Live blog day 1: morning

by Anisse | Jun 1, 2022 | 0 comments

Let's start this 9th edition of Kernel Recipes !

We are happy to welcome you after three years of waiting. Jens Axboe is the godfather for this edition and will also be our first speaker in the morning.

This is the live stream from those who prefer video over text.

What's new with io_uring – Jens Axboe

First of all, what's io_uring ? It's a way for an application to talk with kernel. It's composed of two ring buffers, each doing communication in a single direction.

Underneath, it uses three io_uring_* syscalls. The main goal of io_uring was to replace AIO for async I/O. Its key features are being async, zero-copy and lockless. It's intended to be used through liburing instead of the kernel interface directly.

Jens Axboe at Kernel Recipes 2022

What's new ?

Native workers are now really native: the kernel jobs doing tasks on behalf of the application now use io-threads instead of dangerously trying to assume the apps credentials: it fixed a few security issues and corner cases at the same time.

`TIF_NOTIFY_SIGNAL` is also a new feature to better handle the way signals are sent to io_uring tasks. It was a lot of work to do because it's architecture dependent.

Direct descriptors are a new type of file descriptors that only exist within a ring: they are faster than using file descriptor (less locking), and enable having multiple operations on a file. A more recent change upcoming in 5.19, is to let the kernel manage the direct descriptors instead of having the application do it.

What about bypassing fget/fput locking on opening the ring itself ? It's now possible as well, albeit a bit dangerous to use improperly.

io_uring can also now manage a buffer pool provided by the application to prevent passing a new buffer for each read/write call: a buffer will be picked from the provided buffers. Also in 5.19, there are now ring provided buffers; they are provided by the kernel instead of the application for better performance, but cannot be used at the same time as classic provided buffers; liburing usually takes care of those details.

It's now possible to do custom asynchronous commands in drivers thanks to the `->uring_cmd()` addition. It is used for example for NVME vendor commands to replace what would usually be done in synchronous `ioctl`s.

Cooperative completion scheduling has been added and helps a bit for a few network tasks. For 5.19, cancellations have been improved to match more conditions. Multishot accept requests were also added for accepting many connections asynchronously.

Apart from this, there were many performance optimizations over the last releases.

liburing 2.2 will be synchronized with 5.19 and have all the new features. It now has 80 manpages instead of 8, and more regression tests.

Microsoft added support for "I/O Rings", a design very similar to io_uring. It should open the road to write cross-platform applications with ring I/O.

Of course, this does not stop here: there are still many upcoming improvements queued for 5.20 and planned in the future: true async buffered writes, level triggered poll support, faster io-wq, incrementally consumed provided buffers; and the code split in the kernel for the 13k+ lines `fs/io_uring.c` file.

And Jens says that it might take a long time to move to the Completion-based model of io_uring. It's a long term, 10-year long project.

Make Linux developers fix your kernel bug – Thorsten Leemhuis

The Linux kernel is made by volunteers. It does not necessarily mean hobbyists; even if they are paid, it means that you can't really force anyone to work on a bug.

Most people want to work on building what they care about, but sometimes Linus Torvalds might need to step in if things are badly broken.

In addition, most developers will gladly address issues in their code; but life sometimes gets in the way. You can help by writing decent bug reports: it makes fixing bugs much easier.

Thorsten Leemhuis

How to create a decent report ?

The first thing is to ensure that your kernel is *vanilla*. This is not the case for most kernels in the wild: they are built by distros, which makes them unsuitable for reporting issues to Linux maintainers. In this case you should report this to your distribution. Or better yet: install a vanilla kernel; there are often pre-built ones, and Thorsten maintains the one for Fedora. Or compiling a kernel yourself.

Once you have a vanilla kernel, verify that you still have the issue, and then report it directly upstream, not through your distro.

The second thing is to ensure you kernel is *fresh*. What does that mean ? First test the latest mainline kernel; most of the time, it means -rc releases. Do not use longterm (LTS) or stable kernel series for a report. The only exception is when you have a regression within a stable or longterm series.

Then you need to ensure your kernel integrity: verify that it's not *tainted*. You verify that you have 0 in `/proc/sys/kernel/tainted`. You should also make sure you're not using out-of-tree drivers, even if your kernel is not tainted. Before doing the report, disable all such drivers, and reboot without them to verify if you can still reproduce the issue.

All of this is described in the official kernel documentation on how to report the issues, which Thorsten wrote.

Don't forget to verify you hardware integrity (ex: overclocking), and check your dmesg log for errors.

Before sending: check that you are submitting your bug report to the right place. And this is not a simple task: the official bugzilla.kernel.org is most of the time the *wrong* place. You should check the `MAINTAINERS` file to find the proper place, most often a mailing list.

Finally, make your report as clear, short, and simple as possible.

Kind of issues

The security issues are usually those that people are being obliged to address. Data loss or devastating bugs as well. Regressions are also very important to the Linux kernel development process, and should be fixed very quickly.

Thorsten is working on tracking regressions, and has a dedicated bot for this; the EU funded his work in the past, and now Meta has stepped in to continue. If you see regression, don't forget to copy the regression mailing list when sending your report. Now, the regressions that are part of the rule are the ones visible from userspace; and of course you should have a similar kernel config.

During the reporting process, you will most likely be ask to find the culprit yourself, because you are the one that can reproduce it. This is done with git bisections. Once the commit introducing the issue is found, a fix is pretty much guaranteed; and reverting is always a possibility.

Some issues on the other hand are likely to be ignored. For example, Linux contains a few incomplete drivers. Or some drivers might face real-world issues, like the nouveau driver which might not have the necessary the knowledge to use some hardware features.

Sometimes code does not have an active maintainer. It remains in the kernel because it is useful to people, and removing it might break the no-regression rule. Other times, the maintainer might document that they are only doing Odd fixes, but a report should be sent. When code is fully Orphan, a report should be sent as well, but it's usually expected that no outside fix will happen.

To conclude, Thorsten says you should definitely look at the step-by-step guide to report issues, which is now part of the official documentation.

io_uring: path to zerocopy – Pavel Begunkov

The goal with modern I/O it to have zerocopy for maximum performance, and even peer-to-peer DMA where possible. Pavel has been working with networking and io_uring.

