventions: **bold** is used to highlight words; *italic* is used for file and directory names; `monospace` is used for parts of code and commands.

# Chapter 2

# Background

This chapter provides the necessary information for understanding what was accomplished in this work. We start with an introduction to the Linux kernel and its development process, followed by an overview of some of the tools used for testing it, then narrowing down to unit testing and how it is applied in the context of the Linux kernel. Finally, we head over to the AMD display driver, our object of study, to introduce unit tests.

## 2.1 Linux Kernel Development

The kernel is an essential piece of any operating system since it manages, at a low level, the resources of a computer. In 1991, Linus Torvalds started developing the Linux kernel as a hobby project based on MINIX. He published his work on a MINIX newsgroup (*The Beginning of the Linux Open-Source Operating System* 2022), asking for feedback, and, from that moment on, Linux thrived as a free software project. Nowadays, it powers smartphones, servers, desktops, and even embedded devices such as Ingenuity, the first aircraft on Mars Ackerman, 2022.

As for its organization, Linux is a monolithic kernel, meaning that the entire operating system works in kernel mode (Tanenbaum and Bos, 2014). At first glance, this approach can lead to thinking that adding and removing a new feature is complicated since all system procedures are linked into one single kernel image, which would require changing the whole kernel. To overcome this, Linux disposes of modules, which can be loaded and unloaded at runtime, making it easy to extend the operating system.

The Linux kernel development style has unique features to coordinate the various individuals and companies contributing to Linux. The Linux kernel is divided into different areas with specific components such as file systems, memory management, and device drivers. Within those areas, some subsystems specialize even further into more specific domains. For instance, about file systems, there are the *ext4* and *Btrfs* subsystems, and about GPUs drivers, there are the AMD Display Core, Intel DRM Drivers, etc.

Each kernel subsystem typically has its tree and development style. Changes made into the kernel usually go first through the subsystem to which they belong and are subsequently reviewed and merged by the subsystem maintainers. Mailing lists are usually

used for discussions and sending patches, although there are subsystems that do not necessarily rely on them and use other tools like GitHub. The *MAINTAINERS* file available at the kernel repository is the source for checking information about each subsystem.

```
956 AMD DISPLAY CORE
957 M:  Harry Wentland <harry.wentland@amd.com>
958 M:  Leo Li <sunpeng.li@amd.com>
959 M:  Rodrigo Siqueira <Rodrigo.Siqueira@amd.com>
960 L:  amd-gfx@lists.freedesktop.org
961 S:  Supported
962 T:  git https://gitlab.freedesktop.org/agd5f/linux.git
963 F:  drivers/gpu/drm/amd/display/
```

**Figure 2.1:** *Section example from the MAINTAINERS file*

An example extracted from the MAINTAINERS file is shown in Figure 2.1. The first line (AMD DISPLAY CORE) is the name of the subsystem. The other lines are the section entries, listing information such as the maintainers and their emails (M), relevant mailing list to which send patches and discussions (L), status (S), tree (T) where the subsystem work is based and files (F) that the subsystem cover. Essentially, any change made in files under *drivers/gpu/drm/amd/display/* should go to the listed maintainers and mailing list, and also be compatible with the current state of the indexed tree.

The mainline, the main kernel tree maintained by Linus, is where all of the work developed in the subsystems comes together. There are some not-so-strict rules in this development process: every two to three months, a kernel release is made; most of the new changes introduced in the subsystems are merged during the first two weeks of this cycle, called the merge window. After the merge window closes, the subsequent development weeks prioritize patches that fix problems. A new release candidate kernel is published roughly every new week until one of them is deemed to be good enough for the final release (*2. How the development process works* 2022).

## 2.2 Linux Kernel Testing Overview

In the book *The Art of Software Testing*, Myers *et al.*, 2012 characterizes testing as the "process of executing a program with the intention of finding errors". When bugs and regressions are discovered before hitting production, developers can solve them faster, offering a friendlier experience to the software end-user that will be less likely to deal with a malfunctioning feature, for instance. Test is a common way to validate code, mainly when new changes are introduced. Therefore, testing is undoubtedly an essential step in software engineering practices.

Different types of tests exist, from techniques to testing the behavior of one function exclusively up to testing an entire system. They each serve distinct purposes and, when combined, help assure the reliability of a system. Cohn, 2009, in his book "Succeeding with Agile", introduces the concept of a test pyramid to categorize the different types of software testing: (i) End-to-end tests are at the top of the pyramid and test the application as a whole from the perspective of a user; (ii) Integration tests, responsible for asserting

the integration among the system components, are in the middle of the pyramid; (iii) Unit tests, which test small units of the code, stand at the base of the pyramid and are the focus of this project.

In the 6.0 Linux kernel release, more than 2,000 developers contributed to more than 1,000,000 lines of code (Corbet, 2022). Large projects where many developers work on the same file set can be crowded and hectic, making regressions and bugs almost unavoidable. That is why the Linux kernel counts not only on its users for testing but also relies heavily on the aforementioned automated testing tools.

Most of the millions of lines in the Linux kernel source code come from the *drivers* directory, as illustrated in Figure 2.2. Therefore, it is worth analyzing how this part of the kernel is tested and which tools are used, as one of the goals of this work is to introduce one form of testing into a specific device driver. In this sense, Schmitt, 2022's work characterizes these testing tools, distinguishing between those described in academic works and those found in grey literature.

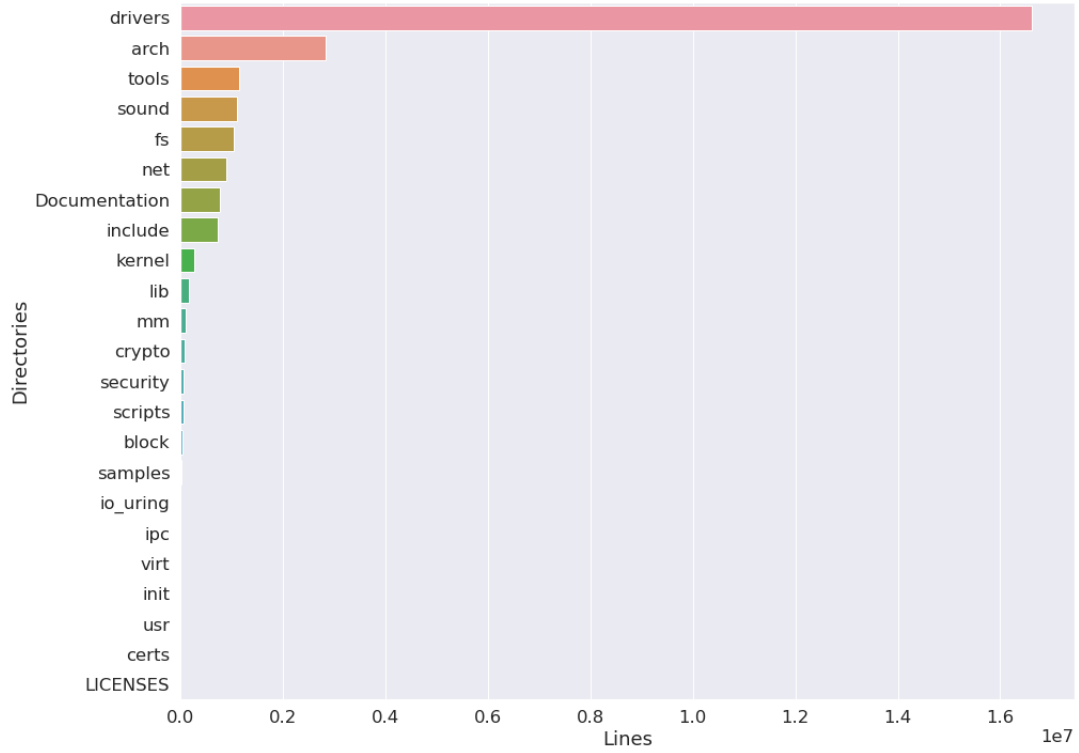



**Figure 2.2:** *Number of lines from the top-level directories of the 6.0 release kernel*

As for the mapped testing tools presented in academic papers, most of them present issues when setting up, installing, and using. Compared to the academic works, the number of testing tools described in the grey literature is much larger. It is also a more reliable and up-to-date source to understand and assess the tools that Linux developers currently use.

From the grey literature, there are reports from the different techniques currently employed to test Linux: this varies from tools that can be used locally by developers and others that report results after a patch is already at the sent stage.

Regarding the tools developers use to test Linux locally, we highlight Sparse[1] and Kselftest[2], the most commonly used tool by kernel maintainers, according to the survey conducted by Schmitt, 2022. **Sparse**, similar to what Smatch[3] and Coccinelle/coccicheck[4] do, is used for static analysis and can be easily run from inside the kernel repository by running `make` with the corresponding option set. **Kselftest** is a framework that can be used for writing many test types: unit, regression, stress, functional, and performance. It is run as a userspace process and is found at *tools/testing/selftests* in the kernel source code.

