ReadME - Linux Kernel Best Practices

Akhil Kumar Mengani
amengan@ncsu.edu
North Carolina State University

Nivedita Lodha
nnlodha@ncsu.edu
North Carolina State University

Sai Naga Vamshi Chidara
schidar@ncsu.edu
North Carolina State University

Uma Gnanasundaram
ugnanas@ncsu.edu
North Carolina State University

Neha Kotcherlakota
nkotche@ncsu.edu
North Carolina State University

ABSTRACT

The Linux kernel is a shining illustration of open source's power and the benefits of collaborative effort. The kernel has been around for three decades, due to its contributing developers from over large number of firms all over the world. The project's durability and success thus far have been unprecedented, especially when considering the development mayhem that occurs on a daily basis. The Linux kernel is undoubtedly the most dynamic kernel on the market, with stable versions issued every 3 months. Of doubt, such a massive undertaking must have had its share of setbacks and difficulties.

CCS CONCEPTS

• **Software Engineering**; • **Best Practices** → *Linux Kernel Development*;

KEYWORDS

Software Engineering, Linux Kernel Best Practices, ReadME

1 INTRODUCTION

The Linux project adheres to a set of essential principles, which we have included in our project guidelines [1]. The following are some guidelines to follow:

- Short Release Cycles
- Zero Internal Boundaries
- The No-Regressions Rule
- Consensus Oriented Model
- Distributed Development

The Linux open source community follows these five guidelines, which have been critical to their success over the previous 30 years. We will try to establish the connection between the items mentioned in the rubric and the Linux Kernel best practices. We will also elaborate on how we followed these practices to achieve efficiency and collaboration in the software development process.

2 LINUX KERNEL BEST PRACTICES

2.1 Short Release Cycles

Major kernel releases used to happen every few years in the early days of kernel development. However, this resulted in a number of bugs and issues during the development process. In Long release cycles, developers were forced to include features in the next release even if they weren't totally stable. As a result, transition to shorter release cycles eliminated all of the problems associated with long release cycles.

Below are items in the rubric that are associated with Linux kernel best practices.

- **Use of version control tools** - We followed version control tool, GitHub, to immediately commit the changes done by a developer.
- **Number of commits by different people**
- **Number of commits** - Each developer worked on their own branch and the create a pull request to merge the changes to main branch.
- **Issues reports, Issues are being closed** - Continuous integration allowed us to create a stable code base.

Since it is only phase 1 of the project, there aren't any releases before this. But we made sure that we followed the approach of short release cycles.

2.2 Zero Internal Boundaries

Developers typically work on certain sections of the kernel, but this does not preclude them from making changes to other parts of the kernel provided the changes are justified. This technique ensures that problems are resolved at the source rather than allowing various workarounds to emerge, which is always bad for kernel stability. Furthermore, it provides developers with a broader view of the kernel as a whole.

Below are the items in the rubric that checks Zero Internal Boundaries.

- **Evidence that the whole team is using the same tools (e.g. config files in the repo, updated by lots of different people)**
- **Evidence that the whole team is using the same tools: everyone can get to all tools and files**
- **Evidence that the members of the team are working across multiple places in the code base**
- **Evidence that the whole team is using the same tools (e.g. tutor can ask anyone to share screen, they demonstrate the system running on their computer)**

All the team members have used Python 3, Django and Angular to create the application. We documented all the commands to be executed to set-up the project. Whenever a developer wants to work on the projects. A virtual environment is created and install all the packages and dependencies using requirements.txt file. Dependencies are updated continuously in the requirements.txt file during the development process. Whatever dependencies the developers encounter while coding the functionalities, they are included immediately in the requirements file, so that other developers will not face any issue. The work is divided in such a way that each

member gets a chance to push code to different parts of the code base. Developers are advised to use appropriate commit messages which might serve as an evidence of contribution.