An example with network send requests: zerocopy is supported when using `MSG_ZEROCOPY`, but it requires locking on the buffer so that the app can process the data. The io_uring model is different.

Pavel Begunkov

It exposes two different models: storage-like and two-step; each has its own pros and cons, and depending on the type of network requests (TCP or UDP), one or the other would be better. The goal would be to have chose a model for networking with io_uring.

In the v1 patches, the storage-like model was chosen, but network developers weren't really pleased with the proposal. v2 used a notification registration model; it had much better performance. It managed to use io_uring registered buffers, bringing io_uring performance advantages. During testing with UDP, the zerocopy patches brought a nice advantage with bigger (4k) payloads.

What's being worked on: peer-to-peer DMA with dmabuf. It's still in discussion, and can feel hacky for now; `p2pdma` would need to be supported in the networking layer.

In the future, zerocopy receive might be worked on as well. Nothing new compared to the current solutions: with mmap, it would be like `TCP_ZEROCOPY_RECEIVE`. With provided buffers, a `zctap/AF_XDP` approach could be taken. But it requires hardware support.

That's it for this morning! Continue reading with the afternoon liveblog.

Live blog day 1: afternoon

by Anisse | Jun 1, 2022 | 0 comments

The live stream is still available. Read the morning liveblog.

Test-driven kernel releases – Guillaume Tucker

The Linux kernel might be heavily tested, because it's used all over the place. But this is a both hidden and duplicated testing effort, that isn't tracked upstream.

There's already some available testing available in the open for mainline though. syszbot is a syscall fuzzing tool with automated bisection and reproducer generation with a public web ui. KernelCI is a tailored CI system, that's distributed, built on kubernetes, has also automated bisection and public Web UI and API. regzbot in another, that focuses on regressions and manual submissions.

The goal of Guillaume's project is to aggregate all those test results for each release, and ideally provide them in-tree. But it's challenging because those need to be done before the release.

Guillaume Tucker

This started with the results reproducible on any hardware: the tests included in the kernel tree. Does the kernel build even with reference toolchains ? Then there is builds with sparse, coccicheck, KUnit, device tree validation.

The first idea (RFC) was to put the results in-tree, and rely on git history for previous results. The second idea was to use a link in release commits. Third proposal was to use git metadata. The goal is always to be compatible with the infamous email-based system linux kernel developers use.

Guillaume is looking for feedback on this concept, or how to move this idea forward. A good part of the audience agreed it would be a good idea, but there a few back and forth on details of implementation which might have an impact on everyone's workflow. For example, once you have the data in-tree, how do you decide what to do when tests don't pass ?

Some preferred the second proposal, with a central link that is referenced in commits but with the results kept out-of-tree. What matters to companies is the paper trail that shows that a given release was tested, not necessarily what is tested and how. Other think it might need to be actionable, not just a simple paper trail. There was also some question on whether or not test results should be stored forever, if it's free to store why not ? It was also added that this type of information is very useful for pushing companies and customers in general to update their kernel.

What's new in ftrace – Steven Rostedt

ftrace was originally the "function tracer". It was designed for embedded, and is greatly portable. You can watch previous kernel recipes talks for an introduction.

ftrace has so many features, that even Steven sometimes forgets about them, even though he is their author. This is a reason for this talk.

trace-cmd is a wrapper around the tracefs directory, and is much more powerful. Steven says you should use it.

kprobe tracing is kind of new (2009), and very powerful; for example it supports adding function argument tracing. Steven showed an example with ftrace and kprobes: how to use argument in kprobe to get a given function argument, dereferencing a pointer in order to get a given field, like the interface name.

uprobe tracing a bit newer (2012), and it's also possible to use them with ftrace. Steven showed to trace system-wide all the calls to malloc, and print the size argument in the ftrace buffer.

Steven Rostedt

Histograms were added in 2016, and have an event “trigger”. They are designed to count events. An example with syscalls was given. Syscalls only have two tracepoints: sys_enter and sys_exit, not for specific syscalls. A histogram with syscalls was created on the sys_enter tracepoint, with mapping from syscall number to symbol done directly in ftrace. Another example used the histograms with the malloc uprobes, with per-process total allocation size, and matching from pid to comm.

In 2018, synthetic events were added: they connect two different events into one event. For example, it’s possible to show the latency between the two events. Steven showed an example with wakeup latency: first, a synthetic event is created, with a name, pid and latency. Then a histogram is created the pid as the key, and the timestamp (from the ftrace ringbuffer) as the value. Then, a second histogram is created with a mapping from the next pid, creating the difference between the sched_waking and sched_switch events. It’s then possible to create new histograms from the synthetic event. Quite simple to use, and Steven is wondering: why is nobody using it ?

In 2020, libtracefs was released: it exposes almost all features of the tracefs system: it was extracted from trace-cmd. It contains an example program called `sqlhist` that helps with synthetic events. You write an sql query, and the program will parse it and convert it into ftrace synthetic events histograms. It’s much more intuitive.

Another nice example with `sqlhist` showed how to create synthetic events to measure how long tasks are blocked, and by which syscall. The first synthetic event got the syscall name that blocked (followed by a sched_switch with task uninterruptible). The second one counted the time it spent offcpu (unscheduled).

Event probes are from 2021: they extend trace events with kprobes. Steven realized he forgot to write the documentation for it (it is now done). He showed an example eprobe to add the interface name (like the first example). Another example showed all the files being opened with an eprobe on `openat`. But it wasn’t as simple because the filename from userspace might not have been mapped. So he used a synthetic event with `sqlhist` to join both the `sys_enter_openat` and `sys_exit_openat`, and get both the filename (already available), and the return code.

bootconfig was added in 2020, and allows loading a config file into a kernel to enable a set of ftrace options during boot.

In 2022, the `CUSTOM_TRACE_EVENT` macro was added to be able to only select a subset of information in an event, saving on ringbuffer memory used by events.

Idmapped Mounts – Christian Brauner

VFS Ownership is usually stored into the filesystem, but not necessarily. Internally, at the filesystem level, this is usually read with `i_uid_read`/`i_uid_write` functions, which then translate from `struct inode` raw uids into kuids or vice versa. That’s where idmappings come into play. They map userspace ids to kernel-space ids, by ranges, usually in a given user_namespace. This is noted with `u:k:r` notation: userpace-id:kernel-id:range.