As for the tools that run integration tests and run them out of the developer local scope, we mention the 0-day test robot[5], KernelCI[6], and LKFT[7]. The **0-day test robot** makes use of some of the testing tools mentioned above, retrieves patches that were sent to mailing lists, applies them to the corresponding trees, compiles kernels with a variety of configurations, and reports back to the developer that sent the patches in case of any failure or warning. **KernelCI**, very similarly, also builds and runs tests against a diverse set of architectures, providing detailed information about the results. **Linux Kernel Functional Testing (LKFT)**, maintained by Linaro, tests and builds kernels with a particular focus on the ARM architecture. In summary, these platforms use some of the testing tools mentioned previously to ensure the quality of the Linux kernel and detect regressions as early as possible.

## 2.3 Unit Testing

Unit testing, sometimes called module testing, is a software testing technique that focuses on testing small code units. In particular, this type of testing concentrates on single functions or even on their smaller, testable parts. As this requires knowing the implementation – as in code – of the application to be tested, it falls under the white-box testing category.

JUnit, the first framework specialized in unit testing for the Java language, dates back to 1997 (Fowler, 2006). Since then, many other frameworks have emerged, such as pytest for Python and Jest for Javascript, offering a structure for concisely writing unit tests.

Because of their simple nature, unit tests are usually faster to run when compared to other types of testing, such as end-to-end testing. It also makes debugging more manageable, especially if the code to debug is covered by tests.

---

[1] https://docs.kernel.org/dev-tools/sparse.html

[2] https://docs.kernel.org/dev-tools/kselftest.html

[3] https://lwn.net/Articles/691882/

[4] https://docs.kernel.org/dev-tools/coccinelle.html

[5] https://01.org/lkp/documentation/0-day-brief-introduction

[6] https://foundation.kernelci.org/

[7] https://lkft.linaro.org/

### 2.3.1 Techniques for Building Test Cases

Several strategies help developers to think efficiently and write unit tests for their code. Myers *et al.*, 2012 recommend using, firstly, white-box strategies and, then, black-box strategies based on the code specification.

#### Black-Box Testing

- **Equivalence Partitioning**: takes advantage of the fact that within the infinite number of possible inputs and outputs of the software, some of those present the same behavior and can be grouped into equivalent classes. By using this technique, we can derive fewer test cases and remain confident that we are still exercising and testing the program enough.
- **Boundary-value Analysis**: is often applied alongside the equivalence partitioning method. This approach suggests analyzing the class range and testing the values on its border and those close to them. The motivation behind this is that more errors are expected in areas where the software changes its behavior.
- **Error Guessing**: suggests guessing and writing tests for special cases that a program may not have handled, such as, for a program that receives a list as input, an empty list can be provided as an input.

#### White-Box Testing

- **Statement Coverage**: concerns having every code statement hit at least once by testing. This is a fragile approach because one piece of code could easily cover all of the statements and still not cover many outputs.
- **Decision Coverage or Branch Coverage**: describes that the true or false outcome of every conditional and loop should be covered by testing.
- **Multiple-condition Coverage**: tests all the combinations of decisions within the program, expanding, even more, the strategy applied by decision coverage.

### 2.3.2 Test-Driven Development (TDD)

In a test-driven development procedure, developers first build the tests, and since there is no functioning code, they fail. Only after the tests are written the code, whose tests were just written, is developed so that the tests can pass. This is done in a cycle, as illustrated in Figure 2.3 until the code is deemed good enough, contrary to the established development workflow where features are first developed and tested.

In these circumstances, unit tests are the type of tests that most fit TDD purposes since they can easily be written before the feature is ready.

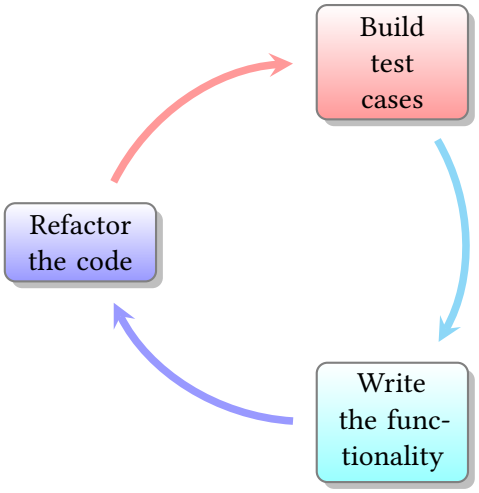



**Figure 2.3:** *TDD cycle*

## 2.4 KUnit

Unit tests existed in the kernel before KUnit, but for those subsystems that used it, each had its way of handling and developing them. KUnit emerged as an alternative to have a unified structure for unit tests within the kernel. Google engineers developed it, the first RFC was sent on October 2018[8], and it was merged into the mainline in 2019, arriving in Linux v5.5.

KUnit tests can be wrapped in a module or be built-in: formerly, the tests are run when the module is loaded, and in the latter, they are run during boot time. When it started, KUnit enabled the unit tests to be run only using the User-Mode Linux (UML) architecture, which allows running a kernel instance as a standalone user-space process, meaning that there is no need to set up, build and install a kernel on a target machine. This option still exists today, but there is also support for running KUnit in other instances.

### 2.4.1 Run KUnit

To run KUnit tests, a Linux kernel repository, newer than version 5.5, is needed. The dependencies for running KUnit are the same as the Linux kernel. The most straightforward way to run KUnit is through the Python script, found under `tools/testing/kunit/kunit.py` in the kernel repository. This script can configure, build and run a kernel with the selected KUnit tests on UML or a virtual machine by using QEMU[9], adding the possibility of running the tests in different architectures easily.

The selection of the tests to run is made through a configuration file, used to tell the kernel which modules and how they should be built. It is a common practice for subsystems that use KUnit to provide a *.kunitconfig* file, defining the tests to be built and the dependencies they may have. Then, when running a set of tests from a specific subsystem is desired, we can pass that configuration file as an argument for the *kunit.py*

[8] https://lore.kernel.org/lkml/20181016235120.138227-1-brendanhiggins@google.com/

[9] Machine emulator and virtualizer

```
┌─[±][tag:v6.0 ✓][linux][]
└─▪ ./tools/testing/kunit/kunit.py run --kunitconfig=drivers/gpu/drm/tests/.kunitconfig
[17:46:25] Configuring KUnit Kernel ...
[17:46:25] Building KUnit Kernel ...
Populating config with:
$ make ARCH=um O=.kunit olddefconfig
Building with:
$ make ARCH=um O=.kunit --jobs=8
[17:46:31] Starting KUnit Kernel (1/1)...
[17:46:31] ============================================================
[17:46:31] ============ drm_format_helper_test (1 subtest) ============
[17:46:31] ================= xrgb8888_to_rgb332_test ==================
[17:46:31] [PASSED] single_pixel_source_buffer
[17:46:31] [PASSED] single_pixel_clip_rectangle
[17:46:31] [PASSED] well_known_colors
[17:46:31] [PASSED] destination_pitch
[17:46:31] ============= [PASSED] xrgb8888_to_rgb332_test =============
[17:46:31] ============= [PASSED] drm_format_helper_test ==============
[17:46:31] ============================================================
[17:46:31] Testing complete. Ran 4 tests: passed: 4
[17:46:31] Elapsed time: 6.280s total, 0.002s configuring, 6.161s building, 0.098s running
```

**Figure 2.4:** *Result output from running KUnit tests from the DRM subsystem*

script to execute, as shown in Figure 2.4.

KUnit tests can also be run on real hardware. To do that, one must select which tests to run through the system configuration file and build and install the kernel onto the system. If the tests are in a module, running `modprobe test_module`, where `test_module` is the name of the module containing the tests, will load the tests and display the results in the kernel buffer. If the tests are built-in, they automatically run during boot and, similarly, are also available in the kernel buffer, which can be accessed by running `dmesg`.

### 2.4.2 Building Tests

The most basic idea in unit testing is making **assertions**, that is, comparing whether an output is behaving accordingly to what is expected from it. For making assertions, KUnit disposes of two classes of macros: `KUNIT_EXPECT_*` and `KUNIT_ASSERT_*`. Assertions are different from expectations: when assertions are not met, the test function stops running completely, not allowing the other tests to be run. Some macros such as `KUNIT_EXPECT_FALSE`, `KUNIT_EXPECT_TRUE` and `KUNIT_EXPECT_NULL` check the state of only one value; others such as `KUNIT_EXPECT_EQ`, `KUNIT_EXPECT_NE`, `KUNIT_EXPECT_STREQ` compare two values against each other.

