Handling regressions

We don't cause regressions -- this document describes what this "first rule of Linux kernel development" means in practice for developers. It complements Documentation/admin-guide/reporting-regressions.rst, which covers the topic from a user's point of view; if you never read that text, go and at least skim over it before continuing here.

The important bits (aka "The TL;DR")

1. Ensure subscribers of the regression mailing list (regressions@lists.linux.dev) quickly become aware of any new regression report:

   - When receiving a mailed report that did not CC the list, bring it into the loop by immediately sending at least a brief "Reply-all" with the list CCed.
   - Forward or bounce any reports submitted in bug trackers to the list.

2. Make the Linux kernel regression tracking bot "regzbot" track the issue (this is optional, but recommended):

   - For mailed reports, check if the reporter included a line like `#regzbot introduced v5.13..v5.14-rc1`. If not, send a reply (with the regressions list in CC) containing a paragraph like the following, which tells regzbot when the issue started to happen:

     ```
     #regzbot ^introduced 1f2e3d4c5b6a
     ```

   - When forwarding reports from a bug tracker to the regressions list (see above), include a paragraph like the following:

     ```
     #regzbot introduced: v5.13..v5.14-rc1
     #regzbot from: Some N. Ice Human <some.human@example.com>
     #regzbot monitor: http://some.bugtracker.example.com/ticket?id=123456789
     ```

3. When submitting fixes for regressions, add "Link:" tags to the patch description pointing to all places where the issue was reported, as mandated by Documentation/process/submitting-patches.rst and :ref:`Documentation/process/5.Posting.rst <development_posting>`.

   System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\(linux-master) (Documentation) (process)handling-regressions.rst`, line 43); *backlink*

   Unknown interpreted text role "ref".

4. Try to fix regressions quickly once the culprit has been identified; fixes for most regressions should be merged within two weeks, but some need to be resolved within two or three days.

All the details on Linux kernel regressions relevant for developers

The important basics in more detail

What to do when receiving regression reports

Ensure the Linux kernel's regression tracker and others subscribers of the regression mailing list (regressions@lists.linux.dev) become aware of any newly reported regression:

- When you receive a report by mail that did not CC the list, immediately bring it into the loop by sending at least a brief "Reply-all" with the list CCed; try to ensure it gets CCed again in case you reply to a reply that omitted the list.
- If a report submitted in a bug tracker hits your Inbox, forward or bounce it to the list. Consider checking the list archives beforehand, if the reporter already forwarded the report as instructed by Documentation/admin-guide/reporting-issues.rst.

When doing either, consider making the Linux kernel regression tracking bot "regzbot" immediately start tracking the issue:

- For mailed reports, check if the reporter included a "regzbot command" like `#regzbot introduced 1f2e3d4c5b6a`. If not, send a reply (with the regressions list in CC) with a paragraph like the following::

  ```
  #regzbot ^introduced: v5.13..v5.14-rc1
  ```

  This tells regzbot the version range in which the issue started to happen; you can specify a range using commit-ids as well or state a single commit-id in case the reporter bisected the culprit.

Note the caret (^) before the "introduced": it tells regzbot to treat the parent mail (the one you reply to) as the initial report for the regression you want to see tracked; that's important, as regzbot will later look out for patches with "Link:" tags pointing to the report in the archives on lore.kernel.org.

- When forwarding a regressions reported to a bug tracker, include a paragraph with these regzbot commands:

  ```
  #regzbot introduced: 1f2e3d4c5b6a
  #regzbot from: Some N. Ice Human <some.human@example.com>
  #regzbot monitor: http://some.bugtracker.example.com/ticket?id=123456789
  ```

  Regzbot will then automatically associate patches with the report that contain "Link:" tags pointing to your mail or the mentioned ticket.

What's important when fixing regressions

You don't need to do anything special when submitting fixes for regression, just remember to do what Documentation/process/submitting-patches.rst, :ref:`Documentation/process/5.Posting.rst <development_posting>`, and Documentation/process/stable-kernel-rules.rst already explain in more detail:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\(linux-master) (Documentation) (process)handling-regressions.rst`, line 110); *backlink*

Unknown interpreted text role "ref".