When a user namespace is created, it has a default idmapping: `u0:k0:r4294967296`. `make_kuid` and `from_kuid` functions are used to do the mapping.

Christian Brauner

When opening an existing file, a lookup is done if the inode is not in the icache. Depending on whether or not the fs is mounted with idmapping, the kernel id might be different from the on-disk user id. The same is done when creating a file: the process `current_fsuid()` is fetched from the current user namespace and converted through the idmapping into the filesystem.

To alter ownership filesystem-wide, for example for a filesystem mounted by an unprivileged user, the relevant idmapping is determined at mount time.

In systemd, it's now possible to have portable home directories: a way to have a home directory that can work on multiple machines. This works by allowing a pseudo-random uid at login time. For this usecase, it recursively changes with `chown()` the permissions if the login uid/gid changed.

For simple process isolation, unprivileged containers are very useful, but make filesystem interactions difficult if sharing outside the container is needed. They also need recursive ownership changes, and it wastes space and makes container startup expensive.

Idmapped mounts allow to fix the issues in these two use cases. The file ownership can be changed locally (per mount), and temporarily (for the lifetime of the mount). Christian showed an example of a bind mount with an idmapping, doing exactly this, using a patched mount binary which should soon be upstream.

The internals are all documented in the idmappings official doc.

The idmappings cannot be changed after being mounted. It currently needs patching in filesystems. It started in linux 5.12 with ext4, fat and xfs, linux 5.15 had btrfs and ntfs3, 5.18 had f2fs and 5.19 will have both erofs and overlayfs support. This feature is heavily tested in xfstests.

Userspace support has already reached many projects: systemd, containerd, crun, runC, LXC, LXD, podman, the OCI spec and mount in util-linux.

Rethinking the kernel camera framework – Ricardo Ribalda

With his team, Ricardo has been working on improving the way camera are used with a new user/kernel framework: CAM.

Unlike standard joystick abstractions, Cameras are special. For example, there is now support for more than 200 video formats in the linux kernel. These can't really be converted on the fly in the kernel: it would be too costly. They also have complex input parameters, and sometimes multiple ways to reach the same result. In addition, the video bandwidth can be huge, and need low latency. So this leads to things being done in hardware. Cameras are also driving the consumer market.

In video4linux, the model has changed over the years: from simple cameras where the driver did all the work, to the internals of modern cameras being exposed and needing a lot of software to configure all of it. This is where libcamera comes into play (see the presentation by Laurent Pinchard at Embedded Recipes).

And Ricardo says that we have been living a lie: sensor data as it comes, raw, already needs a lot of software transformation before getting a useful image.

CAM is a new kernel subsystem (KCAM), which provides much simpler abstractions on top of hardware components: Entities, and Operations. Entities are organized in a tree, have a single register-set, and can throw events. Operations are a way we modify an entity; for example, reading or writing to a regmap, transferring a parameter buffer. An operation can depend on other operations or events; it can create a fence for synchronization.

In practice, a list of operations is sent by an app to the kernel via an ioctl (soon to be changed with an io_uring), and it is run by the kernel.

This new prototype framework is heavily tested: with kunit, libkc, vcam, error injections. It is tested on hardware with the ChromeOS test infra.

The main goal of CAM is to get smaller drivers, moving more work in the userspace (using libcamera, too) app. It's a full blank-slate approach from video4linux, changing the model from streams to operations.

The governance model will be similar to DRM: any driver must have an open source userspace stack before being merged.

The goal is to test this stack in ChromeOS first to benchmark it, and then of course proposing it upstream; this presentation is the first step for the latter, Ricardo says.

That's it for today ! Read the day 2 morning live blog.

Live blog day 2: morning

by Anisse | Jun 2, 2022 | 0 comments

We are back for another day of live-blog ! See yesterday afternoon's liveblog.

You can find the live stream here.

The kernel self-protection project and how you can help – Gustavo A. R. Silva

The kernel self-protection project's (KSPP) focus is hardening the Linux kernel. It is an upstream project, not a downstream fork, with a central repository of patches to try. Its goal is not to fix bugs, but to eliminate entire bug classes and methods of exploitation. Gustavo felt important to mention that the project does not care about CVEs or individual issues.

Coordination happens on the linux-hardening mailing list, which was created in 2020 as a goal to have a list where the discussion on small development tasks and details, not necessarily new ideas. There's also a patchwork instance to keep track of patches and tags. While the project is not a subsystem, it has maintainers.

The project also has an issue tracker, used to keep track of work, document a few things, and having some good first issues for new contributors.

During his first pull request to Linus, Gustavo sent a pull-request with a huge merge commit message, detailing all the changes; but it was too expansive so it had to be redone.

Gustavo A. R. Silva

The project uses Coverity to find defects in kernel code, as well a Coccinelle. The latter is used to make transforms from some buggy idioms to safer alternative, like the migration to the new `struct_size()` macro, for variable-sized structs that end with a 0-size array for appending data.

For some changes, the Kernel Test Robot from Intel's `Build-Tested-by` tag is included: it allows testing wide variety of architectures, configs and toolchains.

It takes sweat and blood to harden the kernel, Gustavo jokes about. It's a project with big goals, and is usually not glamorous. A strategy to fix that is to make the compiler an ally.

Enabling compiler options is the way forward, in order to detect as many problems as possible at build-time. But it takes time when sending patches to convince everyone of the usefulness of the changes, which might turn political in some cases. This is though and thankless work as well, Gustavo says, but it needs to be done.

Changes to fix compiler warnings and cleanups might feel like mechanical changes; maintainers don't like to see mechanical changes outside of their tree, so it means each changes should be sent to the proper maintainer.

Most of the changes related to compiler warnings have a lot false positives, and can feel very noisy. But they also find real bugs. And they help improve compilers as well because the project actually finds corner-cases in the compilers themselves.

The work is hard and long, but is important: it helps improve the quality of the code in the long term.

Ongoing and finished changes

`struct_group()` is changed aimed at expliciting changes that go across struct boundary to adjacent objects, usually through a single `memset` or `memcpy`. This helps enabling compiler warnings like `FORTIFY_SOURCE` which strengthen the use of these operations.

