WolfLease Project and Kernel Practices

Ameya Vaichalkar
agvaicha@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Subodh Gujar
sgujar@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Kunal Patil
kpatil5@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Rohan Shiveshwarkar
rsshives@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Yash Sonar
ysonar@ncsu.edu
North Carolina State University
Raleigh, NC, USA

ABSTRACT

All international students coming to the USA or national students changing states for higher education but don't have an idea about where to live and don't want to lease the complete apartment because of cost. We are introducing WolfLease, easy-to-use software where students and apartment owners can post their rooms for subleasing which helps them to get an idea about the neighborhood and can choose which room to sublease based on their preferences.

CCS CONCEPTS

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

KEYWORDS

csc510, software engineering, kernel best practices

ACM Reference Format:
Ameya Vaichalkar, Subodh Gujar, Kunal Patil, Rohan Shiveshwarkar, and Yash Sonar. 2022. WolfLease Project and Kernel Practices. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Linux Kernel is one of the biggest open source project that has been around for a while and is still used and supported by many individuals all around the world. What enables a kernel that was initially created on a single machine to be utilized and improved upon by several developers are the Linux Kernel's methods for maintaining an open source project. The following are the list of few Linux kernel best practices:

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

2 LINUX KERNEL BEST PRACTICES

2.1 Short Release Cycles

Smaller amounts of code can be added, and a stable version can be released more frequently. It guarantees that the code that is pushed is effective and error-free. It also makes it easier to identify coding problems. Shorter release cycles make it easier to focus on single, smaller milestones. Making brief releases with a limited number of features is also a wonderful method to obtain customer input on each modification. Additionally, alter the features when they are still in the early stages of development, when it is simpler to do so.

In the short development period, it was challenging to keep this practice. However, we took steps to incorporate this approach into our project by regularly making commits to the repository. Multiple issues were created and assigned to team members, which were then discussed and completed in different branches.

2.2 Distributed Development Model

According to this model, the project is divide into parts and they are assigned to team members. This helps to distribute the work and responsibilities among developers. Code review and integration are made easy by allocating different functionalities to different people according to their preferences and skills.

The Distributed Development Model breaks the project into parts and the parts are then assigned to different developers. This keeps in check that the responsibility is distributed and the entire load is not put on a single developer. Assigning different functionalities to different individuals based on their liking and expertise enables seamless code review and integration. The developer gets to learn new technology with the responsibility to only deliver a feature.

This is supported by the Project 1 rubric, which includes criteria like "workload is distributed across the team," "proof that the team members are working across numerous locations in the code base," and "evidence that the team is using the same tools." The quantity of commits and the number of rejected pull requests provide proof of this. Additionally, each file's commit history reveals that various members contributed to it. We divided up the work into distinct project components; each individual started a branch and combined it after the component was finished and approved. We have kept same version of dependencies in requirements.txt file, which ensures that we are using same tools with same versions. A Pull Request is raised when the feature is completed and tested thoroughly. Every Pull Request was reviewed by minimum two members which ensured a bug free and stable code is merged into master branch.

2.3 Consensus-Oriented Model

Before the code is incorporated into the stable version and made public, there should be agreement among the developers. Each integration should uphold the coding standards. Additionally, the software's future development should be decided upon so that all developers are pursuing the same objective.

The rubric specifically states that "Issues are discussed before they are closed" in relation to this requirement. This includes merging pull requests because we associate pull requests with issues/features, and merging pull requests closes issues. Before merging, we made sure at least one reviewer had looked through the code. Another rubric, "Chat channel: existing," is mentioned in the rubric. A communication channel is necessary for consensus to occur. A WhatsApp group, and a Microsoft Teams channel was made by us to discuss current features being implemented and issues occurring during development.

2.4 The No-Regressions Rule

As new updates to a piece of software are released, the No Regressions Rule for Linux Kernel Best Development Practices states that the existing functionality should not be disturbed. We've made sure not to remove any functionality that already exists, but rather to add enhancements to existing features. The rubric includes criteria like "Utilize version control tools," "Store your documentation under revision control with your source code," "Publish your release history, including release data, version numbers, major features of each release, etc. on your website or in your documentation," and "There is a branch of the repository that is always stable." This rubric, documentation and version control can ascertain if any feature are removed or the quality of code is deteriorated between the older iteration.

2.5 Zero Internal Boundaries

This principle sates that every project member should have a high level overview of the whole project structure.According to the Zero Internal Boundaries principle, there should be no restrictions on which contributors can access what areas of a project. Everyone working on a project should have access to the tools and every part of the source code of the project. This principle is in line with the project 1 rubric's criterion for "proof that the team as a whole is utilizing the same tools" and "evidence that team members are working across numerous locations in the code base". Since each team member contributes to the project's numerous source files and because we only use cross-platform open-source tools, we adhere to the zero internal boundaries principle.

For Project 1, we kept the requirements file up to date with the libraries that we used in our project. By using virtual environment to run our project along with the same versions of dependencies, we ensured that everyone has access to all functionality of the software . We also have a WhatsApp group and Microsoft Teams chat channel for meeting notes and resources to discuss issues and any difficulties while development.

It is important to have access to the project's overall perspective. Even if distinct functionalities are the focus of each person's job, they are free to perform changes to any part of the code. By doing this, issues are resolved directly at the root source rather than having to travel through several steps before changing anything. The team benefits from having no internal boundaries by the fact that there won't be any obstacle to limit a person's freedom to contribute, collaborate to the development.