Connection between the Project Rubric and Linux Kernel Development Best Practices

Chandrahas Reddy Mandapati, Sri Pallavi Damuluri, Harini Bharata, Niraj Lavani, Sandesh Aladhalli Shivarudre Gowda
cmandap@ncsu.edu,sdamulu@ncsu.edu,hbharat@ncsu.edu,nrlavani@ncsu.edu,sgaladha@ncsu.edu

ABSTRACT

Linux Kernel Best Practices are wildly used to ensure smooth development cycles with many moving parts. This report addresses the similarities in software development and Linux Kernel Development Best Practices.

KEYWORDS

Linux Kernel, Best Practices, Software Development

1 SHORT RELEASE CYCLES

Software development used to have slower release cycles with major developmental changes. Major changes were seen to be inefficient since large chunks of code needed to be integrated at once. Additionally, larger releases were packed with many shortcomings such as glitches and problems in the developmental cycle. This old practice was replaced with shorter release cycles which solved all the major issues long release cycles possessed. Shorter cycles means smaller code integration which generally means manageable issues. New code is integrated into a stable release with the benefit of not causing any major disruptions. This also alleviates pressure off developers for getting things right in 1 large release cycle.

Our rubric considers a the number of requests per project. This is an important specification since a few commits may lead to a finished program but may indicate longer development cycles and longer development cycles tend to slow progress and may come with unwanted surprises (improper integration, bugs, glitches). The number of commits. The rubric also specifically grades short release cycles to make sure all other team members can reach the progress of the developer.

Smaller projects, which naturally have a smaller amount of commits, should still implement short development cycles. In our project, we implement and push small yet functional changes to the repository so the rest of the team can build from it. As individuals in a team, our goal was to push to the repository upon completion of small self-made milestones.

2 DISTRIBUTED DEVELOPMENT MODEL

The distributed development model is a paradigm where individuals are given different attributes of the development to work on individually. Previously, a single person was in charge of approving and and integrating changes to the kernel. For larger tasks, this model became cumbersome as one person cannot keep up with the complexities of an operating system kernel. With the distributed development model, different portions of the kernel, like networking, device drivers, wireless, etc) are delegated to different developers based on their expertise in the area. This creates efficient, bug-free, and confident commits that do not compromise stability. This proves to work best given the thousands of areas of the kernel that require review and integration.

Our project rubric states in different areas that workload must be spread over the whole team; team members should gravitate to what they are comfortable with and have experience working in. Though our program isn't as complex as a Linux kernel, the same distributed development model can be applied. In our team, Chandrahas, Sri Pallavi, and Harini are adept in back-end programming and therefore, were assigned the role of ensuring proper webscraping functionality in our program. Sandesh and Niraj are new to python programming so they were given front-end development and organizational roles. Through this team distribution, we were able to create a functioning webapp within our defined timeline.

The rubric also mentioned the number of commits by different people. This requirement is there to ensure every member is participating towards the completion of the project. Additional specifications on the rubric (such as: DOI Badge, Docs: what: point descriptions of each class or function, Docs: how: for common use cases XYZ mini tutorials showing worked examples on how to do XYZ, Docs: why: docs tell a story..., Docs: short video) are different areas members can work on.

3 CONSENSUS-ORIENTED MODEL

The Linux kernel community lives by the consensus-oriented model that specifies any asserted code change/implementation can only be integrated into the code base if all respected developers agree on its integration. If a single respected developer is against it, the code must be revisited for alterations or review. This strict paradigm is there to enforce strong kernel integrity; poor review or lazy integration into the code base can threaten the stability of the kernel. This model has proven to be successful as the current Linux kernel is scalable from small chips to super-computers shared across a wide variety of uses.

The rubric specifications includes having a CONTRIBUTING.md file which lists the coding standards and how to extend the system without ruining functionality, and a chat channel that indicates a team is communicating actions that require the approval of all members. Our team has both a CONTRIBUTING.md file and a chat channel on Whatsapp where we discuss functionality. Functionality consensus is also taken over virtual meetings via Zoom. For example, our team decided to first implement a .csv file as a baseline for links connecting the program to different e-commerce websites. We then discussed and agreed to implement APIs to get the links dynamically.

4 TOOLS MATTER

Kernel struggled to scale due to the increasing weight and complexity of the development; with other thousands of facets, it's hard to manage. The Linux kernel then started using BitKeeper, a source-code management system which instantaneously changed the communities practices. BitKeeper is an SCM that tracks a running history of changes to the code base and even helps resolves conflicts between merges by different developers. Through BitKeeper, the kernel was able to scale at a much faster rate.

Our project rubric requires the use of version control tools to track progress and changes made by teammates and prevent any conflicts. Version control assists in comparing files, and making sure all team members are working off the latest version. When it comes to scaling the project after project 1, version control will be needed in reviewing new updates to the program and safely merging them into the code base without fear of breaking functionality.

5 NO-REGRESSIONS RULE

The kernel developer community aims to upgrade the kernel's code base but not at the cost of quality. The No-Regression rule exists to state if the kernel works in a specific setting, all other kernel improvements must also function previous settings. Linux kernel developers also keep tabs on if the regression rule is broken. When it is, developers address the issue and return the kernel state back to the original. The no-regressions rule is an ideal rule but provides motivation in creating net-positive updates.

The rubric's requirement of test cases, existing and routinely executed, is present to follow the no-regressions rule; all improvements to the project must still pass test cases. The rubric also mentions style check, code formatting, syntax, code coverage, automated analysis, which are all used to show if the program ever regresses in functionality, us developers will know and can take action. In our project development, as a team, we made a plan of action so code in our meetings so functionality was never regressing. We made sure all progress brought us towards a working program.

6 ZERO INTERNAL BOUNDARIES

Even though the distributed development model states all developers should work in their own fields related to their expertise, the zero internal boundaries practice exists so all developers can still make changes to different areas of the kernel. This helps fix problems when they are found rather than creating workarounds which generally threaten kernel stability. Zero internal boundaries also give kernel developers a better overarching view of how the kernel works.

In the same way, the project rubric asks for evidence that the whole team is using the same tools where everyone can access and edit the project files. The rubric also states every member should be familiar with the project structure and are working across different parts of the code base. In our project meeting calls, we discuss different areas of the program so everyone is on the same page and understands what's going on under the hood. We also collaborated on the same project files which indicates everyone has a clear picture of what's being implemented.

7 REFERENCES

$https : //medium.com/@Packtpub/linux-kernel-development-best - practices - 11c1474704d6$