`FORTIFY_SOURCE` has also seen improvements in Linux by using `__builtin_object_size()` compiler builtin properly for bounds checking.

Flexible array transformations were used to fix how variable-length arrays were used at the end of structs. The zero-length and 1-length arrays were replaced by with flexible-length arrays and the `struct_size()` length computation macro.

This lead to an issue being found: compilers treated any trailing arrays in struct as flexible arrays, which breaks `FORTIFY_SOURCE`. This is isn't fixed yet in compilers, and some legacy code actually uses this now.

`-Warray-bounds` warning being enabled also helped find a lot of bugs. About 40000 fallthrough warnings were fixed and the warning enabled by default as well.

The project is also looking for help, and has many things that need to be fixed. A good place to start is the public issue tracker.

Trust and the Linux development model – Greg KH

"With enough eyes, all bugs are shallow". That's the old open source motto, and it isn't that much discussed anymore as core advantage of open source. Code can be audited (and fixed!) by anyone, at any time, and it's possible to go back in time and audit whatever happened: both in mailing lists and code history.

This talk is about the "University of Minnesota incident" (UMN), or how to **not** do research. From 2018 to 2020, a team from this university tried to submit "Hypocrite Commits", or intentionally wrong patches. They submitted a total of 5 of those, and only one was accepted. It turned out later that it was, in fact, correct, and not hypocrite. In 2020, a paper was published on this.

It continued again in 2021, and "researchers" continued sending poor quality patches. Greg did not like it at all, and publicly said that they were doing things wrong. All past patches from `@umn.edu` were reviewed, and new ones rejected. The Linux Foundation stepped in and sent a letter to the university, asking to stop the "research", document what they did, and retract the paper.

The paper was retracted in the end, hours before IEEE was about to revoke it. This Linux Foundation Technical Advisory Board (TAB) published a full report on the incident.

In the TAB report, it was documented that all the patches that were wrong were rejected. The accepted "wrong" `patch 1` was in fact correct, but was reverted in the end because it was submitted under a false name, which is against the Linux kernel guidelines.

Almost all of those patches were for obscure drivers in error handling "cleanup paths". These code paths were never hit in the real world. About 20% of patches submitted to fix things were in fact wrong. It falls under Hanlon's razon, Greg says: "Never attribute to malice that which is adequately explained by stupidity."

The biggest issue of the "Hypocrite Commits" is that they signed the legal statement, the Developer Certificate of Origin (DCO, with Signed-off-by) while using a fake name. And it's really not ethical, Greg says, which led to lawyers being involved.

As said before, hypocrite `patch 1` was correct despite the incorrect intention. `patch 2` was detected as incorrect by Greg because it tried to address a syzbot issue that was already fixed. `patch 3` and `4` were caught by maintainers whom suggested a proper way to fix the issues. Those suggestions were ignored, wasting the maintainers' time. Bonus `patch 5` was in fact a correct one, but reject because they sent it from a machine set-up for hypocrite commits, and it had a fake name: James Bond.

Greg KH

All the fake patches were caught. Does that mean the development model works ? Greg thinks that in fact the project got lucky.

In 2021, the UMN responded to the TAB report, saying that Linux was the only project they sent hypocrite patches to, and that it was over. Later in the year, in November, they asked Kees Cook (and Greg) if they could send patches again. They refused. Despite this, patches appeared on the mailing list in December. In January, the developers noticed that patches were wrong. UMN was notified, but they feigned ignorance.

There is now an official researchers guidelines documentation in linux. It asks for more information from researcher on patches, and why and how they fix things.

In April 2022, UMN brought in a kernel developer to help fix their program.

Should you trust the Linux kernel community ? As a basis, no, Greg says: the license says it comes with no warranty. Some people want every contributor to be vetted. But it's done at such a scale, worldwide, that it's impossible. Can you review everything that's going in ? There were 79662 total

commits in 2021, and no single group can review all of that. Most bugs are fixed before they reach a final release anyway.

The top developers of the kernel are also the top fixers. And they are also the top bug introducers. Which is fully expected. Most prolific developers write a ton of security bugs, and Greg includes himself in the bug introducers. So what needs to be done, and is being done is to make it easier and faster to find bugs.

From the outside, you should Trust but, verify; or in developer terms: Trust, but test. But from the inside, the main model of Linux kernel development, is not built around trust that code will be correct. It's trust that people will stick around fix things when bugs inevitably happen.

Developing Tilck, a tiny Linux-compatible kernel – Vladislav Valtchev

The tilck project includes a monolothic kernel written in C, a bootloader, test suites, and buildroot-like scripts for building 3rd party software.

It's partially compatible with Linux, and only runs on 32-bit x86. It's an educational project at the moment, distributed under the BSD-license. It's not an attempt to replace Linux, Vlad says.

Binary compatibility with Linux was chosen to have a measure of robustness by running 3rd party software, and not deal with designing a new syscall interface, or designing new toolchains. It can use binary linux libc builds for example.

It's a very small and simple OS, leading to less memory footprint, low latency. It aims for robustness and deterministic behavior. Developer experience is also a focus, spanning from the project simplicity.

Vladislav Valtchev

Vlad showed a demo of the build toolchain to show how simple it is to build tilck with a single command, and then run it in qemu. It boots in a hundred milliseconds, and then runs a busybox-based system. It has a sysfs implementation, so that tools that need it on Linux can work. Its tty implementation can run vi, and even vim colored syntax highlighting. And of course, it runs doom.

The OS supports a native syscall tracer, with filtering, which was used to debug the missing tty escape sequences in order to be able to run vim. It has multiple built-in testsuites: for userspace, system, and kernel. But it wasn't enough to test all codepaths, so to improve coverage, Vlad wrote tests that analyzed the serial and framebuffer outputs.

Vlad shared a few bug and optimization stories around Tilck: from an overcommit bug, to a framebuffer font rendering optimization war story. While starting quite slow, character rendering in Tilck has been improved on a lot, reaching faster speeds than Linux in the end.

That's it for this morning ! Continued reading this afternoon's live blog.