In KUnit, functions of signature `void function_test(struct kunit *test)` define one **test case**, containing assertions and/or expectations. `function_test` is the name of the function containing the assertions, and `struct kunit` is a structure that stores a test context, with information such as the data needed for the test and the test status. In Figure 2.5, a simple example for building tests for two functions using the `KUNIT_EXPECT_*` macros is displayed.

Each test case needs to be passed to the `KUNIT_CASE` macro to set a `struct kunit_case` for that function. The `struct kunit_case` is a representation of the test function that stores additional information such as the test status and log. Related functions

```c
void prime_test(struct kunit *test)
{
    KUNIT_EXPECT_FALSE(test, is_prime(0));
    KUNIT_EXPECT_FALSE(test, is_prime(1));
    KUNIT_EXPECT_TRUE(test, is_prime(2));
    KUNIT_EXPECT_FALSE(test, is_prime(24));
}

void factorial_test(struct kunit *test)
{
    KUNIT_EXPECT_EQ(test, 1, factorial(0));
    KUNIT_EXPECT_EQ(test, 1, factorial(1));
    KUNIT_EXPECT_EQ(test, 2, factorial(2));
    KUNIT_EXPECT_EQ(test, 120, factorial(5));
    KUNIT_EXPECT_EQ(test, 3628800, factorial(10));
}
```

**Figure 2.5:** *KUnit syntax for building two test cases*

are grouped in an array of `struct kunit_case`, as displayed in Figure 2.6, required to end with `NULL`.

```c
static struct kunit_case math_test_cases[] = {
    KUNIT_CASE(prime_test),
    KUNIT_CASE(factorial_test),
    {}
};
```

**Figure 2.6:** *Test cases should be grouped into an array*

A **test suite**, defined as a `struct kunit_suite`, is then built with the array containing the test cases. We can optionally also specify functions to be run before and after each test case by assigning a function to `.init` and `.exit`, respectively. Functions for setting up and tearing down the test environment before and after the test suite is run can also be set in the `suite_init` and `suite_exit` members. Finally, for the tests in a suite to be recognized and run by KUnit, they must be passed to the `kunit_test_suite` macro. Figure 2.7 shows the declaration of a KUnit test suite built on top of the previous examples.

```c
static struct kunit_suite math_test_suite = {
    .name = "math",
    .init = math_test_init,
    .exit = math_test_exit,
    .suite_init = math_suite_init,
    .suite_exit = math_suite_exit,
    .test_cases = math_test_cases,
};
kunit_test_suite(math_test_suite);
```

**Figure 2.7:** *KUnit suite declaration*

Depending on how the test cases were built and grouped, there may be a need for more

than one test suite in one file. This can be achieved by simply replacing `kunit_test_suite` with `kunit_test_suites` and passing all the `struct kunit_suite` as arguments.

KUnit also supports parameterized testing, providing a simple structure for running the same set of tests for different values. Instead of having `KUNIT_CASE`, this approach uses `KUNIT_CASE_PARAM`, which takes a function besides the test case that iterates over the different cases as a second argument.

Testing every single function of a file through KUnit is not always possible in a uniform way: there are different approaches depending on whether a function is static. Currently, the only way to test static functions is by including the `.c` test file into the source file, using the `#include` directive. If we want to have a module only for tests, the functions to be tested have to be exported so that the test module can access them. If there is no such requirement and no static function to be tested, then the tests can be a part of the module with the functions to test.

All of the macros, functions, and structs needed for building tests and mentioned previously are available in the *kunit/test.h* header. Therefore, every file containing KUnit tests has to include it.

### 2.4.3 KUnit Usage in the Kernel

KUnit adoption in the Linux kernel subsystems is still at a very initial stage. Using the Linux kernel version 6.0 repository as a starting point, we built a set of scripts to discover which subsystems use KUnit. The core idea to determine that was by looking for all files that include the *kunit/test.h* header and, from there, obtain the subsystem to which that file belongs.

In the Linux kernel version 6.0, we found 35 subsystems that adopted KUnit, obtained through the script described in Program A.1. Apart from the specific subsystems, KUnit is also used to test some general utility libraries, such as hash routines and sorting, that do not belong to one specific subsystem and are listed under “THE REST”. From the list, we highlight the tests developed for DRM, the parent subsystem of the AMD display driver.

- APPARMOR SECURITY MODULE
- ASPEED SD/MMC DRIVER
- BITMAP API
- CHROME HARDWARE PLATFORM SUPPORT
- COMMON CLK FRAMEWORK
- DATA ACCESS MONITOR
- DRIVER CORE, KOBJECTS, DEBUGFS AND SYSFS
- DRM DRIVERS
- EXT4 FILE SYSTEM
- FILESYSTEMS (VFS and infrastructure)
- GENERIC INCLUDE/ASM HEADER FILES
- HIBERNATION (aka Software Suspend, aka swsusp)
- HID CORE LAYER
- IIO SUBSYSTEM AND DRIVERS
- KASAN

- KCSAN
- KERNEL UNIT TESTING FRAMEWORK (KUnit)
- KFENCE
- KPROBES
- LANDLOCK SECURITY MODULE
- LINEAR RANGES HELPERS
- LIST KUNIT TEST
- MANAGEMENT COMPONENT TRANSPORT PROTOCOL (MCTP)
- NETWORKING [GENERAL]
- NETWORKING [MPTCP]
- NITRO ENCLAVES (NE)
- PROC SYSCTL
- REAL TIME CLOCK (RTC) SUBSYSTEM
- S390
- SLAB ALLOCATOR
- SOUND - SOC LAYER / DYNAMIC AUDIO POWER MANAGEMENT (ASoC)
- THE REST
- THUNDERBOLT DRIVER
- TIMEKEEPING, CLOCKSOURCE CORE, NTP, ALARMTIMER
- VFAT/FAT/MSDOS FILESYSTEM

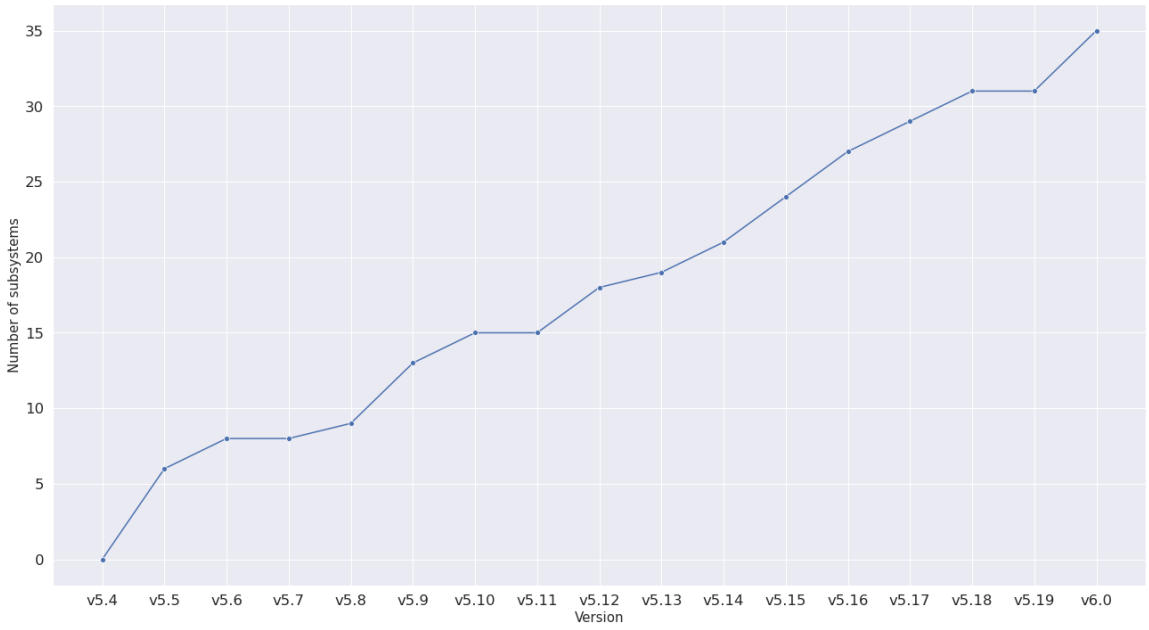



**Figure 2.8:** *Amount of subsystems that adopted KUnit since its release*

From the script in Program A.3, we obtained a plot, shown in figure 2.8, showing the evolution in the number of subsystems that have used KUnit since its release in v5.5.

## 2.5 AMD Display Driver

A Graphics Processing Unit (GPU) is a specific-purpose processor with many small cores and a dedicated memory called VRAM – which stands for Video RAM. GPUs are

mainly used in graphics, dealing with many simple calculations, such as for rendering geometric shapes. As GPUs have many cores, they can execute those many parallel simple operations more efficiently when compared to other types of processors. The display driver performs the job of reading the GPU frames in the VRAM up until displaying it on a screen or display device.

The AMD display driver in the Linux kernel is a part of the AMDGPU module and is found under the *drivers/gpu/drm/amd/display* directory. AMDGPU belongs to the Direct Rendering Manager (DRM) subsystem, which is responsible for handling graphics devices in Linux. The AMD display driver has two main components(*drm/amd/display - Display Core (DC)* 2022):

- **Display Core (DC)**: handles hardware programming and resource management; the implementations do not depend on the operating system it is on.
- **Display Manager (DM)**: implements the AMDGPU base driver and the DRM API, making it dependent on Linux.

The driver supports two architectures: Display Core Engine (DCE) and its successor, Display Core Next (DCN). Figure 2.9 shows the files and directories, in yellow and blue respectively, that make up the AMD display driver, with the DM component living under the *amdgpu_dm* folder and DC in the *dc* one.

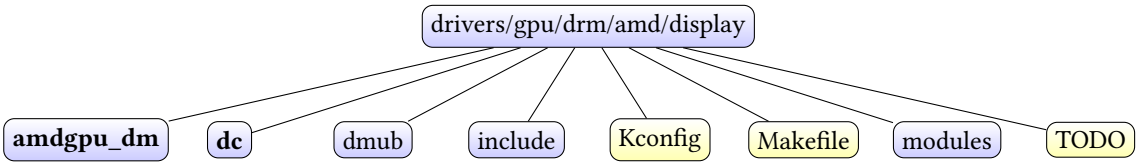



**Figure 2.9:** *AMD display driver directory structure at kernel v6.0*

In the DCE architecture, fixed-point arithmetic is used to calculate mode setting[10] parameters. The specific code for this architecture is spread across the *dc* directory, with a *dce* folder centralizing the common logic for the different versions of the architectures and specific folders for each architecture version released: *dce60* for the DCE 6 and *dce120* for the DCE 12, for instance.

Alternately, the code for the DCN architecture uses floating-point operations handled in the Floating-Point Unit (FPU). Floating-point usage in the kernel can be unsafe, mainly due to it allowing a change of the floating-point state of the CPU. Because of this, the DCN code that uses FPU is isolated in the Display Mode Library (DML) directory, found at *dc*. Thus, it allows for centralized control of the functions that access the FPU. Similar to the DCE case, the code for each DCN version is under *dc – dcn20* corresponds to the DCN 2.0 architecture, as an example. There is the particularity of having directories for each architecture at the *dml* folder.

[10] Operation that sets a display resolution, depth, and frequency.

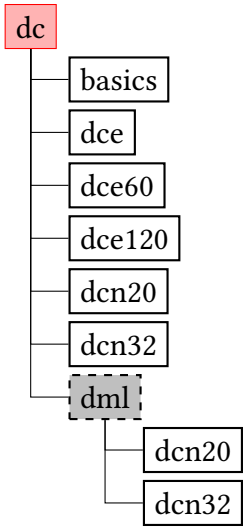
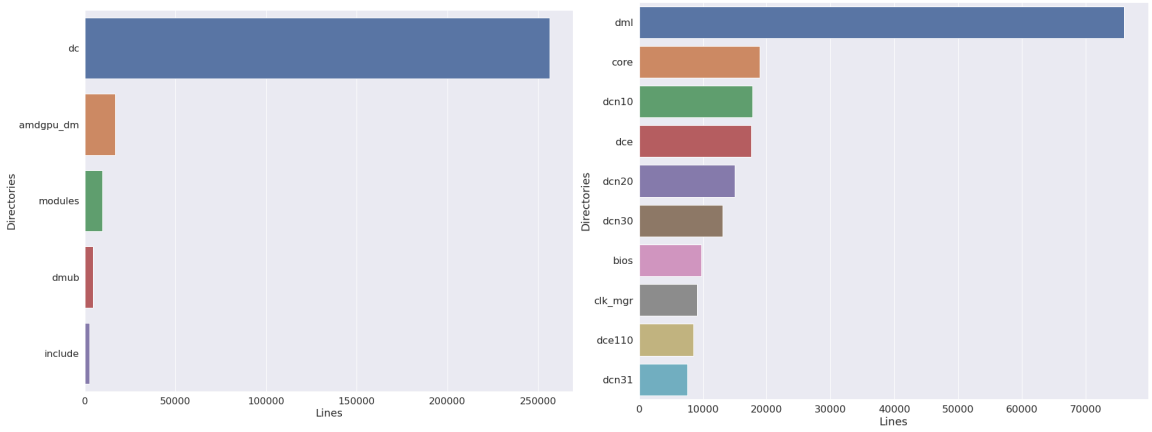



**Figure 2.10:** *DC simplified directory structure*

Another important directory under *dc* is *basics*, responsible for providing a set of utility functions and structures, such as for handling fixed-point operations and vectors. Both DCE and DCN rely on this module.




**(a)** *Directories from the display folder by number of lines*

**(b)** *Top 10 directories in dc in number of lines*

**Figure 2.11**

The Display Core directory has many other files and directories, such as the graphics card BIOS and other architecture-specific files, that we will not cover as they are only partially related to the scope of this work. Figure 2.10 shows a very simplified representation of the DC directory structure, highlighting the *dml* folder, where we wrote most of the unit tests.

To get an idea of the size of the dc directory, we created some scripts, described in A.2, to obtain the number of lines of code for the display folders and move one more level of hierarchy for the dc component. Figure 2.11 depicts the size of the DC component in the number of lines.

# Chapter 3

# Results and Discussion

This chapter covers the contributions made to the Linux kernel: contributions related to introducing unit tests to the AMD display driver and other more general contributions not wholly linked to fulfilling this goal. We also explain how test cases were devised and discuss design choices related to KUnit's adoption into AMDGPU.

## 3.1 General Contributions

As part of the process of getting used to the Linux kernel development workflow, especially concerning the AMD display driver subsystem, we made a few first contributions not directly related to adding unit tests. These were primarily the correction of compilation warnings and the removal of duplicated and unused code. Those contributions are listed in Table 3.1, where the numbers in green represent the number of lines added, and those in red the number of lines removed.

| Age | Commit Message | Files | Lines |
|---|---|---|---|
| 2022-02-02 | drm/amd/display: Use NULL pointer instead of plain integer | 1 | -1 / 1 |
| 2022-02-24 | drm/amd/display: Adjust functions documentation | 1 | -3 / 3 |
| 2022-02-24 | drm/amd/display: Add conditional around function | 1 | -1 / 3 |
| 2022-02-24 | drm/amd/display: Use NULL instead of 0 | 3 | -1 / 4 |
| 2022-02-24 | drm/amd/display: Turn functions into static | 3 | -18 / 3 |
| 2022-05-04 | powerpc: Fix missing declaration of [en/dis]able_kernel_altivec() | 1 | -0 / 9 |
| 2022-08-10 | drm/amd/display: remove DML Makefile duplicate lines | 1 | -2 / 0 |
| 2022-08-10 | drm/amd/display: make variables static | 4 | -6 / 3 |
| 2022-08-10 | drm/amd/display: remove header from source file | 2 | -2 / 2 |
| 2022-08-10 | drm/amd/display: include missing headers | 3 | -0 / 5 |
| 2022-08-22 | drm/amd/display: drm/amd/display: remove unused header | 1 | -34 / 0 |

**Figure 3.1:** *Table listing contributions made to the Linux kernel*

## 3.2 Implementing Unit Tests

Inside the AMD display driver directory structure, we mainly focused on writing unit tests for the Display Mode Library (DML), part of the Display Core (DC) component, due to its mathematical nature. This library is responsible for dealing with "clock, watermark, and bandwidth calculations for DCN"[1]. Apart from DML, we also built tests for the fixed31_32 and DMUB libraries, both part of the Display Core component as well. For thinking and devising test cases, we mostly relied on inspecting **code coverage**, **equivalence partitioning**, and **boundary-value analysis** techniques, described in Section 2.3.1. We also used past regressions in the code we evaluated as references to build test cases.

### 3.2.1 Using Test Design Techniques

The first file we explored was *bw_fixed.c*, which can be found at *drivers/gpu/drm/amd/display/dc/dml/calcs*. It mainly contains basic mathematical functions, such as calculating the absolute value and the ceiling of given numbers and operations that deal with fixed-point arithmetic. The following two subsections discuss how we developed test cases for two functions from this file using equivalence partitioning and boundary-value analysis.