- Point to all places where the issue was reported using "Link:" tags:

  ```
  Link: https://lore.kernel.org/r/30th.anniversary.repost@klaava.Helsinki.FI/
  Link: https://bugzilla.kernel.org/show_bug.cgi?id=1234567890
  ```

- Add a "Fixes:" tag to specify the commit causing the regression.
- If the culprit was merged in an earlier development cycle, explicitly mark the fix for backporting using the `Cc: stable@vger.kernel.org` tag.

All this is expected from you and important when it comes to regression, as these tags are of great value for everyone (you included) that might be looking into the issue weeks, months, or years later. These tags are also crucial for tools and scripts used by other kernel developers or Linux distributions; one of these tools is regzbot, which heavily relies on the "Link:" tags to associate reports for regression with changes resolving them.

Prioritize work on fixing regressions

You should fix any reported regression as quickly as possible, to provide affected users with a solution in a timely manner and prevent more users from running into the issue; nevertheless developers need to take enough time and care to ensure regression fixes do not cause additional damage.

In the end though, developers should give their best to prevent users from running into situations where a regression leaves them only three options: "run a kernel with a regression that seriously impacts usage", "continue running an outdated and thus potentially insecure kernel version for more than two weeks after a regression's culprit was identified", and "downgrade to a still supported kernel series that lack required features".

How to realize this depends a lot on the situation. Here are a few rules of thumb for you, in order or importance:

- Prioritize work on handling regression reports and fixing regression over all other Linux kernel work, unless the latter concerns acute security issues or bugs causing data loss or damage.
- Always consider reverting the culprit commits and reapplying them later together with necessary fixes, as this might be the least dangerous and quickest way to fix a regression.
- Developers should handle regressions in all supported kernel series, but are free to delegate the work to the stable team, if the issue probably at no point in time occurred with mainline.
- Try to resolve any regressions introduced in the current development before its end. If you fear a fix might be too risky to apply only days before a new mainline release, let Linus decide: submit the fix separately to him as soon as possible with the explanation of the situation. He then can make a call and postpone the release if necessary, for example if multiple such changes show up in his inbox.
- Address regressions in stable, longterm, or proper mainline releases with more urgency than regressions in mainline pre-releases. That changes after the release of the fifth pre-release, aka "-rc5": mainline then becomes as important, to ensure all the improvements and fixes are ideally tested together for at least one week before Linus releases a new mainline version.
- Fix regressions within two or three days, if they are critical for some reason -- for example, if the issue is likely to affect many users of the kernel series in question on all or certain architectures. Note, this includes mainline, as

issues like compile errors otherwise might prevent many testers or continuous integration systems from testing the series.

- Aim to fix regressions within one week after the culprit was identified, if the issue was introduced in either:
  - a recent stable/longterm release
  - the development cycle of the latest proper mainline release

  In the latter case (say Linux v5.14), try to address regressions even quicker, if the stable series for the predecessor (v5.13) will be abandoned soon or already was stamped "End-of-Life" (EOL) -- this usually happens about three to four weeks after a new mainline release.
- Try to fix all other regressions within two weeks after the culprit was found. Two or three additional weeks are acceptable for performance regressions and other issues which are annoying, but don't prevent anyone from running Linux (unless it's an issue in the current development cycle, as those should ideally be addressed before the release). A few weeks in total are acceptable if a regression can only be fixed with a risky change and at the same time is affecting only a few users; as much time is also okay if the regression is already present in the second newest longterm kernel series.

Note: The aforementioned time frames for resolving regressions are meant to include getting the fix tested, reviewed, and merged into mainline, ideally with the fix being in linux-next at least briefly. This leads to delays you need to account for.

Subsystem maintainers are expected to assist in reaching those periods by doing timely reviews and quick handling of accepted patches. They thus might have to send git-pull requests earlier or more often than usual; depending on the fix, it might even be acceptable to skip testing in linux-next. Especially fixes for regressions in stable and longterm kernels need to be handled quickly, as fixes need to be merged in mainline before they can be backported to older series.

More aspects regarding regressions developers should be aware of

How to deal with changes where a risk of regression is known

Evaluate how big the risk of regressions is, for example by performing a code search in Linux distributions and Git forges. Also consider asking other developers or projects likely to be affected to evaluate or even test the proposed change; if problems surface, maybe some solution acceptable for all can be found.

