Maintenance of a stable kernel branch

2012/09/21

Willy Tarreau - Exceliance
ALOHA R&D
<wtarreau@exceliance.fr>
http://www.exceliance.fr/

Who am I ?

I'm the maintainer of various old stable kernel branches :

- 2.4 since 2006
- 2.6.27 since 2010
- 2.6.32 since 2012

What's a stable kernel ?

- Fork of the mainline kernel which aims at being more reliable
- Overlap between 2 versions ensures users are not stressed to upgrade.
- What is merged is dictated by several rules in `stable_kernel_rules.txt`:
  - Only small reviewable patches (<100 lines)
  - No new features added
  - fix real bugs that bother people
  - minor driver updates are accepted (eg: support for new PCI IDs)
  - everything merged must already be upstream

What's a longterm supported kernel (LTS) ?

- stable kernel branch maintained longer (between 18 and 24 months)
- originally designed to make distro maintainers' life easier
- rate of patches progressively drops for 2 years and drops faster afterwards
- kernels 2.6.16, 2.6.27, 2.6.32, 3.0 are LTS

What stable and LTS kernels are used for ?

Stable kernels are made to ensure users can update with limited risk, so that they will eventually get rid of known bugs, resulting in a better overall Linux kernel quality in deployments.

- low risk of regression within the same branch means no excuse for not upgrading
- accelerates mainline releases by removing some last minute stress
- useful common base for long or experimental development

Who uses stable kernels and why ?

- stable kernels are used by all those who want reliability and not to bother with bug reports
- prevalent in the areas of servers and embedded devices that are not easily upgradable
- some distros use them and contribute back to them by pointing commit IDs to backport

What happens at the end of a stable cycle ?

For the maintainers :

- no more updates published for the given branch
- ... except the LTS ones that are picked for extended support

For the user :

- mandatory upgrade to next stable kernel, unless running on extended support, where upgrade is just recommended

What's the EOL expectation on these extended LTS kernels ?

2.4, 2.6.27 and 2.6.32 are currently extended LTS kernels

- no identified EOL yet
- 2.4 has no more releases (last 12/2010) but fixes still published (04/2012)
- relatively low maintenance work when most patches are not relevant anymore.

How much work is it to maintain such kernels ?

The amount of work is manageable due to the low frequency of updates :

- generally no emergency to publish fixes since users have to qualify fixes and planify reboots
- bugs generally have been present for so long that they can still wait (eg: leap second+futex)
- many patches are skipped because reverted later or amended
- such a task is improved by batch processing (eg: dedicate a whole week-end)
- some backports are harder to make, find limit between fix and sabotage
- fixing may sometimes be too risky so we prefer to document the bug instead (eg: unix socket local DoS in 2.4)

How did I get involved with this task ?

- long-time need for stable kernels for professional and personal usage
- maintenance of the unofficial "hotfix" tree in parallel to mainline 2.4 in early 2005
- takeover of 2.4.33 maintainership in 2006
- first attempts at extending 2.6.20 and 2.6.25 with Greg's help
- official takeover of 2.6.27, then 2.6.32, the two last LTS reaching EOL

Why I do this

- very interesting and enriching
- allows me to use ultra-stable kernels that have been gathering fixes for 2 years
- allows my company to safely ship products which we know customers will refuse to upgrade for years
- not that much an amount of work, only difficulty is to find contiguous time
- participates to Linux deployment on amazing and unexpected devices which cannot always be upgraded
- and... because Greg is really helpful and users are forgiving !

What the process looks like

Patches are backported from next LTS so that we can guarantee that a user upgrading will not meet the same unfixed bug again. 2.6.32 currently backports from 3.0.

1. Enumerate all patches that were added to 3.0 since last backport session
2. Review all of them one at a time (between 600 and 1000 each time). This is the part which requires a full week-end.
3. Only those which apply, are easy to fix, were explicitly asked for or were the result of a relevant security alert are considered.
4. Remaining ones are ignored, but can be merged upon request
5. Resulting kernel is tested on several archs and configs.

Step 1/6 : enumerating new patches

This process can take a full day!

- `git log x..y` sent to a large file
- All commit messages are read. Think about it next time, detailed commit descriptions are absolutely essential
- Patches are tagged by hand in the log file
- Some scripts help picking them out from the long log

Step 2/6 : integrating patches

This process generally takes a full day too !

- Patches are processed by moving them between queues
- `quilt` is used for applying fixes. Not easy to get first but quite helpful.
- What does not apply is either fixed or quickly moved to a "`failed`" queue
- About 10% of patches generally rely on many other patches. Some quick arbitration needed.
- Sometimes I ask for help (lack of reproducer or code that I don't understand). All subsystem maintainers have always been very helpful with this task. Some of them are amazingly careful about stable kernels.

Step 3/6 : testing

This is the step which causes me small eyes on Monday morning

- Kernels are built for at least 2 architectures, ideally 3 (i386, x86_64, arm) with one or two configs each.
- Many times things go wrong (tests are really useful)
- Many times I don't have enough time to retest all fixes, so some breakage happens...

Step 4/6 : preparing for the public review process

This is the fastest but the most stressful step.

- `quilt mail` is used for this, along with a huge amount of ugly and fragile scripts
- It's easy to get something wrong, processing mboxes with utilities is never fun
- mistakes are generally cascaded into older branches at the same time
- Last review by hand with mutt
- When everything *looks* OK, then send to the world using formail.
- No global -rc patch is provided with these kernels, as end-users don't test them.
- Note: be very careful with `git commit` when fixing rejects, it can make you the apparent author of someone else's commit.

Step 5/6 : peer review and fixes

- This step lasts a few days so that patch authors can suggest missing fixes, removing undesired ones or report an incorrect backport.
- The patch queue is then adjusted based on this feedback.
- Another review might be fired if too many patches were changed.
- Very few users test these patches, so feedback may come much later than the deadline.

Step 6/6 : release

This is the easiest part.

- One or two builds/boots are generally performed if patches were changed from the preview
- A few minutes of scripts and the git tree is ready and tagged
- Kernel tree is pushed to "for-greg" tree on kernel.org
- I ping Greg who pulls/pushes to the final location => kernel is released
- I then feel much better for a few months !

Step 7/6 : release again :-)

It generally happens that a release follows another one because there are more testers, and more build breakage is discovered, sometimes up to 3 months later!

Conclusion

- Interesting and fun to do if you know your limits
- Unfortunately needs more contiguous time than I can devote at the moment
- Needs lots of scripts to avoid biggest errors and to save time
- Needs real support for the community (bug reporters, subsystem maintainers, testers)
- You know users are satisfied when they start bringing patches themselves

Thank you !

Questions ?