#### Function `abs_i64()`

The first function we explored was `abs_i64`, defined as shown in Figure 3.2. This function returns an `uint64_t` with the absolute value of the parameter it receives.

```
1 static uint64_t abs_i64(int64_t arg)
2 {
3   if (arg >= 0)
4     return (uint64_t)(arg);
5   else
6     return (uint64_t)(-arg);
7 }
```

**Figure 3.2:** *abs_i64() definition*

We started off by analyzing the argument `arg`: an `int64_t`, alias for `signed long long`, which can range from $-(2^{63})$ to $2^{63} - 1$. As a first approach, we wanted to test how the function behaves when it deals with these border cases. Since any other values outside the range mentioned above can lead to an integer overflow and the function does not handle these cases, we did not test them.

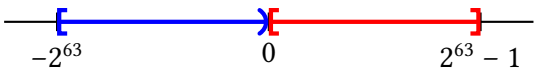



**Figure 3.3:** *Equivalence partitions for abs_i64()*

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=061bfa06a4

By looking at the `if`-statements, we could determine two equivalence classes: one for values greater than or equal to 0 and another for values less than 0. In the first scenario, the function does not make any change to the input, and its return value is the input integer, as it is, cast to `uint64_t`; in the second scenario, the function takes the input and returns its opposite, also cast to `uint64_t`. As the function changes its behavior around 0, we set it as one boundary. We also have the boundaries defined based on the argument type, leaving us with the two categories depicted in Figure 3.3.

Finally, we wanted to test the values on the border and the values close to it so that more than one representative of the partition class would get tested. We ended up with the following cases for `arg`:

- $-2^{63}$, the lower bound of `int64_t`
- 0, value where the function changes its behavior
- 1, value close to one equivalence class boundary
- $2^{63} - 1$, the upper bound of `int64_t`

Figure 3.4 shows how we can build a test for the values above using KUnit features.

```
/**
* 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 3.4:** *KUnit tests cases for abs_i64()*

### Function `bw_floor2()`

This function takes two arguments of type `bw_fixed` (whose structure definition is found in Figure 3.5): the first one, `arg`, is the value that will be rounded down; the second one, `significance`, is the multiple to which `arg` will be rounded. The function definition is shown in Figure 3.6.

```
struct bw_fixed {
  int64_t value;
};
```

**Figure 3.5:** *bw_fixed definition*

```
1  struct bw_fixed bw_floor2(
2    const struct bw_fixed arg,
3    const struct bw_fixed significance)
4  {
5    struct bw_fixed result;
6    int64_t multiplicand;
7
8    multiplicand = div64_s64(arg.value, abs_i64(significance.value));
9    result.value = abs_i64(significance.value) * multiplicand;
10   ASSERT(abs_i64(result.value) <= abs_i64(arg.value));
11   return result;
12 }
```

**Figure 3.6:** *bw_floor2() definition*

The first step we followed to build test cases for this function was defining the partition classes. To do so, we began by analyzing the parameters: both are of type `struct bw_fixed`, a structure consisting of only one member, `value`, of type `int64_t`.

| arg.value | significance.value | result.value |
|---|---|---|
| - | - | - |
| 0 | + | 0 |
| 0 | - | 0 |
| + | + | + |
| + | - | + |

**Figure 3.7:** *Equivalence partitions for `bw_floor2()`*

`value` could be either positive, negative, or zero. Taking into account the function's context, we noticed that, as per lines 8 and 9 of the function definition (Figure 3.6), the sign of `significance`'s `value` variable does not make a difference, since the function only uses its absolute value. On the other hand, the sign of `arg` `value` does interfere with the function output. From here, we defined three equivalence classes, shown in Figure 3.7: one that outputs positive values, another that outputs negative ones, and another that outputs zero. We exercised the border values on the range of the arguments in a similar manner as done for the function `bw_floor2()` described above in 3.2.1.

Furthermore, we added more than one test case for some equivalence classes in a way to document and explain to developers, who might work with this function, how it works. Figure 3.8 shows the final result with all the tests we built for `bw_floor2()`.

```c
/**
* bw_floor2_test - KUnit test for bw_floor2
* @test: represents a running instance of a test.
*/
static void bw_floor2_test(struct kunit *test)
{
  struct bw_fixed arg;
  struct bw_fixed significance;
  struct bw_fixed res;

  /* Round 10 down to the nearest multiple of 3 */
  arg.value = 10;
  significance.value = 3;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, 9, res.value);

  /* Round 10 down to the nearest multiple of 5 */
  arg.value = 10;
  significance.value = 5;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, 10, res.value);

  /* Round 100 down to the nearest multiple of 7 */
  arg.value = 100;
  significance.value = 7;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, 98, res.value);

  /* Round an integer down to its nearest multiple should return itself */
  arg.value = MAX_I64;
  significance.value = MAX_I64;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, MAX_I64, res.value);

  arg.value = MIN_I64;
  significance.value = MIN_I64;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, MIN_I64, res.value);

  /* Value is a multiple of significance, result should be value */
  arg.value = MAX_I64;
  significance.value = MIN_I64 + 1;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, MAX_I64, res.value);

  /* Round 0 down to the nearest multiple of any number should return 0 */
  arg.value = 0;
  significance.value = MAX_I64;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, 0, res.value);