Live blog day 2: afternoon

by Anisse | Jun 2, 2022 | 0 comments

This is the link for this afternoon's livestream.

HID-BPF – Benjamin Tissoires

HID-BPF combines Human Interface Devices (HID) and eBPF (extended Berkeley Packet Filter). It is currently in review upstream.

HID is a simple protocol to report input devices events. It is well supported in Linux, and most devices work with the generic driver. Custom drivers are usually quite short (less than 100 lines), and only do a few transforms.

eBPF is an interesting technology that Benjamin saw at many conferences, and was wondering what to do with it. You'll get a more complete introduction on it with Alexei's talk tomorrow.

HID-BPF works on arrays of bytes, and is not intended to do any smart processing. It aims to work in a Compile-Once Run Everywhere mode, where users don't need to have llvm installed.

In HID, there's a quite involved process to add a new quirk when a device is somewhat broken: a key is not properly reported. There's a long time between the first identification of the issue, to the fix reaching the user's through standard (distro) means.

With HID-BPF, quirks become much simpler to write and deploy: the developer sends a test program (blob + source) to the user, which tests if it fixes the issue, and then can keep using as-is if it works.

Another usecase for HID-BPF is to have some kind of firewall for HID devices: they can work in two directions (can both send and receive events), and this is often used to enable the firmware update mode. For example on linux, Steam opens-up game controllers to everyone (with `uaccess`) in hidraw mode; and SDL is happy with that. It also prevents a random Chrome plugin to initiate a controller firmware upgrade over the network.

Benjamin Tissoires

It's also possible to change dynamically the way a device work. For example, the Surface Dial is reported in the inputs as a Dial, but depending on the use case, one might want to report it as mouse, or mouse wheel; you would need to change the kernel driver to do that. You can also use HID-BPF to do HID tracing, in a more powerful way than just hidraw.

With HID-BPF, you can load programs for various types of events: when a device is plug, or a sends a report: you can for example change the neutral zone of a joystick dynamically.

HID-BPF is built on top of BPF, but outside of it. It uses `ALLOW_ERROR_INJECTION` to enable functions to be tracepoints for BPF programs. Kfuncs are used to export a kernel function as eBPF dynamic API; once the signature is done, the rest is done automatically by eBPF: argument checking, versioning etc.

To conclude, the goal is to simplify the easy fixes in the future. Quirks and live fixes will be simpler without having to update the kernel. Custom uAPIs for devices won't need to be added anymore: they could just use eBPF programs instead.

New brightness uAPI – Hans de Goede

The problems with `/sys/class/backlight` aren't very new: Hans did a presentation on them 8 years ago. With the current uAPI, there is no way to map a given backlight sysfs to a given display, so you might be lost if a you have multiple displays. Sometimes, there might even be multiple sysfs devices for a single display, so userspace won't know which one to use. The value 0 is also undefined: might be low backlight, or no backlight at all.

The new proposal add new display_brightness and display_brightness_max properties on the drm connector object (which handles a screen): this way the mapping between display and backlight is

clear. If the max is 0, it means brightness control is not supported. The latter can also change on hotplug events, for example if a monitor is plugged.

Hans de Goede

The main issue stems from the fact that in x86 laptops, there is a lot of technical debt: multiple control methods were used over the years, from the GPU, to any random firmware interface. The current approach registers a sysfs device for all of them on a laptop; sometimes a kernel heuristic might say a native device should be preferred, and then unregister all others (that were loaded asynchronously); but the rest of the time the userspace needs to pick the sysfs device to pick one (firmware, platform, or native ?). This is all very messy, which is how Hans is now making time to clean this up.

There are also probe ordering issues, between acpi_video backlight and native GPU drivers. But the latter, which are preferred, will take longer to probe; so the acpi_video_backlight userspace interface loading is delayed by up to 8 seconds after probing, giving time to the native driver.

Another ordering issue is between platform(-x86) drivers, and GPU drivers. GPU drivers load earlier, so sometime userspace might try to access backlight, but find it's not available (yet). Hotplugging events are needed here as well.

Looking at yourself – Arnaldo Carvalho de Melo

Linux has instrumentation to be able to look at itself: its own types, etc.; and it's not used only for humans, but can also serve runtime-decisions.

Arnaldo's (kernel) introspection story started a long time ago with `struct sock` which wasn't simple to debug with a complex hierarchy, and per-family variants. He looked at how gdb knew about all the type information, and worked with DWARF, ELF and ORC, to create `pahole`, a tool rebuilding the source with type information, and showing holes in structures, cache boundaries, etc.

ACME

After a while, someone from CERN came along, and added support for C++ to it, as well as a 32 to 64-bit migration.

`pahole` nowadays has the `--reorganize` option to automatically re-order struct fields, optimizing for size.

In addition to DWARF, the CTF format was added from Solaris; the core of `pahole` is now multi-format. DWARF has a few issues: debuginfo files are very big, going to hundred of megabytes. This is mainly due to type duplication: because types are per compilation unit (file).

At some point, BPF needed type info to pretty print maps, and to have CO-RE (Compile Once – Run Everywhere). This lead to the creation of BTF, the BPF Type Format. It reused the infrastructure created for CTF in pahole. BTF is de-duplicated, which makes it much smaller, and is available in `/sys/kernel/btf/vmlinux`. It's about 3MB nowadays, and very fast to load, which is why pahole uses it by default nowadays. For example, `pahole spinlock` will print the struct spinlock from the runtime BTF information.

You can dump the whole BTF information in a C header file with bpftool or pahole. BTF can store program lines for disassembly, and this is supported by `perf` as well.

If BTF is available, it allows perf to do some dynamic things, without patching the kernel. For example, handle the new argument in the `sched_switch` tracepoint dynamically at runtime to support multiple kernel versions.

It's possible to use `bpftool` to list, disassemble bpf programs, or even dump their maps.

Finally, pahole is now built in the kernel and modules build when using BTF. For now, pahole is required, but eventually compilers will produce BTF. In the next version 1.24, pahole will be able to encode BTF in parallel for faster builds.

In Rust, struct fields are reorganized by default (what `pahole --reorganize` does), but if you build BTF from this, the offsets might not be in order, which is not supported by the BPF verifier. Offsets for BTF rust builds, have been removed for now until a fix is found.