2.3 No Regressions Rule

The kernel developer community attempts to improve the kernel code base on a regular basis, but not at the expense of quality. This is why they adhere to the no-regressions rule, which asserts that if one kernel works in one environment, all future kernels must also operate in that environment. If, however, the system is affected by regression, kernel developers respond quickly to address the problem and restore the system to its previous form.

Below are the items in the rubric to checks if the team has followed No Regression Rule.

- **Test cases exist** - Test cases are written for each function/method in the project. Whenever a new method is added, a test case is written for that in the same transaction.
- **Test cases are routinely executed** - All the test cases in the test suite are executed every time a new test case is added. Also whenever a developer push the code to the repository, **Travis CI** is used to test the project. Before every pull request, the project is tested before its merged. If there is any failure in the test cases, build fails and the status of the build is seen in the GitHub repository.
- **The files CONTRIBUTING.md lists coding standards and lots of tips on how to extend the system without screwing things up** - Developers who are interested in contributing to the projects should follow the steps mentioned in the CONTRIBUTING.md file to avoid any breakage in the functionality. This file has all the information to contribute to the project.

2.4 Consensus-Oriented Model

The Linux kernel community follows a consensus-oriented strategy, which specifies that a proposed change cannot be accepted into the code base if a reputable developer opposes it. Although this may annoy individual developers, it ensures that the kernel's integrity is not compromised; no single developer may make changes to the code base at the expense of others. As a result, the kernel's code base remains as adaptable and scalable as it has always been.

Below are the items in the rubric to check if we have followed the consensus-oriented model practice.

- **Is your source code stored in a repository under revision control?** Yes, the entire code base is stored in GitHub repository and all the members of the team are added as collaborators which enables them contribute to the project.
- **Issues are discussed before they are closed** - All the developers in the team get a chance to speak on the issues that are being closed. Whenever a developer raises merge request, the fix is reviewed by at least 2 members of the team. Reviewers comment on the every merge request raised. If there are any issues with the code or merge request, it will be discussed on the discord or zoom call by scheduling a call.
- **Are e-mails to your support e-mail address received by more than one person?** - Yes, We created a separate mail which can be accessed by all the members of the team. One of the team members or whoever checks the mail first will post on the discord channel about the subject of the mail.

We ensured that we followed the consensus-oriented model and all the members of the team had their part in the software development of the project.

2.5 Distributed Development Model

Decentralized teams located across different physical work spaces plan, design, build, test, and manage software in distributed software development. To create software, these groups use web-based collaboration tools and platforms.

Different portions of the code are assigned to different developers based on their familiarity.

Below are items in the rubric that are associated with distributed development practice.

- **Chat channel: exists** - We had a dedicated group for the team member in Discord to discuss about the division and progress of the work.
- **Issues reports**
- **Issues are discussed before they are closed** - Developers create issues in GitHub and are tracked by other developers. Each issue is assigned to multiple developers. If its a software issue, then its fixed by a developer and reviewed by other before merging it into main branch.
- **Workload is distributed across the entire team** - Every Monday, we held Zoom video conferences to track the state of application development and distribute work across the team. To manage in-progress and completed issues assigned to various employees, we used the Projects tab in GitHub. This also aids in determining whether or not the task has been allocated to all the members of the team.

Also, contribution of each member can be seen in GitHub graph.

3 CONCLUSION

Following the Linux Best Practices ensured that our product met open-source development standards. All team members were allowed to work on all aspects of the project because there were no internal limits. Our program was constantly developing and no features were deleted by ensuring there were no regressions. Following a consensus-based approach guaranteed that everyone on the team had a say in every decision. We had talks about all code changes and open communication because we distributed the work. Short Release Cycles enabled continual improvement and delivery of software throughout the software development life cycle.

REFERENCES

[1] Tim Menzies. 2021. *proj1rubric*. Retrieved September 29, 2021 from https://github.com/txt/se21/blob/master/docs/proj1rubric.md