  arg.value = 0;
  significance.value = MIN_I64;
  res = bw_floor2(arg, significance);
  KUNIT_EXPECT_EQ(test, 0, res.value);
}
```

**Figure 3.8:** *bw_floor2() test case*

Lessons and Recommendations

After sending out the *bw_fixed.c* tests for review, an integer overflow problem in the code we were testing was detected and reported[a]. Subsequently, a patch[b] fixing the issue was sent, and merged[c].
All things considered, we summarize these lessons:

- Following **techniques for designing test cases** is quite efficient, especially when not knowing, at first, how to select values for testing. In our case, adopting those techniques proved extremely helpful for the small functions we wrote tests for.
- For the test cases above, we are essentially using the same test and only changing the function input values. This may be a good call for **parameterized testing**, as suggested in the mailing list discussion[d].
- Tests can also serve as an alternative way to **document** the behavior of a function.

[a] https://lists.freedesktop.org/archives/amd-gfx/2022-August/082649.html

[b] https://lists.freedesktop.org/archives/amd-gfx/2022-August/082746.html

[c] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6ae0632d17

[d] https://lists.freedesktop.org/archives/amd-gfx/2022-August/082828.html

### 3.2.2 Covering Regressions

A regression happens when the software or part of it starts behaving unintendedly or stops working after adding a new change. For both functions described below, we searched for events of regressions in the code we intended to test, so we could build test cases that cover those specific situations.

#### Function `dcn21_update_bw_bounding_box()`

After building tests for all functions from *bw_fixed.c*, we moved on to build test cases for different DML files from the DCN 2.0 architecture, which corresponds to the architecture from the GPUs we had at our disposal, at *drivers/gpu/drm/amd/display/dc/dml/dcn20*.

We built tests for this function based on a real-life event, using a Test-Driven Development (TDD) approach to show how unit tests can be integrated into the kernel development process.

A TDD approach can also be adapted when finding a bug or regression in the code. The flowchart in Figure 3.9 illustrates one way to do this, also summarized in the list of steps below.

- After finding a regression, the developer has to investigate and identify why and how it is happening.
- The developer, then, writes one or more test cases that will naturally fail due to the regression but would pass in a regression-free code.

- The regression is then fixed.
- The tests should be run and passed.

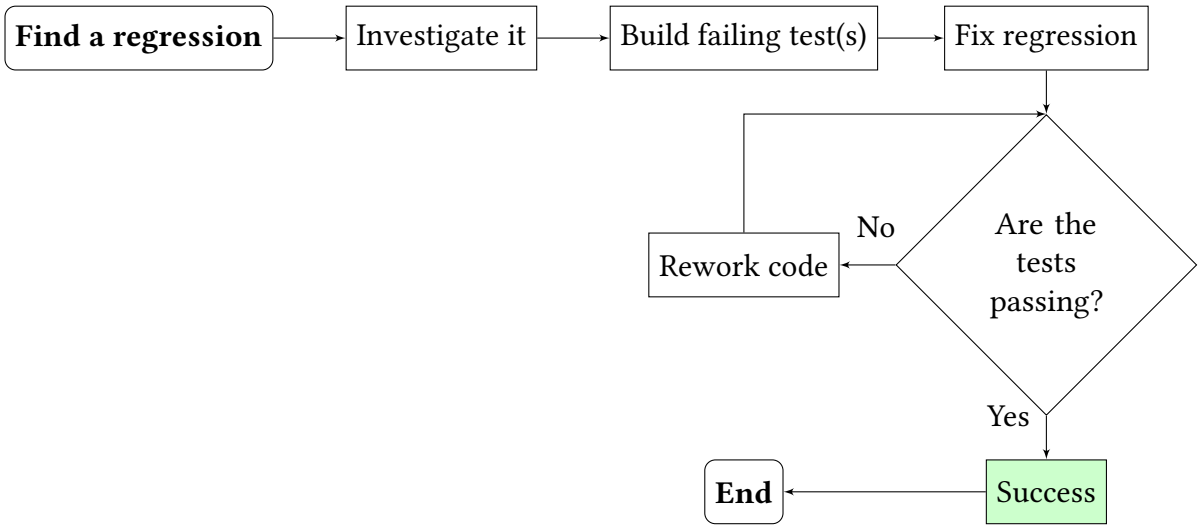



**Figure 3.9:** *Diagram showing how to apply TDD to fix regressions*

To apply this technique in the AMD display driver subsystem, we first identified a regression in the `dcn21_update_bw_bounding_box()` function. At one point, this function exceeded the defined stack size for it, which caused a compilation warning. A first attempt[2] to fix this warning was made, and it fixed the problem. This solution, though, did not preserve the function behavior – that is, the fix caused the function output to be different from what it was before the fix –, causing a regression. A new patch[3] was then sent to fix both the stack size warning as well as the previously introduced regression. We can summarize this set of events into three parts:

1. **Problem**: The stack size of the function triggered a compilation warning.
2. **Regression**: A patch fixed the problem, but it changed the function behavior.
3. **Fix**: A patch was sent to fix the regression and the initial problem.

We decided to use this occurrence as a background for devising a test case for the `dcn21_update_bw_bounding_box()` function. To do so, we first investigated the root cause of the introduced regression. After that, we built a test case that would fail due to the circumstances brought by the tough patch but otherwise would pass if we just reverted it. Then, we applied the patch that fixed the regression and reran the tests to check if the problem was fixed by asserting if the tests had passed. This was an attempt to show by example how developers, be they from the Linux kernel or not, can use tests to fix bugs and regressions in a sophisticated manner.

Another important thing to highlight is that this function has large structures as arguments, so we would stumble upon a stack-size warning when writing the test function in the traditional KUnit manner. To solve that, we decided to take advantage of the

[2] https://lists.freedesktop.org/archives/amd-gfx/2022-June/080004.html

[3] https://lists.freedesktop.org/archives/amd-gfx/2022-June/080214.html

parameterized testing feature provided by KUnit, which, instead of using heap memory, allocates the necessary memory dynamically, avoiding subsequent stack size warnings and also making the process of adding new test cases for the function easier.

#### Function `populate_subvp_cmd_drr_info()`

We also found a regression in the `populate_subvp_cmd_drr_info()` for which we wrote a test case, similarly to the approach applied to devise the test case for the `dcn21_update_bw_bounding_box()` function. This function is found under *drivers/gpu/drm/amd/display/dc/dc_dmub_srv.c*.

In this case, a patch[4] adding a new feature ended up introducing compilation warnings for 32-bit architectures due to the floating-point operations in the code. A new patch[5] was then sent to fix the previous problem, but it turned out to change the function behavior by zeroing some values from a structure it should not have. The summary of events is listed as follows:

1. **Problem**: Floating-point operations caused 32-bit compilation errors.
2. **Regression**: A patch attempted to fix introduced the problem but zeroed all values in the structure the function is supposed to populate.

To build the test case, we studied the expected behavior of the function before the regression was introduced. After that, we devised cases using the values the function produced when reverting the commit that brought the regression, so all the tests would pass. We also applied parameterized testing.

Lessons and Recommendations

- Tests can also serve for **documenting** a function past of regressions.
- One interesting source for building test cases is **covering past regressions** in the code.
- For eventual regressions that may be found in the code, we strongly recommend **writing unit tests** that cover them since the developer who will fix it understands and can think of test cases where that piece of code is not working.

### 3.2.3 Test Coverage

**Test coverage**, or code coverage, is one way to estimate how much source code is covered by tests, measuring the amount of source code that was hit when the tests that cover it are run. Below, in Figure 3.10 we present the folders and files – in purple and yellow, respectively – covered by tests.

[4] https://lore.kernel.org/amd-gfx/20220630191322.909650-3-Rodrigo.Siqueira@amd.com/

[5] https://lore.kernel.org/amd-gfx/20220708052650.1029150-1-alexander.deucher@amd.com/

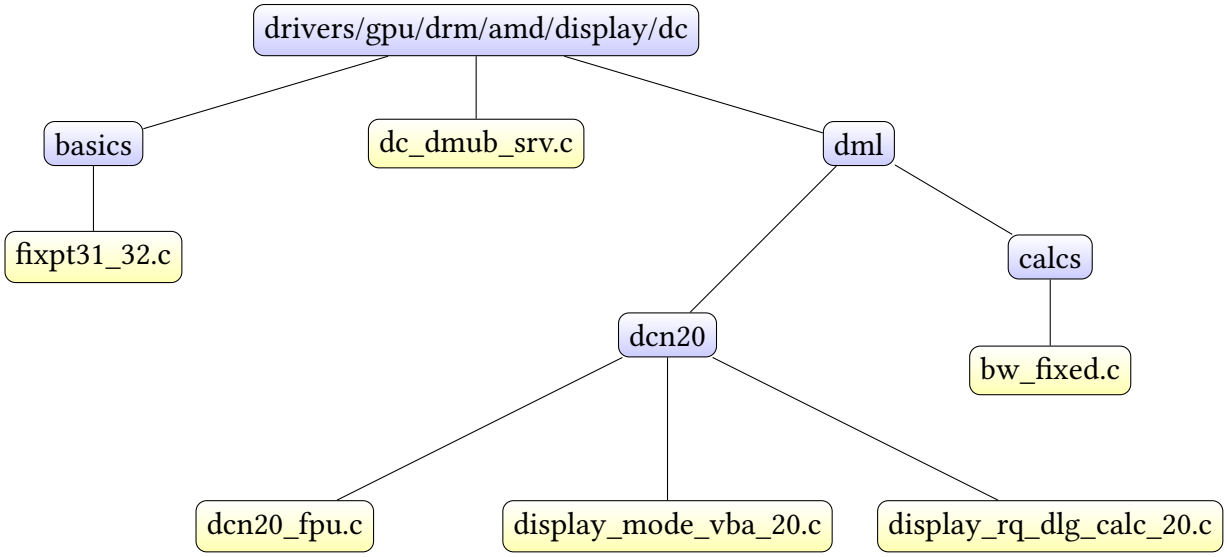



**Figure 3.10:** *Tree showing files for which tests were written*

Using *gcov*, a tool that generates code coverage reports, alongside KUnit, we can obtain information about the percentage of lines and functions covered, in addition to the specific lines that were covered or not. Figures 3.12, 3.11, and 3.13 show the results we got.

| 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 3.11:** *Report showing code coverage from DML*

**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 3.12:** *Report showing code coverage for DMUB*

| 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 3.13:** *Report showing code coverage from fixed31_32*

As stated before, we mostly focused on writing tests for the DML folder. At first glance, Figure 3.11 indicates that little line and function coverages were achieved in that area. When analyzing those numbers, though, we have to keep in mind that this directory is huge in number of lines and many of the functions need to be rewritten and split so that they can be tested. Nevertheless, we have tests that cover 20 functions from the DML folder.

Outside of the DML, in Figure 3.12 we can that see that 10.3% of the runnable lines from the `dc_dmub_srv.c` file were executed when our tests are run. We achieved that by testing 1 out of the 26 functions of the file, in this case, the function `populate_subvp_cmd_drr_info()`, discussed in the topic 3.2.2. As for the *fixed31_32.c* file, which handles fixed-point operations, Figure 3.13 shows that our tests covered 48.7% of the lines by testing 27.8% of the functions.

### 3.2.4 Design Choices

We placed all the tests in a *tests* folder under the *display* directory since this is where all the files we tested live. Furthermore, under *tests*, we replicated the *display* folder structure, following the structure of the files we tested. For example, the tests for the *drivers/gpu/drm/amd/display/dc/dml/dcn20/dcn20_fpu.c* file are in *drivers/gpu/drm/amd/display/tests/dc/dml/dcn20/dcn20_fpu_test.c*.

#### Test Module

When compiling the Linux kernel, we can choose whether the modules will be built-in or loadable. In the first case, it means that whenever the computer is booted, the module will be automatically loaded. In the second case, we can choose, at runtime, when to load and when to unload the module.

The KUnit tests we wrote can either be run as a module or be built-in, both as part of the AMDGPU module. They are also hardware-independent and can run either on virtual machines or on bare metal[6].

When we first approached the task of using KUnit for writing tests for the AMD display driver, we originally wanted the tests to be isolated in one single module, meaning that they could be loaded and unloaded without necessarily depending on the whole AMDGPU module.

KUnit allows tests to be run as a single module, but if we want to use functions from other modules, as in any kernel module, they need to be available in the kernel namespace, meaning that they have to be explicitly exported. That is not the case with the functions in the AMD display driver, so we did not have an out-of-the-box solution for defining an exclusive module for tests.

[6] System with physical hardware

### Static Functions

When writing the unit tests, we focused on testing public functions. As we progressed, the public functions that we had not tested yet were mostly functions with high cyclomatic complexity[7], indicating that we would have to write many equally complex test cases if we wanted to cover those functions well. That left us with the choice of either (i) creating relatively incomplete tests that might cover only a couple of outcomes, (ii) explore the simple static functions that were left or (iii) rewrite the public functions so that they can be more approachable to testing.