Some people are starting to build products that rely on BTF, like Tetragon from Cilium, or HID-BPF, in the presentation we had earlier.

That's it for today ! Continued tomorrow !

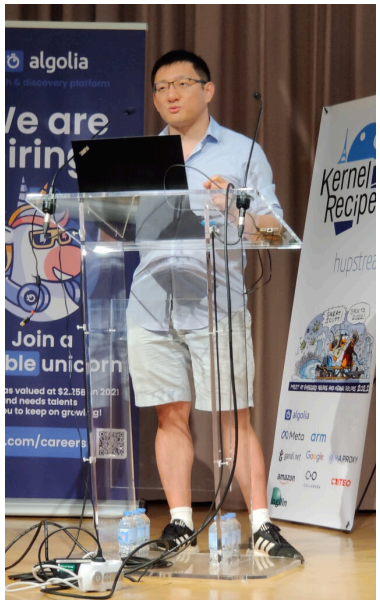
Live blog day 3: morning

by Anisse | Jun 3, 2022 | 0 comments

This is the live stream for this morning.

Single board computers made possible by the community – Da Xue

In 2016, Da had a crazy dream: he wanted to invest into a better future with an ecosystem with upstream development. So in 2017 he found what Neil from BayLibre was doing was awesome, and decided to have a partnership with them for the LePotato board. In 2018, the upstreaming of the video side of Amlogic SoCs with the Bootlin kickstarter was very interesting.



Da Xue

The goal is to get an upstream stack, from the bottom up: starting with the lower layers of the stack: ARM Trusted Firmware (ATF), edk2, optee, u-boot and Linux. Da showed a demo of the Renegade Elite board booting a standard openSUSE distribution.

Once upon an API – Michael Kerrisk

Michael cares about APIs, he says, and he wants them done properly. It all started in the beggining of `time()`, when `SIGCHLD` was sent to a parent process when a child process terminates. Then in 1997, someone decided that it would be nice to have the reverse, so they added a `prctl` flag to do this. And they did add a documentation for this feature.

But there was missing pieces; for example, it wasn't possible to discover if this option was set. So another `prctl` flag was added to get the value of the option. And that's where inconsistencies started. `PR_GET_PDEATHSIG` was a getter added that returned the result differently from `PR_GET_DUMPABLE`: as a value in the second argument instead of a function result. Nowadays, these inconsistencies have become the norm.

Another missing piece: the documentation that was added for the flag mentioned the option being cleared on `fork()`; but what about `execve` ? It wasn't mentioned until the documentation was updated in 2012. Would anyone have noticed the associated security vulnerability if this was documented earlier ? Indeed, if a suid-binary is `execve`-ed, it was then possible to send it signals, which wouldn't be allowed otherwise. This was fixed in 2007, 10 years after the feature was introduced.

In another example, a mis-design on an API was reported 15 years after it was done. And since it was part of the uAPI, it wasn't possible to fix it, so the quirk was just added to the documentation.

Back to the child signal: what happens after the subreaper feature is added ? A subreaper is a parent that wants sub-childs to be re-parented to it. But then, a child can have a series of parents, if there multiple subreapers in a hierarchy. This was documented later properly.

What about now if the the subreaper process is multi-threaded ? What happens when a thread of the subreaper parent terminates ? Usually child processes are parented by individual threads. Reparenting can also happen inside a given subreaper process, when a parent thread terminates.

Should these intricacies actually be documented ? Michael thinks so, because users will eventually depend on a feature, whether or not it is documented.

In this case, the feature exposes accidentally exposes internal linux kernel behavior: when a child gets multiple signals if the parent process is multi-threaded.

Michael Kerrisk

What went wrong in the end ? Multiple things. There was no single owner of the API, or interface. There was not enough documentation, and the interface evolved over time as new features

(subreapers) were added.

Who owns the interface contract ? Should it be the kernel developers, since they own the code. But sometimes, behavior might derive from the actual intention (it's a bug). Should it be the glibc developers, since they do the actual wrappers around it ? But those are pretty thin. Should it be the documentation, man-pages developers (Michael) ? But the documentation might be incomplete, or non-existent. Or should it the user-space developers themselves ? Given enough time, they will find intricacies of the API, and invent new use-cases based on them.

So, Michael wonders, if things had been documented upfront, would things have been different ? He thinks so. Whether it comes in the form of a man-page patch, or a very well written commit message, Documentation is a time multiplier: it makes everyone's life faster.

Most of the time, the issue is the lack of a big-picture view: de-centralized design does not work well, and this example is a perfect illustration. Michael thinks that there should be a paid kernel user-space API maintainer(s).

We won't live forever, what does that mean ? – David Miller

As a maintainer, you need to ensure continuity of the project.

In October 2019, Dave suffered a stroke, and it was a wake-up call, he says. You have to be able to delegate.

But maintainers don't grow on trees, and you can't find them overnight. You need to groom them; this needs to be thought well in advance. And it brings other advantages: once you delegate, you get more breathing room.

Plan for succession, David says. "A goal without a plan is just a wish". Find people you trust, and delegate to them.

David Miller

After a question from the room, David says he's very happy with his co-maintainers now, and trusts Jakub could become maintainer with little change to the workflows. Co-maintainers also need to be recognized in the community, have the ability to push back and be respected when making decisions.

On grooming, there are multiple step; one can become a trusted reviewer without being a maintainer. There are also discussions in the room on how to make sure whether or not younger people are interested in this field. Some thought there wasn't enough, but others raised that there were quite a few younger people in the room.

The story of BPF – Alexei Starovoitov

BPF is an instruction set designed 30 years ago. In 2011, a startup Alexei worked for, wanted to revolutionize Software Defined Networking (SDN). SDN is the equivalent of VMs, but for networking.

The traditional approach for VMs, what to use different kernel modules (kvm) for each feature; but at Alexei's startup Plumgrid, they decided to do it differently, with a single `iovisor.ko` module for switches, router, etc. that would load binary native code dynamically from userspace.

Alexei Starovoitov

But after a critical x86-dependent hard to debug issue, Alexei decided that this binary code injected needed to be verified. There were multiple x86-based iterations for an instruction set. They were always designed to be JIT-first, not interpreted. Then they wanted to have their solution upstream. So Alexei started talking with people, but this new instruction set looked scary to compiler developers, and even scarier to kernel maintainers.

To work around this, they decided to make it familiar. That's how the new instruction set was designed to be as closed as the original BPF, and called "extended" BPF (eBPF). In reality, there is not much in common with BPF, apart from the opcodes.

Before submitting anything, Alexei registered to the `netdev@` mailing list, and read all the messages for 6 months, in order to identify all the key people. His first kernel patch wasn't even related to eBPF: he moved bpf module free into a worker. He continued for a while with a few patches to fix issues to "build his reputation".

Finally, he posted the eBPF patchset. It was rejected. Why ? Mostly because of the UAPI. So it needed a plan B, to add it in the kernel, without adding a new UAPI. They decided to find something to make faster first. So Alexei rewrote the existing BPF interpreter, using eBPF opcode and implementation, but called “internal BPF”. The term “classic BPF” was coined by Daniel Borkman for the historical one.

In 2014, BPF code was converted to iBPF (the internal one), and JIT-ed. Neither eBPF nor the verifier technically existed upstream at the time. There was some arguments at the time on applying iBPF to networking.

So there was another pivot to try to apply it on tracing instead of networking. It started with filtering. Again, the strategy to make the existing code faster was applied. The tree walker filter was much slower than the BPF tracing classifier filter. Finally, by september 2014, the verifier landed: eBPF was born, and the team celebrated.

In only became useful later in the year when eBPF programs could be attached to sockets, then they could be attached to kprobes for tracing a few months later. But this was only the beginning. They had an instruction set, but no compiler.

The LLVM project had very different rules and way of working than the kernel. The fact that the instruction set was in the kernel did not really influence anything: tests had to be written, Alexei went to in-person meetups, etc. The backend monster patch was submitted in the end of 2014, and soon merged after many acks, but only as an experimental backend. It was in-tree, but could be removed or reverted at any time.

To graduate from experimental status, it had to have users, more than one developer, and participate in tree-wide changes. A build-bot had to be contributed as well.

For GCC, things were harder since the backend emitted BPF code directly (which was a blocker), so nothing was done because it was too much work. It was finally done by a separate team in 2019.

Being present at conferences doesn’t necessarily make difference: it is useful, but it won’t help improve the patches: the code matters the most.

To summarize the strategies to land the patches: build reputation, make things look familiar, improve performance, and be ready to compromise.

That’s it for this morning !

Live blog day 3: afternoon

by Anisse | Jun 3, 2022 | 0 comments

This is the live stream for this afternoon.

Checking your work: Linux kernel testing and CI – David Vernet

There are many ways to test the kernel: kselftests, KUnit, xfstests, etc.

kselftests are userspace programs, usually written in C, they are in tree at `tools/testing/selftests`. Some can be simple scripts, others have a kernel module counterpart. They output TAP format results.

KUnit is unit testing framework for testing individual Linux kernel functions. It's compiled in the kernel, and configured at build-time. C source files usual live next to driver code. There's a python script to run them in `tools/testing/kunit`

`xfstests` is a project focused on filesystems. And there are others that out of tree, like Linux Kernel Perfomance (lkp-tests), Phoronix Test suite (pts), Linux Test project(ltp).

For test run projects, there are a few options: KernelCI, Linux kernel test robot, patchwork, syzbot, etc.

KernelCI is now a Linux Foundation project. It builds the kernel across of variety of trees, branches, toolchains, configs and runs tests on lots of different architectures. All the results are shown on a web ui dashboard, with build and run logs available, etc.

Linux Kernel Performance is run by the 0-day team at Intel that also runs the kernel test robot. They also have a dashboard, which is in mailing list format. It test-builds patches before they are merged, which is quite useful.

Patchwork can be combined with some scripts and CI actions to have the build information. For example, the bpf project uses that, with Github action runners.

syszbot is the Google fuzzer; it has a dashboard; it focuses on fuzzing, so it doesn't run developer-written tests.

btrfs tests runs xfstests for btrfs, and hosts a dashboard. It's used to track performance of the project.

All those CI systems have the same goal: testing the kernel. Can they be combined ?

kselftests are nice, but need more work. They were initially intended as a dumping ground for small tests programs. The suites should be upgraded to advertise if a test is flaky or not.

A test system shouldn't annoy maintainers. Its goal is to alleviate pressure on maintainers. So flaky tests should be fixed or removed, as they provide negative value.

As a bonus after Q&A, David showed how to fast and easy it is to write a kselftest. It's integrated in the linux kernel build system.

The complete story of the in-kernel sloppy GPIO logic analyzer – Wolfram Sang

This is a story about hacking to scratch one's itch, Wolfram says. It works in Linux with irqs + preemption disabled by polling GPIOs. It's called "sloppy" for a reason.

For a common task, Wolfram would get a new board, and had to enable the IP cores on the SoC. For this one, he could not test his patches, because he did not have physical access to the board, which sat in Japan while he worked from Germany. Therefore, he could not submit untested work upstream.

He did not have multiple logic analyzers in the lab, or someone to operate them constantly remotely. But someone could setup wires once, and he had a lot of idle CPU cores.

He know that it was possible to build a software logic analyzer, but he wasn't an expert in CPU isolation.

Wolfram Sang

Despite that, he tried to get his work upstream, but it isn't merged yet.

SGLA (Sloppy GPIO Logic Analyzer) is the kernel part: it's configured and accessed through debugfs. It locks the CPU while it runs to do its task. The userspace part is a script that wraps debugfs access, isolates the CPU, and converts the output to sigrok data.

The device-tree configuration was quite simple, it configured the pins on which the analysis should be done, and bound them properly.

The display was done in sigrok, an open source GUI software for logic analyzers.

In the initial test, CPU isolation did not work because SMP support was not done yet on the prototype. So he ran the code on the other CPUs without Linux SMP support anyway.

While doing the testing, he found that he did not need to have custom wires to do the analysis: he could directly read the GPIO value even if the pins were muxed to i^2C on this hardware. But it needed support in the GPIO subsystem: that is supported as non-strict mode; and in the pinctrl driver for the hardware: luckily, this was a simple patch.

On the next hardware revision, the hardware no longer supported non-strict mode with GPIOs.

In the current incarnation, it works reasonably well, although it's still not a logic analyzer. It was fun to create though.

Linux on RISC-V – Drew Fustini

RISC-V is the clean slate design RISC instruction set architecture coming out of Berkeley. It has a 32bits, 64bits as well as 128bits variants for future-proofing.

There are 32 general purpose registers, and many extensions for other features; the most common variant for Linux is RISCV64 **GC**, for general purpose.

Since the RISC-V extensions vary a lot, general "profiles" are in the work for pre-selecting a set of extensions: one for microcontrollers and one for application processors. There are multiple books on the architecture already.

The specifications are controlled by RISC-V International, a non-profit, with 2700+ members. Many vendors have already shipped many RISC-V cores, like NVIDIA in their GPUs, or Seagate in their disk drives.

There is no ISA licensing fee or royalties, avoiding some complexities. The ISA is open, but the cores can be proprietary or open source. The open ISA makes the latter possible, and there are a already few open source cores, from many different groups.

Drew Fustini

It also has well supported software ecosystem: support is in many OSes, toolchains and libraries, languages and runtimes.

The RISC-V architecture has three privileges mode: user, supervisor (OS), and machine (firmware). There are few Control Status Registers to get the machine status and configure it. These are used for many things, like controlling virtual memory, supervisor mode or trap handling.

The spec is still being worked on, and there are for examples future interrupt handlers incoming to enhance the current PLIC and CLIC ones.

There's a non-ISA RISC-V specification for describing the Supervisor Binary Interface (SBI), the calling convention between supervisor and machine mode.

In RISC-V, a Hart is the Hardware Thread. In the context of SMT, a single core might have multiple harts.

There are SBI extensions, and those are used to control low-level platform features: Hart State Management (lifecycle), Performance, system reset. The hypervisor extension adds another level between machine and supervisor, as well as SBI interfaces for communication between levels.

There's an open source implementation of SBI called openSBI.

UEFI Support is also available. Both u-boot and edk2 have support for RISC-V EFI mode, and grub2 can be used as a RISC-V payload. There's a RISC-V EFI Boot Protocol for discovering the boot Hart.

The RISC-V platform specification describes common requirements for hardware in order to be able boot an OS. It has different levels/profiles depending the type of OS: generic, embedded or server. There's a RISC-V ACPI Platform Specification for RISC-V specific tables like Hart and Timer capabilities.

QEMU can easily emulate both RISCV32 and RISCV64, and supports some extensions like hypervisor mode.

In Linux, most relevant architecture-dependent features are implemented in the RISC-V architecture. Recently, KVM support was added, as well as 5-level page tables to have 57 bits of address space. Perf support was improved. In 5.18, cpuidle and suspend drivers now support the SBI HSM extension; and the kernel can list the capabilities of a given CPU. In 5.19, `kexec_file()` support was submitted.

In progress, we have the vector ISA support in Linux for the new Vector 1.0 extension. As well as IPI and SSTC.

Fedora has been working on RISC-V for a while now. In Debian, 95% of the packages are supported in the non-official architecture port. In Ubuntu, official images are provided for some systems. Both Yocto and Buildroot have good support for RISC-V, too.

SiFive shipped a few high-end boards to be able to run the latest chips, but those were expensive are hard to get.

The Kendryte K210 is a much smaller board with 8MB of RAM with support in Buildroot to build a small Linux system.

T-Head is a RISC-V hardware and SoC (C910) from Alibaba with support for Android. They also have smaller SoC as well, the C906, with an official devboard (D1 Nezha) from Allwinner International that is very affordable. Some people have been working on linux mainline support for the D1. Unfortunately, it had a non-standard design (a non-coherent interconnect), which needed special support upstream in the form of the Svpbmt patchset, finally merged in 5.19.

It's possible to join RISC-V International free-of-cost as an individual to participate in the discussions. Drew also runs a bi-weekly virtual meetup for the community.

Powerful and programmable kernel debugging with drgn – Omar Sandoval

drgn is a "programmable debugger": it exposes a REPL with specific helpers instead of custom cli, and can work on both kernel core dumps and live linux kernel.

drgn was created because Omar came up against some very tricky bugs, and wasn't able to use existing tools to do what he needed to do. And it was designed from the start to be used as a library.

Omar Sandoval

The next part was a demo of the debugger. It uses a python shell by default. The processes are accessed through the `prog` variable, using the dictionary syntax. Omar showed how to iterate on the children of given running process, in a VM being debugged by `drgn`.

`drgn` provides many helpers to help with common manipulations: for example to list elements in a list. This is all documented inline within python with `help(drgn.helpers.linux)`; the same thing also available in the official drgn documentation.

Case Study

A report indicated a container creation failure with `ENOSPC`. Using `strace` and `retsnoop`, it was find to come from a limit on the number of IPC namespaces. But there were only a few IPC namespaces alive on the machines.

After analysis of the code, it was found to be an error returned when a given atomic counter in a hashtable reached a maximum value. Using `drgn`, one can search this hashtable for the proper id, and get the value of the counter. The counter did reach the maximum value, while there was only a few of those namespaces being used.

The decrement path has then to be analyzed. It was called in a kernel workqueue, so tasks were enumated with drgn to find if any locked task was in the system. The parameters of the workqueue callback function could then be printed. It was found that the free path of those free_ipc was using `synchronize_rcu`, making the close path slower; and delayed because it was in workqueue. A running crash-looping container was creating IPC namespaces fast enough that it was faster than the free mechanism, reaching the upper limit of IPC namespaces.

All this can be found dynamically in a familiar programming environment of drgn. And the python code doing this can be reused and shared.

Underneath, `drgn` uses a C library called `libdgrn` that does the heavy lifting and core abstraction.

The limitations of `drgn` are that it's racy for live targets, needs to be kept in sync with the kernel, and requires DWARF. The latter is being worked on by Stephen Brennan to use BTF, ORC and kallsyms. But BTF is missing local variable descriptions, which if they are added will make the file bigger: from ~4MB to ~6MB on Stephen's machine.

`drgn` can go beyond debugging, Omar says, thanks to its modular design: as a learning tool, for automation, etc.