Getting started with open source development

Preeti U Murthy <preeti@linux.vnet.ibm.com>

Energy Management Team,

Linux Technology Center, IBM

Problem Statement: Develop feature 'X' in the open source project 'Y'

How to begin with the development of feature 'X'?

Summary:

- Get the source code of project 'Y' onto your local workstation.

- Make changes to the code on your local workstation.

- Post these changes to the mailing list of 'Y'

- Actively discuss your code with the community as they offer suggestions and feedback.

- Incorporate the suggestions back into feature 'X' and post it out again.

- Repeat the above two steps till feature 'X' gets no more corrections. Feature 'X' is now ready to go into project 'Y'.

Make changes to the local copy of the source code

- Any change that you make to project 'Y' should be reflected as a “patch”.
- Feature 'X' could get into project 'Y' as one patch or multiple patches.
- Patch management could be done using git or stgit.
- Demo using stgit to generate a patch.

Example of a patch

```
kernel/sched/fair.c |   41 ++++++++++++++++++++++-------------------
 1 file changed, 22 insertions(+), 19 deletions(-)

--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -4288,6 +4288,7 @@ struct sg_lb_stats {
        unsigned long group_capacity;
        unsigned long idle_cpus;
        unsigned long group_weight;
+       unsigned long group_power;
        int group_imb; /* Is there an imbalance in the group ? */
        int group_has_capacity; /* Is there extra capacity in the group? */
 };
@@ -4546,7 +4547,8 @@ static inline void update_sg_lb_stats(st
                update_group_power(env->sd, env->dst_cpu);

        /* Adjust by relative CPU power of the group */
-       sgs->avg_load = (sgs->group_load*SCHED_POWER_SCALE) / group->sgp->power;
+       sgs->group_power = group->sgp->power;
+       sgs->avg_load = (sgs->group_load*SCHED_POWER_SCALE) / sgs->group_power;
```

Now you have a patch, what next?

- Every patch must be accompanied by a changelog, which tells the idea behind the patch and why it has been implemented the way it has.
- So now you have patch + changelog, who to mail it to?

  - If there is an internal mailing list in IBM LTC pertaining to your work, post out the patch to it. Internal reviews weed out a lot of issues which are relatively easy to solve.

    This ensures that when you post out to the community you have more pressing issues to discuss.

  - After an internal review, mail the patch to the maintainers of the subsystem that your patch is dealing with.

  - Add the mailing list of project 'Y' to the cc list as well.

  - Add the email id of your peer developers within LTC.

Changelogs

- Changelogs could vary in their depth and length depending on what the patch is about.
- If the patch is as trivial as removing a redundant variable, a two line changelog mentioning this would be good enough.

  Eg: https://lkml.org/lkml/2013/8/16/134
- If the patch is a change in some feature which results in better performance, the changelog needs to have the results showing this.

  Eg: https://lkml.org/lkml/2013/8/12/78
- If the patch is a proof of concept, introducing a new feature, it needs to be quite elaborate, but to the point.

  Eg: https://lkml.org/lkml/2013/8/14/240

Best practices while developing a patch

- Your patch will be reviewed by interested developers and maintainers. Respect the challenge of having to review hundreds of patches each day by the maintainers.
- Keep your Changelogs as brief, clear and relevant as possible. Changelogs matter more than the patch itself!
- Before you mail your patch, rebase it on the master branch or mention the commit id of the master branch on top of which your patch applies.

Best practices while developing a patch

- Coding practices are essential in software development. Why is it all the more important in open source development?

  - The growth of open source projects cannot be stopped, more developers come, if they develop good code, it has to go in.

  - The open source projects are bound to get bigger. We cannot then afford non-uniform coding style due to the following reasons:

    a. A new developer who can start development any day in any part of the world should not find the code hard to understand.

    b. An experienced developer who has been on the project for years now, should not look at code 5 years old and find it hard to understand.

- Example of the coding styles documented for linux kernel development. https://www.kernel.org/doc/Documentation/CodingStyle

Best practices while developing a patch

- Coding style, good changelogs and well thought out patches prevent back and forth of emails between the maintainer and the developer.