```
1001 #if IS_ENABLED(CONFIG_AMD_DC_KUNIT_TEST)
1002 #include "../kunit/dc/dc_dmub_srv_test.c"
1003 #endif
```

**Figure 3.14:** *Appending test file into source file*

We chose the option 2 (ii) and decided to test static functions, especially since we were interested in one static function with a regression history. As mentioned in the subsection 2.4.2, the KUnit way for testing static functions means appending the test file source code into the file we are testing, illustrated in Figure 3.14. This approach may not please everyone[8], since, when testing a file, we should try not to modify the original file as much as possible.

Finally, we chose a mixed approach for the tests: we appended the source code of the test file only in the files with static functions we wanted to test; the files whose tested functions were not static remained unchanged.

## 3.3 Summary

We can argue that unit testing is a consolidated practice in userspace applications and, compared to lower-level applications like the Linux kernel, there are also similarities and different challenges.

One of our first concerns was the need to mock devices, which proved unnecessary as we wrote the tests for self-contained functions. This allows the tests to be run on various machines; it does not depend on the specific GPU for which the code is written. Going further, we followed techniques to write test cases also used in userspace software. Finally, running the tests can be as easy as running a script or more challenging if the user chooses to compile and install the kernel with tests coupled in it.

Now that the overall structure for unit testing is ready, it is quite straightforward to follow the examples provided by the available tests and add new tests for both beginners and more experienced developers. There are many opportunities to apply unit testing: when developing a new feature, the developer can take advantage of TDD; if a regression is found, one way to document it and be more confident that it will not come back is

[7] A metric that expresses the number of possible paths that a piece of code can take (McCabe, 1976)

[8] https://lists.freedesktop.org/archives/amd-gfx/2022-August/082643.html

by writing a unit test that covers it; write tests for code that is not yet covered by them, making it possible to test it modularly or even be refactored.

# Chapter 4

# Final Remarks

This work presents a perspective on introducing unit tests in the Linux kernel, focused on the AMD display driver subsystem and using the KUnit framework. The first result of this work is the descriptions of techniques for designing unit tests, which can be applied not only in the Linux kernel context but also in a more general scope for other applications. The second result is more kernel-centric and is about setting up a simple structure that allows for testing any function in the code – not limited to non-static ones – and making it easy for other developers to add any tests as needed.

Going through the formalism of designing test cases can be pretty valuable. Approaches, such as equivalence partitioning and boundary-value analysis, help in thinking and devising valuable tests and make this task less time-consuming than blindly trying to devise the tests without any technique in mind. Tracking and analyzing past regressions from the code that will be under test are also very interesting ways to design test cases. These two forms of creating test cases were used and discussed throughout this work and have found them to be rich ways of documenting how a function behaves and how it has regressed in the past.

Now that we have set up a structure for using KUnit in the AMD display driver, as soon as our patches are merged, developers from the subsystem can take the tests we wrote as example and write new tests, increasing the overall test coverage of the subsystem. We recommend, for these new tests that may arise or in other contexts out of the kernel, using the formal test design techniques and covering regression, as they make the process of building tests less complicated. Furthermore, having KUnit in the subsystem allows developers to write tests for a feature that is not yet implemented, benefiting from the Test-Driven Development process and, consequently, expanding the code coverage.

When compared to userspace applications, some particularities are related to running the tests in the kernel. We tried to touch as little source code as possible, meaning that we did not export any function we wanted to test. To allow any function to be tested, we coupled the tests inside the AMDGPU module – so that the functions were visible to the test files – but still had to append the test file to the source files, which contained static functions we desired to test. These choices are still subject to change.

Moreover, as mentioned previously in Section 2.4.3, KUnit usage across Linux kernel

subsystems is still tiny, opening doors to some future works such as:

- Adds more tests to subsystems that already use KUnit, improving their test coverage.
- Measures the quality of the unit tests in the kernel, evaluating whether good practices in unit testing are followed.
- Refactors code that is already covered by unit tests.
- Analyzes the adoption of KUnit across the subsystems.

We also emphasize two pending technical tasks directly related to this work and points to be analyzed as future work:

- IGT GPU Tools[1] is a set of integration tests for the DRM subsystem. Among its tests, IGT can run DRM unit tests defined inside the Linux kernel repository. Since DRM tests were converted to use KUnit, IGT had to be adapted to continue running these tests. As the AMD display driver is contained within DRM, its KUnit tests should also be run using IGT since it would make it easier to integrate it with existing CI infrastructures that already support IGT. There is already a work[2] for adapting IGT to run KUnit tests, but it still lacks closure.
- Test coverage can be used to detect missing coverage in the code and is a quite helpful metric to evaluate the progress of unit tests in a system. One advantage of KUnit is the ease of running its tests using the *kunit_tool* script available in the kernel repository. It is also possible to use this script with gcov to generate the test coverage reports, but there are a few limitations for it:
    1. For GCC 9+ versions, there happens a linking issue: `mangle_path` from *gcc/gcov-io.** conflicts with the function of the same name defined in the Linux kernel, *fs/seq_file.c*.
    2. GCC 7+ versions are not able to generate gcov coverage information. This happens because of how gcov exit handler is dealt[3].
    3. UML[4] is the only supported architecture for obtaining the test coverage using the script. The script does not support copying the files produced by gcov using QEMU [5] as of now.

Issues 1 and 2 sums up to the script only being able to generate test coverage with a GCC version older than 7. GCC 7 was released back in 2017, and nowadays, users have to compile and install these older versions from the source, which can be a nuisance. From this, solving these first two issues would make it pretty trivial to generate test reports using KUnit and gcov.

[1] https://gitlab.freedesktop.org/drm/igt-gpu-tools

[2] https://groups.google.com/g/kunit-dev/c/fRteQ5_6164

[3] https://lore.kernel.org/all/d36ea54d8c0a8dd706826ba844a6f27691f45d55.camel@sipsolutions.net/

[4] User-Mode Linux

[5] https://lore.kernel.org/all/CAGS_qxpbH6c3OvoYZC6TXFQomLpwZg5q7=EZ9B9k=Rw1mOz=0w@mail.gmail.com/

Regarding issue 3, not all drivers support UML, which is the case with the AMD display driver. To obtain the test coverage from this project using the *kunit_tool*, we had to internally tweak the AMD display driver code a bit to make it support UML[6]. Instead, by working on the way to retrieve the test coverage information from other architectures through the script, we can guarantee a broader range of subsystems that do not support UML being able to use *kunit_tool* to obtain the test coverage.

Beyond the discussions brought in this monograph, we highlight these results and contributions:

- 1 tutorial at FLUSP website (Generate Linux kernel's KUnit test coverage reports[7]).
- 3 posts at personal blog[8].
- 11 authored patches to the Linux kernel[9].
- 1 Reported-by patch [10].
- Full-slot presentation and lightning talk at the X.Org Developer's Conference 2022[11,12].

[6] https://gitlab.freedesktop.org/isinyaaa/linux/-/merge_requests/8

[7] https://flusp.ime.usp.br/kernel/generate-kunit-test-coverage/

[8] https://magalilemes.github.io/

[9] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=author&q=Magali

[10] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=01b537eeb049b98d7efc8f9c93c2608ef26ec338

[11] https://indico.freedesktop.org/event/2/contributions/65/

[12] https://indico.freedesktop.org/event/2/contributions/164/

# Chapter 5

# Personal Appreciation

From having no idea of what Free Software meant until actually contributing to one of the biggest free software projects out there has been such a long road. Working on this project allowed me to learn a lot about different areas in Computer Science, from operating systems until software testing.

I'm very grateful for having had the opportunity to work in this project alongside amazing people and being mentored by awesome folks. Going to my first conference ever and presenting the project with my peers is definitely one of the highlights from all of this.

It has been quite rewarding.

# Appendix A

# Scripts

**Program A.1** Script used to retrieve the subsystems that use KUnit.

```
from pathlib import Path
import re
import subprocess
import sys

def subsystems_with_kunit(linux_repo):
    subsystems_kunit = set()
    paths = Path(linux_repo).rglob('*')

    for path in paths:
        if path.is_file():
            with open(path, 'r', encoding="ISO-8859-1") as f:
                    if "kunit/test.h" in f.read():
                        maintainer = subprocess.run(
                            ["./scripts/get_maintainer.pl", "--subsystem", "--no-email", "--no-l
 ", path],
                            stdout=subprocess.PIPE,
                            cwd=linux_repo,
                            encoding="utf-8")
                        subsystem = re.match("[^\n]+", maintainer.stdout).group(0)

                        subsystems_kunit.add(subsystem)
    return subsystems_kunit

def main():
    print(subsystems_with_kunit(sys.argv[1]))

if __name__ == '__main__':
    main()
```

**Program A.2** Script used to plot number of lines in a directory.

```
from glob import glob
from pathlib import Path

import json
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import subprocess
import sys

sns.set(rc={ 'figure.figsize': (15, 12) })
sns.set(font_scale=1.5)

def loc_plot(directory, n=None):
    dirs_lines = []
    directories = glob(str(directory) + "/*/", recursive=True)

    if n is None:
        n = len(directories)

    for directory in directories:
        tokei = subprocess.run(
            ["tokei", directory, "--output", "json"],
            stdout=subprocess.PIPE,
            encoding="utf-8"
        )
        tokei_json = json.loads(tokei.stdout)
        lines = tokei_json["Total"]["code"]
        dirs_lines.append(lines)

    d = { 'Directories': list(map(lambda d : Path(d).stem, directories)), 'Lines': dirs_lines }
    df = pd.DataFrame(d, columns=['Directories', 'Lines'])

    ax = sns.barplot(
        x='Lines',
        y='Directories',
        data=df,
        order=df.sort_values('Lines', ascending=False).Directories.head(n)
    )

    plt.show()

def main():
    if len(sys.argv) == 2:
        loc_plot(sys.argv[1])
    elif len(sys.argv) == 3:
        loc_plot(sys.argv[1], int(sys.argv[2]))
    else:
        print("python3 lines_of_code.py <path> [<n>]")

if __name__ == '__main__':
    main()
```

**Program A.3** Script used to show the evolution of KUnit's usage.

```python
from glob import glob
from pathlib import Path

import matplotlib.pyplot as plt
import re
import seaborn as sns
import subprocess
import sys

sns.set(rc={ 'figure.figsize': (22, 12) })
sns.set(font_scale=1.5)

def subsystems_with_kunit(linux_repo):
    subsystems_kunit = set()

    paths = Path(linux_repo).rglob('*')
    for path in paths:
        if path.is_file():
            with open(path, 'r', encoding="ISO-8859-1") as f:
                    if "kunit/test.h" in f.read():
                        maintainer = subprocess.run(
                            ["./scripts/get_maintainer.pl", "--subsystem", "--no-email", "--no-l
 ", path],
                            stdout=subprocess.PIPE,
                            cwd=linux_repo,
                            encoding="utf-8")
                        subsystem = re.match("[^\n]+", maintainer.stdout).group(0)

                        subsystems_kunit.add(subsystem)
    return subsystems_kunit

def main():
    if len(sys.argv) != 2:
        print("python3 kunit_usage.py <linux path>")
    else:
        linux_repo = sys.argv[1]
        linux_tags = subprocess.run(
        [
            "git", "tag", "--list", 'v[5-9].[0-9]*',
            "--sort=version:refname"
        ],
        cwd=linux_repo,
        stdout=subprocess.PIPE,
        encoding="utf-8",
    )
    tags = []

    for t in linux_tags.stdout.split("\n")[:-1]:
        final = re.match("v[0-9].[0-9]*$", t)
        if final is not None:
            tags.append(final.group(0))

    kunit_usage = []

    for tag in tags[4:]:
        subprocess.run(
            ["git", "checkout", tag],
            cwd=linux_repo,
            encoding="utf-8",
        )

        kunit_usage.append(len(subsystems_with_kunit(linux_repo)))

    ax = sns.lineplot(x=tags[4:], y=kunit_usage, marker='o')
    ax.set_xlabel("Version", fontsize=15)
    ax.set_ylabel("Number of subsystems", fontsize=15)

    plt.show()

if __name__ == '__main__':
    main()
```

**Program A.4** Script to compare types of contributions in the AMD Display Driver.

```
from glob import glob
from pathlib import Path

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import subprocess
import sys

sns.set(rc={ 'figure.figsize': (1, 1) })
sns.set(font_scale=1.5)

def commits_amount(contributor, linux_repo):

    command_template = "git log --oneline"
    if contributor == "external":
        command_template += " --author='^(?!(.*@amd[.]com))' --perl-regexp"
    elif contributor == "amd":
        command_template += " --author='^(.*@amd[.]com)' --perl-regexp"

    commits = []

    for year in range(2019, 2023):
        command = command_template
        command += " --after=\"" + str(year) + "-01-01\"" + " --until=\"" + str(year) + "
 -12-31\""
        command += " -- drivers/gpu/drm/amd/display | wc -l"

        commits_out = subprocess.run(
            command,
            cwd=linux_repo,
            encoding="utf-8",
            shell=True,
            stdout=subprocess.PIPE,
        )

        commits.append(int(commits_out.stdout))

    return {"year": list(range(2019, 2023)),
            "contributor": [contributor]*4,
            "commits": commits}

def main():
    if len(sys.argv) != 2:
        print("python3 kunit_usage.py <linux path>")
        return

    linux_repo = sys.argv[1]
    df1 = pd.DataFrame(commits_amount("amd", linux_repo))
    df = df1.append(pd.DataFrame(commits_amount("external", linux_repo)), ignore_index=True)

    print(df)

    sns.factorplot(x='year', y='commits', hue='contributor', data=df, kind='bar')

    plt.show()


if __name__ == '__main__':
    main()
```

# Annex A

# Pictures


**Figure A.1:** *AMD Radeon™ RX 5700 XT 50th Anniversary Graphics video card*

# References


[*2. How the development process works* 2022] *2. How the development process works*. URL: https://www.kernel.org/doc/html/latest/process/2.Process.html (visited on 12/22/2022) (cit. on p. 6).

[Ackerman 2022] Evan Ackerman. *How NASA Designed a Helicopter That Could Fly Autonomously on Mars*. URL: https://spectrum.ieee.org/nasa-designed-perseverance-helicopter-rover-fly-autonomously-mars (visited on 12/18/2022) (cit. on p. 5).

[Cohn 2009] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. ISBN 978-0321579362. Addison-Wesley Professional, 2009 (cit. on p. 6).

[Corbet 2022] Jonathan Corbet. *Some 6.0 development statistics*. URL: https://lwn.net/Articles/909625/ (visited on 12/22/2022) (cit. on p. 7).

[*drm/amd/display - Display Core (DC)* 2022] *drm/amd/display - Display Core (DC)*. URL: https://docs.kernel.org/gpu/amdgpu/display/index.html (visited on 12/22/2022) (cit. on p. 15).

[Fowler 2006] Martin Fowler. *Xunit*. 2006. URL: https://martinfowler.com/bliki/Xunit.html (visited on 12/22/2022) (cit. on p. 8).

[Iacono *et al.* 2009] Jessica Iacono, Ann Brown, and Clive Holtham. "Research methods–a case example of participant observation". In: *The Electronic Journal of Business Research Methods Volume* 7 (Jan. 2009), pp. 39–46 (cit. on p. 2).

[McCabe 1976] T.J. McCabe. "A complexity measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837 (cit. on p. 27).

[Myers *et al.* 2012] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. 3rd ed. Hoboken and N.J: John Wiley & Sons, 2012 (cit. on pp. 6, 9).

[Robey and Taylor 2018] Daniel Robey and Wallace T.F Taylor. "Engaged participant observation: an integrative approach to qualitative field research for practitioner-scholars". In: *Engaged Management* 2 (2018) (cit. on p. 2).

[Schmitt 2022] Marcelo Schmitt. “Linux kernel device driver testing”. São Paulo: Universisade de São Paulo, Dec. 2022 (cit. on pp. 1, 7, 8).

[Tanenbaum and Bos 2014] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Boston, MA: Pearson, 2014. isbn: 978-0-13-359162-0 (cit. on p. 5).

[*The Beginning of the Linux Open-Source Operating System* 2022] *The Beginning of the Linux Open-Source Operating System*. url: https://historyofinformation.com/detail.php?entryid=2000 (visited on 12/18/2022) (cit. on p. 5).