If the risk of regressions in the end seems to be relatively small, go ahead with the change, but let all involved parties know about the risk. Hence, make sure your patch description makes this aspect obvious. Once the change is merged, tell the Linux kernel's regression tracker and the regressions mailing list about the risk, so everyone has the change on the radar in case reports trickle in. Depending on the risk, you also might want to ask the subsystem maintainer to mention the issue in his mainline pull request.

What else is there to known about regressions?

Check out Documentation/admin-guide/reporting-regressions.rst, it covers a lot of other aspects you want might want to be aware of:

- the purpose of the "no regressions rule"
- what issues actually qualify as regression
- who's in charge for finding the root cause of a regression
- how to handle tricky situations, e.g. when a regression is caused by a security fix or when fixing a regression might cause another one

Whom to ask for advice when it comes to regressions

Send a mail to the regressions mailing list (regressions@lists.linux.dev) while CCing the Linux kernel's regression tracker (regressions@leemhuis.info); if the issue might better be dealt with in private, feel free to omit the list.

More about regression tracking and regzbot

Why the Linux kernel has a regression tracker, and why is regzbot used?

Rules like "no regressions" need someone to ensure they are followed, otherwise they are broken either accidentally or on purpose. History has shown this to be true for the Linux kernel as well. That's why Thorsten Leemhuis volunteered to keep an eye on things as the Linux kernel's regression tracker, who's occasionally helped by other people. Neither of them are paid to do this, that's why regression tracking is done on a best effort basis.

Earlier attempts to manually track regressions have shown it's an exhausting and frustrating work, which is why they were abandoned after a while. To prevent this from happening again, Thorsten developed regzbot to facilitate the work, with the long term goal to automate regression tracking as much as possible for everyone involved.

How does regression tracking work with regzbot?

The bot watches for replies to reports of tracked regressions. Additionally, it's looking out for posted or committed patches referencing such reports with "Link:" tags; replies to such patch postings are tracked as well. Combined this data provides good insights into the current state of the fixing process.

Regzbot tries to do its job with as little overhead as possible for both reporters and developers. In fact, only reporters are burdened with an extra duty: they need to tell regzbot about the regression report using the `#regzbot introduced` command outlined above; if they don't do that, someone else can take care of that using `#regzbot ^introduced`.

For developers there normally is no extra work involved, they just need to make sure to do something that was expected long before regzbot came to light: add "Link:" tags to the patch description pointing to all reports about the issue fixed.

Do I have to use regzbot?

It's in the interest of everyone if you do, as kernel maintainers like Linus Torvalds partly rely on regzbot's tracking in their work -- for example when deciding to release a new version or extend the development phase. For this they need to be aware of all unfixed regression; to do that, Linus is known to look into the weekly reports sent by regzbot.

Do I have to tell regzbot about every regression I stumble upon?

Ideally yes: we are all humans and easily forget problems when something more important unexpectedly comes up -- for example a bigger problem in the Linux kernel or something in real life that's keeping us away from keyboards for a while. Hence, it's best to tell regzbot about every regression, except when you immediately write a fix and commit it to a tree regularly merged to the affected kernel series.

How to see which regressions regzbot tracks currently?

Check regzbot's web-interface for the latest info; alternatively, search for the latest regression report, which regzbot normally sends out once a week on Sunday evening (UTC), which is a few hours before Linus usually publishes new (pre-)releases.

What places is regzbot monitoring?

Regzbot is watching the most important Linux mailing lists as well as the git repositories of linux-next, mainline, and stable/longterm.

What kind of issues are supposed to be tracked by regzbot?

The bot is meant to track regressions, hence please don't involve regzbot for regular issues. But it's okay for the Linux kernel's regression tracker if you use regzbot to track severe issues, like reports about hangs, corrupted data, or internal errors (Panic, Oops, BUG(), warning, ...).

Can I add regressions found by CI systems to regzbot's tracking?

Feel free to do so, if the particular regression likely has impact on practical use cases and thus might be noticed by users; hence, please don't involve regzbot for theoretical regressions unlikely to show themselves in real world usage.

How to interact with regzbot?

By using a 'regzbot command' in a direct or indirect reply to the mail with the regression report. These commands need to be in their own paragraph (IOW: they need to be separated from the rest of the mail using blank lines).

One such command is `#regzbot introduced <version or commit>`, which makes regzbot consider your mail as a regressions report added to the tracking, as already described above; `#regzbot ^introduced <version or commit>` is another such command, which makes regzbot consider the parent mail as a report for a regression which it starts to track.

Once one of those two commands has been utilized, other regzbot commands can be used in direct or indirect replies to the report. You can write them below one of the *introduced* commands or in replies to the mail that used one of them or itself is a reply to that mail:

- Set or update the title:

  ```
  #regzbot title: foo
  ```

- Monitor a discussion or bugzilla.kernel.org ticket where additions aspects of the issue or a fix are discussed -- for example the posting of a patch fixing the regression:

  ```
  #regzbot monitor: https://lore.kernel.org/all/30th.anniversary.repost@klaava.Helsinki.FI/
  ```

  Monitoring only works for lore.kernel.org and bugzilla.kernel.org; regzbot will consider all messages in that thread or ticket as related to the fixing process.

- Point to a place with further details of interest, like a mailing list post or a ticket in a bug tracker that are slightly related, but about a different topic:

  ```
  #regzbot link: https://bugzilla.kernel.org/show_bug.cgi?id=123456789
  ```

- Mark a regression as fixed by a commit that is heading upstream or already landed:

```
#regzbot fixed-by: 1f2e3d4c5d
```

- Mark a regression as a duplicate of another one already tracked by regzbot:

```
#regzbot dup-of: https://lore.kernel.org/all/30th.anniversary.repost@klaava.Helsinki.FI/
```

- Mark a regression as invalid:

```
#regzbot invalid: wasn't a regression, problem has always existed
```

Is there more to tell about regzbot and its commands?

More detailed and up-to-date information about the Linux kernel's regression tracking bot can be found on its project page, which among others contains a getting started guide and reference documentation which both cover more details than the above section.

Quotes from Linus about regression

Find below a few real life examples of how Linus Torvalds expects regressions to be handled:

- From 2017-10-26 (1/2):

```
If you break existing user space setups THAT IS A REGRESSION.

It's not ok to say "but we'll fix the user space setup".

Really. NOT OK.

[...]

The first rule is:

 - we don't cause regressions

and the corollary is that when regressions *do* occur, we admit to
them and fix them, instead of blaming user space.

The fact that you have apparently been denying the regression now for
three weeks means that I will revert, and I will stop pulling apparmor
requests until the people involved understand how kernel development
is done.
```

- From 2017-10-26 (2/2):

```
People should basically always feel like they can update their kernel
and simply not have to worry about it.

I refuse to introduce "you can only update the kernel if you also
update that other program" kind of limitations. If the kernel used to
work for you, the rule is that it continues to work for you.

There have been exceptions, but they are few and far between, and they
generally have some major and fundamental reasons for having happened,
that were basically entirely unavoidable, and people _tried_hard_ to
avoid them. Maybe we can't practically support the hardware any more
after it is decades old and nobody uses it with modern kernels any
more. Maybe there's a serious security issue with how we did things,
and people actually depended on that fundamentally broken model. Maybe
there was some fundamental other breakage that just _had_ to have a
flag day for very core and fundamental reasons.

And notice that this is very much about *breaking* peoples environments.

Behavioral changes happen, and maybe we don't even support some
feature any more. There's a number of fields in /proc/<pid>/stat that
are printed out as zeroes, simply because they don't even *exist* in
the kernel any more, or because showing them was a mistake (typically
an information leak). But the numbers got replaced by zeroes, so that
the code that used to parse the fields still works. The user might not
see everything they used to see, and so behavior is clearly different,
but things still _work_, even if they might no longer show sensitive
(or no longer relevant) information.

But if something actually breaks, then the change must get fixed or
reverted. And it gets fixed in the *kernel*. Not by saying "well, fix
your user space then". It was a kernel change that exposed the
problem, it needs to be the kernel that corrects for it, because we
have a "upgrade in place" model. We don't have a "upgrade with new
user space".

And I seriously will refuse to take code from people who do not
understand and honor this very simple rule.
```

```
This rule is also not going to change.

And yes, I realize that the kernel is "special" in this respect. I'm
proud of it.

I have seen, and can point to, lots of projects that go "We need to
break that use case in order to make progress" or "you relied on
undocumented behavior, it sucks to be you" or "there's a better way to
do what you want to do, and you have to change to that new better
way", and I simply don't think that's acceptable outside of very early
alpha releases that have experimental users that know what they signed
up for. The kernel hasn't been in that situation for the last two
decades.

We do API breakage _inside_ the kernel all the time. We will fix
internal problems by saying "you now need to do XYZ", but then it's
about internal kernel API's, and the people who do that then also
obviously have to fix up all the in-kernel users of that API. Nobody
can say "I now broke the API you used, and now _you_ need to fix it
up". Whoever broke something gets to fix it too.

And we simply do not break user space.
```

- From 2020-05-21:

```
The rules about regressions have never been about any kind of
documented behavior, or where the code lives.

The rules about regressions are always about "breaks user workflow".

Users are literally the _only_ thing that matters.

No amount of "you shouldn't have used this" or "that behavior was
undefined, it's your own fault your app broke" or "that used to work
simply because of a kernel bug" is at all relevant.

Now, reality is never entirely black-and-white. So we've had things
like "serious security issue" etc that just forces us to make changes
that may break user space. But even then the rule is that we don't
really have other options that would allow things to continue.

And obviously, if users take years to even notice that something
broke, or if we have sane ways to work around the breakage that
doesn't make for too much trouble for users (ie "ok, there are a
handful of users, and they can use a kernel command line to work
around it" kind of things) we've also been a bit less strict.

But no, "that was documented to be broken" (whether it's because the
code was in staging or because the man-page said something else) is
irrelevant. If staging code is so useful that people end up using it,
that means that it's basically regular kernel code with a flag saying
"please clean this up".

The other side of the coin is that people who talk about "API
stability" are entirely wrong. API's don't matter either. You can make
any changes to an API you like - as long as nobody notices.

Again, the regression rule is not about documentation, not about
API's, and not about the phase of the moon.

It's entirely about "we caused problems for user space that used to work".
```

- From 2017-11-05:

```
And our regression rule has never been "behavior doesn't change".
That would mean that we could never make any changes at all.

For example, we do things like add new error handling etc all the
time, which we then sometimes even add tests for in our kselftest
directory.

So clearly behavior changes all the time and we don't consider that a
regression per se.

The rule for a regression for the kernel is that some real user
workflow breaks. Not some test. Not a "look, I used to be able to do
X, now I can't".
```

- From 2018-08-03:

```
YOU ARE MISSING THE #1 KERNEL RULE.
```

```
We do not regress, and we do not regress exactly because your are 100% wrong.

And the reason you state for your opinion is in fact exactly *WHY* you
are wrong.

Your "good reasons" are pure and utter garbage.

The whole point of "we do not regress" is so that people can upgrade
the kernel and never have to worry about it.

> Kernel had a bug which has been fixed

That is *ENTIRELY* immaterial.

Guys, whether something was buggy or not DOES NOT MATTER.

Why?

Bugs happen. That's a fact of life. Arguing that "we had to break
something because we were fixing a bug" is completely insane. We fix
tens of bugs every single day, thinking that "fixing a bug" means that
we can break something is simply NOT TRUE.

So bugs simply aren't even relevant to the discussion. They happen,
they get found, they get fixed, and it has nothing to do with "we
break users".

Because the only thing that matters IS THE USER.

How hard is that to understand?

Anybody who uses "but it was buggy" as an argument is entirely missing
the point. As far as the USER was concerned, it wasn't buggy - it
worked for him/her.

Maybe it worked *because* the user had taken the bug into account,
maybe it worked because the user didn't notice - again, it doesn't
matter. It worked for the user.

Breaking a user workflow for a "bug" is absolutely the WORST reason
for breakage you can imagine.

It's basically saying "I took something that worked, and I broke it,
but now it's better". Do you not see how f*cking insane that statement
is?

And without users, your program is not a program, it's a pointless
piece of code that you might as well throw away.

Seriously. This is *why* the #1 rule for kernel development is "we
don't break users". Because "I fixed a bug" is absolutely NOT AN
ARGUMENT if that bug fix broke a user setup. You actually introduced a
MUCH BIGGER bug by "fixing" something that the user clearly didn't
even care about.

And dammit, we upgrade the kernel ALL THE TIME without upgrading any
other programs at all. It is absolutely required, because flag-days
and dependencies are horribly bad.

And it is also required simply because I as a kernel developer do not
upgrade random other tools that I don't even care about as I develop
the kernel, and I want any of my users to feel safe doing the same
time.

So no. Your rule is COMPLETELY wrong. If you cannot upgrade a kernel
without upgrading some other random binary, then we have a problem.
```

- From 2021-06-05:

```
THERE ARE NO VALID ARGUMENTS FOR REGRESSIONS.

Honestly, security people need to understand that "not working" is not
a success case of security. It's a failure case.

Yes, "not working" may be secure. But security in that case is *pointless*.
```

- From 2011-05-06 (1/3):

```
Binary compatibility is more important.

And if binaries don't use the interface to parse the format (or just
parse it wrongly - see the fairly recent example of adding uuid's to
/proc/self/mountinfo), then it's a regression.

And regressions get reverted, unless there are security issues or
similar that makes us go "Oh Gods, we really have to break things".

I don't understand why this simple logic is so hard for some kernel
developers to understand. Reality matters. Your personal wishes matter
NOT AT ALL.

If you made an interface that can be used without parsing the
interface description, then we're stuck with the interface. Theory
simply doesn't matter.

You could help fix the tools, and try to avoid the compatibility
issues that way. There aren't that many of them.
```

From 2011-05-06 (2/3):

```
it's clearly NOT an internal tracepoint. By definition. It's being
used by powertop.
```

From 2011-05-06 (3/3):

```
We have programs that use that ABI and thus it's a regression if they break.
```

- From 2012-07-06:

```
> Now this got me wondering if Debian _unstable_ actually qualifies as a
> standard distro userspace.

Oh, if the kernel breaks some standard user space, that counts. Tons
of people run Debian unstable
```

- From 2019-09-15:

```
One _particularly_ last-minute revert is the top-most commit (ignoring
the version change itself) done just before the release, and while
it's very annoying, it's perhaps also instructive.

What's instructive about it is that I reverted a commit that wasn't
actually buggy. In fact, it was doing exactly what it set out to do,
and did it very well. In fact it did it _so_ well that the much
improved IO patterns it caused then ended up revealing a user-visible
regression due to a real bug in a completely unrelated area.

The actual details of that regression are not the reason I point that
revert out as instructive, though. It's more that it's an instructive
example of what counts as a regression, and what the whole "no
regressions" kernel rule means. The reverted commit didn't change any
API's, and it didn't introduce any new bugs. But it ended up exposing
another problem, and as such caused a kernel upgrade to fail for a
user. So it got reverted.

The point here being that we revert based on user-reported _behavior_,
not based on some "it changes the ABI" or "it caused a bug" concept.
The problem was really pre-existing, and it just didn't happen to
trigger before. The better IO patterns introduced by the change just
happened to expose an old bug, and people had grown to depend on the
previously benign behavior of that old issue.

And never fear, we'll re-introduce the fix that improved on the IO
patterns once we've decided just how to handle the fact that we had a
bad interaction with an interface that people had then just happened
to rely on incidental behavior for before. It's just that we'll have
to hash through how to do that (there are no less than three different
patches by three different developers being discussed, and there might
be more coming...). In the meantime, I reverted the thing that exposed
the problem to users for this release, even if I hope it will be
re-introduced (perhaps even backported as a stable patch) once we have
consensus about the issue it exposed.

Take-away from the whole thing: it's not about whether you change the
kernel-userspace ABI, or fix a bug, or about whether the old code
"should never have worked in the first place". It's about whether
something breaks existing users' workflow.

Anyway, that was my little aside on the whole regression thing.  Since
it's that "first rule of kernel programming", I felt it is perhaps
worth just bringing it up every once in a while
```