  By making the maintainer pick out the wrong coding practices or asking you to explain your changelog better, you are delaying entry of your patch into project 'Y'!

- The end result of following the best practices is that you end up having discussions with the maintainers and developers about the more important code design and implementation of yours and not about the already documented coding practices.

  Eg: https://lkml.org/lkml/2013/3/29/90 was a redundant exchange of mail and waster of the maintainer's time.

Posting a patch'set'

- Developing feature 'X' as one single patch will make the patch complex and long to review, most of the time.
- Consider splitting the patch into multiple patches.
- When you develop multiple patches, make sure that each patch has a solid purpose and should follow a flow like:

  patch1 solves problem1 of feature 'X' but not problem2; patch 2 therefore is brought in to solve problem2

  or

  patch1 creates a framework to be used by patch2

  i.e. Each patch should make way for the next in the patchset
- Each patch in the patchset should nevertheless still have a changlelog of its own.
- The patchset should be accompanied by a cover letter, which briefly describes all the patches.

  Eg: of a patchset: https://lkml.org/lkml/2013/4/9/675

When to tag a patch with an RFC?

- If the patch you are developing is a bug fix or is a code cleanup, the patch need not have an RFC tag.
- If the patch you are mailing out has gone through multiple reviews, and the developers and maintainers are happy with the current form of the patch, it need not have an RFC tag.
- If your patch requires developer's reviews and suggestions tag it with RFC.
- If your patch has not solved all the issues pointed out by the community, but solves a few of them alone, tag it with an RFC.
- If the first posting of your patch requires optimization that you , yourself believe needs to be there but would take care of it later, tag it with an RFC.

Best email practices in open source development

- Once the patch has been posted, you will receive reviews, comments from the developers and maintainers over a few days after you have posted. Ensure that you reply to them as soon as possible.
- You may not have the answer to all their questions right away, but reply back saying that you will think about it and get back.
- You may have a very strong answer supporting your patch. Debate it out at a technical level and not at a personal level.

Best email practices in open source development

- You may realize that the suggestions are actually better than your present implementation of the patch. Acknowledge it and state that your next version will have their suggestions incorporated.
- Wait for a few days for all the reviews to subside, then work on V2 of your patch. This can take time, its no problem.
- The V2 must be significantly different from V1, and must have solved all the issues that you had promised to solve in V1.

Best email practices in open source development

- The patch may be too long. And your reply relevant to some small function in the patch.

  Snip out the rest of the patch, leaving just that function while replying. Some developers get annoyed if they find a one line in-lined reply to a 100 line email!

  Eg: http://lists-archives.com/linux-kernel/27884933-sched-factor-out-code-to-should_we_balance.html
- Sometimes you may have to review patches. All your reviews and replies must not exceed around 80 characters in a single line, else it is highly annoying for readers! Enable the word wrap feature in your email client.
- Sometimes you will need to attach a patch in your replies. Disable the word wrap feature for the patch, else the patch gets mangled.

Best email practices in open source development

- Read the mailing list of project 'Y' regularly. This is to be aware of the latest happenings in project Y, to observe how people interact over patches and also for you to provide your reviews on others patches.
- Any conversation, apart from the patch itself that you have with the open source community is a purely technical discussion in plain English. This is no different from how we discuss technical issues with each other :) So do not hesitate while discussing in the community. The goal is to make our points clear.
- The open source community works in collaboration. Hence seek help, provide your assistance and acknowledge the contribution of others during the development activities.

**The last demo is on stg mail and patch import.

Backup Slides

Get project 'Y' on your local workstation

- Project 'Y' would be hosted as a git tree on a website.

  Eg:

  http://github.com/rjwysocki/linux-pm.git
  http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
- It can have multiple development branches, choose the branch that is relevant to your work. Eg: master branch, bleeding edge branch, next branch.
- Pull the branch you chose onto your local workstation.
- It is best to have your development branch independent of the branch you pulled.

Summary of the steps

- git remote add origin http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
- git fetch origin
- git branch –track master origin
- git checkout master
- git branch test
- git checkout test

On your local workstation you would now have 2 branches: