aPAS - A Personal Agile Scheduler

Aditi Bhagwat
aabhagwa@ncsu.edu
North Carolina State University
USA

Anumit Garg
agarg24@ncsu.edu
North Carolina State University
USA

Palvit Garg
pgarg5@ncsu.edu
North Carolina State University
USA

Rachit Sharma
rsharm26@ncsu.edu
North Carolina State University
USA

Shree L Ramasubramanian
slramasu@ncsu.edu
North Carolina State University
USA

ABSTRACT

The Linux kernel[7] has been around for 27 years and it serves as a prime example of open source and the merits of constructive collaboration. It is probably the most dynamic kernel and it has had its share of failures and challenges. However, what accounts for the project's success is the Linux kernel community's capacity to stay flexible and keep apace of the latest trends while adhering to several best practices. These practices, if adopted in any software project, can help reduce the development chaos that ensues almost on a daily basis. This document brings out the connection between the project rubric document and the Linux kernel best practices. In addition, it also showcases how our project has undergone some of these itself.

KEYWORDS

Linux Kernel[7], Software Development, Linux Kernel best practices[7]

ACM Reference Format:
Aditi Bhagwat, Anumit Garg, Palvit Garg, Rachit Sharma, and Shree L Ramasubramanian. . aPAS - A Personal Agile Scheduler. In ,. ACM, New York, NY, USA, 2 pages.

1 SHORT RELEASE CYCLES

In the earlier stages of kernel development, major releases would come once every several years. This led to problems in the development cycle as major chunks of code had to be integrated all at once which proved to be inefficient. It also posed a lot of challenges for developers who had to integrate features in the upcoming release even if they were not completely stable or ready. Hence, the Linux kernel development community decided to shift to short release cycles. New code is continually integrated into a stable release which allows introducing fundamental changes into the code base without causing major disruptions.

Our project rubric mentions the number of commits made in the project which is a good indicator of short release cycles. Since our project is a small scale project, we do not have major release cycles. However, what we as a team ensured was that each of us commits the changes that we make in the code to build a working functionality so that everyone else can access it and further build on it.

2 DISTRIBUTED DEVELOPMENT MODEL

Initially, all the changes made to the Linux Kernel's code base were sent to a single person for review and integration. However, this soon proved to be cumbersome, as it is impossible for a single person to be able to keep up with something as complex and varied as an operating system kernel. The community quickly realized that a distributed model is the best way of taking the development ahead. In a distributed development model, different parts of the project are handled by different team members so that a single person is not burdened with keeping apace with every part of the project. Pertaining to this, our project rubric mentions whether the workload was spread over the entire team and how many commits were made by each team member. That is a direct indicator of the "distributed development model". We created issues for every new task and divided the work amongst each other by working on those issues.[1]

3 CONSENSUS ORIENTED MODEL

The consensus-oriented model states that the proposed changes cannot be integrated into the code base if a respected developer is opposed to it. This ensures that no single group can make changes to the code at the expense of other groups. In our project, for the initial project design, we conducted a zoom meeting and we moved ahead with the current design only after each and every member agreed to it. In addition, during the implementation phase, branches were created for different features and the changes were reviewed before merging them into the master branch. This made sure that the new code didn't disrupt the original working code.[8]

4 TOOL MATTER

Kernel development struggled to scale until the advent of the BitKeeper source-code management system changed the community's practices nearly overnight; the switch to Git brought about another leap forward. Without the right tools, a project like the kernel would simply be unable to function without collapsing under its own weight. We have used Git[4] as our version control system to maintain the project and newer features. Intellij[5] was used as the integrated development environment throughout for the

development of the software and for generating Javadocs for documentation. Eclipse Papyrus[6] was used as the UML[10] tool for defining the architecture of the project, this helped in developing various features.

5 'NO REGRESSIONS' RULE

A No-regression rule means that if a code base works on a specific setting, all the subsequent code merges must work there too. Doing short releases and maintaining a consensus-oriented model is one step closer to a no regression rule.The kernel developer community continually strives to upgrade the kernel code base, but not at the cost of quality. This is why they follow the no-regressions rule, which states that if a given kernel works in a specific setting, all the subsequent kernels must work there too. However, if the system does end up affected by regression, kernel developers waste no time in addressing the issue and getting the system back to its original state. We have set up Java CI with Maven pipeline[2] which ensures that every new commit is ran on a set of tests. This makes sure that previous commits are safe from subsequent commits. Confidence on a strong "no regression" rule gives the team to work on new capabilities.

6 NO INTERNAL BOUNDARIES

Developers generally work on specific parts of the kernel; however, this does not prevent them from making changes to any other part as long as the changes are justifiable.Anyone who thinks that they can solve or work on a particular issue can rightly do so. Once that is done someone can review the changes and see if those changes are justified.This practice ensures that problems are fixed where they originate rather than making way for multiple workarounds, which are always a bad news for kernel stability. Moreover, it also gives the developers a wider view of the kernel as a whole. In our project, we created a lot of issues[1] for the pending tasks and the person who thought they could take up a particular issue took it and worked on it. The changes were then reviewed and the issue was closed.

7 CORPORATE PARTICIPATION IN THE PROCESS IS CRUCIAL

We would not have the fast-moving project described here without Corporate Participation in the process is crucial.But it is also important that no single company dominates kernel development. While any company can improve the kernel for its specific needs, no company can drive development in directions that hurt the others or restrict what the kernel can do. This ensures that the project is available and can be extended by other companies. Our project is available on Github[3]and can be extended by anyone who is interested. We have provided documentation[9] support for developers ensuring clear understanding and aiding them to contribute.

8 REFERENCES

[1] https://github.com/elric97/CalBot/issues
[2] https://github.com/elric97/CalBot/actions/workflows/maven.yml
[3] https://github.com/elric97/CalBot
[4] https://github.com
[5] https://www.jetbrains.com/idea/
[6] https://www.eclipse.org/papyrus/
[7] https://go.pardot.com/l/6342/2017-10-24/3xr3f2/6342/188781/Publication_LinuxKernelReport_2017.pdf
[8] https://github.com/elric97/CalBot/pull/13/
[9] https://github.com/elric97/CalBot#readme/
[10] https://www.uml.org/